

Fixed-Point Toolbox™

User's Guide

R2012b

MATLAB®

How to Contact MathWorks



www.mathworks.com
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Fixed-Point Toolbox™ User's Guide

© COPYRIGHT 2004–2012 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Version 1.1 (Release 14SP1)
March 2005	Online only	Version 1.2 (Release 14SP2)
September 2005	Online only	Version 1.3 (Release 14SP3)
October 2005	Second printing	Version 1.3
March 2006	Online only	Version 1.4 (R2006a)
September 2006	Third printing	Version 1.5 (R2006b)
March 2007	Fourth printing	Version 2.0 (R2007a)
September 2007	Online only	Revised for Version 2.1 (R2007b)
March 2008	Online only	Revised for Version 2.2 (R2008a)
October 2008	Online only	Revised for Version 2.3 (R2008b)
March 2009	Online only	Revised for Version 2.4 (R2009a)
September 2009	Online only	Revised for Version 3.0 (R2009b)
March 2010	Online only	Revised for Version 3.1 (R2010a)
September 2010	Online only	Revised for Version 3.2 (R2010b)
April 2011	Online only	Revised for Version 3.3 (R2011a)
September 2011	Online only	Revised for Version 3.4 (R2011b)
March 2012	Online only	Revised for Version 3.5 (R2012a)
September 2012	Online only	Revised for Version 3.6 (R2012b)

Fixed-Point Concepts

1

Fixed-Point Data Types	1-2
Scaling	1-4
Precision and Range	1-5
Range	1-5
Precision	1-6
Arithmetic Operations	1-10
Modulo Arithmetic	1-10
Two's Complement	1-11
Addition and Subtraction	1-12
Multiplication	1-13
Casts	1-19
fi Objects and C Integer Data Types	1-22
Integer Data Types	1-22
Unary Conversions	1-24
Binary Conversions	1-25
Overflow Handling	1-28

Working with fi Objects

2

Ways to Construct fi Objects	2-2
Types of fi Constructors	2-2
Examples of Constructing fi Objects	2-3
Cast fi Objects	2-12
Overwriting by Assignment	2-12

Ways to Cast with MATLAB Software	2-12
fi Object Properties	2-17
Data Properties	2-17
fimath Properties	2-17
numericType Properties	2-19
Setting fi Object Properties	2-20
fi Object Functions	2-23

Fixed-Point Topics

3

Set Up Fixed-Point Objects	3-2
Create Fixed-Point Data	3-2
View Fixed-Point Number Circles	3-18
Perform Binary-Point Scaling	3-31
Develop Fixed-Point Algorithms	3-37
Calculate Fixed-Point Sine and Cosine	3-48
Calculate Fixed-Point Arctangent	3-70
Compute Sine and Cosine Using CORDIC Rotation Kernel	3-96
Perform QR Factorization Using CORDIC	3-102
Compute Square Root Using CORDIC Hyperbolic Kernel	3-142

Convert Cartesian to Polar Using CORDIC Vectoring Kernel	3-148
Set Data Types Using Min/Max Instrumentation	3-154
Convert Fast Fourier Transform (FFT) to Fixed Point	3-168
Detect Limit Cycles in Fixed-Point State-Space Systems	3-179
Compute Quantization Error	3-191
Normalize Data for Lookup Tables	3-199
Implement Fixed-Point Log2 Using Lookup Table	3-205
Implement Fixed-Point Square Root Using Lookup Table	3-210
Set Fixed-Point Math Attributes	3-215

Working with fimath Objects

4

fimath Object Construction	4-2
fimath Object Syntaxes	4-2
Building fimath Object Constructors in a GUI	4-4
fimath Object Properties	4-6
Math, Rounding, and Overflow Properties	4-6
Setting fimath Object Properties	4-7
fimath Properties Usage for Fixed-Point Arithmetic	4-11
fimath Rules for Fixed-Point Arithmetic	4-11
Binary-Point Arithmetic	4-13

[Slope Bias] Arithmetic	4-17
fimath for Rounding and Overflow Modes	4-20
fimath for Sharing Arithmetic Rules	4-22
Default fimath Usage to Share Arithmetic Rules	4-22
Local fimath Usage to Share Arithmetic Rules	4-22
fimath ProductMode and SumMode	4-25
Example Setup	4-25
FullPrecision	4-26
KeepLSB	4-27
KeepMSB	4-28
SpecifyPrecision	4-30

Working with fipref Objects

5

fipref Object Construction	5-2
fipref Object Properties	5-3
Display, Data Type Override, and Logging Properties	5-3
fipref Object Properties Setting	5-3
fi Object Display Preferences Using fipref	5-5
Underflow and Overflow Logging Using fipref	5-7
Logging Overflows and Underflows as Warnings	5-7
Accessing Logged Information with Functions	5-9
Data Type Override Preferences Using fipref	5-12
Overriding the Data Type of fi Objects	5-12
Data Type Override for Fixed-Point Scaling	5-13

Working with numerictype Objects

6

numerictype Object Construction	6-2
numerictype Object Syntaxes	6-2
Example: Construct a numerictype Object with Property Name and Property Value Pairs	6-3
Example: Copy a numerictype Object	6-4
Example: Build numerictype Object Constructors in a GUI	6-5
 numerictype Object Properties	6-7
Data Type and Scaling Properties	6-7
Set numerictype Object Properties	6-8
 numerictype Structure of Fixed-Point Objects	6-11
Valid Values for numerictype Structure Properties	6-11
Properties That Affect the Slope	6-13
Stored Integer Value and Real World Value	6-13
 numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects	6-14
Example 1	6-14
Example 2	6-15

Working with quantizer Objects

7

Constructing quantizer Objects	7-2
 quantizer Object Properties	7-3
 Quantizing Data with quantizer Objects	7-4
 Transformations for Quantized Data	7-6

Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms

8

Code Acceleration and Code Generation from MATLAB	8-3
Requirements for Generating Compiled C Code Files	8-4
Functions Supported for Code Acceleration or Generation	8-5
Fixed-Point Code Acceleration and Generation Workflow	8-14
Set Up Compiler to Generate Compiled C Code Functions	8-15
Accelerate Code Using fiaccel	8-16
Speeding Up Fixed-Point Execution with fiaccel	8-16
Running fiaccel	8-16
Generated Files and Locations	8-17
Data Type Override Using fiaccel	8-20
File Infrastructure and Paths Setup	8-21
Compile Path Search Order	8-21
When to Use the Code Generation Path	8-21
Add Files to the Code Generation Path	8-22
Adding Folders to Search Paths	8-22
Naming Conventions	8-22
Detect and Debug Code Generation Errors	8-25
Debugging Strategies	8-25
Error Detection at Design Time	8-26
Error Detection at Compile Time	8-26
Set Up C Code Compilation Options	8-28
C Code Compiler Configuration Object	8-28

Compilation Options Modification at the Command Line	
Using Dot Notation	8-28
How fiaccel Resolves Conflicting Options	8-29
MEX Configuration Dialog Box Options	8-30
See Also	8-35
Specify Primary Function Input Properties	8-36
Why You Must Specify Input Properties	8-36
Properties to Specify	8-36
Rules for Specifying Properties of Primary Inputs	8-39
Methods for Defining Properties of Primary Inputs	8-40
Input Properties Definition by Example at the Command Line	8-40
Best Practices for Accelerating Fixed-Point Code	8-48
Recommended Compilation Options for fiaccel	8-48
Build Scripts	8-49
Check Code Interactively Using MATLAB Code Analyzer	8-50
Separating Your Test Bench from Your Function Code ...	8-51
Preserving Your Code	8-51
File Naming Conventions	8-51
Create and Use Fixed-Point Code Generation	
Reports	8-52
Code Generation Report Creation	8-52
Code Generation Report Opening	8-53
Viewing Your MATLAB Code	8-53
Viewing Variables in the Variables Tab	8-55
See Also	8-56
Generate C Code from Code Containing Global Data ..	8-57
Workflow Overview	8-57
Declaring Global Variables	8-57
Defining Global Data	8-58
Synchronizing Global Data with MATLAB	8-59
Limitations of Using Global Data	8-62
Define Input Properties Programmatically in MATLAB File	8-63

How to Use <code>assert</code>	8-63
Rules for Using <code>assert</code> Function	8-67
Example: Specifying Properties of Primary Fixed-Point Inputs	8-68
Example: Specifying Class and Size of Scalar Structure ..	8-69
Example: Specifying Class and Size of Structure Array ..	8-70
Control Run-Time Checks	8-71
Types of Run-Time Checks	8-71
When to Disable Run-Time Checks	8-72
How to Disable Run-Time Checks	8-72
Generation with MATLAB Coder	8-74
Code Generation with MATLAB Function Block	8-75
Composing MATLAB Language Function in Simulink Model	8-75
MATLAB Function Block with Data Type Override	8-75
Fixed-Point Data Types with MATLAB Function Block ...	8-76
Generate Fixed-Point FIR Code Using MATLAB Function Block	8-84
Program the MATLAB Function Block	8-84
Prepare the Inputs	8-85
Create the Model	8-85
Define the <code>fimath</code> Object Using the Model Explorer	8-87
Run the Simulation	8-88
Fixed-Point FFT Code Example Parameter Values	8-89
Accelerate Code for Variable-Size Data	8-92
Disable Support for Variable-Size Data	8-92
Control Dynamic Memory Allocation	8-93
Accelerate Code for MATLAB Functions with Variable-Size Data	8-94
Accelerate Code for a MATLAB Function That Expands a Vector in a Loop	8-96
Accelerate C Code for Fixed-Point Mean Value (TBD)	8-101

Create Embeddable C Code for Summing Values (TBD)	8-102
Propose Fixed-Point Data Types in a MATLAB Coder Project	8-103
Apply Fixed-Point Data Types in a MATLAB Coder Project	8-113
Code Generation Readiness Tool	8-119
What Information Does the Code Generation Readiness Tool Provide?	8-119
Summary Tab	8-120
Code Structure Tab	8-121
See Also	8-124
Check Code Using the Code Generation Readiness Tool	8-125
Run Code Generation Readiness Tool at the Command Line	8-125
Run the Code Generation Readiness Tool From the Current Folder Browser	8-125
See Also	8-125

Interoperability with Other Products

9

fi Objects with Simulink	9-2
Reading Fixed-Point Data from the Workspace	9-2
Writing Fixed-Point Data to the Workspace	9-2
Setting the Value and Data Type of Block Parameters	9-6
Logging Fixed-Point Signals	9-6
Accessing Fixed-Point Block Data During Simulation	9-6
fi Objects with DSP System Toolbox	9-7
Reading Fixed-Point Signals from the Workspace	9-7
Writing Fixed-Point Signals to the Workspace	9-7
fi Objects with dfilt Objects	9-11

Ways to Generate Code	9-12
-----------------------------	------

Calling Functions for Code Generation

10

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

Code Generation for MATLAB Classes

11

MATLAB Classes Definition for Code Generation	11-2
Language Limitations	11-2
Code Generation Features Not Compatible with Classes ..	11-4
Defining Class Properties for Code Generation	11-5

Calls to Base Class Constructor	11-6
Classes That Support Code Generation	11-9
Memory Allocation Requirements	11-10
Generate Code for MATLAB Value Classes	11-11
Generate Code for MATLAB Handle Classes and System Objects	11-17
MATLAB Classes in Code Generation Reports	11-20
What Reports Tell You About Classes	11-20
How Classes Appear in Code Generation Reports	11-20
How to Generate a Code Generation Report	11-22
Troubleshooting Issues with MATLAB Classes	11-23
Class <code>class</code> does not have a property with name <i>name</i> ...	11-23

Defining Data for Code Generation

12

Data Definition for Code Generation	12-2
Code Generation for Complex Data	12-4
Restrictions When Defining Complex Variables	12-4
Expressions Containing Complex Operands Yield Complex Results	12-5
Code Generation for Characters	12-6

Defining Functions for Code Generation

13

Specify Variable Numbers of Arguments	13-2
Supported Index Expressions	13-3
Apply Operations to a Variable Number of Arguments	13-4
When to Force Loop Unrolling	13-4
Using Variable Numbers of Arguments in a for-Loop	13-5
Implement Wrapper Functions	13-7
Passing Variable Numbers of Arguments from One Function to Another	13-7
Pass Property/Value Pairs	13-8
Variable Length Argument Lists for Code Generation	13-10

Defining MATLAB Variables for C/C++ Code Generation

14

Variables Definition for Code Generation	14-2
Best Practices for Defining Variables for C/C++ Code Generation	14-3
Define Variables By Assignment Before Using Them	14-3
Use Caution When Reassigning Variables	14-6
Use Type Cast Operators in Variable Definitions	14-6
Define Matrices Before Assigning Indexed Variables	14-6
Eliminate Redundant Copies of Variables in Generated Code	14-7

When Redundant Copies Occur	14-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables	14-7
Defining Uninitialized Variables	14-8
Reassignment of Variable Properties	14-9
Define and Initialize Persistent Variables	14-10
Reuse the Same Variable with Different Properties ...	14-11
When You Can Reuse the Same Variable with Different Properties	14-11
When You Cannot Reuse Variables	14-12
Limitations of Variable Reuse	14-14
Avoid Overflows in for-Loops	14-16
Supported Variable Types	14-18

Design Considerations for C/C++ Code Generation

15

When to Generate Code from MATLAB Algorithms ...	15-2
When Not to Generate Code from MATLAB Algorithms ..	15-2
Which Code Generation Feature to Use	15-4
Prerequisites for C/C++ Code Generation from MATLAB	15-6
MATLAB Code Design Considerations for Code Generation	15-7
See Also	15-8

Expected Differences in Behavior After Compiling	
MATLAB Code	15-9
Why Are There Differences?	15-9
Character Size	15-9
Order of Evaluation in Expressions	15-9
Termination Behavior	15-10
Size of Variable-Size N-D Arrays	15-10
Size of Empty Arrays	15-11
Floating-Point Numerical Results	15-11
NaN and Infinity Patterns	15-12
Code Generation Target	15-12
MATLAB Class Initial Values	15-12
Variable-Size Support for Code Generation	15-12
 MATLAB Language Features Supported for C/C++ Code	
Generation	15-13
MATLAB Language Features Not Supported for C/C++	
Code Generation	15-14

Code Generation for Enumerated Data

16

Enumerated Data Definition for Code Generation	16-2
 Enumerated Types Supported for Code Generation ...	16-3
Enumerated Type Based on int32	16-3
Enumerated Type Based on Simulink.IntEnumType	16-4
 When to Use Enumerated Data for Code Generation ..	16-6
 Generate Code for Enumerated Data from MATLAB	
Algorithms	16-7
How to Generate Code for Enumerated Data	16-7
 Generate Code for Enumerated Data from MATLAB	
Function Blocks	16-9
 Define Enumerated Data for Code Generation	16-10

Naming Enumerated Types for Code Generation	16-11
Instantiate Enumerated Types for Code Generation ..	16-12
Operations on Enumerated Data Allowed for Code	
Generation	16-13
Assignment Operator, =	16-13
Relational Operators, < > <= >= == ~=	16-13
Cast Operation	16-14
Indexing Operation	16-14
Control Flow Statements: if, switch, while	16-15
Include Enumerated Data in Control Flow	
Statements	16-16
if Statement with Enumerated Data Types	16-16
switch Statement with Enumerated Data Types	16-17
while Statement with Enumerated Data Types	16-20
Customize Enumerated Types Based on int32	16-22
About Customizing Enumerated Types	16-22
Specify a Default Enumerated Value	16-24
Specify a Header File	16-25
Customize Enumerated Types Based on	
Simulink.IntEnumType	16-28
Control Names of Enumerated Type Values in	
Generated Code	16-29
Change and Reload Enumerated Data Types	16-31
Restrictions on Use of Enumerated Data in	
for-Loops	16-32
Toolbox Functions That Support Enumerated Types for	
Code Generation	16-33

Code Generation for Function Handles

17

Function Handles Definition for Code Generation	17-2
Define and Pass Function Handles for Code Generation	17-3
Define and Pass Function Handles for Code Acceleration	17-5
Function Handle Limitations for Code Generation . . .	17-7

Generating Efficient and Reusable Code

18

Unroll for-Loops	18-2
Inline Functions	18-3
Eliminate Redundant Copies of Function Inputs	18-4
Generate Reusable Code	18-6

Code Generation for MATLAB Structures

19

Structure Definition for Code Generation	19-2
Structure Operations Allowed for Code Generation . . .	19-3
Define Scalar Structures for Code Generation	19-4

Restrictions When Using <code>struct</code>	19-4
Restrictions When Defining Scalar Structures by Assignment	19-4
Adding Fields in Consistent Order on Each Control Flow Path	19-4
Restriction on Adding New Fields After First Use	19-5
Define Arrays of Structures for Code Generation	19-7
Ensuring Consistency of Fields	19-7
Using <code>repmat</code> to Define an Array of Structures with Consistent Field Properties	19-7
Defining an Array of Structures Using Concatenation	19-8
Make Structures Persistent	19-9
Index Substructures and Fields	19-10
Assign Values to Structures and Fields	19-12
Pass Large Structures as Input Parameters	19-13

Functions Supported for Code Generation

20

Functions Supported for Code Generation — Alphabetical List	20-2
Functions Supported for Code Generation — Categorical List	20-75
Aerospace Toolbox Functions	20-76
Arithmetic Operator Functions	20-76
Bit-Wise Operation Functions	20-77
Casting Functions	20-77
Communications System Toolbox Functions	20-78
Complex Number Functions	20-78
Computer Vision System Toolbox Functions	20-79
Data Type Functions	20-80
Derivative and Integral Functions	20-80

Discrete Math Functions	20-81
Error Handling Functions	20-81
Exponential Functions	20-81
Filtering and Convolution Functions	20-82
Fixed-Point Toolbox Functions	20-82
Histogram Functions	20-91
Image Processing Toolbox Functions	20-91
Input and Output Functions	20-92
Interpolation and Computational Geometry	20-92
Linear Algebra	20-92
Logical Operator Functions	20-93
MATLAB Compiler Functions	20-93
Matrix and Array Functions	20-94
Nonlinear Numerical Methods	20-98
Polynomial Functions	20-98
Relational Operator Functions	20-98
Rounding and Remainder Functions	20-99
Set Functions	20-99
Signal Processing Functions in MATLAB	20-100
Signal Processing Toolbox Functions	20-100
Special Values	20-105
Specialized Math	20-105
Statistical Functions	20-106
String Functions	20-106
Structure Functions	20-107
Trigonometric Functions	20-107

Code Generation for Variable-Size Data

21

What Is Variable-Size Data?	21-2
Variable-Size Data Definition for Code Generation ...	21-3
Bounded Versus Unbounded Variable-Size Data	21-4
Control Memory Allocation of Variable-Size Data	21-5

Specify Variable-Size Data Without Dynamic Memory	
Allocation	21-6
Fixing Upper Bounds Errors	21-6
Specifying Upper Bounds for Variable-Size Data	21-6
 Variable-Size Data in Code Generation Reports	21-10
What Reports Tell You About Size	21-10
How Size Appears in Code Generation Reports	21-11
How to Generate a Code Generation Report	21-11
 Define Variable-Size Data for Code Generation	21-12
When to Define Variable-Size Data Explicitly	21-12
Using a Matrix Constructor with Nonconstant Dimensions	21-13
Inferring Variable Size from Multiple Assignments	21-13
Defining Variable-Size Data Explicitly Using coder.varsize	21-14
 C Code Interface for Arrays	21-19
C Code Interface for Statically Allocated Arrays	21-19
C Code Interface for Dynamically Allocated Arrays	21-20
Utility Functions for Creating emxArray Data Structures	21-21
 Troubleshooting Issues with Variable-Size Data	21-23
Diagnosing and Fixing Size Mismatch Errors	21-23
Diagnosing and Fixing Errors in Detecting Upper Bounds	21-25
 Incompatibilities with MATLAB in Variable-Size	
Support for Code Generation	21-27
Incompatibility with MATLAB for Scalar Expansion	21-27
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	21-29
Incompatibility with MATLAB in Determining Size of Empty Arrays	21-30
Incompatibility with MATLAB in Vector-Vector Indexing	21-31
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation	21-32
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks	21-34

Restrictions on Variable Sizing in Toolbox Functions	
Supported for Code Generation	21-35
Common Restrictions	21-35
Toolbox Functions with Variable Sizing Restrictions	21-36

Primary Functions

22

Primary Function Input Specification	22-2
When to Specify Input Properties	22-2
Why You Must Specify Input Properties	22-2
Properties to Specify	22-3
Rules for Specifying Properties of Primary Inputs	22-8
Methods for Defining Properties of Primary Inputs	22-8
Define Input Properties by Example at the Command Line	22-9
Specify Constant Inputs at the Command Line	22-12
Specify Variable-Size Inputs at the Command Line	22-14
 Define Input Properties Programmatically in the	
MATLAB File	22-16
How to Use assert with MATLAB Coder	22-16
Rules for Using assert Function	22-23
Specifying General Properties of Primary Inputs	22-24
Specifying Properties of Primary Fixed-Point Inputs	22-25
Specifying Class and Size of Scalar Structure	22-25
Specifying Class and Size of Structure Array	22-26
 Accelerate Code for Variable-Size Data (TBD)	23-2
 Enable and Disable Dynamic Memory Allocation (TBD)	23-3
 Fix Runtime Stack Overflows (TBD)	23-4
 Check Code Using the MATLAB Code Analyzer	23-5
 Fix Errors Detected at Code Generation Time	23-6

See Also	23-6
----------------	------

24

System Objects Supported for Code Generation

System Objects Supported for Code Generation	24-2
Code Generation for System Objects	24-2
Computer Vision System Toolbox System Objects	24-2
Communications System Toolbox System Objects	24-7
DSP System Toolbox System Objects	24-13

25

System Objects

Create System Objects	25-2
Create a System object	25-3
Define a New System object	25-3
Change a System object Property	25-4
Check if a System object Property Has Changed	25-4
Run a System object	25-4
Display Available System Objects	25-5
 Set Up System Objects	 25-6
Create a New System object	25-6
Retrieve System object Property Values	25-6
Set System object Property Values	25-7
 Process Data Using System Objects	 25-11
What are System object Methods?	25-11
The Step Method	25-11
Common Methods	25-13
Advantages of Using Methods	25-15
 Tuning System object Properties in MATLAB	 25-16
Understand System object Modes	25-16

Change Properties While Running System Objects	25-17
Change System object Input Complexity or Dimensions . .	25-18
Find Help and Examples for System Objects	25-19
Use System Objects in MATLAB Code Generation	25-21
Considerations for Using System Objects in Generated Code	25-21
Use System Objects with codegen	25-26
Use System Objects with the MATLAB Function Block . . .	25-26
Use System Objects with MATLAB Compiler	25-26

Index

Fixed-Point Concepts

- “Fixed-Point Data Types” on page 1-2
- “Scaling” on page 1-4
- “Precision and Range” on page 1-5
- “Arithmetic Operations” on page 1-10
- “fi Objects and C Integer Data Types” on page 1-22

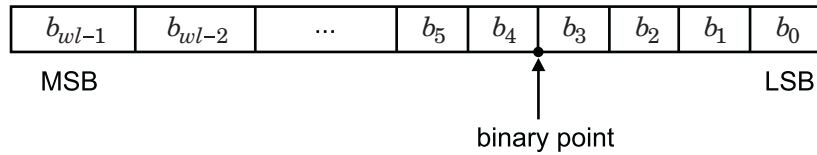
Fixed-Point Data Types

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. This chapter discusses many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the i^{th} binary digit.
- wl is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is the only representation used by Fixed-Point Toolbox™ documentation. Refer to “Two's Complement” on page 1-11 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

$$\text{real-world value} = (\text{slope} \times \text{integer}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In Fixed-Point Toolbox documentation, the negative of the fixed exponent is often referred to as the *fraction length*.

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in [Slope Bias] representation that has a bias equal to zero and a slope adjustment factor equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

$$\text{real-world value} = 2^{\text{fixed exponent}} \times \text{integer}$$

or

$$\text{real-world value} = 2^{-\text{fixed exponent}} \times \text{integer}$$

Fixed-Point Toolbox software supports both binary point-only scaling and [Slope Bias] scaling.

Note For examples of binary point-only scaling, see the Fixed-Point Toolbox Binary-Point Scaling example.

Precision and Range

In this section...

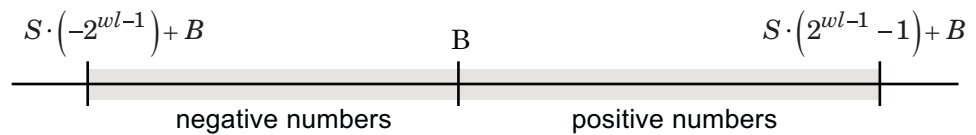
“Range” on page 1-5

“Precision” on page 1-6

Note You must pay attention to the precision and range of the fixed-point data types and scalings you choose in order to know whether rounding methods will be invoked or if overflows or underflows will occur.

Range

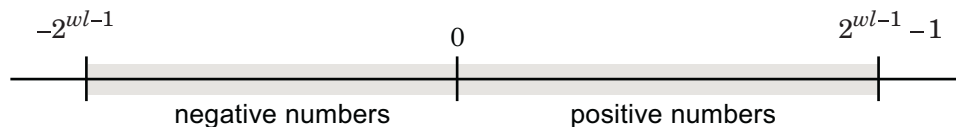
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two’s complement fixed-point number of word length wl , scaling S and bias B is illustrated below:



For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl} .

For example, in two’s complement, negative numbers must be represented as well as zero, so the maximum value is $2^{wl-1} - 1$. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1} :

For slope = 1 and bias = 0:



Overflow Handling

Because a fixed-point data type represents numbers within a finite range, overflows and underflows can occur if the result of an operation is larger or smaller than the numbers in that range.

Fixed-Point Toolbox software allows you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. Refer to “Modulo Arithmetic” on page 1-10 for more information.

When you create a `fi` object, any overflows are saturated. The `OverflowMode` property of the default `fi` object is `saturate`. You can log overflows and underflows by setting the `LoggingMode` property of the `fipref` object to `on`. Refer to “LoggingMode” for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Methods

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, a rounding method is used to cast the value to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding method itself. To provide you with

greater flexibility in the trade-off between cost and bias, Fixed-Point Toolbox software currently supports the following rounding methods:

- `ceil` rounds to the closest representable number in the direction of positive infinity.
- `convergent` rounds to the closest representable number. In the case of a tie, `convergent` rounds to the nearest even number. This is the least biased rounding method provided by the toolbox.
- `fix` rounds to the closest representable number in the direction of zero.
- `floor`, which is equivalent to two's complement truncation, rounds to the closest representable number in the direction of negative infinity.
- `nearest` rounds to the closest representable number. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity. This rounding method is the default for `fi` object creation and `fi` arithmetic.
- `round` rounds to the closest representable number. In the case of a tie, the `round` method rounds:
 - Positive numbers to the closest representable number in the direction of positive infinity.
 - Negative numbers to the closest representable number in the direction of negative infinity.

Choosing a Rounding Method. Each rounding method has a set of inherent properties. Depending on the requirements of your design, these properties could make the rounding method more or less desirable to you. By knowing the requirements of your design and understanding the properties of each rounding method, you can determine which is the best fit for your needs. The most important properties to consider are:

- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?
 - Low — The method requires few processing cycles.
 - Moderate — The method requires a moderate number of processing cycles.
 - High — The method requires more processing cycles.

Note The cost estimates provided here are hardware independent. Some processors have rounding modes built-in, so consider carefully the hardware you are using before calculating the true cost of each rounding mode.

- Bias — What is the expected value of the rounded values minus the original values: $E(\hat{\theta} - \theta)$?
 - $E(\hat{\theta} - \theta) < 0$ — The rounding method introduces a negative bias.
 - $E(\hat{\theta} - \theta) = 0$ — The rounding method is unbiased.
 - $E(\hat{\theta} - \theta) > 0$ — The rounding method introduces a positive bias.
- Possibility of Overflow — Does the rounding method introduce the possibility of overflow?
 - Yes — The rounded values may exceed the minimum or maximum representable value.
 - No — The rounded values will never exceed the minimum or maximum representable value.

The following table shows a comparison of the different rounding methods available in both Fixed-Point Toolbox and Simulink® Fixed Point™ products.

Fixed-Point Toolbox Rounding Method	Simulink Fixed Point Rounding Mode	Cost	Bias	Possibility of Overflow
ceil	Ceiling	Low	Large positive	Yes
convergent	Convergent	High	Unbiased	Yes
fix	Zero	Low	<ul style="list-style-type: none"> • Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples 	No
floor	Floor	Low	Large negative	No
nearest	Nearest	Moderate	Small positive	Yes
round	Round	High	<ul style="list-style-type: none"> • Small negative for negative samples • Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples 	Yes
N/A	Simplest (Simulink Fixed Point only)	Low	Depends on the operation	No

Arithmetic Operations

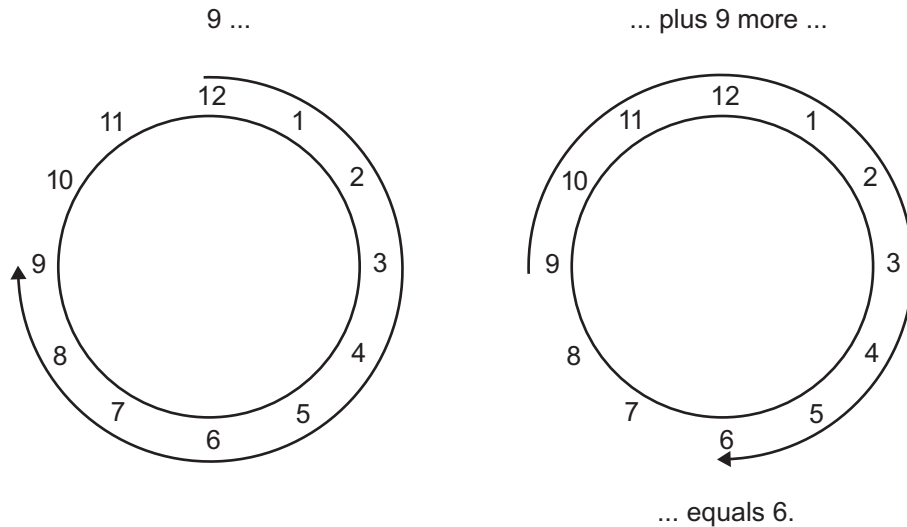
In this section...
“Modulo Arithmetic” on page 1-10
“Two’s Complement” on page 1-11
“Addition and Subtraction” on page 1-12
“Multiplication” on page 1-13
“Casts” on page 1-19

Note These sections will help you understand what data type and scaling choices result in overflows or a loss of precision.

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the “clock” system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:



Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped “around the circle” to either 0 or 1.

Two’s Complement

Two’s complement is a way to interpret a binary number. In two’s complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit of a two’s complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two’s complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 = (0 + 2^0) = 1$$

$$11 = \left((-2^1) + (2^0) \right) = (-2 + 1) = -1$$

To compute the negative of a binary number using two’s complement,

- 1 Take the one’s complement, or “flip the bits.”

- 2** Add a $2^{(-FL)}$ using binary math, where FL is the fraction length.
- 3** Discard any bits carried beyond the original word length.

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

$$11010 \rightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

$$\begin{array}{r} 00101 \\ +1 \\ \hline 00110 \end{array} \quad (6)$$

Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

$$\begin{array}{r} 010010.1 \quad (18.5) \\ +0110.110 \quad (6.75) \\ \hline 011001.010 \quad (25.25) \end{array}$$

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign-extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

$$\begin{array}{r} 010010.100 \quad (18.5) \\ -0110.110 \quad (6.75) \\ \hline \end{array} \quad \xrightarrow[\text{and sign extension}]{\text{two's complement}} \quad \begin{array}{r} 010010.100 \quad (18.5) \\ +111001.010 \quad (-6.75) \\ \hline 1001011.110 \quad (11.75) \end{array}$$

Carry bit
is discarded

The default `fimath` has a value of 1 (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

If `CastBeforeSum` has a value of 0 (false), the addends are added with full precision maintained. After the addition the sum is then quantized.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign-extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

The diagram illustrates the multiplication of 10.11 (-1.25) and 011 (3). The first factor, 10.11, is shown with a sign extension of 1 to the left of the binary point, resulting in 11011. The second factor, 011, is shown with a sign extension of 0 to the left of the binary point, resulting in 1011. The product is 1100.01, which is the sum of the two intermediate results. The diagram includes two annotations: one pointing to the extra 1 in the first factor, stating 'The extra 1 is the result of necessary sign extension.', and another pointing to the fractional part of the product, stating 'The number of fractional bits of the result is the sum of the number of fractional bits of the factors.'

$$\begin{array}{r}
 10.11 \text{ } (-1.25) \\
 \quad 011 \text{ } (3) \\
 \hline
 11011 \\
 \quad 1011 \\
 \hline
 1100.01 \text{ } (-3.75)
 \end{array}$$

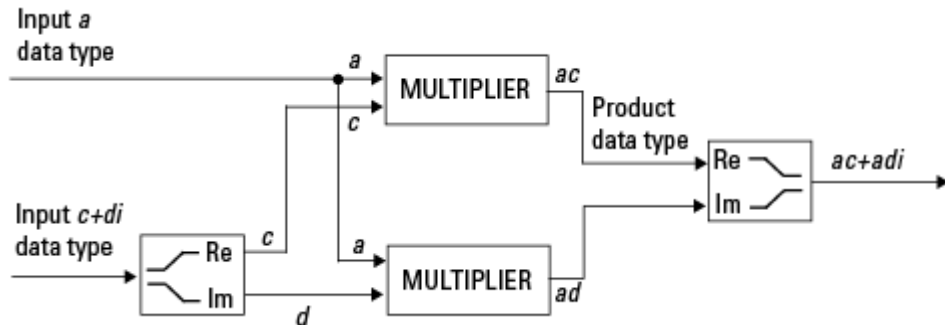
Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication using Fixed-Point Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication.

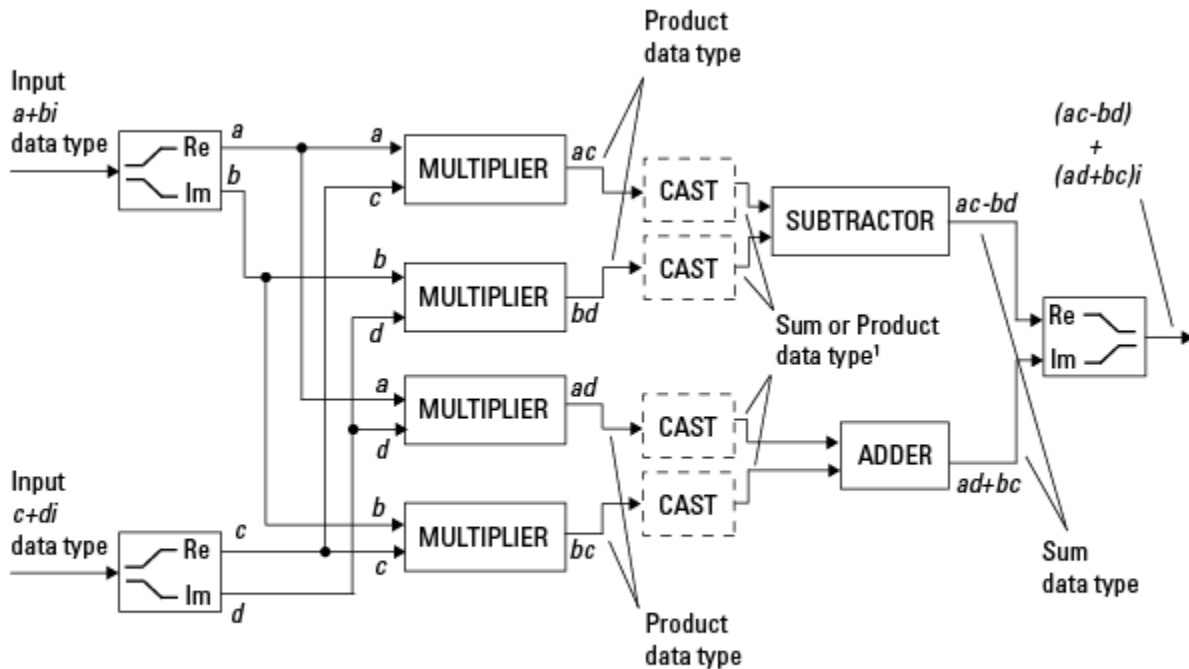
Real-Real Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of two real numbers. The software returns the output of this operation in the product data type, which is governed by the fimath object ProductMode property.



Real-Complex Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of a real and a complex fixed-point number. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product data type, which is governed by the fimath object ProductMode property:



Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers. Note that the software returns the output of this operation in the sum data type, which is governed by the fimath object SumMode property. The intermediate product data type is determined by the fimath object ProductMode property.



¹ Sum data type if CastBeforeSum is true,
Product data type if CastBeforeSum is false

When the fimath object CastBeforeSum property is true, the casts to the sum data type are present after the multipliers in the preceding diagram. In C code, this is equivalent to

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator. When the CastBeforeSum property is false, the casts are not present, and the data remains in the product data type before the subtraction and addition operations.

Multiplication with fimath

In the following examples, let

```
F = fimath('ProductMode','FullPrecision',...
'SumMode','FullPrecision')
T1 = numerictype('WordLength',24,'FractionLength',20)
T2 = numerictype('WordLength',16,'FractionLength',10)
```

Real*Real. Notice that the word length and fraction length of the result z are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the `fimath` `SumMode` and `ProductMode` properties are set to `FullPrecision`:

```
P = fipref;
P.FimathDisplay = 'none';
x = fi(5, T1, F)
```

$x =$

5

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 24
FractionLength: 20
```

```
y = fi(10, T2, F)
```

$y =$

10

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 10
```

```
z = x*y
```

z =

50

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 40
        FractionLength: 30

```

Real*Complex. Notice that the word length and fraction length of the result z are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the fimath SumMode and ProductMode properties are set to FullPrecision:

x = fi(5,T1,F)

x =

5

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 24
        FractionLength: 20

```

y = fi(10+2i,T2,F)

y =

10.0000 + 2.0000i

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 10

```

$z = x \cdot y$

$z =$

$50.0000 + 10.0000i$

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 40
FractionLength: 30

Complex*Complex. Complex-complex multiplication involves an addition as well as multiplication, so the word length of the full-precision result has one more bit than the sum of the word lengths of the multiplicands:

$x = \text{fi}(5+6i, T1, F)$

$x =$

$5.0000 + 6.0000i$

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 24
FractionLength: 20

$y = \text{fi}(10+2i, T2, F)$

$y =$

$10.0000 + 2.0000i$

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 10

```
z = x*y
```

```
z =
```

```
38.0000 +70.0000i
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 41  
FractionLength: 30
```

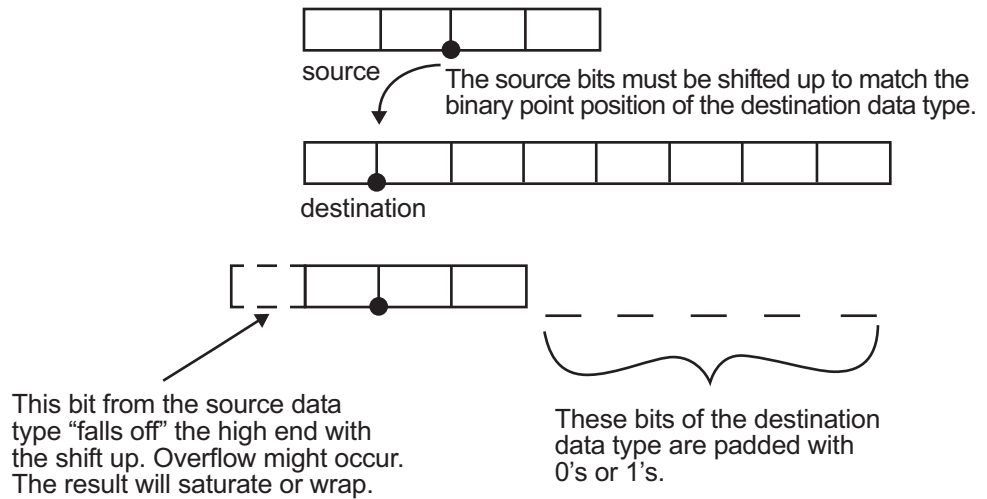
Casts

The `fimath` object allows you to specify the data type and scaling of intermediate sums and products with the `SumMode` and `ProductMode` properties. It is important to keep in mind the ramifications of each cast when you set the `SumMode` and `ProductMode` properties. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples demonstrate cases where overflow and rounding can occur.

Note For more examples of casting, see “Cast fi Objects” on page 2-12.

Casting from a Shorter Data Type to a Longer Data Type

Consider the cast of a nonzero number, represented by a 4-bit data type with two fractional bits, to an 8-bit data type with seven fractional bits:



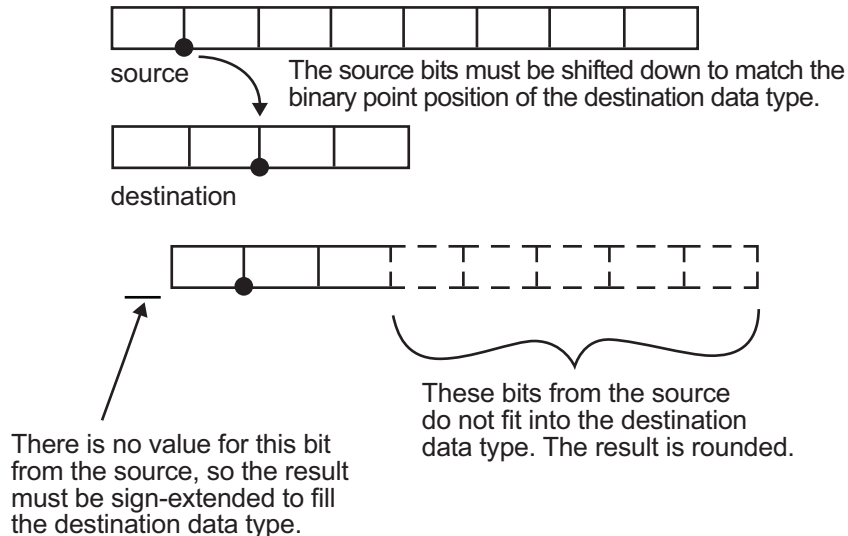
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.
- If wrapping occurs, the empty bits are padded with 0's.
- If saturation occurs,
 - The empty bits of a positive number are padded with 1's.
 - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow can still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

Casting from a Longer Data Type to a Shorter Data Type

Consider the cast of a nonzero number, represented by an 8-bit data type with seven fractional bits, to a 4-bit data type with two fractional bits:



As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so sign extension is used to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision can be lost as the result is rounded.

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow can occur.

The worst case occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.

fi Objects and C Integer Data Types

In this section...
“Integer Data Types” on page 1-22
“Unary Conversions” on page 1-24
“Binary Conversions” on page 1-25
“Overflow Handling” on page 1-28

Note The sections in this topic compare the `fi` object with fixed-point data types and operations in C. In these sections, the information on ANSI C is adapted from Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual*, 3rd ed., Prentice Hall, 1991.

Integer Data Types

This section compares the numerical range of `fi` integer data types to the minimum numerical range of C integer data types, assuming a “Two’s Complement” on page 1-11 representation.

C Integer Data Types

Many C compilers support a two’s complement representation of signed integer data types. The following table shows the minimum ranges of C integer data types using a two’s complement representation. The integer ranges can be larger than or equal to those shown, but cannot be smaller. The range of a `long` must be larger than or equal to the range of an `int`, which must be larger than or equal to the range of a `short`.

In the two’s complement representation, a signed integer with n bits has a range from -2^{n-1} to $2^{n-1} - 1$, inclusive. An unsigned integer with n bits has a range from 0 to $2^n - 1$, inclusive. The negative side of the range has one more value than the positive side, and zero is represented uniquely.

Integer Type	Minimum	Maximum
signed char	-128	127
unsigned char	0	255
short int	-32,768	32,767
unsigned short	0	65,535
int	-32,768	32,767
unsigned int	0	65,535
long int	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295

fi Integer Data Types

The following table lists the numerical ranges of the integer data types of the `fi` object, in particular those equivalent to the C integer data types. The ranges are large enough to accommodate the two's complement representation, which is the only signed binary encoding technique supported by Fixed-Point Toolbox software.

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
<code>fi(x,1,n,0)</code>	Yes	n (2 to 65,535)	0	-2^{n-1}	$2^{n-1} - 1$	N/A
<code>fi(x,0,n,0)</code>	No	n (2 to 65,535)	0	0	$2^n - 1$	N/A
<code>fi(x,1,8,0)</code>	Yes	8	0	-128	127	signed char
<code>fi(x,0,8,0)</code>	No	8	0	0	255	unsigned char
<code>fi(x,1,16,0)</code>	Yes	16	0	-32,768	32,767	short int
<code>fi(x,0,16,0)</code>	No	16	0	0	65,535	unsigned short

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
fi(x,1,32,0)	Yes	32	0	−2,147,483,648	2,147,483,647	long int
fi(x,0,32,0)	No	32	0	0	4,294,967,295	unsigned long

Unary Conversions

Unary conversions dictate whether and how a single operand is converted before an operation is performed. This section discusses unary conversions in ANSI C and of fi objects.

ANSI C Usual Unary Conversions

Unary conversions in ANSI C are automatically applied to the operands of the unary !, −, ~, and * operators, and of the binary << and >> operators, according to the following table:

Original Operand Type	ANSI C Conversion
char or short	int
unsigned char or unsigned short	int or unsigned int ¹
float	float
Array of T	Pointer to T
Function returning T	Pointer to function returning T

¹If type int cannot represent all the values of the original data type without overflow, the converted type is unsigned int.

fi Usual Unary Conversions

The following table shows the fi unary conversions:

C Operator	fi Equivalent	fi Conversion
!x	<code>~x = not(x)</code>	Result is logical.
~x	<code>bitcmp(x)</code>	Result is same numeric type as operand.
*x	No equivalent	N/A
x<<n	<code>bitshift(x,n)</code> positive n	Result is same numeric type as operand. Round mode is always <code>floor</code> . Overflow mode is obeyed. 0-valued bits are shifted in on the right.
x>>n	<code>bitshift(x,-n)</code>	Result is same numeric type as operand. Round mode is always <code>floor</code> . Overflow mode is obeyed. 0-valued bits are shifted in on the left if the operand is unsigned or signed and positive. 1-valued bits are shifted in on the left if the operand is signed and negative.
+x	+x	Result is same numeric type as operand.
-x	-x	Result is same numeric type as operand. Overflow mode is obeyed. For example, overflow might occur when you negate an unsigned fi or the most negative value of a signed fi.

Binary Conversions

This section describes the conversions that occur when the operands of a binary operator are different data types.

ANSI C Usual Binary Conversions

In ANSI C, operands of a binary operator must be of the same type. If they are different, one is converted to the type of the other according to the first applicable conversion in the following table:

Type of One Operand	Type of Other Operand	ANSI C Conversion
long double	Any	long double
double	Any	double
float	Any	float
unsigned long	Any	unsigned long
long	unsigned	long or unsigned long ¹
long	int	long
unsigned	int or unsigned	unsigned
int	int	int

¹Type long is only used if it can represent all values of type unsigned.

fi Usual Binary Conversions

When one of the operands of a binary operator (+, −, *, .*) is a `fi` object and the other is a MATLAB built-in numeric type, then the non-`fi` operand is converted to a `fi` object before the operation is performed, according to the following table:

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	double or single	<ul style="list-style-type: none"> Signed = same as the original <code>fi</code> operand WordLength = same as the original <code>fi</code> operand FractionLength = set to best precision possible
fi	int8	<ul style="list-style-type: none"> Signed = 1 WordLength = 8 FractionLength = 0

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	uint8	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 8 • FractionLength = 0
fi	int16	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 16 • FractionLength = 0
fi	uint16	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 16 • FractionLength = 0
fi	int32	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 32 • FractionLength = 0
fi	uint32	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 32 • FractionLength = 0
fi	int64	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 64 • FractionLength = 0
fi	uint64	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 64 • FractionLength = 0

Overflow Handling

The following sections compare how ANSI C and Fixed-Point Toolbox software handle overflows.

ANSI C Overflow Handling

In ANSI C, the result of signed integer operations is whatever value is produced by the machine instruction used to implement the operation. Therefore, ANSI C has no rules for handling signed integer overflow.

The results of unsigned integer overflows wrap in ANSI C.

fi Overflow Handling

Addition and multiplication with `fi` objects yield results that can be exactly represented by a `fi` object, up to word lengths of 65,535 bits or the available memory on your machine. This is not true of division, however, because many ratios result in infinite binary expressions. You can perform division with `fi` objects using the `divide` function, which requires you to explicitly specify the numeric type of the result.

The conditions under which a `fi` object overflows and the results then produced are determined by the associated `fimath` object. You can specify certain overflow characteristics separately for sums (including differences) and products. Refer to the following table:

fimath Object Properties Related to Overflow Handling	Property Value	Description
OverflowMode	'saturate'	Overflows are saturated to the maximum or minimum value in the range.
	'wrap'	Overflows wrap using modulo arithmetic if unsigned, two's complement wrap if signed.

fimath Object Properties Related to Overflow Handling	Property Value	Description
ProductMode	'FullPrecision'	<p>Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than MaxProductWordLength.</p> <p>The rules for computing the resulting product word and fraction lengths are given in “ProductMode” in the Property Reference.</p>
	'KeepLSB'	<p>The least significant bits of the product are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations.</p> <p>The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If ProductWordLength is less than is necessary for the full-precision product, then overflow occurs.</p> <p>The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepMSB'	<p>The most significant bits of the product are kept. Overflow is prevented, but precision may be lost.</p> <p>The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the most significant bits. If ProductWordLength is less than is necessary for the full-precision product, then rounding occurs.</p> <p>The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.</p>
	'SpecifyPrecision'	You can specify both the word length and the fraction length of the resulting product.
ProductWordLength	Positive integer	The word length of product results when ProductMode is 'KeepLSB', 'KeepMSB', or 'SpecifyPrecision'.
MaxProductWordLength	Positive integer	The maximum product word length allowed when ProductMode is 'FullPrecision'. The default is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.
ProductFractionLength	Integer	The fraction length of product results when ProductMode is 'Specify Precision'.

fimath Object Properties Related to Overflow Handling	Property Value	Description
SumMode	'FullPrecision'	<p>Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than MaxSumWordLength.</p> <p>The rules for computing the resulting sum word and fraction lengths are given in “SumMode” in the Property Reference.</p>
	'KeepLSB'	<p>The least significant bits of the sum are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations.</p> <p>The resulting word length is determined by the SumWordLength property. If SumWordLength is greater than is necessary for the full-precision sum, then the result is stored in the least significant bits. If SumWordLength is less than is necessary for the full-precision sum, then overflow occurs.</p> <p>The rule for computing the resulting sum fraction length is given in “SumMode” in the Property Reference.</p>
	'KeepMSB'	<p>The most significant bits of the sum are kept. Overflow is prevented, but precision may be lost.</p> <p>The resulting word length is determined by the SumWordLength property. If SumWordLength is greater than is necessary for the full-precision sum, then the result is stored in the most significant bits. If SumWordLength is less than is necessary for the full-precision sum, then rounding occurs.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
		The rule for computing the resulting sum fraction length is given in “SumMode” in the Property Reference.
	'SpecifyPrecision'	You can specify both the word length and the fraction length of the resulting sum.
SumWordLength	Positive integer	The word length of sum results when SumMode is 'KeepLSB', 'KeepMSB', or 'SpecifyPrecision'.
MaxSumWordLength	Positive integer	The maximum sum word length allowed when SumMode is 'FullPrecision'. The default is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.
SumFractionLength	Integer	The fraction length of sum results when SumMode is 'SpecifyPrecision'.

Working with fi Objects

- “Ways to Construct fi Objects” on page 2-2
- “Cast fi Objects” on page 2-12
- “fi Object Properties” on page 2-17
- “fi Object Functions” on page 2-23

Ways to Construct fi Objects

In this section...
“Types of fi Constructors” on page 2-2
“Examples of Constructing fi Objects” on page 2-3

Types of fi Constructors

You can create `fi` objects using Fixed-Point Toolbox software in any of the following ways:

- You can use the `fi` constructor function to create a new `fi` object.
- You can use the `sfi` constructor function to create a new signed `fi` object.
- You can use the `ufi` constructor function to create a new unsigned `fi` object.
- You can use any of the `fi` constructor functions to copy an existing `fi` object.

To get started, type

```
a = fi(0)
```

to create a `fi` object with the default data type and a value of 0.

```
a =
```

```
0
```

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
           WordLength: 16
    FractionLength: 15

```

This constructor syntax creates a signed `fi` object with a value of 0, word length of 16 bits, and fraction length of 15 bits. Because you did not specify any `fimath` object properties in the `fi` constructor, the resulting `fi` object `a` has no local `fimath`.

To see all of the `fi`, `sfi`, and `ufi` constructor syntaxes, refer to the respective reference pages.

Note For information on the display format of `fi` objects, refer to “View Fixed-Point Data”.

Examples of Constructing fi Objects

The following examples show you several different ways to construct `fi` objects. For other, more basic examples of constructing `fi` objects, see the Examples section of the following constructor function reference pages:

- `fi`
- `sfi`
- `ufi`

Note The `fi` constructor creates the `fi` object using a `RoundingMethod` of `Nearest` and an `OverflowAction` of `Saturate`. If you construct a `fi` from floating-point values, the default `RoundingMethod` and `OverflowAction` property settings are not used.

Constructing a fi Object with Property Name/Property Value Pairs

You can use property name/property value pairs to set `fi` and `fimath` object properties when you create the `fi` object:

```
a = fi(pi, 'RoundingMethod','Floor', 'OverflowAction','Wrap')
```

```
a =
```

```
3.1415
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16
```

```
FractionLength: 13

RoundingMethod: floor
OverflowAction: wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

You do not have to specify every `fimath` object property in the `fi` constructor. The `fi` object uses default values for all unspecified `fimath` object properties.

- If you specify at least one `fimath` object property in the `fi` constructor, the `fi` object will have a local `fimath` object. The `fi` object uses default values for the remaining unspecified `fimath` object properties.
- If you do not specify any `fimath` object properties in the `fi` object constructor, the `fi` object uses default `fimath` values.

Constructing a fi Object Using a numerictype Object

You can use a `numerictype` object to define a `fi` object:

```
T = numerictype
```

```
T =
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 15
```

```
a = fi(pi, T)
```

```
a =
```

```
1.0000
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
```

FractionLength: 15

You can also use a `fimath` object with a `numericType` object to define a `fi` object:

```
F = fimath('RoundingMethod', 'Nearest',...  
          'OverflowAction', 'Saturate',...  
          'ProductMode', 'FullPrecision',...  
          'SumMode', 'FullPrecision')
```

F =

```
          RoundingMethod: Nearest  
          OverflowAction: Saturate  
          ProductMode: FullPrecision  
          SumMode: FullPrecision
```

```
a = fi(pi, T, F)
```

a =

1.0000

```
          DataTypeMode: Fixed-point: binary point scaling  
          Signedness: Signed  
          WordLength: 16  
          FractionLength: 15
```

```
          RoundingMethod: Nearest  
          OverflowAction: Saturate  
          ProductMode: FullPrecision  
          SumMode: FullPrecision
```

Note The syntax `a = fi(pi,T,F)` is equivalent to `a = fi(pi,F,T)`. You can use both statements to define a `fi` object using a `fimath` object and a `numericType` object.

Constructing a fi Object Using a fimath Object

You can create a `fi` object using a specific `fimath` object. When you do so, a local `fimath` object is assigned to the `fi` object you create. If you do not specify any `numericType` object properties, the word length of the `fi` object defaults to 16 bits. The fraction length is determined by best precision scaling:

```
F = fimath('RoundingMethod', 'Nearest',...
'OverflowAction', 'Saturate',...
'ProductMode', 'FullPrecision',...
'SumMode', 'FullPrecision',...)
```

```
F =
```

```
    RoundingMethod: Nearest
  OverflowAction: Saturate
    ProductMode: FullPrecision
        SumMode: FullPrecision
```

```
F.OverflowAction = 'Wrap'
```

```
F =
```

```
    RoundingMethod: Nearest
  OverflowAction: Wrap
    ProductMode: FullPrecision
        SumMode: FullPrecision
```

```
a = fi(pi, F)
```

```
a =
```

```
    3.1416
```

```
    DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
    FractionLength: 13
```



```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

You can also create fi objects using a fimath object while specifying various numericity properties at creation time:

```
b = fi(pi, 0, F)
```

```
b =
```

```
3.1416
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 16
FractionLength: 14
```

```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

```
c = fi(pi, 0, 8, F)
```

```
c =
```

```
3.1406
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 8
FractionLength: 6
```

```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

```
d = fi(pi, 0, 8, 6, F)

d =

    3.1406

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Unsigned
    WordLength: 8
    FractionLength: 6

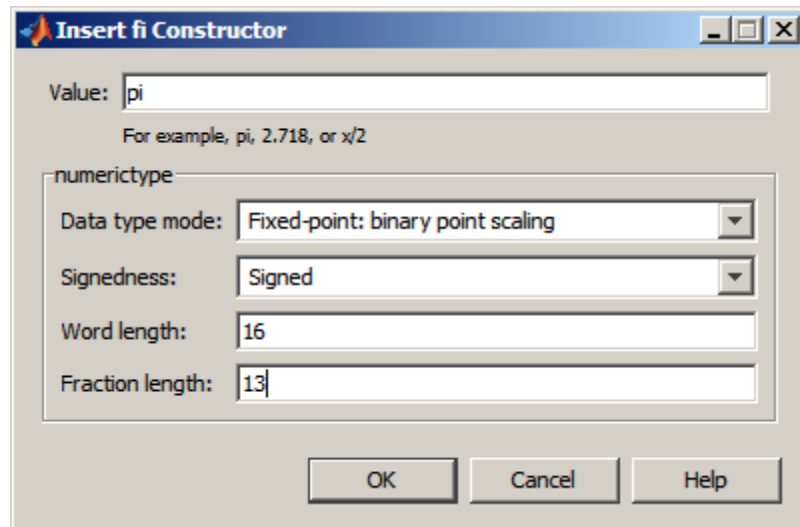
    RoundingMethod: Nearest
    OverflowAction: wrap
    ProductMode: FullPrecision
    SumMode: FullPrecision
```

Building fi Object Constructors in a GUI

When you are working with files in MATLAB®, you can build your `fi` object constructors using the **Insert fi Constructor** dialog box. After specifying the value and properties of the `fi` object in the dialog box, you can insert the prepopulated `fi` object constructor string at a specific location in your file.

For example, to create a signed `fi` object with a value of `pi`, a word length of 16 bits and a fraction length of 13 bits, perform the following steps:

- 1 Open the **Insert fi Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert fi Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the `fi` object:
 - **Value** = `pi`
 - **Data type mode** = Fixed-point: binary point scaling
 - **Signedness** = Signed
 - **Word length** = 16
 - **Fraction length** = 13



- 3** To insert the `fi` object constructor string in your file, place your cursor at the desired location in the file, and click **OK** on the **Insert fi Constructor** dialog box. Clicking **OK** closes the **Insert fi Constructor** dialog box and automatically populates the `fi` object constructor string in your file:

```
7      fi(pi, 1, 16, 13)
```

Determining Property Precedence

The value of a property is taken from the last time it is set. For example, create a `numerictype` object with a value of `true` for the `Signed` property and a fraction length of 14:

```
T = numerictype('Signed', true, 'FractionLength', 14)
```

```
T =
```

```

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 14

```

Now, create the following `fi` object in which you specify the `numerictype` property *after* the `Signed` property, so that the resulting `fi` object is signed:

```
a = fi(pi, 'Signed', false, 'numerictype', T)

a =

    1.9999

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 14
```

Contrast the `fi` object in this code sample with the `fi` object in the following code sample. The `numerictype` property in the following code sample is specified *before* the `Signed` property, so the resulting `fi` object is unsigned:

```
b = fi(pi, 'numerictype', T, 'Signed', false)

b =

    3.1416

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Unsigned
    WordLength: 16
    FractionLength: 14
```

Copying a fi Object

To copy a `fi` object, simply use assignment, as in the following example:

```
a = fi(pi)

a =

    3.1416
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13
```

```
b = a
```

```
b =
```

```
3.1416
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13
```

Cast fi Objects

In this section...
“Overwriting by Assignment” on page 2-12
“Ways to Cast with MATLAB Software” on page 2-12

Overwriting by Assignment

Because MATLAB software does not have type declarations, an assignment like `A = B` replaces the type and content of `A` with the type and content of `B`. If `A` does not exist at the time of the assignment, MATLAB creates the variable `A` and assigns it the same type and value as `B`. Such assignment happens with all types in MATLAB—objects and built-in types alike—including `fi`, `double`, `single`, `int8`, `uint8`, `int16`, etc.

For example, the following code overwrites the value and `int8` type of `A` with the value and `int16` type of `B`:

```
A = int8(0);  
B = int16(32767);  
A = B
```

```
A =
```

```
    32767
```

```
class(A)
```

```
ans =
```

```
int16
```

Ways to Cast with MATLAB Software

You may find it useful to cast data into another type—for example, when you are casting data from an accumulator to memory. There are several ways to cast data in MATLAB. The following sections provide examples of three different methods:

- Casting by Subscripted Assignment
- Casting by Conversion Function
- Casting with the Fixed-Point Toolbox `reinterprecast` Function

Casting by Subscripted Assignment

The following subscripted assignment statement retains the type of `A` and saturates the value of `B` to an `int8`:

```
A = int8(0);
B = int16(32767);
A(:) = B
```

```
A =
    127

class(A)
```

```
ans =

int8
```

The same is true for `fi` objects:

```
fipref('NumericTypeDisplay', 'short');
A = fi(0, true, 8, 0);
B = fi(32767, true, 16, 0);
A(:) = B
```

```
A =

    127
      s8,0
```

Note For more information on subscripted assignments, see the `subsasgn` function.

Casting by Conversion Function

You can convert from one data type to another by using a conversion function. In this example, A does not have to be predefined because it is overwritten.

```
B = int16(32767);  
A = int8(B)
```

```
A =  
  
    127  
  
class(A)
```

```
ans =  
  
int8
```

The same is true for `fi` objects:

```
B = fi(32767, true, 16, 0)  
A = fi(B, 1, 8, 0)
```

```
B =  
  
    32767  
    s16,0
```

```
A =  
  
    127  
    s8,0
```

Using a `numerictype` Object in the `fi` Conversion Function. Often a specific `numerictype` is used in many places, and it is convenient to predefine `numerictype` objects for use in the conversion functions. Predefining these objects is a good practice because it also puts the data type specification in one place.

```
T8 = numerictype(1,8,0)  
  
T8 =
```



```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 8
        FractionLength: 0

T16 = numerictype(1,16,0)

T16 =

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 0

B = fi(32767,T16)

B =

      32767
    s16,0

A = fi(B, T8)

A =

      127
    s8,0

```

Casting with the reinterpretcast Function

You can convert fixed-point and built-in data types without changing the underlying data. The Fixed-Point Toolbox `reinterpretcast` function performs this type of conversion.

In the following example, `B` is an unsigned `fi` object with a word length of 8 bits and a fraction length of 5 bits. The `reinterpretcast` function converts `B` into a signed `fi` object `A` with a word length of 8 bits and a fraction length of 1

bit. The real-world values of A and B differ, but their binary representations are the same.

```
B = fi([pi/4 1 pi/2 4], false, 8, 5)
T = numerictype(true, 8, 1);
A = reinterpretcast(B, T)
```

B =

```
    0.7813    1.0000    1.5625    4.0000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Unsigned
      WordLength: 8
      FractionLength: 5
```

A =

```
   12.5000   16.0000   25.0000  -64.0000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 1
```

To verify that the underlying data has not changed, compare the binary representations of A and B:

```
binary_B = bin(B)
binary_A = bin(A)
```

binary_A =

```
00011001  00100000  00110010  10000000
```

binary_B =

```
00011001  00100000  00110010  10000000
```

fi Object Properties

In this section...
“Data Properties” on page 2-17
“fimath Properties” on page 2-17
“numericity Properties” on page 2-19
“Setting fi Object Properties” on page 2-20

Data Properties

The data properties of a `fi` object are always writable.

- `bin` — Stored integer value of a `fi` object in binary
- `data` — Numerical real-world value of a `fi` object
- `dec` — Stored integer value of a `fi` object in decimal
- `double` — Real-world value of a `fi` object, stored as a MATLAB `double` data type
- `hex` — Stored integer value of a `fi` object in hexadecimal
- `oct` — Stored integer value of a `fi` object in octal

To learn more about these properties, see “fi Object Properties” in the Fixed-Point Toolbox Reference.

fimath Properties

In general, the `fimath` properties associated with `fi` objects depend on how you create the `fi` object:

- When you specify one or more `fimath` object properties in the `fi` constructor, the resulting `fi` object has a local `fimath` object.
- When you do not specify any `fimath` object properties in the `fi` constructor, the resulting `fi` object has no local `fimath`.

To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

The `fimath` properties associated with `fi` objects determine how fixed-point arithmetic is performed. These `fimath` properties can come from a local `fimath` object or from default `fimath` property values. To learn more about `fimath` objects in fixed-point arithmetic, see “`fimath` Rules for Fixed-Point Arithmetic” on page 4-11.

The following `fimath` properties are, by transitivity, also properties of the `fi` object. You can set these properties for individual `fi` objects. The following `fimath` properties are always writable.

- `CastBeforeSum` — Whether both operands are cast to the sum data type before addition

Note This property is hidden when the `SumMode` is set to `FullPrecision`.

- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowAction` — Action to take on overflow
- `ProductBias` — Bias of the product data type
- `ProductFixedExponent` — Fixed exponent of the product data type
- `ProductFractionLength` — Fraction length, in bits, of the product data type
- `ProductMode` — Defines how the product data type is determined
- `ProductSlope` — Slope of the product data type
- `ProductSlopeAdjustmentFactor` — Slope adjustment factor of the product data type
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundingMethod` — Rounding method

- **SumBias** — Bias of the sum data type
- **SumFixedExponent** — Fixed exponent of the sum data type
- **SumFractionLength** — Fraction length, in bits, of the sum data type
- **SumMode** — Defines how the sum data type is determined
- **SumSlope** — Slope of the sum data type
- **SumSlopeAdjustmentFactor** — Slope adjustment factor of the sum data type
- **SumWordLength** — The word length, in bits, of the sum data type

To learn more about these properties, see the “fmath Object Properties” in the Fixed-Point Toolbox Reference.

numerictype Properties

When you create a **fi** object, a **numerictype** object is also automatically created as a property of the **fi** object:

numerictype — Object containing all the data type information of a **fi** object, Simulink signal or model parameter

The following **numerictype** properties are, by transitivity, also properties of a **fi** object. The following properties of the **numerictype** object become read only after you create the **fi** object. However, you can create a copy of a **fi** object with new values specified for the **numerictype** properties:

- **Bias** — Bias of a **fi** object
- **DataType** — Data type category associated with a **fi** object
- **DataTypeMode** — Data type and scaling mode of a **fi** object
- **FixedExponent** — Fixed-point exponent associated with a **fi** object
- **FractionLength** — Fraction length of the stored integer value of a **fi** object in bits
- **Scaling** — Fixed-point scaling mode of a **fi** object
- **Signed** — Whether a **fi** object is signed or unsigned

- **Signedness** — Whether a `fi` object is signed or unsigned

Note `numericType` objects can have a `Signedness` of `Auto`, but all `fi` objects must be `Signed` or `Unsigned`. If a `numericType` object with `Auto Signedness` is used to create a `fi` object, the `Signedness` property of the `fi` object automatically defaults to `Signed`.

- **Slope** — Slope associated with a `fi` object
- **SlopeAdjustmentFactor** — Slope adjustment associated with a `fi` object
- **WordLength** — Word length of the stored integer value of a `fi` object in bits

For further details on these properties, see the “`fi` Object Properties” on page 2-17.

There are two ways to specify properties for `fi` objects in Fixed-Point Toolbox software. Refer to the following sections:

- “Setting Fixed-Point Properties at Object Creation” on page 2-20
- “Using Direct Property Referencing with `fi`” on page 2-21

Setting `fi` Object Properties

You can set `fi` object properties in two ways:

- Setting the properties when you create the object
- Using direct property referencing

Setting Fixed-Point Properties at Object Creation

You can set properties of `fi` objects at the time of object creation by including properties after the arguments of the `fi` constructor function. For example, to set the overflow action to `Wrap` and the rounding method to `Convergent`,

```
a = fi(pi, 'OverflowAction', 'Wrap', ...  
      'RoundingMethod', 'Convergent')
```

```
a =
```

3.1416

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 13

RoundingMethod: Convergent
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

Using Direct Property Referencing with fi

You can reference directly into a property for setting or retrieving `fi` object property values using MATLAB structure-like referencing. You do so by using a period to index into a property by name.

For example, to get the `WordLength` of `a`,

```
a.WordLength
```

```
ans =
```

```
16
```

To set the `OverflowMode` of `a`,

```
a.OverflowAction = 'Wrap'
```

```
a =
```

```
3.1416
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
```

```
FractionLength: 13

RoundingMethod: Convergent
OverflowAction: wrap
    ProductMode: FullPrecision
        SumMode: FullPrecision
```

If you have a `fi` object `b` with a local `fimath` object, you can remove the local `fimath` object and force `b` to use default `fimath` values:

```
b = fi(pi, 1, 'RoundingMethod', 'Floor')

b =
    3.1415

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 13

    RoundingMethod: Floor
    OverflowAction: Saturate
    ProductMode: FullPrecision
    SumMode: FullPrecision

b.fimath = []

b =
    3.1415

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 13

isfimathlocal(b)

ans =
    0
```


fi Object Functions

In addition to functions that operate on fi objects, you can use the following functions to access data in a fi object using dot notation.

- `bin`
- `data`
- `dec`
- `double`
- `hex`
- `storedInteger`
- `storedIntegerToDouble`
- `oct`

For example,

```
a = fi(pi);  
n = storedInteger(a)
```

```
n =
```

```
25736
```

```
h = hex(a)
```

```
h =
```

```
6488
```

```
a.hex
```

```
ans =
```

```
6488
```


Fixed-Point Topics

- “Set Up Fixed-Point Objects” on page 3-2
- “View Fixed-Point Number Circles” on page 3-18
- “Perform Binary-Point Scaling” on page 3-31
- “Develop Fixed-Point Algorithms” on page 3-37
- “Calculate Fixed-Point Sine and Cosine” on page 3-48
- “Calculate Fixed-Point Arctangent” on page 3-70
- “Compute Sine and Cosine Using CORDIC Rotation Kernel” on page 3-96
- “Perform QR Factorization Using CORDIC” on page 3-102
- “Compute Square Root Using CORDIC Hyperbolic Kernel” on page 3-142
- “Convert Cartesian to Polar Using CORDIC Vectoring Kernel” on page 3-148
- “Set Data Types Using Min/Max Instrumentation” on page 3-154
- “Convert Fast Fourier Transform (FFT) to Fixed Point” on page 3-168
- “Detect Limit Cycles in Fixed-Point State-Space Systems” on page 3-179
- “Compute Quantization Error” on page 3-191
- “Normalize Data for Lookup Tables” on page 3-199
- “Implement Fixed-Point Log2 Using Lookup Table” on page 3-205
- “Implement Fixed-Point Square Root Using Lookup Table” on page 3-210
- “Set Fixed-Point Math Attributes” on page 3-215

Set Up Fixed-Point Objects

Create Fixed-Point Data

This example shows the basics of how to use the fixed-point numeric object `fi`.

Notation

The fixed-point numeric object is called `fi` because J.H. Wilkinson used `fi` to denote fixed-point computations in his classic texts *Rounding Errors in Algebraic Processes* (1963), and *The Algebraic Eigenvalue Problem* (1965).

Setup

This example may use display settings or preferences that are different from what you are currently using. To ensure that your current display settings and preferences are not changed by running this example, the example automatically saves and restores them. The following code captures the current states for any display settings or properties that the example changes.

```
format loose
format long g
% Capture the current state of and reset the fi display and logging
% preferences to the factory settings.
fiprefAtStartOfThisExample = get(fipref);
reset(fipref);
```

Default Fixed-Point Attributes

To assign a fixed-point data type to a number or variable with the default fixed-point parameters, use the `fi` constructor. The resulting fixed-point value is called a `fi` object.

For example, the following creates `fi` objects `a` and `b` with attributes shown in the display, all of which we can specify when the variables are constructed. Note that when the `FractionLength` property is not specified, it is set automatically to "best precision" for the given word length, keeping the most-significant bits of the value. When the `WordLength` property is not specified it defaults to 16 bits.

```
a = fi(pi)
```

```
a =
```

```
3.1416015625
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 13
```

```
b = fi(0.1)
```

```
b =
```

```
0.0999984741210938
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 18
```

Specifying Signed and WordLength Properties

The second and third numeric arguments specify **Signed** (true or 1 = signed, false or 0 = unsigned), and **WordLength** in bits, respectively.

```
% Signed 8-bit
a = fi(pi, 1, 8)
```

```
a =
```

```
3.15625
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 8
```

```
FractionLength: 5
```

The `sfi` constructor may also be used to construct a signed `fi` object

```
a1 = sfi(pi,8)
```

```
a1 =
```

```
3.15625
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 8  
FractionLength: 5
```

```
% Unsigned 20-bit  
b = fi(exp(1), 0, 20)
```

```
b =
```

```
2.71828079223633
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned  
WordLength: 20  
FractionLength: 18
```

The `ufi` constructor may be used to construct an unsigned `fi` object

```
b1 = ufi(exp(1), 20)
```

```
b1 =
```

```
2.71828079223633
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned
```

```
WordLength: 20
FractionLength: 18
```

Precision

The data is stored internally with as much precision as is specified. However, it is important to be aware that initializing high precision fixed-point variables with double-precision floating-point variables may not give you the resolution that you might expect at first glance. For example, let's initialize an unsigned 100-bit fixed-point variable with 0.1, and then examine its binary expansion:

```
a = ufi(0.1, 100);
```

 $\text{bin}(a)$

ans =

1100110011001100110011001100110011001100110011010000000000000000000000

Note that the infinite repeating binary expansion of 0.1 gets cut off at the 52nd bit (in fact, the 53rd bit is significant and it is rounded up into the 52nd bit). This is because double-precision floating-point variables (the default MATLAB data type), are stored in 64-bit floating-point format, with 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa plus one "hidden" bit for an effective 53 bits of precision. Even though double-precision floating-point has a very large range, its precision is limited to 53 bits. For more information on floating-point arithmetic, refer to Chapter 1 of Cleve Moler's book, Numerical Computing with MATLAB. The pdf version can be found here: <http://www.mathworks.com/company/aboutus/founders/clevemoler.html>

So, why have more precision than floating-point? Because most fixed-point processors have data stored in a smaller precision, and then compute with larger precisions. For example, let's initialize a 40-bit unsigned `fi` and multiply using full-precision for products.

Note that the full-precision product of 40-bit operands is 80 bits, which is greater precision than standard double-precision floating-point.

```
a = fi(0.1, 0, 40);  
bin(a)
```

```
ans =
```

```
1100110011001100110011001100110011001101
```

```
b = a*a
```

```
b =
```

```
0.01000000000000045
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned  
WordLength: 80  
FractionLength: 86
```

```
bin(b)
```

```
ans =
```

```
101000111101011100001010001111010111000011110101110000101000111101011100001
```

Access to Data

The data can be accessed in a number of ways which map to built-in data types and binary strings. For example,

DOUBLE(A)

```
a = fi(pi);  
double(a)
```



```
ans =
```

```
3.1416015625
```

returns the double-precision floating-point "real-world" value of `a`, quantized to the precision of `a`.

A.DOUBLE = ...

We can also set the real-world value in a double.

```
a.double = exp(1)
```

```
a =
```

```
2.71826171875
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 13
```

sets the real-world value of `a` to `e`, quantized to `a`'s numeric type.

STOREDINTEGER(A)

```
storedInteger(a)
```

```
ans =
```

```
22268
```

returns the "stored integer" in the smallest built-in integer type available, up to 64 bits.

Relationship Between Stored Integer Value and Real-World Value

In `BinaryPoint` scaling, the relationship between the stored integer value and the real-world value is

$$\text{Real-world value} = (\text{Stored integer}) \cdot 2^{-\text{Fraction length}}$$

There is also `SlopeBias` scaling, which has the relationship

$$\text{Real-world value} = (\text{Stored integer}) \cdot \text{Slope} + \text{Bias}$$

where

$$\text{Slope} = (\text{Slope adjustment factor}) \cdot 2^{\text{Fixed exponent}}$$

and

$$\text{Fixed exponent} = -\text{Fraction length}.$$

The math operators of `fi` work with `BinaryPoint` scaling and real-valued `SlopeBias` scaled `fi` objects.

`BIN(A)`, `OCT(A)`, `DEC(A)`, `HEX(A)`

return the stored integer in binary, octal, unsigned decimal, and hexadecimal strings, respectively.

`bin(a)`

`ans =`

0101011011111100

`oct(a)`

`ans =`

053374

```
dec(a)
```

```
ans =
```

```
22268
```

```
hex(a)
```

```
ans =
```

```
56fc
```

A.BIN = ..., A.OCT = ..., A.DEC = ..., A.HEX = ...

set the stored integer from binary, octal, unsigned decimal, and hexadecimal strings, respectively.

```
fi( $\pi$ )
```

```
a.bin = '0110010010001000'
```

```
a =
```

```
3.1416015625
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 13
```

```
fi( $\phi$ )
```

```
a.oct = '031707'
```

```
a =  
  
1.6180419921875  
  
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 13
```

```
fi(e)
```

```
a.dec = '22268'
```

```
a =  
  
2.71826171875  
  
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 13
```

```
fi(0.1)
```

```
a.hex = '0333'
```

```
a =  
  
0.0999755859375  
  
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 13
```

Specifying FractionLength

When the `FractionLength` property is not specified, it is computed to be the best precision for the magnitude of the value and given word length. You may also specify the fraction length directly as the fourth numeric argument in the `fi` constructor or the third numeric argument in the `sfi` or `ufi` constructor. In the following, compare the fraction length of `a`, which was explicitly set to 0, to the fraction length of `b`, which was set to best precision for the magnitude of the value.

```
a = sfi(10,16,0)
```

```
a =
```

```
10
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 0

```

```
b = sfi(10,16)
```

```
b =
```

```
10
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 11

```

Note that the stored integer values of `a` and `b` are different, even though their real-world values are the same. This is because the real-world value of `a` is the stored integer scaled by $2^0 = 1$, while the real-world value of `b` is the stored integer scaled by $2^{-11} = 0.00048828125$.

```
storedInteger(a)
```

```
ans =  
  
    10
```

```
storedInteger(b)
```

```
ans =  
  
    20480
```

Specifying Properties with Parameter/Value Pairs

Thus far, we have been specifying the numeric type properties by passing numeric arguments to the `fi` constructor. We can also specify properties by giving the name of the property as a string followed by the value of the property:

```
a = fi(pi, 'WordLength', 20)
```

```
a =  
  
    3.14159393310547  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 20  
    FractionLength: 17
```

For more information on `fi` properties, type

```
help fi
```

```
or
```

```
doc fi
```

at the MATLAB command line.

Numeric Type Properties

All of the numeric type properties of `fi` are encapsulated in an object named `numericitytype`:

```
T = numericitytype
```

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

The numeric type properties can be modified either when the object is created by passing in parameter/value arguments

```
T = numericitytype('WordLength',40,'FractionLength',37)
```

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 40
      FractionLength: 37
```

or they may be assigned by using the dot notation

```
T.Signed = false
```

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling
```

```
Signedness: Unsigned
WordLength: 40
FractionLength: 37
```

All of the numeric type properties of a `fi` may be set at once by passing in the `numerictype` object. This is handy, for example, when creating more than one `fi` object that share the same numeric type.

```
a = fi(pi,'numerictype',T)
```

```
a =
```

```
3.14159265359194
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 40
FractionLength: 37
```

```
b = fi(exp(1),'numerictype',T)
```

```
b =
```

```
2.71828182845638
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 40
FractionLength: 37
```

The `numerictype` object may also be passed directly to the `fi` constructor

```
a1 = fi(pi,T)
```

```
a1 =
```

```
3.14159265359194
```



```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 40
        FractionLength: 37

```

For more information on `numerictype` properties, type

```
help numerictype
```

or

```
doc numerictype
```

at the MATLAB command line.

Display Preferences

The display preferences for `fi` can be set with the `fipref` object. They can be saved between MATLAB sessions with the `savefipref` command.

Display of Real-World Values

When displaying real-world values, the closest double-precision floating-point value is displayed. As we have seen, double-precision floating-point may not always be able to represent the exact value of high-precision fixed-point number. For example, an 8-bit fractional number can be represented exactly in doubles

```
a = sfi(1,8,7)
```

```
a =
```

```
0.9921875
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 8
        FractionLength: 7

```

01111111

1

[illegible]

so that as much precision as is possible will be displayed.

There are also other display options to make a more shorthand display of the numeric type properties, and options to control the display of the value (as real-world value, binary, octal, decimal integer, or hex).

For more information on display preferences, type

```
help fipref
help savefipref
help format
```

or

```
doc fipref
doc savefipref
doc format
```

at the MATLAB command line.

Cleanup

The following code sets any display settings or preferences that the example changed back to their original states.

```
% Reset the fi display and logging preferences
fipref(fiprefAtStartOfThisExample);
```

View Fixed-Point Number Circles

This example shows how to define unsigned and signed two's complement integer and fixed-point numbers.

Fixed-Point Number Definitions

This example illustrates the definitions of unsigned and signed-two's-complement integer and fixed-point numbers.

Unsigned Integers.

Unsigned integers are represented in the binary number system in the following way. Let

$$b = [b(n) \ b(n-1) \ \dots \ b(2) \ b(1)]$$

be the binary digits of an n-bit unsigned integer, where each $b(i)$ is either one or zero. Then the value of b is

$$u = b(n) \cdot 2^{(n-1)} + b(n-1) \cdot 2^{(n-2)} + \dots + b(2) \cdot 2^{(1)} + b(1) \cdot 2^{(0)}$$

For example, let's define a 3-bit unsigned integer quantizer, and enumerate its range.

```
q = quantizer('ufixed',[3 0]);  
[a,b] = range(q);  
u = (a:eps(q):b)'
```

```
% Now, let's display those values in binary.  
b = num2bin(q,u)
```

```
u =
```

```
0  
1  
2  
3  
4  
5
```

6
7

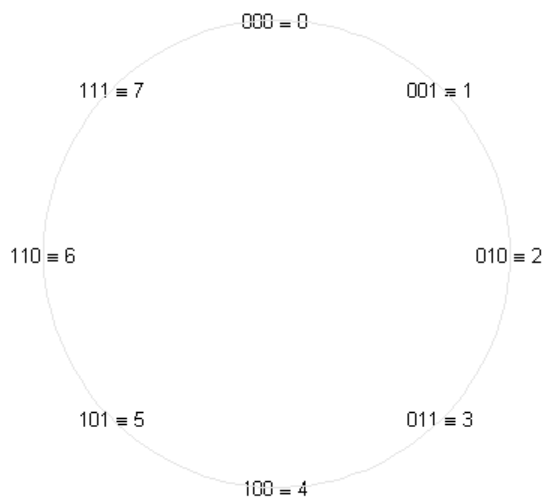
b =

000
001
010
011
100
101
110
111

Unsigned Integer Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

```
fidemo.numbercircle(q);
```



Unsigned Fixed-Point.

Unsigned fixed-point values are unsigned integers that are scaled by a power of two. We call the negative exponent of the power of two the "fractionlength".

If the unsigned integer u is defined as before, and the fractionlength is f , then the value of the unsigned fixed-point number is

$$uf = u \cdot 2^{-f}$$

For example, let's define a 3-bit unsigned fixed-point quantizer with a fractionlength of 1, and enumerate its range.

```
q = quantizer('ufixed',[3 1]);  
[a,b] = range(q);  
uf = (a:eps(q):b)'
```

```
% Now, let's display those values in binary.  
b = num2bin(q,uf)
```

uf =

0
0.5
1
1.5
2
2.5
3
3.5

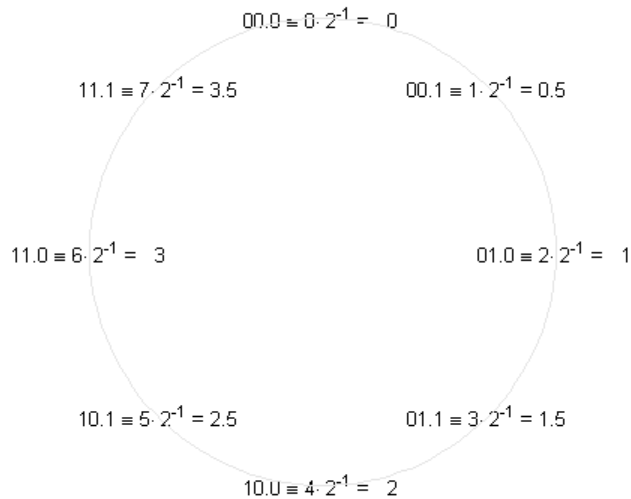
b =

000
001
010
011
100
101
110
111

Unsigned Fixed-Point Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

```
fidemo.numbercircle(q);
```



Unsigned Fractional Fixed-Point.

Unsigned fractional fixed-point numbers are fixed-point numbers whose fractionlength f is equal to the wordlength n , which produces a scaling such that the range of numbers is between 0 and $1-2^{-f}$, inclusive. This is the most common form of fixed-point numbers because it has the nice property that all of the numbers are less than one, and the product of two numbers less than one is a number less than one, and so multiplication does not overflow.

Thus, the definition of unsigned fractional fixed-point is the same as unsigned fixed-point, with the restriction that $f=n$, where n is the wordlength in bits.

$$uf = u \cdot 2^{-f}$$

For example, let's define a 3-bit unsigned fractional fixed-point quantizer, which implies a fractionlength of 3.

```
q = quantizer('ufixed',[3 3]);
[a,b] = range(q);
uf = (a:eps(q):b)'
```



```
% Now, let's display those values in binary.  
b = num2bin(q,uf)
```

```
uf =
```

```
0  
0.125  
0.25  
0.375  
0.5  
0.625  
0.75  
0.875
```

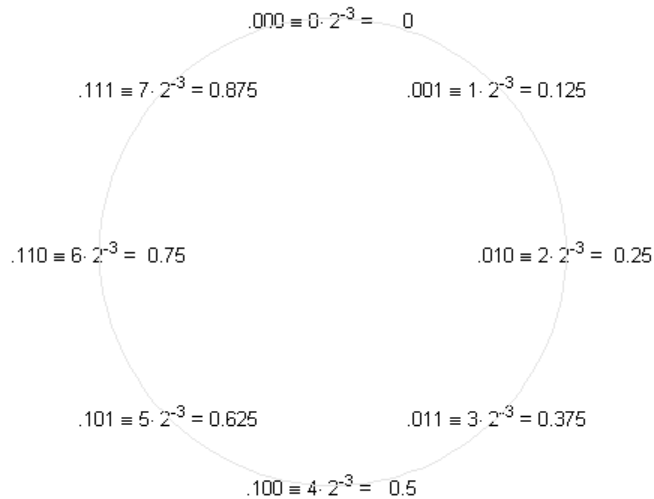
```
b =
```

```
000  
001  
010  
011  
100  
101  
110  
111
```

Unsigned Fractional Fixed-Point Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

```
fidemo.numbercircle(q);
```



Signed Two's-Complement Integers.

Signed integers are represented in two's-complement in the binary number system in the following way. Let

$$b = [b(n) \ b(n-1) \ \dots \ b(2) \ b(1)]$$

be the binary digits of an n -bit signed integer, where each $b(i)$ is either one or zero. Then the value of b is

$$s = -b(n) \cdot 2^{(n-1)} + b(n-1) \cdot 2^{(n-2)} + \dots + b(2) \cdot 2^{(1)} + b(1) \cdot 2^{(0)}$$

Note that the difference between this and the unsigned number is the negative weight on the most-significant-bit (MSB).

For example, let's define a 3-bit signed integer quantizer, and enumerate its range.

```
q = quantizer('fixed',[3 0]);
[a,b] = range(q);
```

```
s = (a:eps(q):b) '

% Now, let's display those values in binary.
b = num2bin(q,s)

% Note that the most-significant-bit of negative numbers is 1, and positive
% numbers is 0.
```

```
s =

    -4
    -3
    -2
    -1
     0
     1
     2
     3
```

```
b =

100
101
110
111
000
001
010
011
```

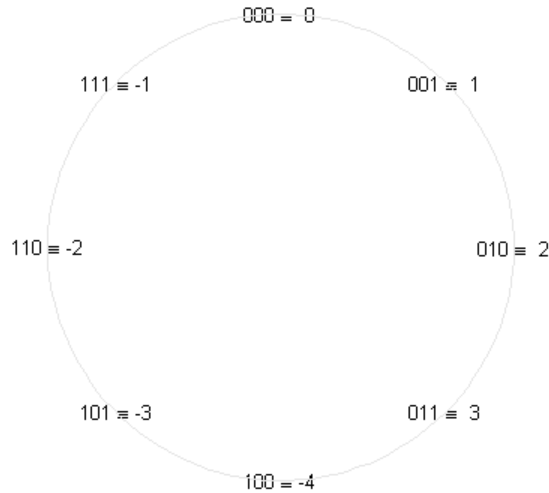
Signed Two's-Complement Integer Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

The reason for this ungainly looking definition of negative numbers is that addition of all numbers, both positive and negative, is carried out as if they

were all positive, and then the $n+1$ carry bit is discarded. The result will be correct if there is no overflow.

```
fidemo.numbercircle(q);
```



Signed Fixed-Point.

Signed fixed-point values are signed integers that are scaled by a power of two. We call the negative exponent of the power of two the "fractionlength".

If the signed integer s is defined as before, and the fractionlength is f , then the value of the signed fixed-point number is

$$sf = s \cdot 2^{-f}$$

For example, let's define a 3-bit signed fixed-point quantizer with a fractionlength of 1, and enumerate its range.

```
q = quantizer('fixed',[3 1]);
[a,b] = range(q);
```

```
sf = (a:eps(q):b)';

% Now, let's display those values in binary.
b = num2bin(q,sf)
```

```
sf =

    -2
   -1.5
    -1
   -0.5
     0
    0.5
     1
    1.5
```

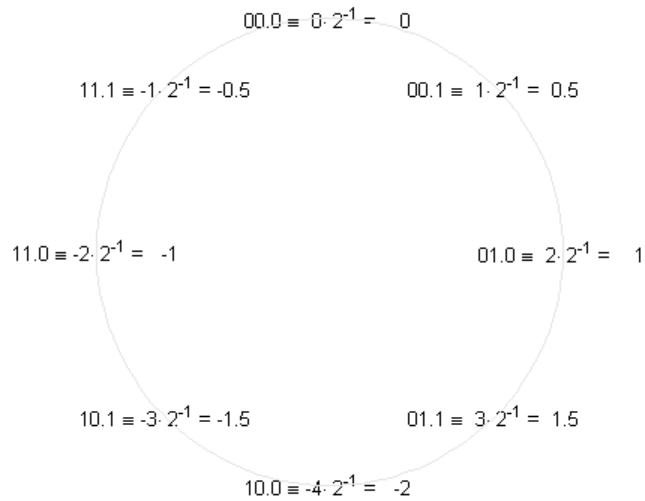
```
b =

100
101
110
111
000
001
010
011
```

Signed Fixed-Point Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

```
fidemo.numbercircle(q);
```



Signed Fractional Fixed-Point.

Signed fractional fixed-point numbers are fixed-point numbers whose fractionlength f is one less than the wordlength n , which produces a scaling such that the range of numbers is between -1 and $1 \cdot 2^{-f}$, inclusive. This is the most common form of fixed-point numbers because it has the nice property that the product of two numbers less than one is a number less than one, and so multiplication does not overflow. The only exception is the case when we are multiplying -1 by -1 , because $+1$ is not an element of this number system. Some processors have a special multiplication instruction for this situation, and some add an extra bit in the product to guard against this overflow.

Thus, the definition of signed fractional fixed-point is the same as signed fixed-point, with the restriction that $f=n-1$, where n is the wordlength in bits.

$$sf = s \cdot 2^{-f}$$

For example, let's define a 3-bit signed fractional fixed-point quantizer, which implies a fractionlength of 2.

```

q = quantizer('fixed',[3 2]);
[a,b] = range(q);
sf = (a:eps(q):b)'

% Now, let's display those values in binary.
b = num2bin(q,sf)

```

```

sf =

    -1
 -0.75
 -0.5
 -0.25
     0
    0.25
    0.5
    0.75

```

```

b =

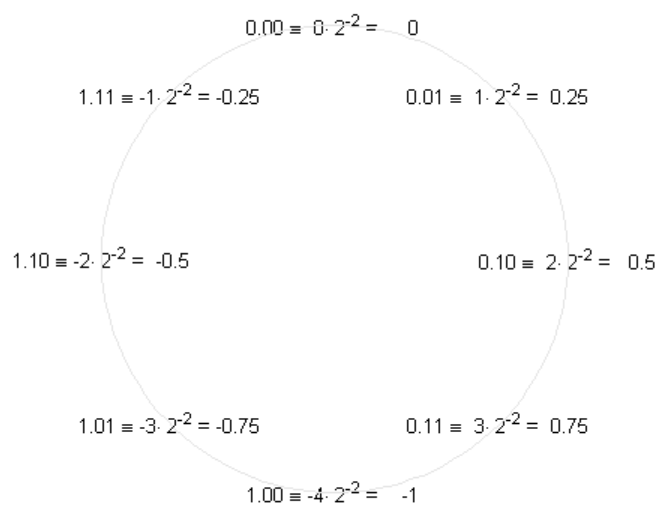
100
101
110
111
000
001
010
011

```

Signed Fractional Fixed-Point Number Circle.

Let's array them around a clock face with their corresponding binary and decimal values.

```
fidemo.numbercircle(q);
```



Perform Binary-Point Scaling

This example shows how to perform binary point scaling in FI.

FI Construction

`a = fi(v,s,w,f)` returns a `fi` with value `v`, signedness `s`, word length `w`, and fraction length `f`.

If `s` is true (signed) the leading or most significant bit (MSB) in the resulting `fi` is always the sign bit.

Fraction length `f` is the scaling $2^{(-f)}$.

For example, create a signed 8-bit long `fi` with a value of 0.5 and a scaling of $2^{(-7)}$:

```
a = fi(0.5,true,8,7)
```

```
a =
```

```
0.5
```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 8
FractionLength: 7

```

Fraction Length and the Position of the Binary Point

The fraction length or the scaling determines the position of the binary point in the `fi` object.

The Fraction Length is Positive and Less than the Word Length

When the fraction length `f` is positive and less than the word length, the binary point lies `f` places to the left of the least significant bit (LSB) and within the word.

For example, in a signed 3-bit `fi` with fraction length of 1 and value -0.5, the binary point lies 1 place to the left of the LSB. In this case each bit is set to 1 and the binary equivalent of the `fi` with its binary point is 11.1 .

The real world value of -0.5 is obtained by multiplying each bit by its scaling factor, starting with the LSB and working up to the signed MSB.

$$(1*2^{-1}) + (1*2^0) + (-1*2^1) = -0.5$$

`storedInteger(a)` returns the stored signed, unscaled integer value -1.

$$(1*2^0) + (1*2^1) + (-1*2^2) = -1$$

```
a = fi(-0.5,true,3,1)
bin(a)
storedInteger(a)
```

```
a =
```

```
-0.5
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 3
FractionLength: 1
```

```
ans =
```

```
111
```

```
ans =
```

```
-1
```

The Fraction Length is Positive and Greater than the Word Length

When the fraction length f is positive and greater than the word length, the binary point lies f places to the left of the LSB and outside the word.

For example the binary equivalent of a signed 3-bit word with fraction length of 4 and value of -0.0625 is `._111`. Here `_` in the `._111` denotes an unused bit that is not a part of the 3-bit word. The first 1 after the `_` is the MSB or the sign bit.

The real world value of -0.0625 is computed as follows (LSB to MSB).

$$(1 \cdot 2^{-4}) + (1 \cdot 2^{-3}) + (-1 \cdot 2^{-2}) = -0.0625$$

`bin(b)` will return 111 at the MATLAB prompt and `storedInteger(b) = -1`

```
b = fi(-0.0625,true,3,4)
bin(b)
storedInteger(b)
```

```
b =
```

```
-0.0625
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 3
FractionLength: 4
```

```
ans =
```

```
111
```

```
ans =
```

```
-1
```

The Fraction Length is a Negative Integer and Less than the Word Length

When the fraction length f is negative the binary point lies f places to the right of LSB and is outside the physical word.

For instance in `c = fi(-4,true,3,-2)` the binary point lies 2 places to the right of the LSB `111__`.. Here the two right most spaces are unused bits that are not part of the 3-bit word. The right most 1 is the LSB and the leading 1 is the sign bit.

The real world value of -4 is obtained by multiplying each bit by its scaling factor $2^{(-f)}$, i.e. $2^{(-(-2))} = 2^{(2)}$ for the LSB, and then adding the products together.

$$(1*2^2) + (1*2^3) + (-1*2^4) = -4$$

`bin(c)` and `storedInteger(c)` will still give 111 and -1 as in the previous two examples.

```
c = fi(-4,true,3,-2)
bin(c)
storedInteger(c)
```

```
c =
```

```
-4
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 3
      FractionLength: -2
```

```
ans =
```

```
111
```

```
ans =
```

```
-1
```

The Fraction Length is Set Automatically to the Best Precision Possible and is Negative

In this example we create a signed 3-bit `fi` where the fraction length is set automatically depending on the value that the `fi` is supposed to contain. The resulting `fi` has a value of 6, with a wordlength of 3 bits and a fraction length of -1. Here the binary point is 1 place to the right of the LSB: 011_.. The _ is again an unused bit and the first 1 before the _ is the LSB. The leading 1 is the sign bit.

The real world value (6) is obtained as follows:

$$(1 \cdot 2^1) + (1 \cdot 2^2) + (-0 \cdot 2^3) = 6$$

`bin(d)` and `storedInteger(d)` will give 011 and 3 respectively.

```
d = fi(5,true,3)
bin(d)
storedInteger(d)
```

```
d =
```

```
6
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 3
      FractionLength: -1
```

```
ans =
```

```
011
```

```
ans =
```

```
3
```

Interactive FI Binary Point Scaling Example

This is an interactive example that allows the user to change the fraction length of a 3-bit fixed-point number by moving the binary point using a slider. The fraction length can be varied from -3 to 5 and the user can change the value of the 3 bits to '0' or '1' for either signed or unsigned numbers.

The "Scaling factors" above the 3 bits display the scaling or weight that each bit is given for the specified signedness and fraction length. The `fi` code, the double precision real-world value and the fixed-point attributes are also displayed.

Type `fibinscaling` at the MATLAB prompt to run this example.

Develop Fixed-Point Algorithms

This example shows how to develop and verify a simple fixed-point algorithm.

Simple Example of Algorithm Development

This example shows the development and verification of a simple fixed-point filter algorithm. We will follow the following steps:

- 1) Implement a second order filter algorithm and simulate in double-precision floating-point.
- 2) Instrument the code to visualize the dynamic range of the output and state.
- 3) Convert the algorithm to fixed-point by changing the data type of the variables - the algorithm itself does not change.
- 4) Compare and plot the fixed-point and floating-point results.

Floating-Point Variable Definitions

We develop our algorithm in double-precision floating-point. We will use a second-order lowpass filter to remove the high frequencies in the input signal.

```
b = [ 0.25 0.5      0.25 ]; % Numerator coefficients
a = [ 1      0.09375 0.28125 ]; % Denominator coefficients
% Random input that has both high and low frequencies.
s = rng; rng(0,'v5uniform');
x = randn(1000,1);
rng(s); % restore RNG state
% Pre-allocate the output and state for speed.
y = zeros(size(x));
z = [0;0];
```

Data-Type-Independent Algorithm

This is a second-order filter that implements the standard difference equation:

$$y(n) = b(1)*x(n) + b(2)*x(n-1) + b(3)*x(n-2) - a(2)*y(n-1) - a(3)*y(n-2)$$

for $k=1:\text{length}(x)$

```
        y(k) = b(1)*x(k) + z(1);
        z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
        z(2) = b(3)*x(k)          - a(3)*y(k);
    end

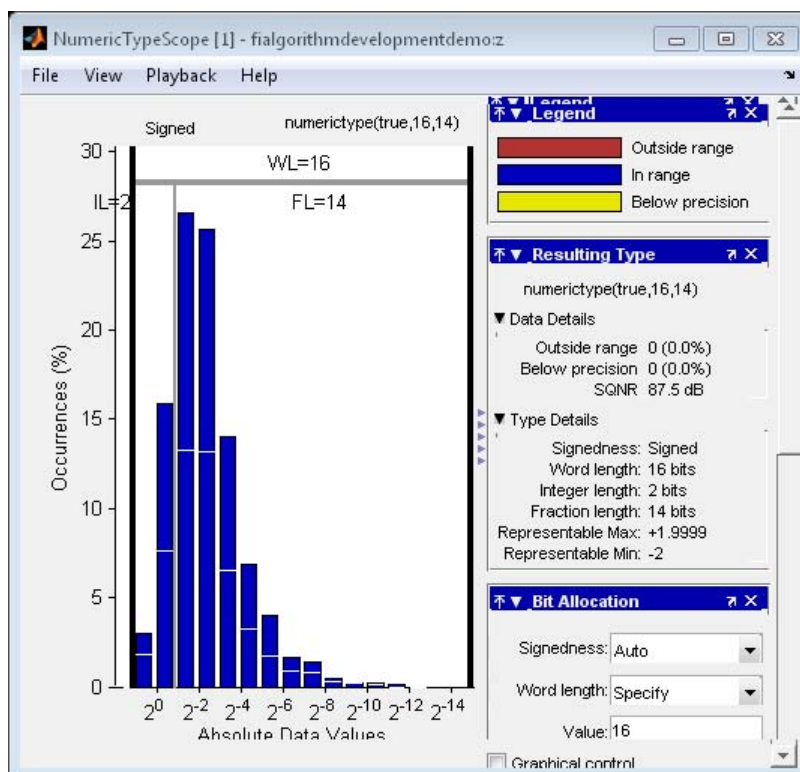
    % Save the Floating-Point Result
    ydouble = y;
```

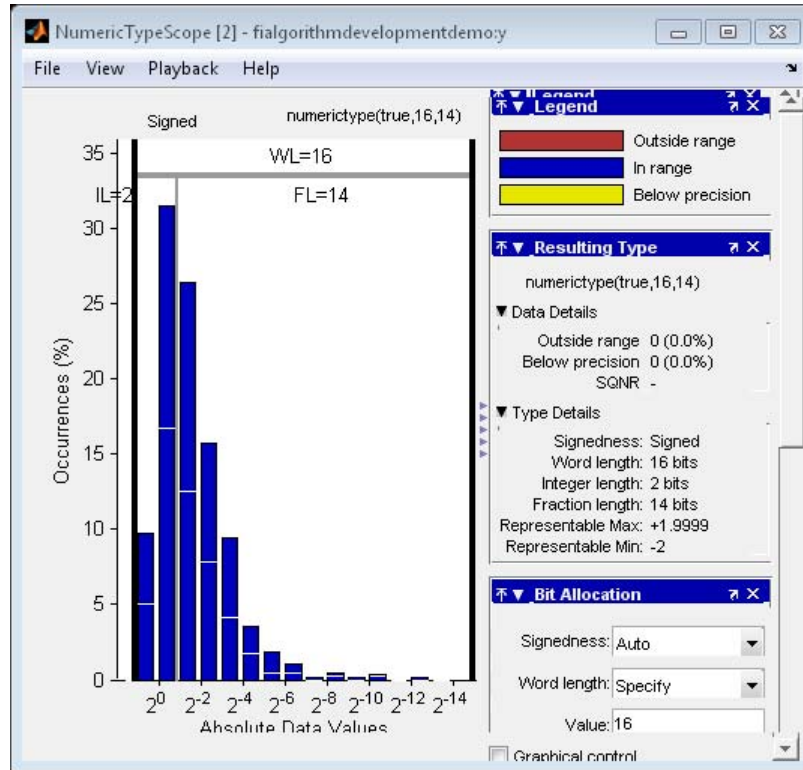
Visualize Dynamic Range

In order to convert to fixed-point, we need to know the range of the variables. Depending on the complexity of an algorithm, this task can be simple or quite challenging. In this example, the range of the input value is known, so selecting an appropriate fixed-point data type is simple. We will concentrate on the output (y) and states (z) since their range is unknown. To view the dynamic range of the output and states, we will modify the code slightly to instrument it. We will create two `NumericTypeScope` objects and view the dynamic range of the output (y) and states (z) simultaneously.

Instrument Floating-Point Code

```
hscope1 = NumericTypeScope;
hscope2 = NumericTypeScope;
for k=1:length(x)
    y(k) = b(1)*x(k) + z(1);
    z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
    z(2) = b(3)*x(k)          - a(3)*y(k);
    % process the data and update the visual.
    step(hscope1,z);
end
step(hscope2,y);
```



Analyze Information in the Scope

Let us first analyze the information displayed for variable z (state). From the histogram we can see that the dynamic range lies between $(2^1 2^{-20}]$.

By default, the scope uses a word length of 16 bits with zero tolerable overflows. This results in a data type of `numerictype(true,16, 14)` since we need at least 1 integer bit to avoid overflows. With this suggested type, values that require more than 14 fractional bits to represent itself will cause an underflow, which is 0.1% in this case and is negligible. You can get more information on the statistical data from the Input Data and Resulting Type panels. From the Input Data panel we can see that the data has both positive and negative values and hence a signed quantity which is reflected in the suggested `numerictype`. Also, the maximum data value is 1.8 which can be represented by the suggested type.

Next, let us look at variable y (output). From the histogram plot we see that the dynamic range lies between $(2^2 \ 2^{-11}]$.

By default, the scope uses a word length of 16 bits with zero tolerable overflows. This results in a data type of `numeric_type(true,16, 13)` since we need at least 2 integer bits to avoid overflows. With this suggested type you see no overflows or underflows. Supposing the application is tolerant to a small amount of overflows, we can optimize the wordsize further. By setting the Maximum Overflow parameter on the Bit Allocation panel to 0.5%, we can reduce the integer length by 1 bit and gain more precision.

Fixed-Point Variable Definitions

We convert variables to fixed-point and run the algorithm again. We will turn on logging to see the overflows and underflows introduced by the selected data types.

```
% Turn on logging to see overflows/underflows.
fp = fipref;
default_loggingmode = fp.LoggingMode;
fp.LoggingMode = 'On';
% Capture the present state of and reset the global fimath to the factory
% settings.
globalFimathAtStart = fimath;
resetglobalfimath;
% Define the fixed-point types for the variables in the below format:
%   fi(Data, Signed, WordLength, FractionLength)
b = fi(b, 1, 8, 6);
a = fi(a, 1, 8, 6);

x = fi(x, 1, 16, 13);
y = fi(zeros(size(x)), 1, 16, 13);
z = fi([0;0], 1, 16, 14);
```

Same Data-Type-Independent Algorithm

```
for k=1:length(x)
    y(k) = b(1)*x(k) + z(1);
    z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
    z(2) = b(3)*x(k) - a(3)*y(k);
```

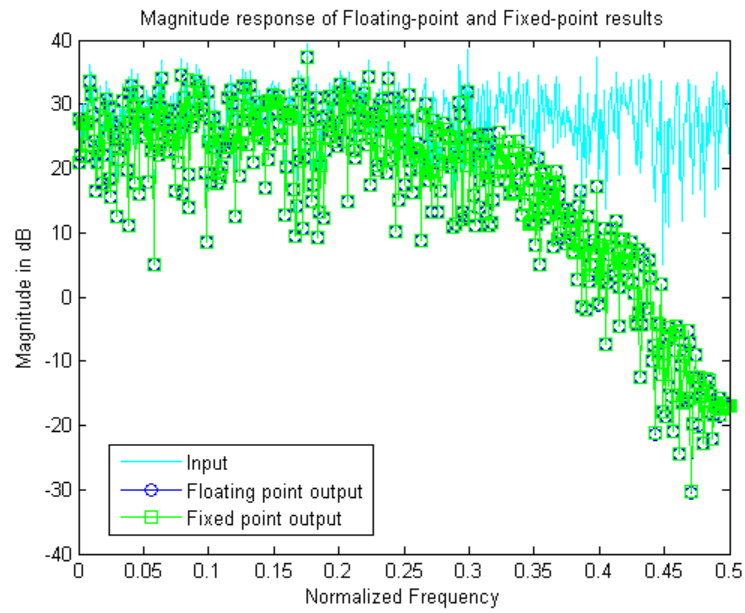
```
end
% Reset the logging mode.
fp.LoggingMode = default_loggingmode;
```

In this example, we have redefined the fixed-point variables with the same names as the floating-point so that we could inline the algorithm code for clarity. However, it is a better practice to enclose the algorithm code in a MATLAB file function that could be called with either floating-point or fixed-point variables. See `filimitcycledemo.m` for an example of writing and using a datatype-agnostic algorithm.

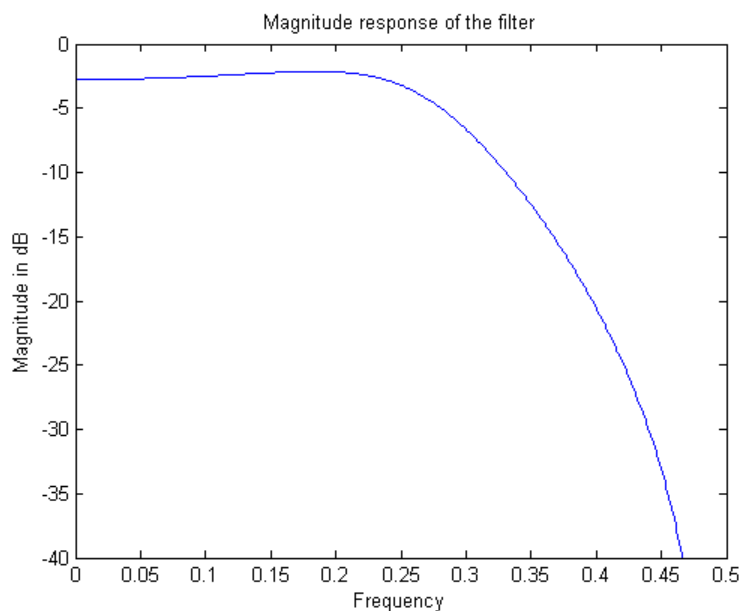
Compare and Plot the Floating-Point and Fixed-Point Results

We will now plot the magnitude response of the floating-point and fixed-point results and the response of the filter to see if the filter behaves as expected when it is converted to fixed-point.

```
n = length(x);
f = linspace(0,0.5,n/2);
x_response = 20*log10(abs(fft(double(x))));
ydouble_response = 20*log10(abs(fft(ydouble())));
y_response = 20*log10(abs(fft(double(y))));
plot(f,x_response(1:n/2),'c-',...
      f,ydouble_response(1:n/2),'bo-',...
      f,y_response(1:n/2),'gs-');
ylabel('Magnitude in dB');
xlabel('Normalized Frequency');
legend('Input','Floating point output','Fixed point output','Location','Bes
title('Magnitude response of Floating-point and Fixed-point results');
```



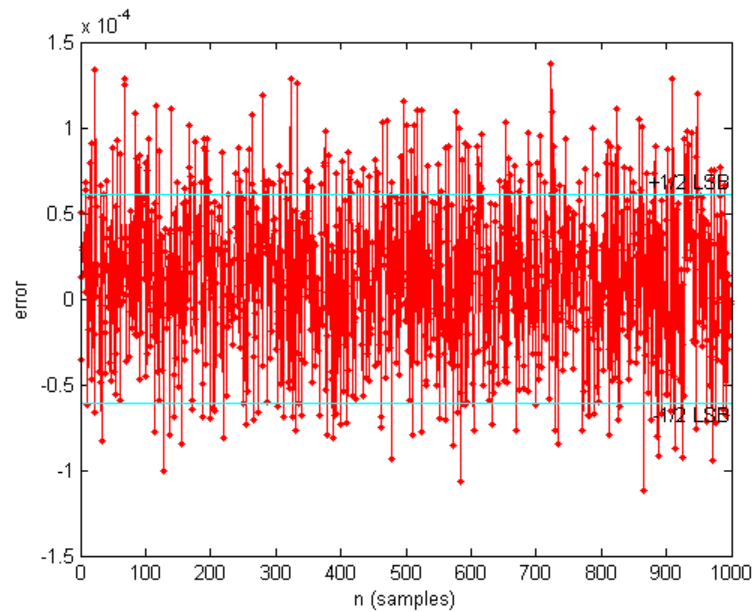
```
h = freqz(double(b),double(a),n/2);
clf
hax = axes;
plot(hax,f,20*log10(abs(h)));
set(hax,'YLim',[-40 0]);
title('Magnitude response of the filter');
ylabel('Magnitude in dB')
xlabel('Frequency');
```



Notice that the high frequencies in the input signal are attenuated by the low-pass filter which is the expected behavior.

Plot the Error

```
clf
n = (0:length(y)-1)';
e = double(lsb(y));
plot(n,double(y)-ydouble,'.-r', ...
     [n(1) n(end)],[e/2 e/2],'c', ...
     [n(1) n(end)],[-e/2 -e/2],'c')
text(n(end),e/2,'+1/2 LSB','HorizontalAlignment','right','VerticalAlignment','bottom')
text(n(end),-e/2,'-1/2 LSB','HorizontalAlignment','right','VerticalAlignment','top')
xlabel('n (samples)'); ylabel('error')
```



Simulink

If you have Simulink and Simulink Fixed Point™, you can run this model, which is the equivalent of the algorithm above. The output, `y_sim` is a fixed-point variable equal to the variable `y` calculated above in MATLAB code.

As in the MATLAB code, the fixed-point parameters in the blocks can be modified to match an actual system; these have been set to match the MATLAB code in the example above. Double-click on the blocks to see the settings.

```
if fidemo.hasSimulinkFixedPointLicense

    % Set up the From Workspace variable
    x_sim.time = n;
    x_sim.signals.values = x;
    x_sim.signals.dimensions = 1;

    % Run the simulation
```

```

out_sim = sim('fitdf2filter_demo', 'SaveOutput', 'on', ...
             'SrcWorkspace', 'current');

% Open the model
fitdf2filter_demo

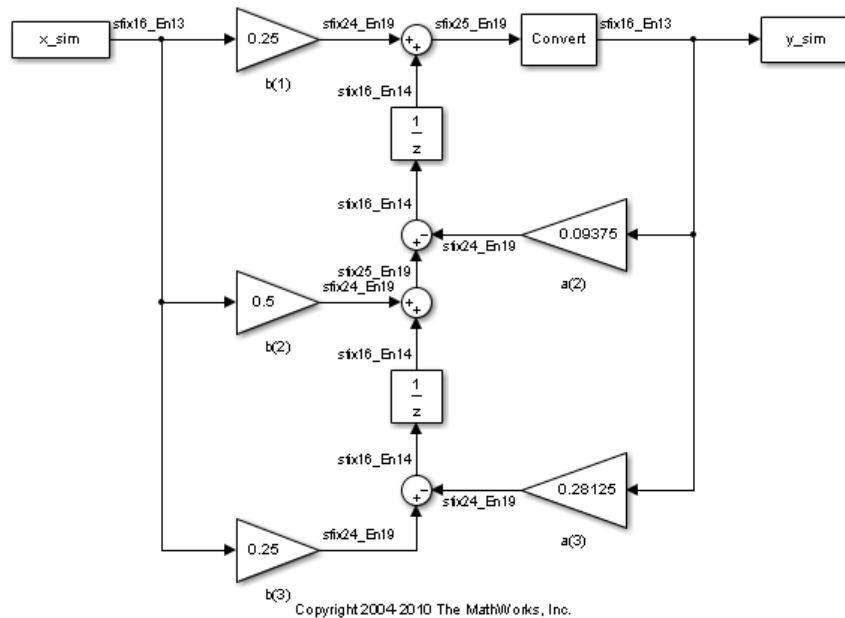
% Verify that the Simulink results are the same as the MATLAB file
isequal(y, out_sim.get('y_sim'))

end

```

```
ans =
```

```
1
```



Assumptions Made for this Example

In order to simplify the example, we have taken the default math parameters: round-to-nearest, saturate on overflow, full precision products and sums. We can modify all of these parameters to match an actual system.

The settings were chosen as a starting point in algorithm development. Save a copy of this MATLAB file, start playing with the parameters, and see what effects they have on the output. How does the algorithm behave with a different input? See the help for `fi`, `fimath`, and `numerictype` for information on how to set other parameters, such as rounding mode, and overflow mode.

```
close all force;  
bdclose all;  
% Reset the global fimath  
globalfimath(globalFimathAtStart);
```

Calculate Fixed-Point Sine and Cosine

This example shows how to use both CORDIC-based and lookup table-based algorithms provided by the Fixed-Point Toolbox™ to approximate the MATLAB sine (SIN) and cosine (COS) functions. Efficient fixed-point sine and cosine algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Calculating Sine and Cosine Using the CORDIC Algorithm

Introduction

The `cordicexp`, `cordicsincos`, `cordicsin`, and `cordiccos` functions approximate the MATLAB `sin` and `cos` functions using a CORDIC-based algorithm. CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm (see [1,2]) is one of the most hardware efficient algorithms because it only requires iterative shift-add operations. The CORDIC algorithm eliminates the need for explicit multipliers, and is suitable for calculating a variety of functions, such as sine, cosine, arcsine, arccosine, arctangent, vector magnitude, divide, square root, hyperbolic and logarithmic functions.

You can use the CORDIC rotation computing mode to calculate sine and cosine, and also polar-to-cartesian conversion operations. In this mode, the vector magnitude and an angle of rotation are known and the coordinate (X-Y) components are computed after rotation.

CORDIC Rotation Computation Mode

The CORDIC rotation mode algorithm begins by initializing an angle accumulator with the desired rotation angle. Next, the rotation decision at each CORDIC iteration is done in a way that decreases the magnitude of the residual angle accumulator. The rotation decision is based on the sign of the residual angle in the angle accumulator after each iteration.

In rotation mode, the CORDIC equations are:

$$z_{i+1} = z_i - d_i * \text{atan}(2^{-i})$$

$$x_{i+1} = x_i - y_i * d_i * 2^{-i}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{-i}$$

where $d_i = -1$ if $z_i < 0$, and $+1$ otherwise;

$i = 0, 1, \dots, N-1$, and N is the total number of iterations.

This provides the following result as N approaches $+\infty$:

$$z_N = 0$$

$$x_N = A_N(x_0 \cos z_0 - y_0 \sin z_0)$$

$$y_N = A_N(y_0 \cos z_0 + x_0 \sin z_0)$$

Where:

$$A_N = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$$

In rotation mode, the CORDIC algorithm is limited to rotation angles between $-\pi/2$ and $\pi/2$. To support angles outside of that range, the `cordicexp`, `cordicsincos`, `cordicsin`, and `cordiccos` functions use quadrant correction (including possible extra negation) after the CORDIC iterations are completed.

Understanding the CORDICSINCOS Sine and Cosine Code

Introduction

The `cordicsincos` function calculates the sine and cosine of input angles in the range $[-2\pi, 2\pi]$ using the CORDIC algorithm. This function takes an angle θ (radians) and the number of iterations as input arguments. The function returns approximations of sine and cosine.

The CORDIC computation outputs are scaled by the rotator gain. This gain is accounted for by pre-scaling the initial $1/A_N$ constant value.

Initialization

The `cordicsincos` function performs the following initialization steps:

- The angle input look-up table `inpLUT` is set to $\text{atan}(2^{-i})$ for $i = 0:N-1$.
- z_0 is set to the θ input argument value.
- x_0 is set to $1/A_N$.
- y_0 is set to zero.

The judicious choice of initial values allows the algorithm to directly compute both sine and cosine simultaneously. After N iterations, these initial values lead to the following outputs as N approaches $+\infty$:

$$x_N \approx \cos(\theta)$$

$$y_N \approx \sin(\theta)$$

Shared Fixed-Point and Floating-Point CORDIC Kernel Code

The MATLAB code for the CORDIC algorithm (rotation mode) kernel portion is as follows (for the case of scalar x , y , and z). This same code is used for both fixed-point and floating-point operations:

```
function [x, y, z] = cordic_rotation_kernel(x, y, z, inpLUT, n)
% Perform CORDIC rotation kernel algorithm for N kernel iterations.
xtmp = x;
ytmp = y;
for idx = 1:n
    if z < 0
        z(:) = accumpos(z, inpLUT(idx));
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
    else
        z(:) = accumneg(z, inpLUT(idx));
        x(:) = accumneg(x, ytmp);
        y(:) = accumpos(y, xtmp);
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end
```

Visualizing the Sine-Cosine Rotation Mode CORDIC Iterations

The CORDIC algorithm is usually run through a specified (constant) number of iterations since ending the CORDIC iterations early would break pipelined code, and the CORDIC gain A_n would not be constant because n would vary.

For very large values of n , the CORDIC algorithm is guaranteed to converge, but not always monotonically. As will be shown in the following example, intermediate iterations occasionally produce more accurate results than later iterations. You can typically achieve greater accuracy by increasing the total number of iterations.

Example

In the following example, iteration 5 provides a better estimate of the result than iteration 6, and the CORDIC algorithm converges in later iterations.

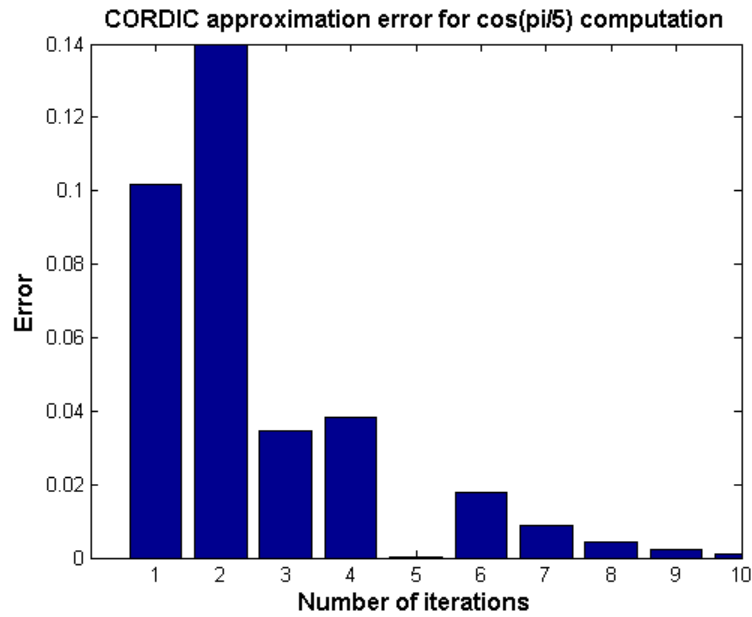
```
theta = pi/5; % input angle in radians
nitters = 10; % number of iterations
sinTh = sin(theta); % reference result
cosTh = cos(theta); % reference result
y_sin = zeros(nitters, 1);
sin_err = zeros(nitters, 1);
x_cos = zeros(nitters, 1);
cos_err = zeros(nitters, 1);
fprintf('\n\nNITERS \tERROR\n');
fprintf('-----\t-----\n');
for n = 1:nitters
    [y_sin(n), x_cos(n)] = cordicsincos(theta, n);
    sin_err(n) = abs(y_sin(n) - sinTh);
    cos_err(n) = abs(x_cos(n) - cosTh);
    if n < 10
        fprintf('  %d \t %1.8f\n', n, cos_err(n));
    else
        fprintf('  %d \t %1.8f\n', n, cos_err(n));
    end
end
fprintf('\n');
```

```
NITERS  ERROR
```

```
-----  
1  0.10191021  
2  0.13966630  
3  0.03464449  
4  0.03846157  
5  0.00020393  
6  0.01776952  
7  0.00888037  
8  0.00436052  
9  0.00208192  
10 0.00093798
```

Plot the CORDIC approximation error on a bar graph

```
figure(1); clf;  
bar(1:niters, cos_err(1:niters));  
xlabel('Number of iterations','fontsize',12,'fontweight','b');  
ylabel('Error','fontsize',12,'fontweight','b');  
title('CORDIC approximation error for cos(pi/5) computation',...  
      'fontsize',12,'fontweight','b');  
axis([0 niters 0 0.14]);
```

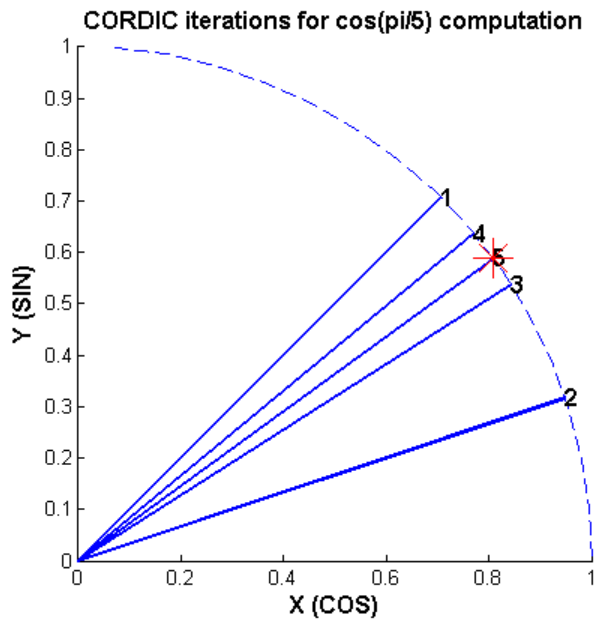


Plot the X-Y results for 5 iterations

```

Niter2Draw = 5;
figure(2), clf, hold on
plot(cos(0:0.1:pi/2), sin(0:0.1:pi/2), 'b--'); % semi-circle
for i=1:Niter2Draw
    plot([0 x_cos(i)], [0 y_sin(i)], 'LineWidth', 2); % CORDIC iteration res
    text(x_cos(i), y_sin(i), int2str(i), 'fontsize', 12, 'fontweight', 'b');
end
plot(cos(theta), sin(theta), 'r*', 'MarkerSize', 20); % IDEAL result
xlabel('X (COS)', 'fontsize', 12, 'fontweight', 'b')
ylabel('Y (SIN)', 'fontsize', 12, 'fontweight', 'b')
title('CORDIC iterations for cos(pi/5) computation', ...
      'fontsize', 12, 'fontweight', 'b')
axis equal;
axis square;

```



Computing Fixed-point Sine with cordicsin

Create 1024 points between $[-2\pi, 2\pi]$

```
stepSize = pi/256;
thRadDb1 = (-2*pi):stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDb1, 12); % signed, 12-bit fixed-point values
sinThRef = sin(double(thRadFxp)); % reference results
```

Compare fixed-point CORDIC vs. double-precision trig function results

Use 12-bit quantized inputs and vary number of iterations from 4 to 10.

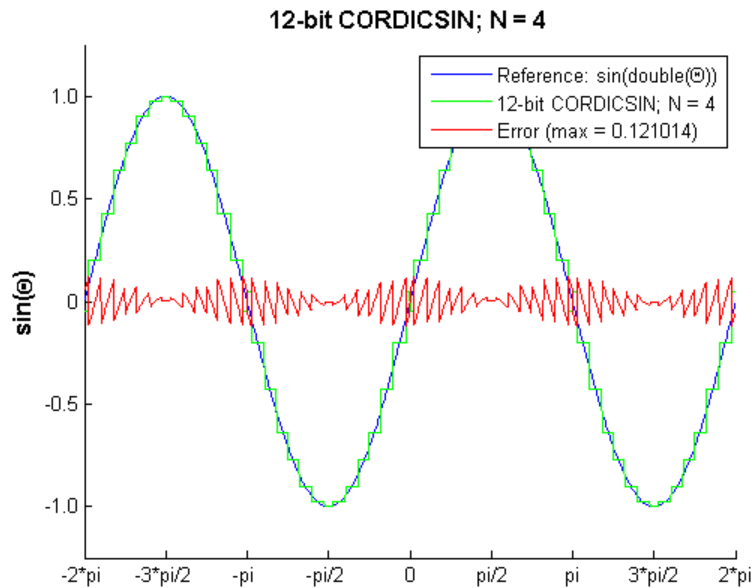
```
for niters = 4:3:10
    cdcSinTh = cordicsin(thRadFxp, niters);
    errCdcRef = sinThRef - double(cdcSinTh);
    figure; hold on; axis([-2*pi 2*pi -1.25 1.25]);
    plot(thRadFxp, sinThRef, 'b');
    plot(thRadFxp, cdcSinTh, 'g');
```

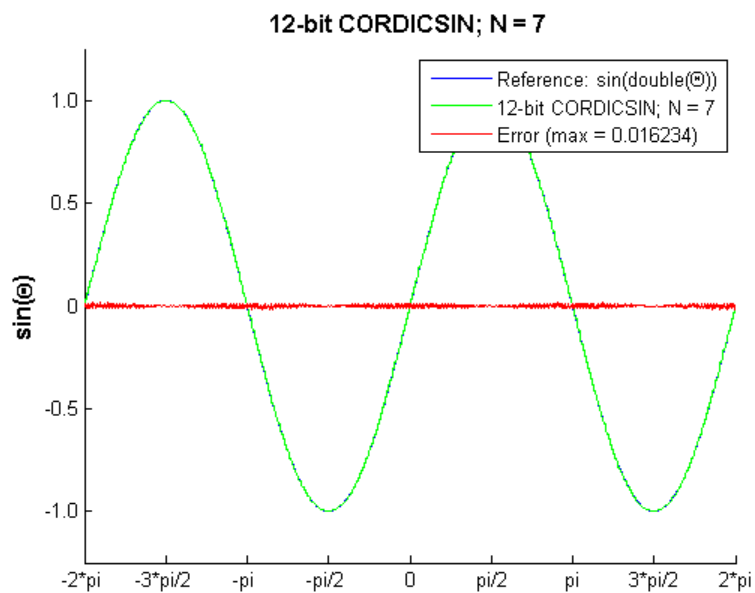


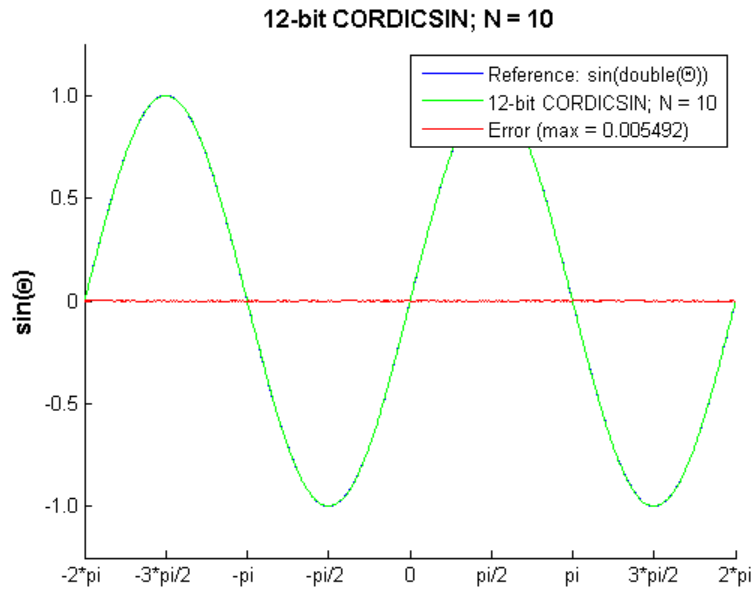
```

plot(thRadFxp, errCdcRef, 'r');
ylabel('sin(\Theta)','fontsize',12,'fontweight','b');
set(gca,'XTick',-2*pi:pi/2:2*pi);
set(gca,'XTickLabel',...
    {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
    '0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
set(gca,'YTick',-1:0.5:1);
set(gca,'YTickLabel',{'-1.0', '-0.5', '0', '0.5', '1.0'});
ref_str = 'Reference: sin(double(\Theta))';
cdc_str = sprintf('12-bit CORDICSIN; N = %d', niters);
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
legend(ref_str, cdc_str, err_str);
title(cdc_str,'fontsize',12,'fontweight','b');
end

```

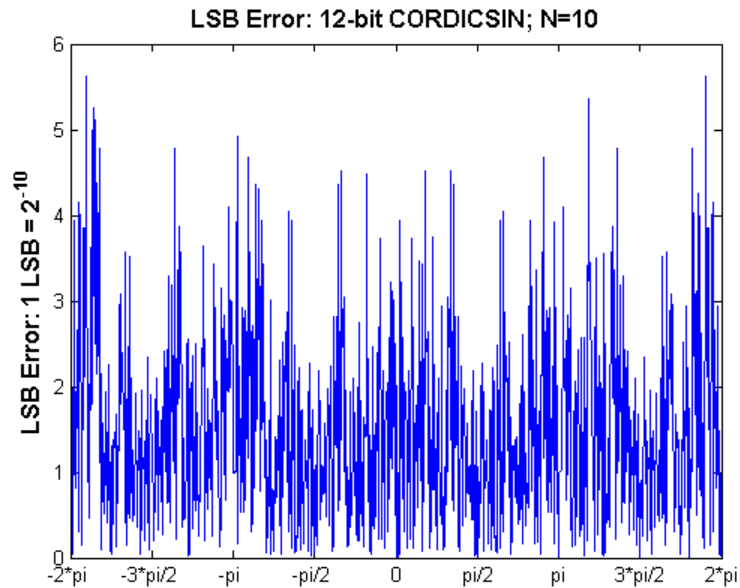






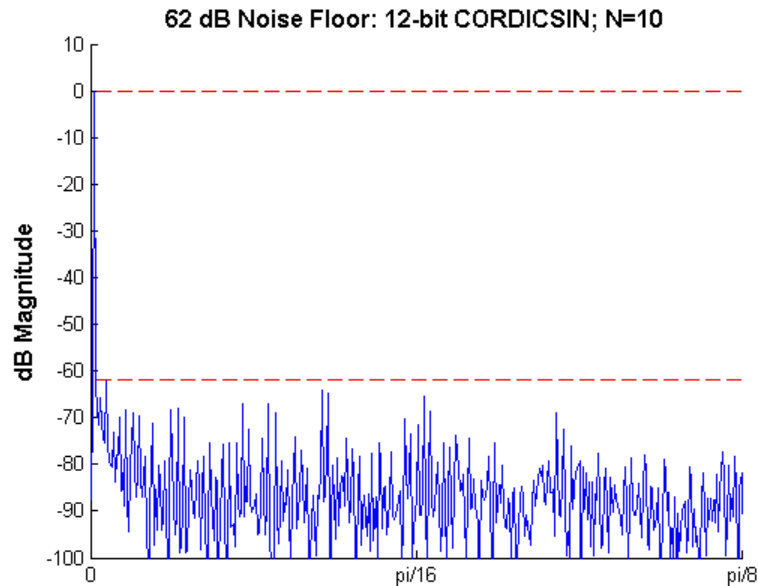
Compute the LSB Error for N = 10

```
figure;
fracLen = cdcSinTh.FractionLength;
plot(thRadFxp, abs(errCdcRef) * pow2(fracLen));
set(gca,'XTick',-2*pi:pi/2:2*pi);
set(gca,'XTickLabel',...
    {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
     '0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
ylabel(sprintf('LSB Error: 1 LSB = 2^{-%d}',fracLen),'fontsize',12,'fontwei
title('LSB Error: 12-bit CORDICSIN; N=10','fontsize',12,'fontweight','b');
axis([-2*pi 2*pi 0 6]);
```



Compute Noise Floor

```
fft_mag = abs(fft(double(cdcSinTh)));
max_mag = max(fft_mag);
mag_db = 20*log10(fft_mag/max_mag);
figure;
hold on;
plot(0:1023, mag_db);
plot(0:1023, zeros(1,1024),'r--'); % Normalized peak (0 dB)
plot(0:1023, -62.*ones(1,1024),'r--'); % Noise floor level
ylabel('dB Magnitude','fontsize',12,'fontweight','b');
title('62 dB Noise Floor: 12-bit CORDICSIN; N=10',...
      'fontsize',12,'fontweight','b');
% axis([0 1023 -120 0]); full FFT
axis([0 round(1024*(pi/8)) -100 10]); % zoom in
set(gca,'XTick',[0 round(1024*pi/16) round(1024*pi/8)]);
set(gca,'XTickLabel',{'0','pi/16','pi/8'});
```



Accelerating the Fixed-Point CORDICSINCOS Function with FIACCEL

You can generate a MEX function from MATLAB code using the MATLAB `fiaccel` function. Typically, running a generated MEX function can improve the simulation speed, although the actual speed improvement depends on the simulation platform being used. The following example shows how to accelerate the fixed-point `cordicsincos` function using `fiaccel`.

The `fiaccel` function compiles the MATLAB code into a MEX function. This step requires the creation of a temporary directory and write permissions in this directory.

```
tempdirObj = fidemo.fiTempdir('fi_sin_cos_demo');
```

When you declare the number of iterations to be a constant (e.g., 10) using `coder.newtype('constant',10)`, the compiled angle look-up table will also be constant, and thus won't be computed at each iteration. Also, when you call `cordicsincos_mex`, you will not need to give it the input argument

for the number of iterations. If you pass in the number of iterations, the MEX-function will error.

The data type of the input parameters determines whether the `cordicsincos` function performs fixed-point or floating-point calculations. When MATLAB generates code for this file, code is only generated for the specific data type. For example, if the `THETA` input argument is fixed point, then only fixed-point code is generated.

```
inp = {thRadFxp, coder.newtype('constant',10)}; % example inputs for the fu
fiaccel('cordicsincos', '-o', 'cordicsincos_mex', '-args', inp)
```

First, calculate sine and cosine by calling `cordicsincos`.

```
tstart = tic;
cordicsincos(thRadFxp,10);
telapsed_Mcordicsincos = toc(tstart);
```

Next, calculate sine and cosine by calling the MEX-function `cordicsincos_mex`.

```
cordicsincos_mex(thRadFxp); % load the MEX file
tstart = tic;
cordicsincos_mex(thRadFxp);
telapsed_MEXcordicsincos = toc(tstart);
```

Now, compare the speed. Type the following at the MATLAB command line to see the speed improvement on your platform:

```
fiaccel_speedup = telapsed_Mcordicsincos/telapsed_MEXcordicsincos;
```

To clean up the temporary directory, run the following commands:

```
clear cordicsincos_mex;
status = tempdirObj.cleanUp;
```

Calculating SIN and COS Using Lookup Tables

There are many lookup table-based approaches that may be used to implement fixed-point sine and cosine approximations. The following is a

low-cost approach based on a single real-valued lookup table and simple nearest-neighbor linear interpolation.

Single Lookup Table Based Approach

The `sin` and `cos` methods of the `fi` object in the Fixed-Point Toolbox approximate the MATLAB builtin floating-point `sin` and `cos` functions, using a lookup table-based approach with simple nearest-neighbor linear interpolation between values. This approach allows for a small real-valued lookup table and uses simple arithmetic.

Using a single real-valued lookup table simplifies the index computation and the overall arithmetic required to achieve very good accuracy of the results. These simplifications yield relatively high speed performance and also relatively low memory requirements.

Understanding the Lookup Table Based SIN and COS Implementation

Lookup Table Size and Accuracy

Two important design considerations of a lookup table are its size and its accuracy. It is not possible to create a table for every possible input value u . It is also not possible to be perfectly accurate due to the quantization of $\sin(u)$ or $\cos(u)$ lookup table values.

As a compromise, the Fixed-Point Toolbox `SIN` and `COS` methods of `FI` use an 8-bit lookup table as part of their implementation. An 8-bit table is only 256 elements long, so it is small and efficient. Eight bits also corresponds to the size of a byte or a word on many platforms. Used in conjunction with linear interpolation, and 16-bit output (lookup table value) precision, an 8-bit-addressable lookup table provides both very good accuracy and performance.

Initializing the Constant SIN Lookup Table Values

For implementation simplicity, table value uniformity, and speed, a full sinewave table is used. First, a quarter-wave `SIN` function is sampled at 64 uniform intervals in the range $[0, \pi/2)$ radians. Choosing a signed 16-bit fractional fixed-point data type for the table values, i.e., `tb1ValsNT = numericity(1,16,15)`, produces best precision results in the `SIN` output

range [-1.0, 1.0). The values are pre-quantized before they are set, to avoid overflow warnings.

```
tblValsNT = numerictype(1,16,15);
quarterSinDblFltPtVals = (sin(2*pi*((0:63) ./ 256)))';
endpointQuantized_Plus1 = 1.0 - double(eps(fi(0,tblValsNT)));

halfSinWaveDblFltPtVals = ...
    [quarterSinDblFltPtVals; ...
     endpointQuantized_Plus1; ...
     flipud(quarterSinDblFltPtVals(2:end))];

fullSinWaveDblFltPtVals = ...
    [halfSinWaveDblFltPtVals; -halfSinWaveDblFltPtVals];

FI_SIN_LUT = fi(fullSinWaveDblFltPtVals, tblValsNT);
```

Overview of Algorithm Implementation

The implementation of the Fixed-Point Toolbox `sin` and `cos` methods of `fi` objects involves first casting the fixed-point angle inputs u (in radians) to a pre-defined data type in the range [0, 2pi]. For this purpose, a modulo-2pi operation is performed to obtain the fixed-point input value `inpValInRange` in the range [0, 2pi] and cast to in the best precision binary point scaled unsigned 16-bit fixed-point type `numerictype(0,16,13)`:

```
% Best UNSIGNED type for real-world value range [0, 2*pi],
% which maps to fixed-point stored integer vals [0, 51472].
inpInRangeNT = numerictype(0,16,13);
```

Next, we get the 16-bit stored unsigned integer value from this in-range fixed-point FI angle value:

```
idxUFIX16 = fi(storedInteger(inpValInRange), numerictype(0,16,0));
```

We multiply the stored integer value by a normalization constant, 65536/51472. The resulting integer value will be in a full-scale `uint16` index range:

```
normConst_NT = numerictype(0,32,31);
normConstant = fi(65536/51472, normConst_NT);
```



```
fullScaleIdx = normConstant * idxUFX16;
idxUFX16(:) = fullScaleIdx;
```

The top 8 most significant bits (MSBs) of this full-scale unsigned 16-bit index `idxUFX16` are used to directly index into the 8-bit sine lookup table. Two table lookups are performed, one at the computed table index location `lutValBelow`, and one at the next index location `lutValAbove`:

```
idxUint8MSBs = uint8(storedInteger(bitsliceget(idxUFX16, 16, 9)));
zeroBasedIdx = int16(idxUint8MSBs);
lutValBelow = FI_SIN_LUT(zeroBasedIdx + 1);
lutValAbove = FI_SIN_LUT(zeroBasedIdx + 2);
```

The remaining 8 least significant bits (LSBs) of `idxUFX16` are used to interpolate between these two table values. The LSB values are treated as a normalized scaling factor with 8-bit fractional data type `rFracNT`:

```
rFracNT = numerictype(0,8,8); % fractional remainder data type
idxFrac8LSBs = reinterpretcast(bitsliceget(idxUFX16,8,1), rFracNT);
rFraction = idxFrac8LSBs;
```

A real multiply is used to determine the weighted difference between the two points. This results in a simple calculation (equivalent to one product and two sums) to obtain the interpolated fixed-point sine result:

```
temp = rFraction * (lutValAbove - lutValBelow);
rslt = lutValBelow + temp;
```

Example

Using the above algorithm, here is an example of the lookup table and linear interpolation process used to compute the value of SIN for a fixed-point input `inpValInRange = 0.425` radians:

```
% Use an arbitrary input value (e.g., 0.425 radians)
inpInRangeNT = numerictype(0,16,13); % best precision, [0, 2*pi] radian
inpValInRange = fi(0.425, inpInRangeNT); % arbitrary fixed-point input angl

% Normalize its stored integer to get full-scale unsigned 16-bit integer in
idxUFX16 = fi(storedInteger(inpValInRange), numerictype(0,16,0));
normConst_NT = numerictype(0,32,31);
```

```

normConstant = fi(65536/51472, normConst_NT);
fullScaleIdx = normConstant * idxUFX16;
idxUFX16(:) = fullScaleIdx;

% Do two table lookups using unsigned 8-bit integer index (i.e., 8 MSBs)
idxUint8MSBs = uint8(storedInteger(bitsliceget(idxUFX16, 16, 9)));
zeroBasedIdx = int16(idxUint8MSBs); % zero-based table index value
lutValBelow = FI_SIN_LUT(zeroBasedIdx + 1); % 1st table lookup value
lutValAbove = FI_SIN_LUT(zeroBasedIdx + 2); % 2nd table lookup value

% Do nearest-neighbor interpolation using 8 LSBs (treat as fractional remainder)
rFracNT = numerictype(0,8,8); % fractional remainder data type
idxFrac8LSBs = reinterpretcast(bitsliceget(idxUFX16,8,1), rFracNT);
rFraction = idxFrac8LSBs; % fractional value for linear interpolation
temp = rFraction * (lutValAbove - lutValBelow);
rslt = lutValBelow + temp;

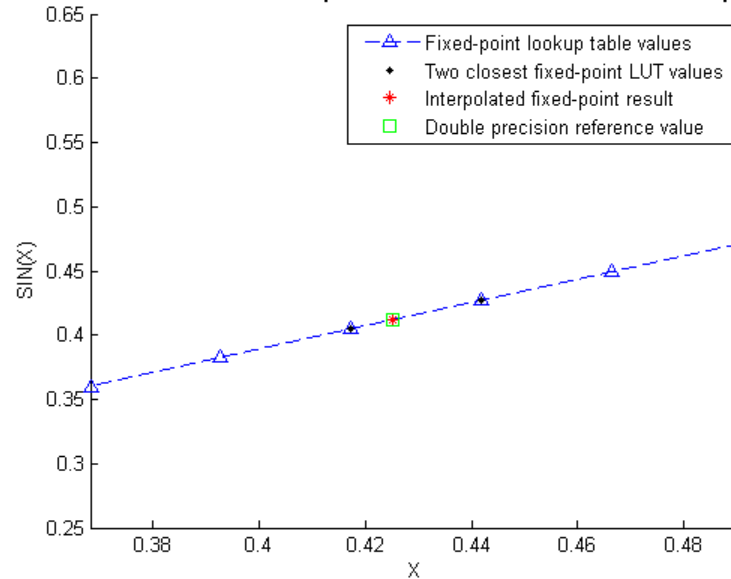
```

Here is a plot of the algorithm results:

```

x_vals = 0:(pi/128):(pi/4);
xIdxLo = zeroBasedIdx - 1;
xIdxHi = zeroBasedIdx + 4;
figure; hold on; axis([x_vals(xIdxLo) x_vals(xIdxHi) 0.25 0.65]);
plot(x_vals(xIdxLo:xIdxHi), double(FI_SIN_LUT(xIdxLo:xIdxHi)), 'b^--');
plot([x_vals(zeroBasedIdx+1) x_vals(zeroBasedIdx+2)], ...
     [lutValBelow lutValAbove], 'k. '); % Closest values
plot(0.425, double(rslt), 'r*'); % Interpolated fixed-point result
plot(0.425, sin(0.425), 'gs'); % Double precision reference result
xlabel('X'); ylabel('SIN(X)');
lut_val_str = 'Fixed-point lookup table values';
near_str = 'Two closest fixed-point LUT values';
interp_str = 'Interpolated fixed-point result';
ref_str = 'Double precision reference value';
legend(lut_val_str, near_str, interp_str, ref_str);
title('Fixed-Point Toolbox Lookup Table Based SIN with Linear Interpolation',
      'fontsize',12,'fontweight','b');

```

Fixed-Point Toolbox Lookup Table Based SIN with Linear Interpolation**Computing Fixed-point Sine Using SIN**

Create 1024 points between $[-2\pi, 2\pi]$

```
stepSize = pi/256;
thRadDb1 = (-2*pi):stepSize:(2*pi - stepSize); % double precision floating-
thRadFxp = sfi(thRadDb1, 12); % signed, 12-bit fixed-point inputs
```

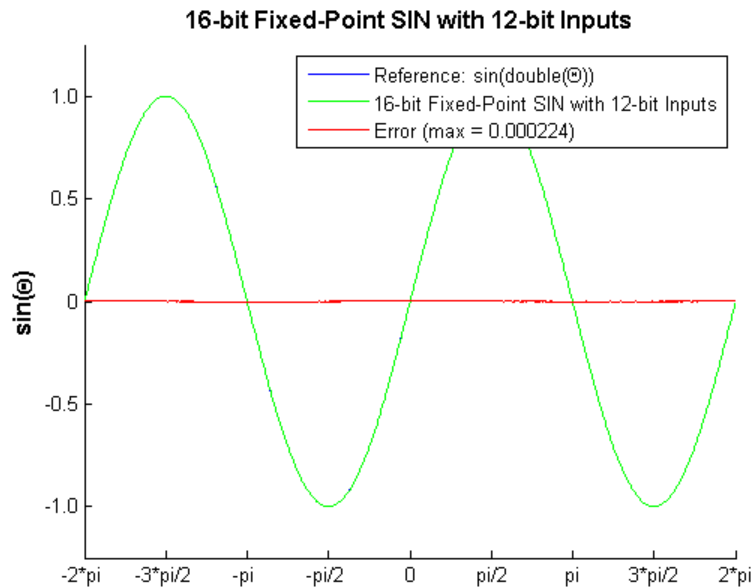
Compare fixed-point SIN vs. double-precision SIN results

```
fxpSinTh = sin(thRadFxp); % fixed-point results
sinThRef = sin(double(thRadFxp)); % reference results
errSinRef = sinThRef - double(fxpSinTh);
figure; hold on; axis([-2*pi 2*pi -1.25 1.25]);
plot(thRadFxp, sinThRef, 'b');
plot(thRadFxp, fxpSinTh, 'g');
plot(thRadFxp, errSinRef, 'r');
ylabel('sin(\Theta)', 'fontsize', 12, 'fontweight', 'b');
set(gca, 'XTick', -2*pi:pi/2:2*pi);
set(gca, 'XTickLabel', ...
```

```

        {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
        '0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
set(gca,'YTick',-1:0.5:1);
set(gca,'YTickLabel',{'-1.0','-0.5','0','0.5','1.0'});
ref_str = 'Reference: sin(double(\Theta))';
fxp_str = sprintf('16-bit Fixed-Point SIN with 12-bit Inputs');
err_str = sprintf('Error (max = %f)', max(abs(errSinRef)));
legend(ref_str, fxp_str, err_str);
title(fxp_str,'fontsize',12,'fontweight','b');

```



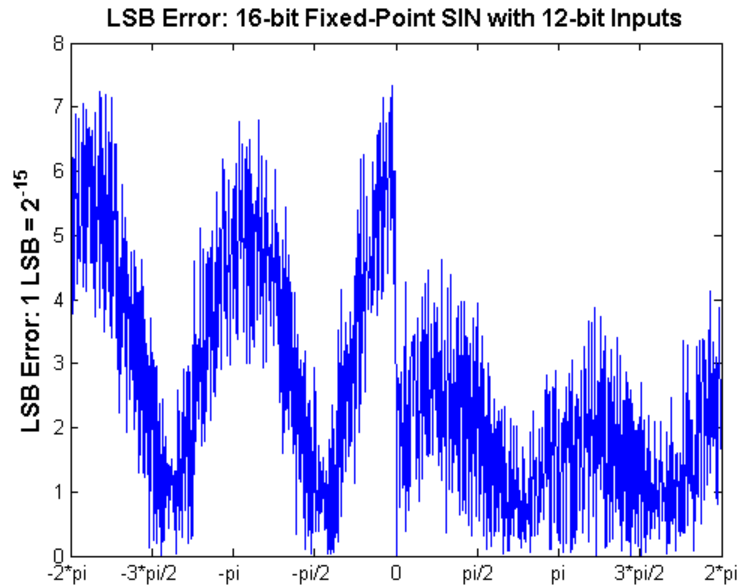
Compute the LSB Error

```

figure;
fracLen = fxpSinTh.FractionLength;
plot(thRadFxp, abs(errSinRef) * pow2(fracLen));
set(gca,'XTick',-2*pi:pi/2:2*pi);
set(gca,'XTickLabel',...
        {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
        '0', 'pi/2', 'pi', '3*pi/2', '2*pi'});

```

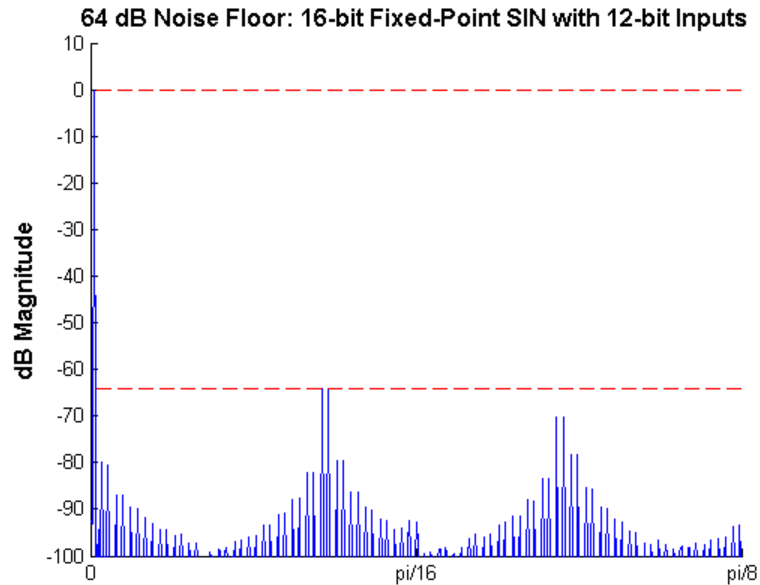
```
ylabel(sprintf('LSB Error: 1 LSB = 2^{-%d}',fracLen),'fontsize',12,'fontwei
title('LSB Error: 16-bit Fixed-Point SIN with 12-bit Inputs','fontsize',12,
axis([-2*pi 2*pi 0 8]));
```



Compute Noise Floor

```
fft_mag = abs(fft(double(fxpSinTh)));
max_mag = max(fft_mag);
mag_db = 20*log10(fft_mag/max_mag);
figure;
hold on;
plot(0:1023, mag_db);
plot(0:1023, zeros(1,1024),'r--'); % Normalized peak (0 dB)
plot(0:1023, -64.*ones(1,1024),'r--'); % Noise floor level (dB)
ylabel('dB Magnitude','fontsize',12,'fontweight','b');
title('64 dB Noise Floor: 16-bit Fixed-Point SIN with 12-bit Inputs',...
'fontsize',12,'fontweight','b');
% axis([0 1023 -120 0]); full FFT
axis([0 round(1024*(pi/8)) -100 10]); % zoom in
```

```
set(gca,'XTick',[0 round(1024*pi/16) round(1024*pi/8)]);
set(gca,'XTickLabel',{'0','pi/16','pi/8'});
```



Comparing the Costs of the Fixed-Point Approximation Algorithms

The fixed-point CORDIC algorithm requires the following operations:

- 1 table lookup **per iteration**
- 2 shifts **per iteration**
- 3 additions **per iteration**

The simplified single lookup table algorithm with nearest-neighbor linear interpolation requires the following operations:

- 2 table lookups
- 1 multiplication
- 2 additions

In real world applications, selecting an algorithm for the fixed-point trigonometric function calculations typically depends on the required accuracy, cost and hardware constraints.

```
close all; % close all figure windows
```

References

- 1** Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.
- 2** Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Calculate Fixed-Point Arctangent

This example shows how to use the CORDIC algorithm, polynomial approximation, and lookup table approaches to calculate the fixed-point, four quadrant inverse tangent. These implementations are approximations to the MATLAB built-in function `atan2`. An efficient fixed-point arctangent algorithm to estimate an angle is critical to many applications, including control of robotics, frequency tracking in wireless communications, and many more.

Calculating `atan2(y,x)` Using the CORDIC Algorithm

Introduction

The `cordicatan2` function approximates the MATLAB `atan2` function, using a CORDIC-based algorithm. CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm (see [1,2]) is one of the most hardware efficient algorithms because it only requires iterative shift-add operations. The CORDIC algorithm eliminates the need for explicit multipliers, and is suitable for calculating a variety of functions, such as sine, cosine, arcsine, arccosine, arctangent, vector magnitude, divide, square root, hyperbolic and logarithmic functions.

CORDIC Vectoring Computation Mode

The CORDIC vectoring mode equations are widely used to calculate `atan(y/x)`. In vectoring mode, the CORDIC rotator rotates the input vector towards the positive X-axis to minimize the *y* component of the residual vector. For each iteration, if the *y* coordinate of the residual vector is positive, the CORDIC rotator rotates clockwise (using a negative angle); otherwise, it rotates counter-clockwise (using a positive angle). If the angle accumulator is initialized to 0, at the end of the iterations, the accumulated rotation angle is the angle of the original input vector.

In vectoring mode, the CORDIC equations are:

$$x_{i+1} = x_i - y_i * d_i * 2^{-i}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{-i}$$

$z_{i+1} = z_i + d_i * \text{atan}(2^{-i})$ is the angle accumulator

where $d_i = +1$ if $y_i < 0$, and -1 otherwise;

$i = 0, 1, \dots, N-1$, and N is the total number of iterations.

As N approaches $+\infty$:

$$x_N = A_N \sqrt{x_0^2 + y_0^2}$$

$$y_N = 0$$

$$z_N = z_0 + \text{atan}(y_0/x_0)$$

$$A_N = 1/(\cos(\text{atan}(2^0)) * \cos(\text{atan}(2^{-1})) * \dots * \cos(\text{atan}(2^{-(N-1)}))) = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$$

As explained above, the arctangent can be directly computed using the vectoring mode CORDIC rotator with the angle accumulator initialized to zero, i.e., $z_0 = 0$, and $z_N \approx \text{atan}(y_0/x_0)$.

Understanding the CORDICATAN2 Code

Introduction

The `cordicatan2` function computes the four quadrant arctangent of the elements of x and y , where $-\pi \leq \text{ATAN2}(y, x) \leq +\pi$. `cordicatan2` calculates the arctangent using the vectoring mode CORDIC algorithm, according to the above CORDIC equations.

Initialization

The `cordicatan2` function performs the following initialization steps:

- x_0 is set to the initial X input value.
- y_0 is set to the initial Y input value.
- z_0 is set to zero.

After N iterations, these initial values lead to $z_N \approx \text{atan}(y_0/x_0)$

Shared Fixed-Point and Floating-Point CORDIC Kernel Code

The MATLAB code for the CORDIC algorithm (vectoring mode) kernel portion is as follows (for the case of scalar x , y , and z). This same code is used for both fixed-point and floating-point operations:

```
function [x, y, z] = cordic_vectoring_kernel(x, y, z, inpLUT, n)
% Perform CORDIC vectoring kernel algorithm for N kernel iterations.
xtmp = x;
ytmp = y;
for idx = 1:n
    if y < 0
        x(:) = accumneg(x, ytmp);
        y(:) = accumpos(y, xtmp);
        z(:) = accumneg(z, inpLUT(idx));
    else
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
        z(:) = accumpos(z, inpLUT(idx));
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end
```

Visualizing the Vectoring Mode CORDIC Iterations

The CORDIC algorithm is usually run through a specified (constant) number of iterations since ending the CORDIC iterations early would break pipelined code, and the CORDIC gain A_n would not be constant because n would vary.

For very large values of n , the CORDIC algorithm is guaranteed to converge, but not always monotonically. As will be shown in the following example, intermediate iterations occasionally rotate the vector closer to the positive X-axis than the following iteration does. You can typically achieve greater accuracy by increasing the total number of iterations.

Example

In the following example, iteration 5 provides a better estimate of the angle than iteration 6, and the CORDIC algorithm converges in later iterations.

Initialize the input vector with angle $\theta = 43$ degrees, magnitude = 1

```
origFormat = get(0, 'format'); % store original format setting;
                                % restore this setting at the end.
format short
%
theta = 43*pi/180; % input angle in radians
Niter = 10;        % number of iterations
inX   = cos(theta); % x coordinate of the input vector
inY   = sin(theta); % y coordinate of the input vector
%
% pre-allocate memories
zf = zeros(1, Niter);
xf = [inX, zeros(1, Niter)];
yf = [inY, zeros(1, Niter)];
angleLUT = atan(2.^(0:Niter-1)); % pre-calculate the angle lookup table
%
% Call CORDIC vectoring kernel algorithm
for k = 1:Niter
    [xf(k+1), yf(k+1), zf(k)] = fixed.internal.cordic_vectoring_kernel_privat
end
```

The following output shows the CORDIC angle accumulation (in degrees) through 10 iterations. Note that the 5th iteration produced less error than the 6th iteration, and that the calculated angle quickly converges to the actual input angle afterward.

```
angleAccumulator = zf*180/pi; angleError = angleAccumulator - theta*180/pi;
fprintf('Iteration: %2d, Calculated angle: %7.3f, Error in degrees: %10g, Error in radians: %10g\n', (1:Niter); angleAccumulator(:)'; angleError(:)'; log2(abs(zf(:)')-th
```

Iteration:	1,	Calculated angle:	45.000,	Error in degrees:	2, Error in radians:
Iteration:	2,	Calculated angle:	18.435,	Error in degrees:	-24.5651, Error in radians:
Iteration:	3,	Calculated angle:	32.471,	Error in degrees:	-10.5288, Error in radians:
Iteration:	4,	Calculated angle:	39.596,	Error in degrees:	-3.40379, Error in radians:
Iteration:	5,	Calculated angle:	43.173,	Error in degrees:	0.172543, Error in radians:
Iteration:	6,	Calculated angle:	41.383,	Error in degrees:	-1.61737, Error in radians:

```
Iteration: 7, Calculated angle: 42.278, Error in degrees: -0.722194, Err
Iteration: 8, Calculated angle: 42.725, Error in degrees: -0.27458, Err
Iteration: 9, Calculated angle: 42.949, Error in degrees: -0.0507692, Err
Iteration: 10, Calculated angle: 43.061, Error in degrees: 0.0611365, Err
```

As N approaches $+\infty$, the CORDIC rotator gain A_N approaches 1.64676. In this example, the input (x_0, y_0) was on the unit circle, so the initial rotator magnitude is 1. The following output shows the rotator magnitude through 10 iterations:

```
rotatorMagnitude = sqrt(xf.^2+yf.^2); % CORDIC rotator gain through iterati
fprintf('Iteration: %2d, Rotator magnitude: %g\n',...
    [(0:Niter); rotatorMagnitude(:)']);
```

```
Iteration: 0, Rotator magnitude: 1
Iteration: 1, Rotator magnitude: 1.41421
Iteration: 2, Rotator magnitude: 1.58114
Iteration: 3, Rotator magnitude: 1.6298
Iteration: 4, Rotator magnitude: 1.64248
Iteration: 5, Rotator magnitude: 1.64569
Iteration: 6, Rotator magnitude: 1.64649
Iteration: 7, Rotator magnitude: 1.64669
Iteration: 8, Rotator magnitude: 1.64674
Iteration: 9, Rotator magnitude: 1.64676
Iteration: 10, Rotator magnitude: 1.64676
```

Note that y_n approaches 0, and x_n approaches $A_n \sqrt{x_0^2 + y_0^2} = A_n$, because $\sqrt{x_0^2 + y_0^2} = 1$.

```
y_n = yf(end)
```

```
y_n =
```

```
-0.0018
```

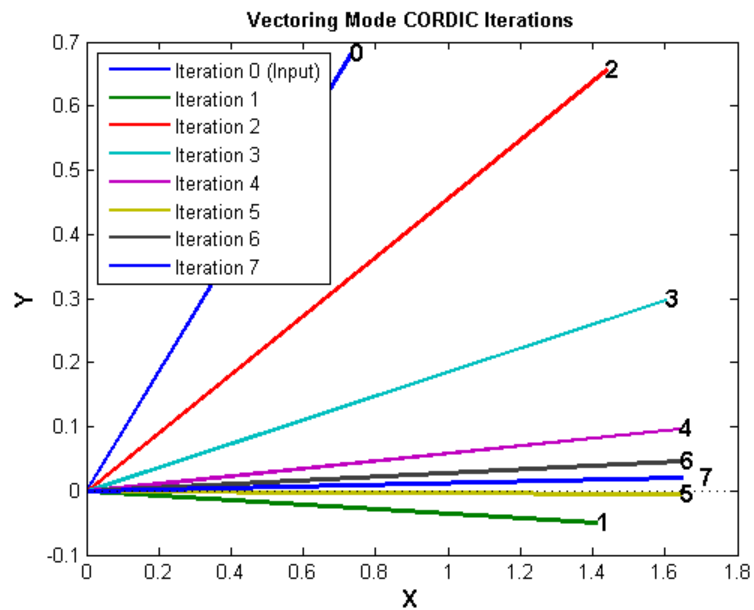
```
x_n = xf(end)
```

```
x_n =
```

```
1.6468
```

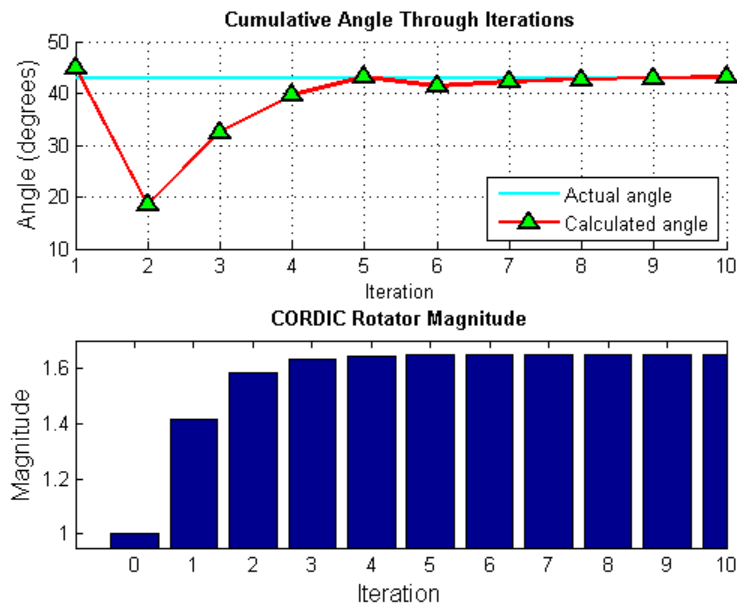
```
figno = 1;
```

```
fidemo.fixpt_atan2_demo_plot(figno, xf, yf) %Vectoring Mode CORDIC Iteratio
```



```
figno = figno + 1; %Cumulative Angle and Rotator Magnitude Through Iteratio
```

```
fidemo.fixpt_atan2_demo_plot(figno,Niter, theta, angleAccumulator, rotatorM
```



Performing Overall Error Analysis of the CORDIC Algorithm

The overall error consists of two parts:

- 1 The algorithmic error that results from the CORDIC rotation angle being represented by a finite number of basic angles.
- 2 The quantization or rounding error that results from the finite precision representation of the angle lookup table, and from the finite precision arithmetic used in fixed-point operations.

Calculate the CORDIC Algorithmic Error

```
theta = (-178:2:180)*pi/180; % angle in radians
inXflt = cos(theta); % generates input vector
inYflt = sin(theta);
Niter = 12; % total number of iterations
zflt = cordicatan2(inYflt, inXflt, Niter); % floating-point results
```

Calculate the maximum magnitude of the CORDIC algorithmic error by comparing the CORDIC computation to the builtin `atan2` function.

```
format long
cordic_algErr_real_world_value = max(abs((atan2(inYflt, inXflt) - zflt)))

cordic_algErr_real_world_value =

    4.753112306290497e-04
```

The log base 2 error is related to the number of iterations. In this example, we use 12 iterations (i.e., accurate to 11 binary digits), so the magnitude of the error is less than 2^{-11}

```
cordic_algErr_bits = log2(cordic_algErr_real_world_value)

cordic_algErr_bits =

    -11.038839889583048
```

Relationship Between Number of Iterations and Precision

Once the quantization error dominates the overall error, i.e., the quantization error is greater than the algorithmic error, increasing the total number of iterations won't significantly decrease the overall error of the fixed-point CORDIC algorithm. You should pick your fraction lengths and total number of iterations to ensure that the quantization error is smaller than the algorithmic error. In the CORDIC algorithm, the precision increases by one bit every iteration. Thus, there is no reason to pick a number of iterations greater than the precision of the input data.

Another way to look at the relationship between the number of iterations and the precision is in the right-shift step of the algorithm. For example, on the counter-clockwise rotation

```
x(:) = x0 - bitsra(y,i);
```

```
y(:) = y + bitsra(x0,i);
```

if i is equal to the word length of y and $x0$, then $\text{bitsra}(y,i)$ and $\text{bitsra}(x0,i)$ shift all the way to zero and do not contribute anything to the next step.

To measure the error from the fixed-point algorithm, and not the differences in input values, compute the floating-point reference with the same inputs as the fixed-point CORDIC algorithm.

```
inXfix = sfi(inXflt, 16, 14);
inYfix = sfi(inYflt, 16, 14);
zref = atan2(double(inYfix), double(inXfix));
zfix8 = cordicatan2(inYfix, inXfix, 8);
zfix10 = cordicatan2(inYfix, inXfix, 10);
zfix12 = cordicatan2(inYfix, inXfix, 12);
zfix14 = cordicatan2(inYfix, inXfix, 14);
zfix15 = cordicatan2(inYfix, inXfix, 15);
cordic_err = bsxfun(@minus,zref,double([zfix8;zfix10;zfix12;zfix14;zfix15]))
```

The error depends on the number of iterations and the precision of the input data. In the above example, the input data is in the range $[-1, +1]$, and the fraction length is 14. From the following tables showing the maximum error at each iteration, and the figure showing the overall error of the CORDIC algorithm, you can see that the error decreases by about 1 bit per iteration until the precision of the data is reached.

```
iterations = [8, 10, 12, 14, 15];
max_cordicErr_real_world_value = max(abs(cordic_err'));
fprintf('Iterations: %2d, Max error in real-world-value: %g\n',...
        [iterations; max_cordicErr_real_world_value]);
```

```
Iterations: 8, Max error in real-world-value: 0.00773633
Iterations: 10, Max error in real-world-value: 0.00187695
Iterations: 12, Max error in real-world-value: 0.000501175
Iterations: 14, Max error in real-world-value: 0.000244621
Iterations: 15, Max error in real-world-value: 0.000244621
```

```
max_cordicErr_bits = log2(max_cordicErr_real_world_value);
fprintf('Iterations: %2d, Max error in bits: %g\n',[iterations; max_cordicErr_bits]);
```



```

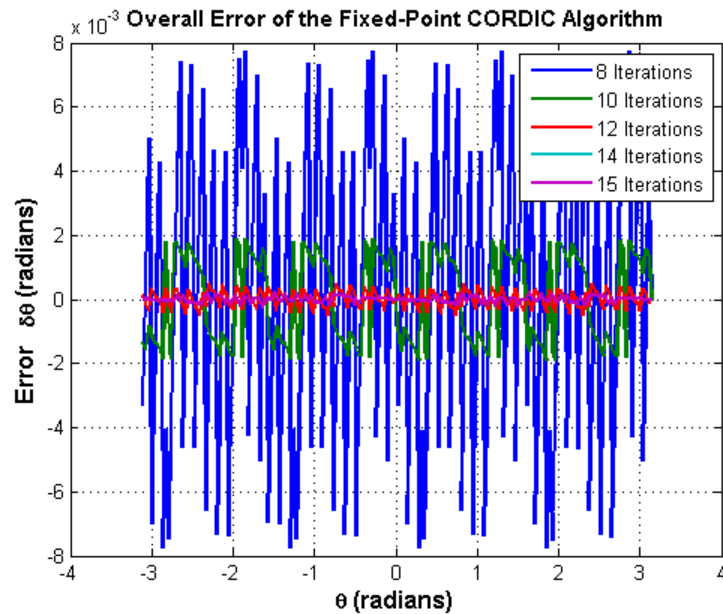
Iterations: 8, Max error in bits: -7.01414
Iterations: 10, Max error in bits: -9.05739
Iterations: 12, Max error in bits: -10.9624
Iterations: 14, Max error in bits: -11.9972
Iterations: 15, Max error in bits: -11.9972

```

```

figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, cordic_err)

```



Accelerating the Fixed-Point CORDICATAN2 Algorithm Using FIACCEL

You can generate a MEX function from MATLAB code using the MATLAB `fiacel` command. Typically, running a generated MEX function can improve the simulation speed, although the actual speed improvement depends on the simulation platform being used. The following example shows how to accelerate the fixed-point `cordicatan2` algorithm using `fiacel`.

The `fiaccel` function compiles the MATLAB code into a MEX function. This step requires the creation of a temporary directory and write permissions in that directory.

```
tempdirObj = fidemo.fiTempdir('fixpt_atan2_demo');
```

When you declare the number of iterations to be a constant (e.g., 12) using `coder.newtype('constant',12)`, the compiled angle lookup table will also be constant, and thus won't be computed at each iteration. Also, when you call the compiled MEX file `cordicatan2_mex`, you will not need to give it the input argument for the number of iterations. If you pass in the number of iterations, the MEX function will error.

The data type of the input parameters determines whether the `cordicatan2` function performs fixed-point or floating-point calculations. When MATLAB generates code for this file, code is only generated for the specific data type. For example, if the inputs are fixed point, only fixed-point code is generated.

```
inp = {inYfix, inXfix, coder.newtype('constant',12)}; % example inputs for  
fiaccel('cordicatan2', '-o', 'cordicatan2_mex', '-args', inp)
```

First, calculate a vector of 4 quadrant `atan2` by calling `cordicatan2`.

```
tstart = tic;  
cordicatan2(inYfix,inXfix,Niter);  
telapsed_Mcordicatan2 = toc(tstart);
```

Next, calculate a vector of 4 quadrant `atan2` by calling the MEX-function `cordicatan2_mex`

```
cordicatan2_mex(inYfix,inXfix); % load the MEX file  
tstart = tic;  
cordicatan2_mex(inYfix,inXfix);  
telapsed_MEXcordicatan2 = toc(tstart);
```

Now, compare the speed. Type the following in the MATLAB command window to see the speed improvement on your specific platform:

```
fiaccel_speedup = telapsed_Mcordicatan2/telapsed_MEXcordicatan2;
```

To clean up the temporary directory, run the following commands:

```
clear cordicatan2_mex;
status = tempdirObj.cleanUp;
```

Calculating atan2(y,x) Using Chebyshev Polynomial Approximation

Polynomial approximation is a multiply-accumulate (MAC) centric algorithm. It can be a good choice for DSP implementations of non-linear functions like atan(x).

For a given degree of polynomial, and a given function $f(x) = \text{atan}(x)$ evaluated over the interval of $[-1, +1]$, the polynomial approximation theory tries to find the polynomial that minimizes the maximum value of $|P(x) - f(x)|$, where $P(x)$ is the approximating polynomial. In general, you can obtain polynomials very close to the optimal one by approximating the given function in terms of Chebyshev polynomials and cutting off the polynomial at the desired degree.

The approximation of arctangent over the interval of $[-1, +1]$ using the Chebyshev polynomial of the first kind is summarized in the following formula:

$$\text{atan}(x) = 2 \sum_{n=0}^{\infty} \frac{(-1)^n q^{2n+1}}{(2n+1)} T_{2n+1}(x)$$

where

$$q = 1/(1 + \sqrt{2})$$

$$x \in [-1, +1]$$

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Therefore, the 3rd order Chebyshev polynomial approximation is

$$\text{atan}(x) = 0.970562748477141 * x - 0.189514164974601 * x^3.$$

The 5th order Chebyshev polynomial approximation is

$$\text{atan}(x) = 0.994949366116654 * x - 0.287060635532652 * x^3 + 0.078037176446441 * x^5$$

The 7th order Chebyshev polynomial approximation is

$$\begin{aligned} \text{atan}(x) = & 0.999133448222780 * x - 0.320533292381664 * x^3 \\ & + 0.144982490144465 * x^5 - 0.038254464970299 * x^7. \end{aligned}$$

You can obtain four quadrant output through angle correction based on the properties of the arctangent function.

Comparing the Algorithmic Error of the CORDIC and Polynomial Approximation Algorithms

In general, higher degrees of polynomial approximation produce more accurate final results. However, higher degrees of polynomial approximation also increase the complexity of the algorithm and require more MAC operations and more memory. To be consistent with the CORDIC algorithm and the MATLAB atan2 function, the input arguments consist of both x and y coordinates instead of the ratio y/x.

To eliminate quantization error, floating-point implementations of the CORDIC and Chebyshev polynomial approximation algorithms are used in the comparison below. An algorithmic error comparison reveals that increasing the number of CORDIC iterations results in less error. It also reveals that the CORDIC algorithm with 12 iterations provides a slightly better angle estimation than the 5th order Chebyshev polynomial approximation. The approximation error of the 3rd order Chebyshev Polynomial is about 8 times larger than that of the 5th order Chebyshev polynomial. You should choose the order or degree of the polynomial based on the required accuracy of the angle estimation and the hardware constraints.

The coefficients of the Chebyshev polynomial approximation for atan(x) are shown in ascending order of x.

```
constA3 = [0.970562748477141, -0.189514164974601]; % 3rd order
constA5 = [0.994949366116654, -0.287060635532652, 0.078037176446441]; % 5th order
constA7 = [0.999133448222780, -0.320533292381664, 0.144982490144465, ...
           -0.038254464970299]; % 7th order
```

```

theta    = (-90:1:90)*pi/180; % angle in radians
inXflt   = cos(theta);
inYflt   = sin(theta);
zfltRef  = atan2(inYflt, inXflt); % Ideal output from ATAN2 function
zfltp3   = fidemo.poly_atan2(inYflt,inXflt,3,constA3); % 3rd order polynomial
zfltp5   = fidemo.poly_atan2(inYflt,inXflt,5,constA5); % 5th order polynomial
zfltp7   = fidemo.poly_atan2(inYflt,inXflt,7,constA7); % 7th order polynomial
zflt8    = cordicatan2(inYflt, inXflt, 8); % CORDIC alg with 8 iterations
zflt12   = cordicatan2(inYflt, inXflt, 12); % CORDIC alg with 12 iterations

```

The maximum algorithmic error magnitude (or infinity norm of the algorithmic error) for the CORDIC algorithm with 8 and 12 iterations is shown below:

```

cordic_algErr    = [zfltRef;zfltRef] - [zflt8;zflt12];
max_cordicAlgErr = max(abs(cordic_algErr));
fprintf('Iterations: %2d, CORDIC algorithmic error in real-world-value: %g\n',...
        [[8,12]; max_cordicAlgErr(:)']);

```

```

Iterations: 8, CORDIC algorithmic error in real-world-value: 0.00772146
Iterations: 12, CORDIC algorithmic error in real-world-value: 0.000483258

```

The log base 2 error shows the number of binary digits of accuracy. The 12th iteration of the CORDIC algorithm has an estimated angle accuracy of 2^{-11} :

```

max_cordicAlgErr_bits = log2(max_cordicAlgErr);
fprintf('Iterations: %2d, CORDIC algorithmic error in bits: %g\n',...
        [[8,12]; max_cordicAlgErr_bits(:)']);

```

```

Iterations: 8, CORDIC algorithmic error in bits: -7.01691
Iterations: 12, CORDIC algorithmic error in bits: -11.0149

```

The following code shows the magnitude of the maximum algorithmic error of the polynomial approximation for orders 3, 5, and 7:

```

poly_algErr    = [zfltRef;zfltRef;zfltRef] - [zfltp3;zfltp5;zfltp7];
max_polyAlgErr = max(abs(poly_algErr));
fprintf('Order: %d, Polynomial approximation algorithmic error in real-world-value: %g\n',...
        [3:2:7; max_polyAlgErr(:)']);

```

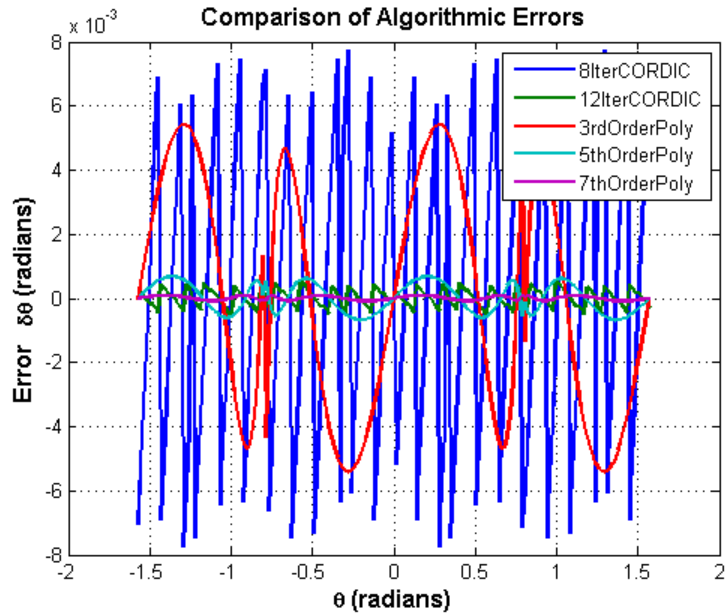
```
Order: 3, Polynomial approximation algorithmic error in real-world-value: 0
Order: 5, Polynomial approximation algorithmic error in real-world-value: 0
Order: 7, Polynomial approximation algorithmic error in real-world-value: 9
```

The log base 2 error shows the number of binary digits of accuracy.

```
max_polyAlgErr_bits = log2(max_polyAlgErr);
fprintf('Order: %d, Polynomial approximation algorithmic error in bits: %g\
      [3:2:7; max_polyAlgErr_bits(:)']');
```

```
Order: 3, Polynomial approximation algorithmic error in bits: -7.52843
Order: 5, Polynomial approximation algorithmic error in bits: -10.5235
Order: 7, Polynomial approximation algorithmic error in bits: -13.414
```

```
figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, cordic_algErr, poly_algErr)
```



Converting the Floating-Point Chebyshev Polynomial Approximation Algorithm to Fixed Point

Assume the input and output word lengths are constrained to 16 bits by the hardware, and the 5th order Chebyshev polynomial is used in the approximation. Because the dynamic range of inputs x , y and y/x are all within $[-1, +1]$, you can avoid overflow by picking a signed fixed-point input data type with a word length of 16 bits and a fraction length of 14 bits. The coefficients of the polynomial are purely fractional and within $(-1, +1)$, so we can pick their data types as signed fixed point with a word length of 16 bits and a fraction length of 15 bits (best precision). The algorithm is robust because $(y/x)^n$ is within $[-1, +1]$, and the multiplication of the coefficients and $(y/x)^n$ is within $(-1, +1)$. Thus, the dynamic range will not grow, and due to the pre-determined fixed-point data types, overflow is not expected.

Similar to the CORDIC algorithm, the four quadrant polynomial approximation-based `atan2` algorithm outputs estimated angles within $[-\pi, \pi]$. Therefore, we can pick an output fraction length of 13 bits to avoid overflow and provide a dynamic range of $[-4, +3.9998779296875]$.

The basic floating-point Chebyshev polynomial approximation of arctangent over the interval $[-1, +1]$ is implemented as the `chebyPoly_atan_fltpt` local function in the `poly_atan2.m` file.

```
function z = chebyPoly_atan_fltpt(y,x,N,constA,Tz,RoundingMethodStr)

tmp = y/x;
switch N
    case 3
        z = constA(1)*tmp + constA(2)*tmp^3;
    case 5
        z = constA(1)*tmp + constA(2)*tmp^3 + constA(3)*tmp^5;
    case 7
        z = constA(1)*tmp + constA(2)*tmp^3 + constA(3)*tmp^5 + constA(4)*tmp^7;
    otherwise
        disp('Supported order of Chebyshev polynomials are 3, 5 and 7');
end
```

The basic fixed-point Chebyshev polynomial approximation of arctangent over the interval $[-1, +1]$ is implemented as the `chebyPoly_atan_fixpt` local function in the `poly_atan2.m` file.

```
function z = chebyPoly_atan_fixpt(y,x,N,constA,Tz,RoundingMethodStr)

z = fi(0,'numerictype', Tz, 'RoundingMethod', RoundingMethodStr);
Tx = numerictype(x);
tmp = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
tmp(:) = Tx.divide(y, x); % y/x;

tmp2 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
tmp3 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
tmp2(:) = tmp*tmp; % (y/x)^2
tmp3(:) = tmp2*tmp; % (y/x)^3

z(:) = constA(1)*tmp + constA(2)*tmp3; % for order N = 3

if (N == 5) || (N == 7)
    tmp5 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
    tmp5(:) = tmp3 * tmp2; % (y/x)^5
    z(:) = z + constA(3)*tmp5; % for order N = 5

    if N == 7
        tmp7 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
        tmp7(:) = tmp5 * tmp2; % (y/x)^7
        z(:) = z + constA(4)*tmp7; %for order N = 7
    end
end
```

The universal four quadrant `atan2` calculation using Chebyshev polynomial approximation is implemented in the `poly_atan2.m` file.

```
function z = poly_atan2(y,x,N,constA,Tz,RoundingMethodStr)

if nargin < 5
    % floating-point algorithm
    fhandle = @chebyPoly_atan_fltpt;
    Tz = [];
    RoundingMethodStr = [];
    z = zeros(size(y));
else
```



```

        % fixed-point algorithm
        fhandle = @chebyPoly_atan_fixpt;
        %pre-allocate output
        z = fi(zeros(size(y)), 'numericType', Tz, 'RoundingMethod', RoundingMethod);
    end

    % Apply angle correction to obtain four quadrant output
    for idx = 1:length(y)
        % first quadrant
        if abs(x(idx)) >= abs(y(idx))
            % (0, pi/4]
            z(idx) = feval(fhandle, abs(y(idx)), abs(x(idx)), N, constA, Tz, RoundingMethod);
        else
            % (pi/4, pi/2)
            z(idx) = pi/2 - feval(fhandle, abs(x(idx)), abs(y(idx)), N, constB, Tz, RoundingMethod);
        end

        if x(idx) < 0
            % second and third quadrant
            if y(idx) < 0
                z(idx) = -pi + z(idx);
            else
                z(idx) = pi - z(idx);
            end
        else % fourth quadrant
            if y(idx) < 0
                z(idx) = -z(idx);
            end
        end
    end
end

```

Performing the Overall Error Analysis of the Polynomial Approximation Algorithm

Similar to the CORDIC algorithm, the overall error of the polynomial approximation algorithm consists of two parts - the algorithmic error and the quantization error. The algorithmic error of the polynomial approximation algorithm was analyzed and compared to the algorithmic error of the CORDIC algorithm in a previous section.

Calculate the Quantization Error

Compute the quantization error by comparing the fixed-point polynomial approximation to the floating-point polynomial approximation.

Quantize the inputs and coefficients with convergent rounding:

```
inXfix = fi(fi(inXflt, 1, 16, 14, 'RoundingMethod', 'Convergent'), 'fimath', [
inYfix = fi(fi(inYflt, 1, 16, 14, 'RoundingMethod', 'Convergent'), 'fimath', [
constAfix3 = fi(fi(constA3, 1, 16, 'RoundingMethod', 'Convergent'), 'fimath', [
constAfix5 = fi(fi(constA5, 1, 16, 'RoundingMethod', 'Convergent'), 'fimath', [
constAfix7 = fi(fi(constA7, 1, 16, 'RoundingMethod', 'Convergent'), 'fimath', [
```

Calculate the maximum magnitude of the quantization error using Floor rounding:

```
ord      = 3:2:7; % using 3rd, 5th, 7th order polynomials
Tz       = numerictype(1, 16, 13); % output data type
zfix3p = fidemo.poly_atan2(inYfix,inXfix,ord(1),constAfix3,Tz,'Floor'); % 3
zfix5p = fidemo.poly_atan2(inYfix,inXfix,ord(2),constAfix5,Tz,'Floor'); % 5
zfix7p = fidemo.poly_atan2(inYfix,inXfix,ord(3),constAfix7,Tz,'Floor'); % 7
poly_quantErr = bsxfun(@minus, [zfltp3;zfltp5;zfltp7], double([zfix3p;zfix5p;zfix7p]));
max_polyQuantErr_real_world_value = max(abs(poly_quantErr));
max_polyQuantErr_bits = log2(max_polyQuantErr_real_world_value);
fprintf('PolyOrder: %2d, Quant error in bits: %g\n',...
        [ord; max_polyQuantErr_bits]);
```

```
PolyOrder:  3, Quant error in bits: -12.7101
PolyOrder:  5, Quant error in bits: -12.325
PolyOrder:  7, Quant error in bits: -11.8416
```

Calculate the Overall Error

Compute the overall error by comparing the fixed-point polynomial approximation to the builtin `atan2` function. The ideal reference output is `zfltRef`. The overall error of the 7th order polynomial approximation is dominated by the quantization error, which is due to the finite precision of the input data, coefficients and the rounding effects from the fixed-point arithmetic operations.

```

poly_err = bsxfun(@minus, zfltRef, double([zfix3p;zfix5p;zfix7p]));
max_polyErr_real_world_value = max(abs(poly_err));
max_polyErr_bits = log2(max_polyErr_real_world_value);
fprintf('PolyOrder: %2d, Overall error in bits: %g\n',...
        [ord; max_polyErr_bits]);

```

```

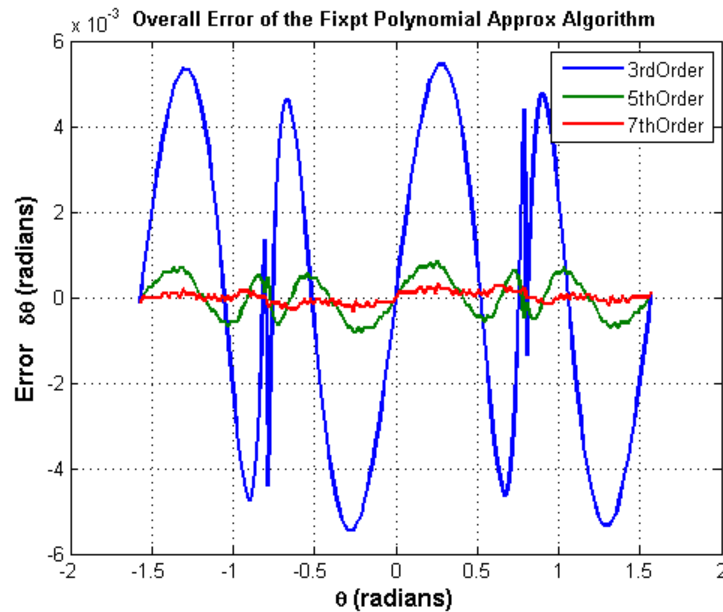
PolyOrder:  3, Overall error in bits: -7.51907
PolyOrder:  5, Overall error in bits: -10.2497
PolyOrder:  7, Overall error in bits: -11.5883

```

```

figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, poly_err)

```



The Effect of Rounding Modes in Polynomial Approximation

Compared to the CORDIC algorithm with 12 iterations and a 13-bit fraction length in the angle accumulator, the fifth order Chebyshev polynomial approximation gives a similar order of quantization error. In the following

example, Nearest, Round and Convergent rounding modes give smaller quantization errors than the Floor rounding mode.

Maximum magnitude of the quantization error using Floor rounding

```
poly5_quantErrFloor = max(abs(poly_quantErr(2,:)));  
poly5_quantErrFloor_bits = log2(poly5_quantErrFloor)
```

```
poly5_quantErrFloor_bits =
```

```
-12.324996933210334
```

For comparison, calculate the maximum magnitude of the quantization error using Nearest rounding:

```
zfixp5n = fidemo.poly_atan2(inYfix,inXfix,5,constAfix5,Tz,'Nearest');  
poly5_quantErrNearest = max(abs(zfltp5 - double(zfixp5n)));  
poly5_quantErrNearest_bits = log2(poly5_quantErrNearest)  
set(0, 'format', origFormat); % reset MATLAB output format
```

```
poly5_quantErrNearest_bits =
```

```
-13.175966487895451
```

Calculating atan2(y,x) Using Lookup Tables

There are many lookup table based approaches that may be used to implement fixed-point argtangent approximations. The following is a low-cost approach based on a single real-valued lookup table and simple nearest-neighbor linear interpolation.

Single Lookup Table Based Approach

The atan2 method of the fi object in the Fixed-Point Toolbox approximates the MATLAB builtin floating-point atan2 function, using a single lookup table based approach with simple nearest-neighbor linear interpolation between

values. This approach allows for a small real-valued lookup table and uses simple arithmetic.

Using a single real-valued lookup table simplifies the index computation and the overall arithmetic required to achieve very good accuracy of the results. These simplifications yield a relatively high speed performance as well as relatively low memory requirements.

Understanding the Lookup Table Based ATAN2 Implementation

Lookup Table Size and Accuracy

Two important design considerations of a lookup table are its size and its accuracy. It is not possible to create a table for every possible y/x input value. It is also not possible to be perfectly accurate due to the quantization of the lookup table values.

As a compromise, the `atan2` method of the Fixed-Point Toolbox `fi` object uses an 8-bit lookup table as part of its implementation. An 8-bit table is only 256 elements long, so it is small and efficient. Eight bits also corresponds to the size of a byte or a word on many platforms. Used in conjunction with linear interpolation, and 16-bit output (lookup table value) precision, an 8-bit-addressable lookup table provides very good accuracy as well as performance.

Overview of Algorithm Implementation

To better understand the Fixed-Point Toolbox implementation, first consider the symmetry of the four-quadrant `atan2(y,x)` function. If you always compute the arctangent in the first-octant of the x-y space (i.e., between angles 0 and $\pi/4$ radians), then you can perform octant correction on the resulting angle for any y and x values.

As part of the pre-processing portion, the signs and relative magnitudes of y and x are considered, and a division is performed. Based on the signs and magnitudes of y and x, only one of the following values is computed: y/x , x/y , $-y/x$, $-x/y$, $-y/-x$, $-x/-y$. The unsigned result that is guaranteed to be non-negative and purely fractional is computed, based on the a priori knowledge of the signs and magnitudes of y and x. An unsigned 16-bit fractional fixed-point type is used for this value.

The 8 most significant bits (MSBs) of the stored unsigned integer representation of the purely-fractional unsigned fixed-point result is then used to directly index an 8-bit (length-256) lookup table value containing angle values between 0 and $\pi/4$ radians. Two table lookups are performed, one at the computed table index location `lutValBelow`, and one at the next index location `lutValAbove`:

```
idxUint8MSBs = uint8(bitsliceget(idxUFX16, 16, 9));
zeroBasedIdx = int16(idxUint8MSBs);
lutValBelow  = FI_ATAN_LUT(zeroBasedIdx + 1);
lutValAbove  = FI_ATAN_LUT(zeroBasedIdx + 2);
```

The remaining 8 least significant bits (LSBs) of `idxUFX16` are used to interpolate between these two table values. The LSB values are treated as a normalized scaling factor with 8-bit fractional data type `rFracNT`:

```
rFracNT      = numerictype(0,8,8); % fractional remainder data type
idxFrac8LSBs = reinterpretcast(bitsliceget(idxUFX16,8,1), rFracNT);
rFraction    = idxFrac8LSBs;
```

The two lookup table values, with the remainder (`rFraction`) value, are used to perform a simple nearest-neighbor linear interpolation. A real multiply is used to determine the weighted difference between the two points. This results in a simple calculation (equivalent to one product and two sums) to obtain the interpolated fixed-point result:

```
temp = rFraction * (lutValAbove - lutValBelow);
rslt = lutValBelow + temp;
```

Finally, based on the original signs and relative magnitudes of `y` and `x`, the output result is formed using simple octant-correction logic and arithmetic. The first-octant $[0, \pi/4]$ angle value results are added or subtracted with constants to form the octant-corrected angle outputs.

Computing Fixed-point Argtangent Using `ATAN2`

You can call the `atan2` function directly using fixed-point or floating-point inputs. The lookup table based algorithm is used for the fixed-point `atan2` implementation:

```
zFxpLUT = atan2(inYfix,inXfix);
```

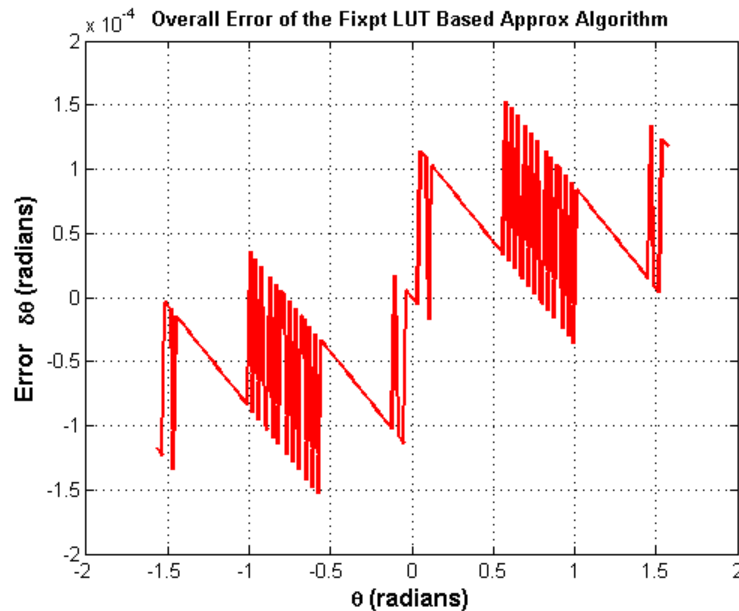
Calculate the Overall Error

You can compute the overall error by comparing the fixed-point lookup table based approximation to the builtin `atan2` function. The ideal reference output is `zfltRef`.

```
lut_err = bsxfun(@minus, zfltRef, double(zFxpLUT));
max_lutErr_real_world_value = max(abs(lut_err'));
max_lutErr_bits = log2(max_lutErr_real_world_value);
fprintf('Overall error in bits: %g\n', max_lutErr_bits);
```

```
Overall error in bits: -12.6743
```

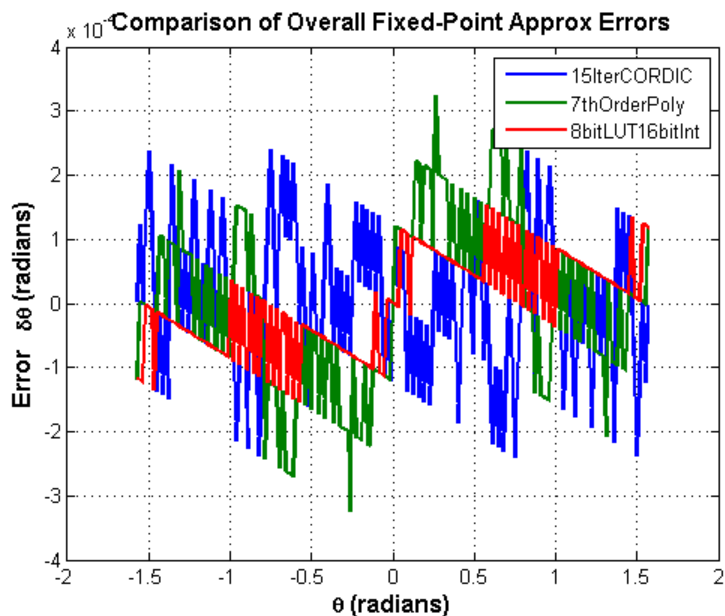
```
figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, lut_err)
```



Comparison of Overall Error Between the Fixed-Point Implementations

As was done previously, you can compute the overall error by comparing the fixed-point approximation(s) to the builtin `atan2` function. The ideal reference output is `zfltRef`.

```
zfixCDC15      = cordicatan2(inYfix, inXfix, 15);
cordic_15I_err = bsxfun(@minus, zfltRef, double(zfixCDC15));
poly_7p_err    = bsxfun(@minus, zfltRef, double(zfix7p));
figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, cordic_15I_err, poly_7p_err, lut
```



Comparing the Costs of the Fixed-Point Approximation Algorithms

The fixed-point CORDIC algorithm requires the following operations:

- 1 table lookup **per iteration**
- 2 shifts **per iteration**
- 3 additions **per iteration**

The N-th order fixed-point Chebyshev polynomial approximation algorithm requires the following operations:

- 1 division
- (N+1) multiplications
- (N-1)/2 additions

The simplified single lookup table algorithm with nearest-neighbor linear interpolation requires the following operations:

- 1 division
- 2 table lookups
- 1 multiplication
- 2 additions

In real world applications, selecting an algorithm for the fixed-point arctangent calculation typically depends on the required accuracy, cost and hardware constraints.

```
close all; % close all figure windows
```

References

- 1** Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.
- 2** Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Compute Sine and Cosine Using CORDIC Rotation Kernel

This example shows how to compute sine and cosine using a CORDIC rotation kernel in MATLAB. CORDIC-based algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Introduction

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm (see [1,2]) is one of the most hardware efficient algorithms because it only requires iterative shift-add operations. The CORDIC algorithm eliminates the need for explicit multipliers, and is suitable for calculating a variety of functions, such as sine, cosine, arcsine, arccosine, arctangent, vector magnitude, divide, square root, hyperbolic and logarithmic functions.

The fixed-point CORDIC algorithm requires the following operations:

- 1 table lookup **per iteration**
- 2 shifts **per iteration**
- 3 additions **per iteration**

CORDIC Kernel Algorithm Using the Rotation Computation Mode

You can use a CORDIC rotation computing mode algorithm to calculate sine and cosine simultaneously, compute polar-to-cartesian conversions, and for other operations. In the rotation mode, the vector magnitude and an angle of rotation are known and the coordinate (X-Y) components are computed after rotation.

The CORDIC rotation mode algorithm begins by initializing an angle accumulator with the desired rotation angle. Next, the rotation decision at each CORDIC iteration is done in a way that decreases the magnitude of the residual angle accumulator. The rotation decision is based on the sign of the residual angle in the angle accumulator after each iteration.

In rotation mode, the CORDIC equations are:

$$z_{i+1} = z_i - d_i * \text{atan}(2^{-i})$$

$$x_{i+1} = x_i - y_i * d_i * 2^{-i}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{-i}$$

where $d_i = -1$ if $z_i < 0$, and $+1$ otherwise;

$i = 0, 1, \dots, N-1$, and N is the total number of iterations.

This provides the following result as N approaches $+\infty$:

$$z_N = 0$$

$$x_N = A_N(x_0 \cos z_0 - y_0 \sin z_0)$$

$$y_N = A_N(y_0 \cos z_0 + x_0 \sin z_0)$$

Where:

$$A_N = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$$

Typically N is chosen to be a large-enough constant value. Thus, A_N may be pre-computed.

In rotation mode, the CORDIC algorithm is limited to rotation angles between $-\pi/2$ and $\pi/2$. To support angles outside of that range, quadrant correction is often used.

Efficient MATLAB Implementation of a CORDIC Rotation Kernel Algorithm

A MATLAB code implementation example of the CORDIC Rotation Kernel algorithm follows (for the case of scalar x , y , and z). This same code can be used for both fixed-point and floating-point operation.

CORDIC Rotation Kernel

```
function [x, y, z] = cordic_rotation_kernel(x, y, z, inPLUT, n)
```

```
% Perform CORDIC rotation kernel algorithm for N iterations.
xtmp = x;
ytmp = y;
for idx = 1:n
    if z < 0
        z(:) = accumpos(z, inpLUT(idx));
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
    else
        z(:) = accumneg(z, inpLUT(idx));
        x(:) = accumneg(x, ytmp);
        y(:) = accumpos(y, xtmp);
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end
```

CORDIC-Based Sine and Cosine Computation Using Normalized Inputs

Sine and Cosine Computation Using the CORDIC Rotation Kernel

The judicious choice of initial values allows the CORDIC kernel rotation mode algorithm to directly compute both sine and cosine simultaneously.

First, the following initialization steps are performed:

- The angle input look-up table `inpLUT` is set to $\text{atan}(2^{-i})$ for $i = 0:N-1$.
- z_0 is set to the θ input argument value.
- x_0 is set to $1/A_N$.
- y_0 is set to zero.

After N iterations, these initial values lead to the following outputs as N approaches $+\infty$:

- $x_N \approx \cos(\theta)$
- $y_N \approx \sin(\theta)$

Other rotation-kernel-based function approximations are possible via pre- and post-processing and using other initial conditions (see [1,2]).

The CORDIC algorithm is usually run through a specified (constant) number of iterations since ending the CORDIC iterations early would break pipelined code, and the CORDIC gain A_n would not be constant because n would vary.

For very large values of n , the CORDIC algorithm is guaranteed to converge, but not always monotonically. You can typically achieve greater accuracy by increasing the total number of iterations.

Example

Suppose that you have a rotation angle sensor (e.g. in a servo motor) that uses formatted integer values to represent measured angles of rotation. Also suppose that you have a 16-bit integer arithmetic unit that can perform add, subtract, shift, and memory operations. With such a device, you could implement the CORDIC rotation kernel to efficiently compute cosine and sine (equivalently, cartesian X and Y coordinates) from the sensor angle values, without the use of multiplies or large lookup tables.

```
sumWL = 16; % CORDIC sum word length
thNorm = -1.0:(2^-8):1.0; % Normalized [-1.0, 1.0] angle values
theta = fi(thNorm, 1, sumWL); % Fixed-point angle values (best precision)

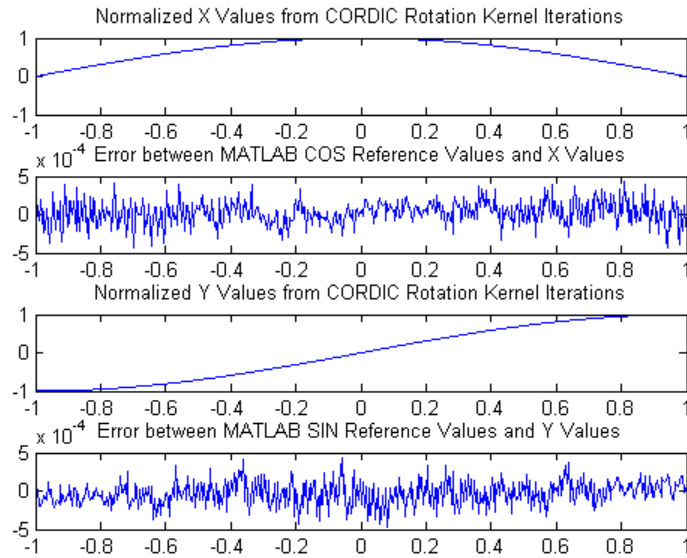
z_NT = numerictype(theta); % Data type for Z
xyNT = numerictype(1, sumWL, sumWL-2); % Data type for X-Y
x_out = fi(zeros(size(theta)), xyNT); % X array pre-allocation
y_out = fi(zeros(size(theta)), xyNT); % Y array pre-allocation
z_out = fi(zeros(size(theta)), z_NT); % Z array pre-allocation

nitters = 13; % Number of CORDIC iterations
inpLUT = fi(atan(2.^(-(0:(nitters-1))))).*(2/pi), z_NT); % Normalized
AnGain = prod(sqrt(1+2.^(-2*(0:(nitters-1))))); % CORDIC gain
inv_An = 1 / AnGain; % 1/A_n inverse of CORDIC gain

for idx = 1:length(theta)
    % CORDIC rotation kernel iterations
    [x_out(idx), y_out(idx), z_out(idx)] = ...
        fidemo.cordic_rotation_kernel(...
```

```
        fi(inv_An, xyNT), fi(0, xyNT), theta(idx), inpLUT, niters);
end

% Plot the CORDIC-approximated sine and cosine values
figure;
subplot(411);
plot(thNorm, x_out);
axis([-1 1 -1 1]);
title('Normalized X Values from CORDIC Rotation Kernel Iterations');
subplot(412);
thetaRadians = pi/2 .* thNorm; % real-world range [-pi/2 pi/2] angle values
plot(thNorm, cos(thetaRadians) - double(x_out));
title('Error between MATLAB COS Reference Values and X Values');
subplot(413);
plot(thNorm, y_out);
axis([-1 1 -1 1]);
title('Normalized Y Values from CORDIC Rotation Kernel Iterations');
subplot(414);
plot(thNorm, sin(thetaRadians) - double(y_out));
title('Error between MATLAB SIN Reference Values and Y Values');
```



References

- 1 Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.
- 2 Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Perform QR Factorization Using CORDIC

This example shows how to write MATLAB code that works for both floating-point and fixed-point data types. The algorithm used in this example is the QR factorization implemented via CORDIC (Coordinate Rotation Digital Computer).

A good way to write an algorithm intended for a fixed-point target is to write it in MATLAB using builtin floating-point types so you can verify that the algorithm works. When you refine the algorithm to work with fixed-point types, then the best thing to do is to write it so that the same code continues working with floating-point. That way, when you are debugging, then you can switch the inputs back and forth between floating-point and fixed-point types to determine if a difference in behavior is because of fixed-point effects such as overflow and quantization versus an algorithmic difference. Even if the algorithm is not well suited for a floating-point target (as is the case of using CORDIC in the following example), it is still advantageous to have your MATLAB code work with floating-point for debugging purposes.

In contrast, you may have a completely different strategy if your target is floating point. For example, the QR algorithm is often done in floating-point with Householder transformations and row or column pivoting. But in fixed-point it is often more efficient to use CORDIC to apply Givens rotations with no pivoting.

This example addresses the first case, where your target is fixed-point, and you want an algorithm that is independent of data type because it is easier to develop and debug.

In this example you will learn various coding methods that can be applied across systems. The significant design patterns used in this example are the following:

- **Data Type Independence:** the algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.
- **Overflow Prevention:** method to guarantee not to overflow. This demonstrates how to prevent overflows in fixed-point.

- Solving Systems of Equations: method to use computational efficiency. Narrow your code scope by isolating what you need to define.

The main part in this example is an implementation of the QR factorization in fixed-point arithmetic using CORDIC for the Givens rotations. The algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.

The QR factorization of M-by-N matrix A produces an M-by-N upper triangular matrix R and an M-by-M orthogonal matrix Q such that $A = Q \cdot R$. A matrix is upper triangular if it has all zeros below the diagonal. An M-by-M matrix Q is orthogonal if $Q' \cdot Q = \text{eye}(M)$, the identity matrix.

The QR factorization is widely used in least-squares problems, such as the recursive least squares (RLS) algorithm used in adaptive filters.

The CORDIC algorithm is attractive for computing the QR algorithm in fixed-point because you can apply orthogonal Givens rotations with CORDIC using only shift and add operations.

Setup

So this example does not change your preferences or settings, we store the original state here, and restore them at the end.

```
originalFormat = get(0, 'format'); format short
originalFipref = fipref;          reset(fipref);
originalGlobalFimath = fimath;    resetglobalfimath;
```

Defining the CORDIC QR Algorithm

The CORDIC QR algorithm is given in the following MATLAB function, where A is an M-by-N real matrix, and niter is the number of CORDIC iterations. Output Q is an M-by-M orthogonal matrix, and R is an M-by-N upper-triangular matrix such that $Q \cdot R = A$.

```
function [Q,R] = cordicqr(A,niter)
    Kn = inverse_cordic_growth_constant(niter);
    [m,n] = size(A);
```

```

R = A;
Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
Q(:) = eye(m); % Initialize Q
for j=1:n
    for i=j+1:m
        [R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
            cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
    end
end
end

```

This function was written to be independent of data type. It works equally well with builtin floating-point types (double and single) and with the fixed-point `fi` object.

One of the trickiest aspects of writing data-type independent code is to specify data type and size for a new variable. In order to preserve data types without having to explicitly specify them, the output `R` was set to be the same as input `A`, like this:

```
R = A;
```

In addition to being data-type independent, this function was written in such a way that MATLAB Coder™ will be able to generate efficient C code from it. In MATLAB, you most often declare and initialize a variable in one step, like this:

```
Q = eye(m)
```

However, `Q=eye(m)` would always produce `Q` as a double-precision floating point variable. If `A` is fixed-point, then we want `Q` to be fixed-point; if `A` is single, then we want `Q` to be single; etc.

Hence, you need to declare the type and size of `Q` in one step, and then initialize it in a second step. This gives MATLAB Coder the information it needs to create an efficient C program with the correct types and sizes. In the finished code you initialize output `Q` to be an `M`-by-`M` identity matrix and the same data type as `A`, like this:

```

Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
Q(:) = eye(m); % Initialize Q

```

The `coder.nullcopy` function declares the size and type of `Q` without initializing it. The expansion of the first column of `A` with `repmat` won't appear in code generated by MATLAB; it is only used to specify the size. The `repmat` function was used instead of `A(:,1:m)` because `A` may have more rows than columns, which will be the case in a least-squares problem. You have to be sure to always assign values to every element of an array when you declare it with `coder.nullcopy`, because if you don't then you will have uninitialized memory.

You will notice this pattern of assignment again and again. This is another key enabler of data-type independent code.

The heart of this function is applying orthogonal Givens rotations in-place to the rows of `R` to zero out sub-diagonal elements, thus forming an upper-triangular matrix. The same rotations are applied in-place to the columns of the identity matrix, thus forming orthogonal `Q`. The Givens rotations are applied using the `cordicgivens` function, as defined in the next section. The rows of `R` and columns of `Q` are used as both input and output to the `cordicgivens` function so that the computation is done in-place, overwriting `R` and `Q`.

```
[R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
    cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
```

Defining the CORDIC Givens Rotation

The `cordicgivens` function applies a Givens rotation by performing CORDIC iterations to rows `x=R(j,j:end)`, `y=R(i,j:end)` around the angle defined by `x(1)=R(j,j)` and `y(1)=R(i,j)` where `i>j`, thus zeroing out `R(i,j)`. The same rotation is applied to columns `u = Q(:,j)` and `v = Q(:,i)`, thus forming the orthogonal matrix `Q`.

```
function [x,y,u,v] = cordicgivens(x,y,u,v,niter,Kn)
    if x(1)<0
        % Compensation for 3rd and 4th quadrants
        x(:) = -x; u(:) = -u;
        y(:) = -y; v(:) = -v;
    end
    for i=0:niter-1
        x0 = x;
```

```
u0 = u;
if y(1)<0
    % Counter-clockwise rotation
    % x and y form R,          u and v form Q
    x(:) = x - bitsra(y, i); u(:) = u - bitsra(v, i);
    y(:) = y + bitsra(x0,i); v(:) = v + bitsra(u0,i);
else
    % Clockwise rotation
    % x and y form R,          u and v form Q
    x(:) = x + bitsra(y, i); u(:) = u + bitsra(v, i);
    y(:) = y - bitsra(x0,i); v(:) = v - bitsra(u0,i);
end
end
% Set y(1) to exactly zero so R will be upper triangular without round off
% showing up in the lower triangle.
y(1) = 0;
% Normalize the CORDIC gain
x(:) = Kn * x; u(:) = Kn * u;
y(:) = Kn * y; v(:) = Kn * v;
end
```

The advantage of using CORDIC in fixed-point over the standard Givens rotation is that CORDIC does not use square root or divide operations. Only bit-shifts, addition, and subtraction are needed in the main loop, and one scalar-vector multiply at the end to normalize the CORDIC gain. Also, CORDIC rotations work well in pipelined architectures.

The bit shifts in each iteration are performed with the bit shift right arithmetic (`bitsra`) function instead of `bitshift`, multiplication by 0.5, or division by 2, because `bitsra`

- generates more efficient embedded code,
- works equally well with positive and negative numbers,
- works equally well with floating-point, fixed-point and integer types, and
- keeps this code independent of data type.

It is worthwhile to note that there is a difference between sub-scripted assignment (subsasgn) into a variable `a(:) = b` versus overwriting a variable `a = b`. Sub-scripted assignment into a variable like this

```
x(:) = x + bitsra(y, i);
```

always preserves the type of the left-hand-side argument `x`. This is the recommended programming style in fixed-point. For example fixed-point types often grow their word length in a sum, which is governed by the `SumMode` property of the `fimath` object, so that the right-hand-side `x + bitsra(y,i)` can have a different data type than `x`.

If, instead, you overwrite the left-hand-side like this

```
x = x + bitsra(y, i);
```

then the left-hand-side `x` takes on the type of the right-hand-side sum. This programming style leads to changing the data type of `x` in fixed-point code, and is discouraged.

Defining the Inverse CORDIC Growth Constant

This function returns the inverse of the CORDIC growth factor after `niter` iterations. It is needed because CORDIC rotations grow the values by a factor of approximately 1.6468, depending on the number of iterations, so the gain is normalized in the last step of `cordicgivens` by a multiplication by the inverse $K_n = 1/1.6468 = 0.60725$.

```
function Kn = inverse_cordic_growth_constant(niter)
    Kn = 1/prod(sqrt(1+2.^(-2*(0:double(niter)-1))));
end
```

Exploring CORDIC Growth as a Function of Number of Iterations

The function for CORDIC growth is defined as

```
growth = prod(sqrt(1+2.^(-2*(0:double(niter)-1))));
```

and the inverse is

```
inverse_growth = 1 ./ growth
```

Growth is a function of the number of iterations `niter`, and quickly converges to approximately 1.6468, and the inverse converges to approximately 0.60725. You can see in the following table that the difference from one iteration to the next ceases to change after 27 iterations. This is because the calculation hit the limit of precision in double floating-point at 27 iterations.

niter	growth	diff(growth)	1./growth	diff(1./growth)
0	1.0000000000000000	0	1.0000000000000000	
1	1.414213562373095	0.414213562373095	0.707106781186547	-0.292893
2	1.581138830084190	0.166925267711095	0.632455532033676	-0.074651
3	1.629800601300662	0.048661771216473	0.613571991077896	-0.018883
4	1.642484065752237	0.012683464451575	0.608833912517752	-0.004738
5	1.645688915757255	0.003204850005018	0.607648256256168	-0.001185
6	1.646492278712479	0.000803362955224	0.607351770141296	-0.000296
7	1.646693254273644	0.000200975561165	0.607277644093526	-0.000074
8	1.646743506596901	0.000050252323257	0.607259112298893	-0.000018
9	1.646756070204878	0.000012563607978	0.607254479332562	-0.000004
10	1.646759211139822	0.000003140934944	0.607253321089875	-0.000001
11	1.646759996375617	0.000000785235795	0.607253031529134	-0.000000
12	1.646760192684695	0.000000196309077	0.607252959138945	-0.000000
13	1.646760241761972	0.000000049077277	0.607252941041397	-0.000000
14	1.646760254031292	0.000000012269320	0.607252936517010	-0.000000
15	1.646760257098622	0.000000003067330	0.607252935385914	-0.000000
16	1.646760257865455	0.000000000766833	0.607252935103139	-0.000000
17	1.646760258057163	0.000000000191708	0.607252935032446	-0.000000
18	1.646760258105090	0.000000000047927	0.607252935014772	-0.000000
19	1.646760258117072	0.000000000011982	0.607252935010354	-0.000000
20	1.646760258120067	0.000000000002995	0.607252935009249	-0.000000
21	1.646760258120816	0.000000000000749	0.607252935008973	-0.000000
22	1.646760258121003	0.000000000000187	0.607252935008904	-0.000000
23	1.646760258121050	0.000000000000047	0.607252935008887	-0.000000
24	1.646760258121062	0.000000000000012	0.607252935008883	-0.000000
25	1.646760258121065	0.000000000000003	0.607252935008882	-0.000000
26	1.646760258121065	0.000000000000001	0.607252935008881	-0.000000
27	1.646760258121065	0	0.607252935008881	
28	1.646760258121065	0	0.607252935008881	
29	1.646760258121065	0	0.607252935008881	
30	1.646760258121065	0	0.607252935008881	
31	1.646760258121065	0	0.607252935008881	

```
32      1.646760258121065      0      0.607252935008881
```

Comparing CORDIC to the Standard Givens Rotation

The `cordicgivens` function is numerically equivalent to the following standard Givens rotation algorithm from Golub & Van Loan, *Matrix Computations*. In the `cordicqr` function, if you replace the call to `cordicgivens` with a call to `givensrotation`, then you will have the standard Givens QR algorithm.

```
function [x,y,u,v] = givensrotation(x,y,u,v)
    a = x(1); b = y(1);
    if b==0
        % No rotation necessary.  c = 1; s = 0;
        return;
    else
        if abs(b) > abs(a)
            t = -a/b; s = 1/sqrt(1+t^2); c = s*t;
        else
            t = -b/a; c = 1/sqrt(1+t^2); s = c*t;
        end
    end
    x0 = x;          u0 = u;
    % x and y form R,  u and v form Q
    x(:) = c*x0 - s*y; u(:) = c*u0 - s*v;
    y(:) = s*x0 + c*y; v(:) = s*u0 + c*v;
end
```

The `givensrotation` function uses division and square root, which are expensive in fixed-point, but good for floating-point algorithms.

Example of CORDIC Rotations

Here is a 3-by-3 example that follows the CORDIC rotations through each step of the algorithm. The algorithm uses orthogonal rotations to zero out the subdiagonal elements of R using the diagonal elements as pivots. The same rotations are applied to the identity matrix, thus producing orthogonal Q such that $Q^*R = A$.

Let A be a random 3-by-3 matrix, and initialize $R = A$, and $Q = \text{eye}(3)$.

$$R = A = \begin{bmatrix} -0.8201 & 0.3573 & -0.0100 \\ -0.7766 & -0.0096 & -0.7048 \\ -0.7274 & -0.6206 & -0.8901 \end{bmatrix}$$

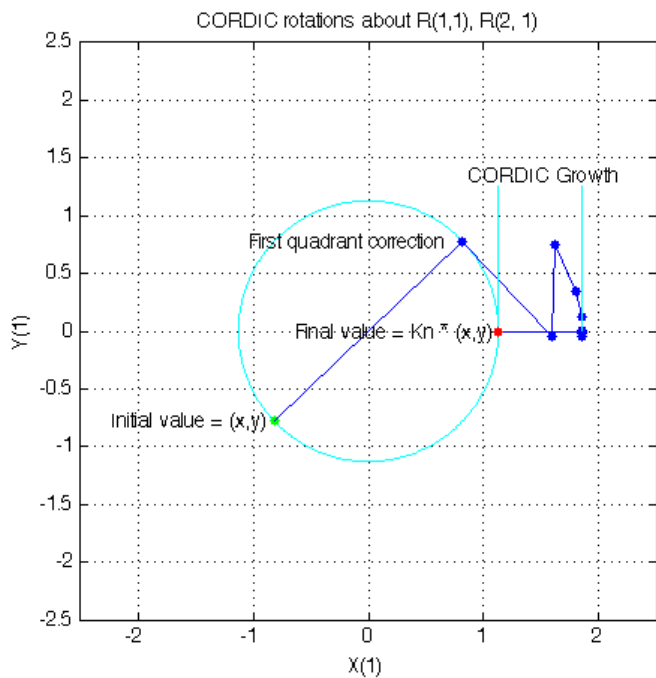
$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The first rotation is about the first and second row of R and the first and second column of Q. Element R(1,1) is the pivot and R(2,1) rotates to 0.

R before the first rotation				R after the first rotation		
x	[-0.8201 0.3573 -0.0100]	->	x	[1.1294 -0.2528 0.4918]		
y	[-0.7766 -0.0096 -0.7048]	->	y	[0 0.2527 0.5049]		
	-0.7274 -0.6206 -0.8901			-0.7274 -0.6206 -0.8901		

Q before the first rotation				Q after the first rotation		
u	v			u	v	
[1]	[0]	0	->	[-0.7261]	[0.6876]	0
[0]	[1]	0		[-0.6876]	[-0.7261]	0
[0]	[0]	1		[0]	[0]	1

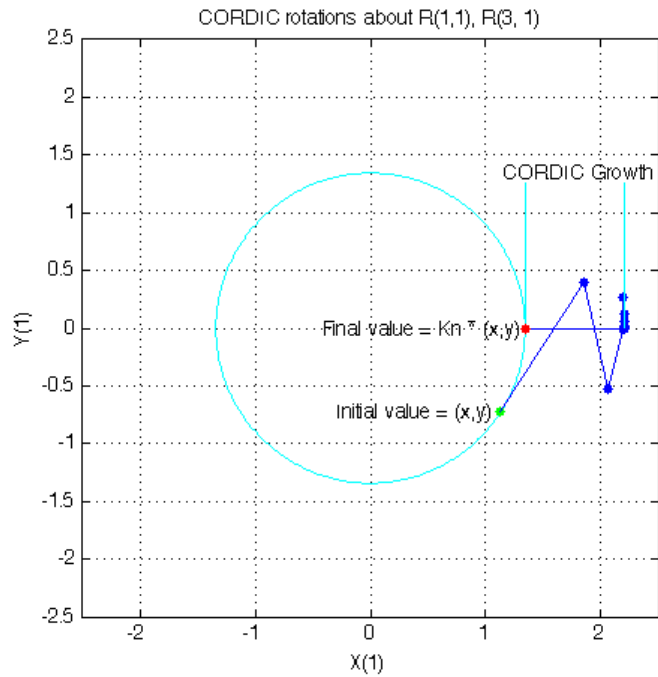
In the following plot, you can see the growth in x in each of the CORDIC iterations. The growth is factored out at the last step by multiplying it by $K_n = 0.60725$. You can see that $y(1)$ iterates to 0. Initially, the point $[x(1), y(1)]$ is in the third quadrant, and is reflected into the first quadrant before the start of the CORDIC iterations.



The second rotation is about the first and third row of R and the first and third column of Q. Element $R(1, 1)$ is the pivot and $R(3, 1)$ rotates to 0.

R before the second rotation				R after the second rotation					
x	[1.1294	-0.2528	0.4918]	->	x	[1.3434	0.1235	0.8954]	
	0	0.2527	0.5049			0	0.2527	0.5049	
y	[-0.7274]	-0.6206	-0.8901	->	y	[0	-0.6586	-0.4820]

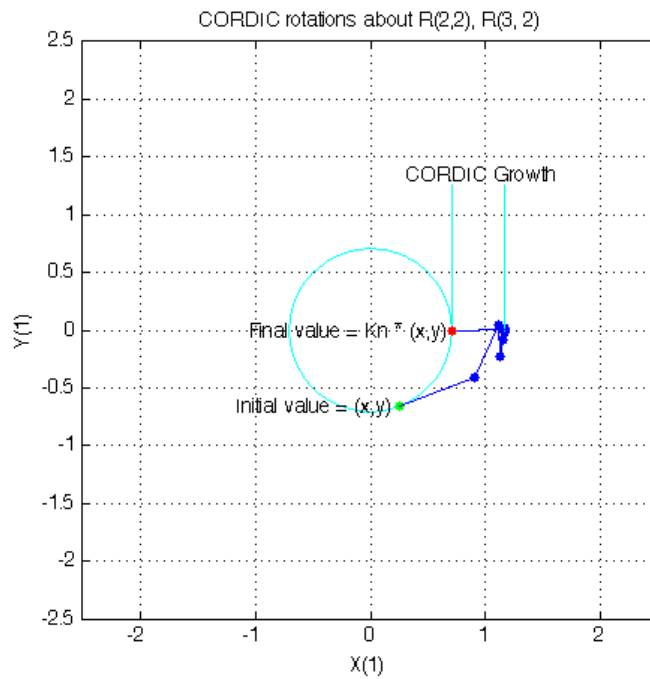
Q before the second rotation				Q after the second rotation			
u		v		u		v	
[-0.7261]	0.6876	[0]		[-0.6105]	0.6876	[-0.3932]	
[-0.6876]	-0.7261	[0]	->	[-0.5781]	-0.7261	[-0.3723]	
[0]	0	[1]		[-0.5415]	0	[0.8407]



The third rotation is about the second and third row of R and the second and third column of Q . Element $R(2,2)$ is the pivot and $R(3,2)$ rotates to 0.

R before the third rotation					R after the third rotation				
	1.3434	0.1235	0.8954			1.3434	0.1235	0.8954	
x	0	[0.2527	0.5049]	->	x	0	[0.7054	0.6308]	
y	0	[-0.6586	-0.4820]	->	y	0	[0	0.2987]	

Q before the third rotation					Q after the third rotation				
		u	v			u	v		
-0.6105	[0.6876]	[-0.3932]			-0.6105	[0.6134]	[0.5011]		
-0.5781	[-0.7261]	[-0.3723]	->		-0.5781	[0.0875]	[-0.8113]		
-0.5415	[0	[0.8407]			-0.5415	[-0.7849]	[0.3011]		



This completes the QR factorization. R is upper triangular, and Q is orthogonal.

$$R = \begin{bmatrix} 1.3434 & 0.1235 & 0.8954 \\ 0 & 0.7054 & 0.6308 \\ 0 & 0 & 0.2987 \end{bmatrix}$$

$$Q = \begin{bmatrix} -0.6105 & 0.6134 & 0.5011 \\ -0.5781 & 0.0875 & -0.8113 \\ -0.5415 & -0.7849 & 0.3011 \end{bmatrix}$$

You can verify that Q is within roundoff error of being orthogonal by multiplying and seeing that it is close to the identity matrix.

$$Q*Q' = \begin{bmatrix} 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.0000 & 0 \\ 0.0000 & 0 & 1.0000 \end{bmatrix}$$

$$Q'*Q = \begin{bmatrix} 1.0000 & 0.0000 & -0.0000 \\ 0.0000 & 1.0000 & -0.0000 \\ -0.0000 & -0.0000 & 1.0000 \end{bmatrix}$$

You can see the error difference by subtracting the identity matrix.

$$Q*Q' - \text{eye}(\text{size}(Q)) = \begin{bmatrix} 0 & 2.7756\text{e-}16 & 3.0531\text{e-}16 \\ 2.7756\text{e-}16 & 4.4409\text{e-}16 & 0 \\ 3.0531\text{e-}16 & 0 & 6.6613\text{e-}16 \end{bmatrix}$$

You can verify that $Q*R$ is close to A by subtracting to see the error difference.

$$Q*R - A = \begin{bmatrix} -3.7802\text{e-}11 & -7.2325\text{e-}13 & -2.7756\text{e-}17 \\ -3.0512\text{e-}10 & 1.1708\text{e-}12 & -4.4409\text{e-}16 \\ 3.6836\text{e-}10 & -4.3487\text{e-}13 & -7.7716\text{e-}16 \end{bmatrix}$$

Determining the Optimal Output Type of Q for Fixed Word Length

Since Q is orthogonal, you know that all of its values are between -1 and +1. In floating-point, there is no decision about the type of Q : it should be the same floating-point type as A . However, in fixed-point, you can do better than making Q have the identical fixed-point type as A . For example, if A has word length 16 and fraction length 8, and if we make Q also have word length 16 and fraction length 8, then you force Q to be less accurate than it could be and waste the upper half of the fixed-point range.

The best type for Q is to make it have full range of its possible outputs, plus accommodate the 1.6468 CORDIC growth factor in intermediate calculations. Therefore, assuming that the word length of Q is the same as the word length of input A , then the best fraction length for Q is 2 bits less than the word length (one bit for 1.6468 and one bit for the sign).

Hence, our initialization of Q in `cordicqr` can be improved like this.

```
if isfi(A) && (isfixed(A) || isscaledouble(A))
    Q = fi(one*eye(m), get(A,'NumericType'), ...
        'FractionLength',get(A,'WordLength')-2);
```

```

else
    Q = coder.nullcopy(repmat(A(:,1),1,m));
    Q(:) = eye(m);
end

```

A slight disadvantage is that this section of code is dependent on data type. However, you gain a major advantage by picking the optimal type for Q , and the main algorithm is still independent of data type. You can do this kind of input parsing in the beginning of a function and leave the main algorithm data-type independent.

Preventing Overflow in Fixed Point R

This section describes how to determine a fixed-point output type for R in order to prevent overflow. In order to pick an output type, you need to know how much the magnitude of the values of R will grow.

Given real matrix A and its QR factorization computed by Givens rotations without pivoting, an upper-bound on the magnitude of the elements of R is the square-root of the number of rows of A times the magnitude of the largest element in A . Furthermore, this growth will never be greater during an intermediate computation. In other words, let $[m,n]=\text{size}(A)$, and $[Q,R]=\text{givensqr}(A)$. Then

$$\max(\text{abs}(R(:))) \leq \sqrt{m} * \max(\text{abs}(A(:))).$$

This is true because the each element of R is formed from orthogonal rotations from its corresponding column in A , so the largest that any element $R(i,j)$ can get is if all of the elements of its corresponding column $A(:,j)$ were rotated to a single value. In other words, the largest possible value will be bounded by the 2-norm of $A(:,j)$. Since the 2-norm of $A(:,j)$ is equal to the square-root of the sum of the squares of the m elements, and each element is less-than-or-equal-to the largest element of A , then

$$\text{norm}(A(:,j)) \leq \sqrt{m} * \max(\text{abs}(A(:))).$$

That is, for all j

$$\begin{aligned}
 \text{norm}(A(:,j)) &= \sqrt{A(1,j)^2 + A(2,j)^2 + \dots + A(m,j)^2} \\
 &\leq \sqrt{m * \max(\text{abs}(A(:)))^2} \\
 &= \sqrt{m} * \max(\text{abs}(A(:))).
 \end{aligned}$$

and so for all i,j

$$\text{abs}(R(i,j)) \leq \text{norm}(A(:,j)) \leq \sqrt{m} * \max(\text{abs}(A(:))).$$

Hence, it is also true for the largest element of R

$$\max(\text{abs}(R(:))) \leq \sqrt{m} * \max(\text{abs}(A(:))).$$

This becomes useful in fixed-point where the elements of A are often very close to the maximum value attainable by the data type, so we can set a tight upper bound without knowing the values of A . This is important because we want to set an output type for R with a minimum number of bits, only knowing the upper bound of the data type of A . You can use `fi` method `upperbound` to get this value.

Therefore, for all i,j

$$\text{abs}(R(i,j)) \leq \sqrt{m} * \text{upperbound}(A)$$

Note that $\sqrt{m} * \text{upperbound}(A)$ is also an upper bound for the elements of A :

$$\text{abs}(A(i,j)) \leq \text{upperbound}(A) \leq \sqrt{m} * \text{upperbound}(A)$$

Therefore, when picking fixed-point data types, $\sqrt{m} * \text{upperbound}(A)$ is an upper bound that will work for both A and R .

Attaining the maximum is easy and common. The maximum will occur when all elements get rotated into a single element, like the following matrix with orthogonal columns:

$$A = \begin{bmatrix} 7 & -7 & 7 & 7 \\ 7 & 7 & -7 & 7 \\ 7 & -7 & -7 & -7 \\ 7 & 7 & 7 & -7 \end{bmatrix};$$

Its maximum value is 7 and its number of rows is $m=4$, so we expect that the maximum value in R will be bounded by $\max(\text{abs}(A(:))) * \sqrt{m} = 7 * \sqrt{4} = 14$. Since A in this example is orthogonal, each column gets rotated to the max value on the diagonal.

$$\text{niter} = 52;$$

```
[Q,R] = cordicqr(A,niter)
```

Q =

```
0.5000    -0.5000    0.5000    0.5000
0.5000     0.5000   -0.5000    0.5000
0.5000   -0.5000   -0.5000   -0.5000
0.5000     0.5000    0.5000   -0.5000
```

R =

```
14.0000    0.0000   -0.0000   -0.0000
      0   14.0000   -0.0000    0.0000
      0      0   14.0000    0.0000
      0      0      0   14.0000
```

Another simple example of attaining maximum growth is a matrix that has all identical elements, like a matrix of all ones. A matrix of ones will get rotated into $1 \cdot \sqrt{m}$ in the first row and zeros elsewhere. For example, this 9-by-5 matrix will have all $1 \cdot \sqrt{9}=3$ in the first row of R.

```
m = 9; n = 5;
A = ones(m,n)
niter = 52;
[Q,R] = cordicqr(A,niter)
```

A =

```
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
```

1 1 1 1 1

Q =

Columns 1 through 7

0.3333	0.5567	-0.6784	0.3035	-0.1237	0.0503	0.0158
0.3333	0.0296	0.2498	-0.1702	-0.6336	0.1229	-0.3012
0.3333	0.2401	0.0562	-0.3918	0.4927	0.2048	-0.5395
0.3333	0.0003	0.0952	-0.1857	0.2148	0.4923	0.7080
0.3333	0.1138	0.0664	-0.2263	0.1293	-0.8348	0.2510
0.3333	-0.3973	-0.0143	0.3271	0.4132	-0.0354	-0.2165
0.3333	0.1808	0.3538	-0.1012	-0.2195	0	0.0824
0.3333	-0.6500	-0.4688	-0.2380	-0.2400	0	0
0.3333	-0.0740	0.3400	0.6825	-0.0331	0	0

Columns 8 through 9

0.0056	-0.0921
-0.5069	-0.1799
0.0359	0.3122
-0.2351	-0.0175
-0.2001	0.0610
-0.0939	-0.6294
0.7646	-0.2849
0.2300	0.2820
0	0.5485

R =

3.0000	3.0000	3.0000	3.0000	3.0000
0	0.0000	0.0000	0.0000	0.0000
0	0	0.0000	0.0000	0.0000
0	0	0	0.0000	0.0000
0	0	0	0	0.0000
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

0 0 0 0 0

As in the `cordicqr` function, the Givens QR algorithm is often written by overwriting A in-place with R, so being able to cast A into R's data type at the beginning of the algorithm is convenient.

In addition, if you compute the Givens rotations with CORDIC, there is a growth-factor that converges quickly to approximately 1.6468. This growth factor gets normalized out after each Givens rotation, but you need to accommodate it in the intermediate calculations. Therefore, the number of additional bits that are required including the Givens and CORDIC growth are $\log_2(1.6468 * \sqrt{m})$. The additional bits of head-room can be added either by increasing the word length, or decreasing the fraction length.

A benefit of increasing the word length is that it allows for the maximum possible precision for a given word length. A disadvantage is that the optimal word length may not correspond to a native type on your processor (e.g. increasing from 16 to 18 bits), or you may have to increase to the next larger native word size which could be quite large (e.g. increasing from 16 to 32 bits, when you only needed 18).

A benefit of decreasing fraction length is that you can do the computation in-place in the native word size of A. A disadvantage is that you lose precision.

Another option is to pre-scale the input by right-shifting. This is equivalent to decreasing the fraction length, with the additional disadvantage of changing the scaling of your problem. However, this may be an attractive option to you if you prefer to only work in fractional arithmetic or integer arithmetic.

Example of Fixed Point Growth in R

If you have a fixed-point input matrix A, you can define fixed-point output R with the growth defined in the previous section.

Start with a random matrix X.

```
X = [0.0513   -0.2097    0.9492    0.2614
      0.8261    0.6252    0.3071   -0.9415
      1.5270    0.1832    0.1352   -0.1623]
```

```
0.4669    -1.0298    0.5152    -0.1461];
```

Create a fixed-point A from X.

```
A = sfi(X)
```

```
A =
```

```
0.0513    -0.2097    0.9492    0.2614
0.8261     0.6252    0.3071   -0.9415
1.5270     0.1832    0.1352   -0.1623
0.4669    -1.0298    0.5152   -0.1461
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 14
```

```
m = size(A,1)
```

```
m =
```

```
4
```

The growth factor is 1.6468 times the square-root of the number of rows of A. The bit growth is the next integer above the base-2 logarithm of the growth.

```
bit_growth = ceil(log2(cordic_growth_constant * sqrt(m)))
```

```
bit_growth =
```

```
2
```

Initialize R with the same values as A, and a word length increased by the bit growth.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =
```

```
    0.0513    -0.2097    0.9492    0.2614
    0.8261     0.6252    0.3071   -0.9415
    1.5270     0.1832    0.1352   -0.1623
    0.4669    -1.0298    0.5152   -0.1461
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 18
        FractionLength: 14
```

Use R as input and overwrite it.

```
niter = get(R,'WordLength') - 1
[Q,R] = cordicqr(R, niter)
```

```
niter =
```

```
    17
```

```
Q =
```

```
    0.0284   -0.1753    0.9110    0.3723
    0.4594    0.4470    0.3507   -0.6828
    0.8490    0.0320   -0.2169    0.4808
    0.2596   -0.8766   -0.0112   -0.4050
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 18
        FractionLength: 16
```

```
R =
```

```

1.7989    0.1694    0.4166   -0.6008
      0    1.2251   -0.4764   -0.3438
      0      0    0.9375   -0.0555
      0      0      0    0.7214

```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 18
FractionLength: 14

```

Verify that $Q \cdot Q'$ is near the identity matrix.

```
double(Q)*double(Q')
```

```
ans =
```

```

1.0000   -0.0001    0.0000    0.0000
-0.0001    1.0001    0.0000   -0.0000
0.0000    0.0000    1.0000   -0.0000
0.0000   -0.0000   -0.0000    1.0000

```

Verify that $Q \cdot R - A$ is small relative to the precision of A .

```
err = double(Q)*double(R) - double(A)
```

```
err =
```

```

1.0e-03 *
-0.1048   -0.2355    0.1829   -0.2146
 0.3472    0.2949    0.0260   -0.2570
 0.2776   -0.1740   -0.1007    0.0966
 0.0138   -0.1558    0.0417   -0.0362

```

Increasing Precision in R

The previous section showed you how to prevent overflow in R while maintaining the precision of A. If you leave the fraction length of R the same as A, then R cannot have more precision than A, and your precision requirements may be such that the precision of R must be greater.

An extreme example of this is to define a matrix with an integer fixed-point type (i.e. fraction length is zero). Let matrix X have elements that are the full range for signed 8 bit integers, between -128 and +127.

```
X = [-128 -128 -128 127
      -128 127 127 -128
       127 127 127 127
       127 127 -128 -128];
```

Define fixed-point A to be equivalent to an 8-bit integer.

```
A = sfi(X,8,0)
```

```
A =
```

```
-128 -128 -128 127
-128 127 127 -128
 127 127 127 127
 127 127 -128 -128
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 0
```

```
m = size(A,1)
```

```
m =
```

```
4
```

The necessary growth is 1.6468 times the square-root of the number of rows of A.

```
bit_growth = ceil(log2(cordic_growth_constant*sqrt(m)))
```

```
bit_growth =
```

```
2
```

Initialize R with the same values as A, and allow for bit growth like you did in the previous section.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =
```

```
-128 -128 -128 127
-128 127 127 -128
127 127 127 127
127 127 -128 -128
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 10
FractionLength: 0
```

Compute the QR factorization, overwriting R.

```
niter = get(R,'WordLength') - 1;
[Q,R] = cordicqr(R, niter)
```

```
Q =
```

```
-0.5039 -0.2930 -0.4063 -0.6914
-0.5039 0.8750 0.0039 0.0078
0.5000 0.2930 0.3984 -0.7148
```

```
0.4922    0.2930   -0.8203    0.0039
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 10
FractionLength: 8
```

```
R =
```

```
257    126     -1     -1
   0    225    151   -148
   0     0    211    104
   0     0     0   -180
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 10
FractionLength: 0
```

Notice that R is returned with integer values because you left the fraction length of R at 0, the same as the fraction length of A.

The scaling of the least-significant bit (LSB) of A is 1, and you can see that the error is proportional to the LSB.

```
err = double(Q)*double(R)-double(A)
```

```
err =
```

```
-1.5039   -1.4102   -1.4531   -0.9336
-1.5039    6.3828    6.4531   -1.9961
 1.5000    1.9180    0.8086   -0.7500
-0.5078    0.9336   -1.3398   -1.8672
```

You can increase the precision in the QR factorization by increasing the fraction length. In this example, you needed 10 bits for the integer part (8 bits to start with, plus 2 bits growth), so when you increase the fraction length you still need to keep the 10 bits in the integer part. For example, you can

increase the word length to 32 and set the fraction length to 22, which leaves 10 bits in the integer part.

```
R = sfi(A, 32, 22)
```

```
R =
```

```
-128 -128 -128 127
-128 127 127 -128
127 127 127 127
127 127 -128 -128
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 22
```

```
niter = get(R,'WordLength') - 1;
[Q,R] = cordicqr(R, niter)
```

```
Q =
```

```
-0.5020 -0.2913 -0.4088 -0.7043
-0.5020 0.8649 0.0000 0.0000
0.4980 0.2890 0.4056 -0.7099
0.4980 0.2890 -0.8176 0.0000
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 30
```

```
R =
```

```
255.0020 127.0029 0.0039 0.0039
0 220.5476 146.8413 -147.9930
0 0 208.4793 104.2429
0 0 0 -179.6037
```



```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 22

```

Now you can see fractional parts in R, and $Q \cdot R - A$ is small.

```
err = double(Q)*double(R)-double(A)
```

```
err =
```

```
1.0e-05 *
```

```

-0.1234  -0.0014  -0.0845   0.0267
-0.1234   0.2574   0.1260  -0.1094
 0.0720   0.0289  -0.0400  -0.0684
 0.0957   0.0818  -0.1034   0.0095

```

The number of bits you choose for fraction length will depend on the precision requirements for your particular algorithm.

Picking Default Number of Iterations

The number of iterations is dependent on the desired precision, but limited by the word length of A. With each iteration, the values are right-shifted one bit. After the last bit gets shifted off and the value becomes 0, then there is no additional value in continuing to rotate. Hence, the most precision will be attained by choosing `niter` to be one less than the word length.

For floating-point, the number of iterations is bounded by the size of the mantissa. In double, 52 iterations is the most you can do to continue adding to something with the same exponent. In single, it is 23. See the reference page for `eps` for more information about floating-point accuracy.

Thus, we can make our code more usable by not requiring the number of iterations to be input, and assuming that we want the most precision possible by changing `cordicqr` to use this default for `niter`.

```
function [Q,R] = cordicqr(A,varargin)
    if nargin>=2 && ~isempty(varargin{1})
        niter = varargin{1};
    elseif isa(A,'double') || isfi(A) && isdouble(A)
        niter = 52;
    elseif isa(A,'single') || isfi(A) && issingle(A)
        niter = single(23);
    elseif isfi(A)
        niter = int32(get(A,'WordLength') - 1);
    else
        assert(0,'First input must be double, single, or fi.');
```

A disadvantage of doing this is that this makes a section of our code dependent on data type. However, an advantage is that the function is much more convenient to use because you don't have to specify `niter` if you don't want to, and the main algorithm is still data-type independent. Similar to picking an optimal output type for `Q`, you can do this kind of input parsing in the beginning of a function and leave the main algorithm data-type independent.

Here is an example from a previous section, without needing to specify an optimal `niter`.

```
A = [7    -7    7    7
      7    7   -7    7
      7   -7   -7   -7
      7    7    7   -7];
[Q,R] = cordicqr(A)
```

`Q =`

```
0.5000   -0.5000    0.5000    0.5000
0.5000    0.5000   -0.5000    0.5000
0.5000   -0.5000   -0.5000   -0.5000
0.5000    0.5000    0.5000   -0.5000
```

`R =`

```

14.0000    0.0000   -0.0000   -0.0000
      0   14.0000   -0.0000    0.0000
      0      0   14.0000    0.0000
      0      0      0   14.0000

```

Example: QR Factorization Not Unique

When you compare the results from `cordicqr` and the `QR` function in MATLAB, you will notice that the QR factorization is not unique. It is only important that **Q** is orthogonal, **R** is upper triangular, and $\mathbf{Q}^*\mathbf{R} - \mathbf{A}$ is small.

Here is a simple example that shows the difference.

```

m = 3;
A = ones(m)

```

A =

```

1      1      1
1      1      1
1      1      1

```

The built-in `QR` function in MATLAB uses a different algorithm and produces:

```
[Q0,R0] = qr(A)
```

Q0 =

```

-0.5774   -0.5774   -0.5774
-0.5774    0.7887   -0.2113
-0.5774   -0.2113    0.7887

```

R0 =

```

-1.7321   -1.7321   -1.7321

```

```

0      0      0
0      0      0

```

And the `cordicqr` function produces:

```
[Q,R] = cordicqr(A)
```

Q =

```

0.5774    0.7495    0.3240
0.5774   -0.6553    0.4871
0.5774   -0.0942   -0.8110

```

R =

```

1.7321    1.7321    1.7321
0      0.0000    0.0000
0      0      -0.0000

```

Notice that the elements of Q from function `cordicqr` are different from Q0 from built-in QR. However, both results satisfy the requirement that Q is orthogonal:

```
Q0*Q0'
```

ans =

```

1.0000    0.0000    0
0.0000    1.0000    0
0      0      1.0000

```

```
Q*Q'
```

ans =

1.0000	0.0000	0.0000
0.0000	1.0000	-0.0000
0.0000	-0.0000	1.0000

And they both satisfy the requirement that $Q^*R - A$ is small:

$Q_0^*R_0 - A$

ans =

1.0e-15 *		
-0.1110	-0.1110	-0.1110
-0.1110	-0.1110	-0.1110
-0.1110	-0.1110	-0.1110

$Q^*R - A$

ans =

1.0e-15 *		
-0.2220	0.2220	0.2220
0.4441	0	0
0.2220	0.2220	0.2220

Solving Systems of Equations Without Forming Q

Given matrices A and B, you can use the QR factorization to solve for X in the following equation:

$$A^*X = B.$$

If A has more rows than columns, then X will be the least-squares solution. If X and B have more than one column, then several solutions can be computed at the same time. If $A = Q \cdot R$ is the QR factorization of A , then the solution can be computed by back-solving

$$R \cdot X = C$$

where $C = Q' \cdot B$. Instead of forming Q and multiplying to get $C = Q' \cdot B$, it is more efficient to compute C directly. You can compute C directly by applying the rotations to the rows of B instead of to the columns of an identity matrix. The new algorithm is formed by the small modification of initializing $C = B$, and operating along the rows of C instead of the columns of Q .

```
function [R,C] = cordicrc(A,B,niter)
    Kn = inverse_cordic_growth_constant(niter);
    [m,n] = size(A);
    R = A;
    C = B;
    for j=1:n
        for i=j+1:m
            [R(j,j:end),R(i,j:end),C(j,:),C(i,:)] = ...
                cordicgivens(R(j,j:end),R(i,j:end),C(j,:),C(i,:),niter,Kn);
        end
    end
end
```

You can verify the algorithm with this example. Let A be a random 3-by-3 matrix, and B be a random 3-by-2 matrix.

```
A = [ -0.8201    0.3573   -0.0100
      -0.7766   -0.0096   -0.7048
      -0.7274   -0.6206   -0.8901];
```

```
B = [ -0.9286    0.3575
       0.6983    0.5155
       0.8680    0.4863];
```

Compute the QR factorization of A .

```
[Q,R] = cordicqr(A)
```

Q =

-0.6105	0.6133	0.5012
-0.5781	0.0876	-0.8113
-0.5415	-0.7850	0.3011

R =

1.3434	0.1235	0.8955
0	0.7054	0.6309
0	0	0.2988

Compute $C = Q' * B$ directly.

`[R,C] = cordicrc(A,B)`

R =

1.3434	0.1235	0.8955
0	0.7054	0.6309
0	0	0.2988

C =

-0.3068	-0.7795
-1.1897	-0.1173
-0.7706	-0.0926

Subtract, and you will see that the error difference is on the order of roundoff.

$Q' * B - C$

ans =

```
1.0e-15 *
-0.0555    0.3331
      0      0
0.1110    0.2914
```

Now try the example in fixed-point. Declare A and B to be fixed-point types.

```
A = sfi(A)
```

```
A =
```

```
-0.8201    0.3573   -0.0100
-0.7766   -0.0096   -0.7048
-0.7274   -0.6206   -0.8901
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

```
B = sfi(B)
```

```
B =
```

```
-0.9286    0.3575
0.6983    0.5155
0.8680    0.4863
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

The necessary growth is 1.6468 times the square-root of the number of rows of A.


```
bit_growth = ceil(log2(cordic_growth_constant*sqrt(m)))
```

```
bit_growth =
```

```
2
```

Initialize R with the same values as A, and allow for bit growth.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =
```

```
-0.8201    0.3573   -0.0100
-0.7766   -0.0096   -0.7048
-0.7274   -0.6206   -0.8901
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 18
      FractionLength: 15
```

The growth in C is the same as R, so initialize C and allow for bit growth the same way.

```
C = sfi(B, get(B,'WordLength')+bit_growth, get(B,'FractionLength'))
```

```
C =
```

```
-0.9286    0.3575
 0.6983    0.5155
 0.8680    0.4863
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 18
      FractionLength: 15
```

Compute $C = Q'B$ directly, overwriting R and C.

```
[R,C] = cordicrc(R,C)
```

R =

```
1.3435    0.1233    0.8954
         0    0.7055    0.6308
         0         0    0.2988
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 18
      FractionLength: 15
```

C =

```
-0.3068   -0.7796
-1.1898   -0.1175
-0.7706   -0.0926
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 18
      FractionLength: 15
```

An interesting use of this algorithm is that if you initialize B to be the identity matrix, then output argument C is Q' . You may want to use this feature to have more control over the data type of Q. For example,

```
A = [-0.8201    0.3573   -0.0100
      -0.7766   -0.0096   -0.7048
      -0.7274   -0.6206   -0.8901];
B = eye(size(A,1))
```

B =

1	0	0
0	1	0
0	0	1

`[R,C] = cordicrc(A,B)`

R =

1.3434	0.1235	0.8955
0	0.7054	0.6309
0	0	0.2988

C =

-0.6105	-0.5781	-0.5415
0.6133	0.0876	-0.7850
0.5012	-0.8113	0.3011

Then C is orthogonal

`C'*C`

ans =

1.0000	0.0000	0.0000
0.0000	1.0000	-0.0000
0.0000	-0.0000	1.0000

and `R = C*A`

`R - C*A`

ans =

```
1.0e-15 *  
  
0.6661    -0.0139    -0.1110  
0.5551    -0.2220     0.6661  
-0.2220    -0.1110     0.2776
```

Links to the Documentation

Fixed-Point Toolbox™

- `bitsra` Bit shift right arithmetic
- `fi` Construct fixed-point numeric object
- `fimath` Construct `fimath` object
- `fipref` Construct `fipref` object
- `get` Property values of object
- `globalfimath` Configure global `fimath` and return handle object
- `isfi` Determine whether variable is `fi` object
- `sfi` Construct signed fixed-point numeric object
- `upperbound` Upper bound of range of `fi` object
- `fiaccel` Accelerate fixed-point code

MATLAB

- `bitshift` Shift bits specified number of places
- `ceil` Round toward positive infinity
- `double` Convert to double precision floating point
- `eps` Floating-point relative accuracy
- `eye` Identity matrix
- `log2` Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

- `prod` Product of array elements
- `qr` Orthogonal-triangular factorization
- `repmat` Replicate and tile array
- `single` Convert to single precision floating point
- `size` Array dimensions
- `sqrt` Square root
- `subsasgn` Subscripted assignment

Functions Used in this Example

These are the MATLAB functions used in this example.

CORDICQR computes the QR factorization using CORDIC.

- `[Q,R] = cordicqr(A)` chooses the number of CORDIC iterations based on the type of A.
- `[Q,R] = cordicqr(A,niter)` uses `niter` number of CORDIC iterations.

CORDICRC computes R from the QR factorization of A, and also returns $C = Q' * B$ without computing Q.

- `[R,C] = cordicrc(A,B)` chooses the number of CORDIC iterations based on the type of A.
- `[R,C] = cordicrc(A,B,niter)` uses `niter` number of CORDIC iterations.

CORDIC_GROWTH_CONSTANT returns the CORDIC growth constant.

- `cordic_growth = cordic_growth_constant(niter)` returns the CORDIC growth constant as a function of the number of CORDIC iterations, `niter`.

GIVENSQR computes the QR factorization using standard Givens rotations.

- `[Q,R] = givensqr(A)`, where A is M-by-N, produces an M-by-N upper triangular matrix R and an M-by-M orthogonal matrix Q so that $A = Q * R$.

CORDICQR_MAKEPLOTS makes the plots in this example by executing the following from the MATLAB command line.

```
load A_3_by_3_for_cordicqr_demo.mat
niter=32;
[Q,R] = cordicqr_makeplots(A,niter)
```

References

- 1** Ray Andraka, "A survey of CORDIC algorithms for FPGA based computers," 1998, ACM 0-89791-978-5/98/01.
- 2** Anthony J Cox and Nicholas J Higham, "Stability of Householder QR factorization for weighted least squares problems," in Numerical Analysis, 1997, Proceedings of the 17th Dundee Conference, Griffiths DF, Higham DJ, Watson GA (eds). Addison-Wesley, Longman: Harlow, Essex, U.K., 1998; 57-73.
- 3** Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, 3rd ed, Johns Hopkins University Press, 1996, section 5.2.3 Givens QR Methods.
- 4** Daniel V. Rabinkin, William Song, M. Michael Vai, and Huy T. Nguyen, "Adaptive array beamforming with fixed-point arithmetic matrix inversion using Givens rotations," Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE) -- Volume 4474 Advanced Signal Processing Algorithms, Architectures, and Implementations XI, Franklin T. Luk, Editor, November 2001, pp. 294--305.
- 5** Jack E. Volder, "The CORDIC Trigonometric Computing Technique," Institute of Radio Engineers (IRE) Transactions on Electronic Computers, September, 1959, pp. 330-334.
- 6** Musheng Wei and Qiaohua Liu, "On growth factors of the modified Gram-Schmidt algorithm," Numerical Linear Algebra with Applications, Vol. 15, issue 7, September 2008, pp. 621-636.

Cleanup

```
fipref(originalFipref);
globalfimath(originalGlobalFimath);
close all
```

```
set(0, 'format', originalFormat);
```

Compute Square Root Using CORDIC Hyperbolic Kernel

This example shows how to compute square root using a CORDIC hyperbolic kernel algorithm in MATLAB. CORDIC-based algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Introduction

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm (see [1,2]) is one of the most hardware efficient algorithms because it only requires iterative shift-add operations. The CORDIC algorithm eliminates the need for explicit multipliers, and is suitable for calculating a variety of functions, such as sine, cosine, arcsine, arccosine, arctangent, vector magnitude, divide, square root, hyperbolic and logarithmic functions.

The fixed-point CORDIC algorithm requires the following operations:

- 1 table lookup **per iteration**
- 2 shifts **per iteration**
- 3 additions **per iteration**

Note that for hyperbolic CORDIC-based algorithms, such as square root, certain iterations ($i = 4, 7, 10, \dots, 3k+1, \dots$) are repeated to achieve result convergence. There is an additional cost of 3 additions for each of those repeated iterations.

CORDIC Kernel Algorithms Using Hyperbolic Computation Modes

You can use a CORDIC computing mode algorithm to calculate hyperbolic functions, such as hyperbolic trigonometric, square root, log, exp, etc.

In hyperbolic rotation mode, the CORDIC equations are:

$$x_{i+1} = x_i + y_i * d_i * 2^{-i}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{-i}$$

$$z_{i+1} = z_i - d_i * \operatorname{atanh}(2^{-i})$$

where $d_i = -1$ if $z_i < 0$, and $+1$ otherwise;

$i = 0, 1, \dots, N-1$, and N is the total number of iterations.

This mode provides the following result as N approaches $+\infty$:

- $x_N \approx A_N(x_0 \cosh z_0 + y_0 \sinh z_0)$
- $y_N \approx A_N(y_0 \cosh z_0 + x_0 \sinh z_0)$
- $z_N \approx 0$

Where:

$$A_N = \prod_{i=0}^{N-1} \sqrt{1 - 2^{-2i}}$$

Typically N is chosen to be a large-enough constant value. Thus, A_N may be pre-computed.

In hyperbolic vectoring mode, the CORDIC equations are as above, but with $d_i = +1$ if $y_i < 0$, and -1 otherwise;

The vectoring mode provides the following result as N approaches $+\infty$:

- $x_N \approx A_N \sqrt{x_0^2 - y_0^2}$
- $y_N \approx 0$
- $z_N \approx z_0 + \operatorname{atanh}(y_0/x_0)$

Note that the rotations in the hyperbolic coordinate system do not converge. It has been shown, however, that convergence is achieved if certain iterations ($i = 4, 7, 10, \dots, 3k+1, \dots$) are repeated.

Efficient MATLAB Implementation of a CORDIC Hyperbolic Vectoring Algorithm

A MATLAB code implementation example of the CORDIC Hyperbolic Vectoring algorithm follows (for the case of scalar x , y , and z). This same code can be used for both fixed-point and floating-point operation.

CORDIC Hyperbolic Vectoring Kernel

```
function [x, y, z] = cordic_hyperbolic_vectoring_kernel(x, y, z, inpLUT, n)
% Perform CORDIC hyperbolic vectoring kernel algorithm for N iterations.
k = 3; % Used for REPEAT rotations at (idx == 4, 7, 10, ..., 3k+1, ...)
for idx = 1:n
    xtmp = bitsra(x, idx); % multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % multiply by 2^(-idx)

    if y < 0
        x(:) = accumpos(x, ytmp);
        y(:) = accumpos(y, xtmp);
        z(:) = accumneg(z, inpLUT(idx));
    else
        x(:) = accumneg(x, ytmp);
        y(:) = accumneg(y, xtmp);
        z(:) = accumpos(z, inpLUT(idx));
    end

    if k > 0
        k = k-1; % Decrease '3k+1' counter
    else
        k = 3; % Re-start '3k+1' counter and REPEAT rotation
        if y < 0
            x(:) = accumpos(x, ytmp);
            y(:) = accumpos(y, xtmp);
            z(:) = accumneg(z, inpLUT(idx));
        else
            x(:) = accumneg(x, ytmp);
            y(:) = accumneg(y, xtmp);
            z(:) = accumpos(z, inpLUT(idx));
        end
    end
end % idx loop
```

CORDIC-Based Square Root Computation

Square Root Computation Using the CORDIC Hyperbolic Kernel

The judicious choice of initial values allows the CORDIC kernel hyperbolic vectoring mode algorithm to compute square root.

First, the following initialization steps are performed:

- The input look-up table `inpLUT` is set to $\text{atanh}(2^{-N})$.
- x_0 is set to $v + 0.25$.
- y_0 is set to $v - 0.25$.
- z_0 is set to zero.

After N iterations, these initial values lead to the following output as N approaches $+\infty$:

$$x_N \approx A_N \sqrt{(v + 0.25)^2 - (v - 0.25)^2} \approx A_N \sqrt{v}$$

For many square root algorithms, the input value v is typically normalized to a $[0.5, 2)$ range, using a fixed word length normalization. This additional pre-processing step may be used to support large input value ranges, since arbitrary inputs u may be expressed as $u = v * 2^w$, where w is an even integer value. Simple post-processing may then be used to adjust corresponding output values.

Example

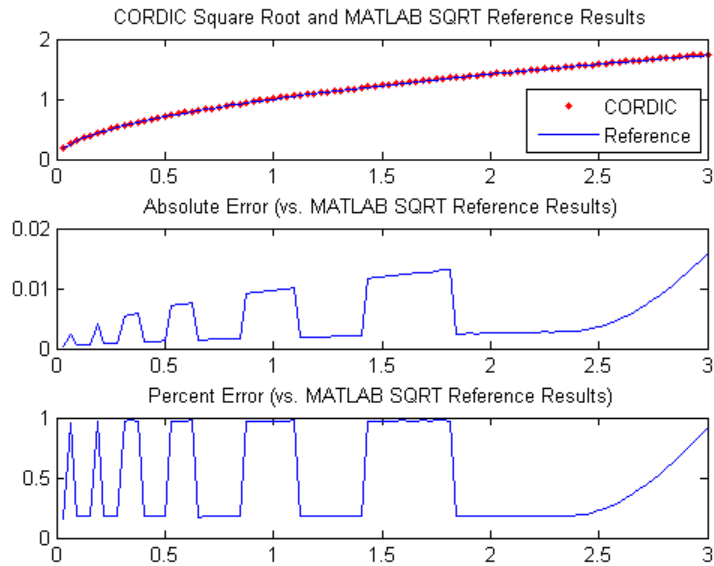
Use CORDIC to compute the square root of `v_fix`:

```
xyNT = numerictype(1,20,16);
v_fix = fi((2^-5):(2^-5):3.0), xyNT); % fixed-point input values
niter = 10; % note that iterations 4, 7, and 10 will be repeated
hpLUT = atanh(2.^ -(1:niter));
z_NT = numerictype(1,24,23);
lutFP = fi(hpLUT, z_NT);
x_sqr = fi(zeros(size(v_fix)), xyNT); % X array pre-allocation
y_sqr = fi(zeros(size(v_fix)), xyNT); % Y array pre-allocation
z_sqr = fi(zeros(size(v_fix)), z_NT); % Z array pre-allocation
```

```
for idx = 1:length(v_fix)
    x_in = fi(accumpos(v_fix(idx), 0.25)); % v + 0.25 in same data type
    y_in = fi(accumneg(v_fix(idx), 0.25)); % v - 0.25 in same data type
    z_in = fi(0, z_NT);

    [x_sqr(idx), y_sqr(idx), z_sqr(idx)] = ...
        fidemo.cordic_hyperbolic_vectoring_kernel(...
            x_in, y_in, z_in, lutFP, niter);
end

% Get the Real World Value (RWV) of the CORDIC outputs for comparison
% and plot the error between the MATLAB reference and CORDIC sqrt values
An_hp = 0.5 .* prod(sqrt(1+2.^(-2*(0:(niter-1)))));
x_cdc = double(x_sqr) ./ An_hp; % CORDIC sqrt results (scaled by An_hp)
v_ref = double(v_fix);
x_ref = sqrt(v_ref); % MATLAB sqrt reference results
figure;
subplot(311);
plot(v_ref, x_cdc, 'r. ');
hold on;
plot(v_ref, x_ref, 'b- ');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root and MATLAB SQRT Reference Results');
hold off;
subplot(312);
absErr = abs(x_ref - x_cdc);
plot(v_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');
subplot(313);
plot(v_ref, 100 .* (absErr ./ x_ref));
title('Percent Error (vs. MATLAB SQRT Reference Results)');
```



References

- 1 Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.
- 2 Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Convert Cartesian to Polar Using CORDIC Vectoring Kernel

This example shows how to convert Cartesian to polar coordinates using a CORDIC vectoring kernel algorithm in MATLAB. CORDIC-based algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Introduction

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm (see [1,2]) is one of the most hardware efficient algorithms because it only requires iterative shift-add operations. The CORDIC algorithm eliminates the need for explicit multipliers, and is suitable for calculating a variety of functions, such as sine, cosine, arcsine, arccosine, arctangent, vector magnitude, divide, square root, hyperbolic and logarithmic functions.

The fixed-point CORDIC algorithm requires the following operations:

- 1 table lookup **per iteration**
- 2 shifts **per iteration**
- 3 additions **per iteration**

CORDIC Kernel Algorithm Using the Vectoring Computation Mode

You can use a CORDIC vectoring computing mode algorithm to calculate $\text{atan}(y/x)$, compute cartesian-polar to cartesian conversions, and for other operations. In vectoring mode, the CORDIC rotator rotates the input vector towards the positive X-axis to minimize the y component of the residual vector. For each iteration, if the y coordinate of the residual vector is positive, the CORDIC rotator rotates clockwise (using a negative angle); otherwise, it rotates counter-clockwise (using a positive angle). Each rotation uses a progressively smaller angle value. If the angle accumulator is initialized to 0, at the end of the iterations, the accumulated rotation angle is the angle of the original input vector.

In vectoring mode, the CORDIC equations are:

$$x_{i+1} = x_i - y_i * d_i * 2^{-i}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{-i}$$

$$z_{i+1} = z_i + d_i * \text{atan}(2^{-i}) \text{ is the angle accumulator}$$

where $d_i = +1$ if $y_i < 0$, and -1 otherwise;

$i = 0, 1, \dots, N-1$, and N is the total number of iterations.

As N approaches $+\infty$:

$$x_N = A_N \sqrt{x_0^2 + y_0^2}$$

$$y_N = 0$$

$$z_N = z_0 + \text{atan}(y_0/x_0)$$

Where:

$$A_N = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}.$$

Typically N is chosen to be a large-enough constant value. Thus, A_N may be pre-computed.

Efficient MATLAB Implementation of a CORDIC Vectoring Kernel Algorithm

A MATLAB code implementation example of the CORDIC Vectoring Kernel algorithm follows (for the case of scalar x , y , and z). This same code can be used for both fixed-point and floating-point operation.

CORDIC Vectoring Kernel

```
function [x, y, z] = cordic_vectoring_kernel(x, y, z, inPLUT, n)
% Perform CORDIC vectoring kernel algorithm for N iterations.
xtmp = x;
ytmp = y;
for idx = 1:n
    if y < 0
        x(:) = accumneg(x, ytmp);
```

```

        y(:) = accumpos(y, xtmp);
        z(:) = accumneg(z, inpLUT(idx));
    else
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
        z(:) = accumpos(z, inpLUT(idx));
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end

```

CORDIC-Based Cartesian to Polar Conversion Using Normalized Input Units

Cartesian to Polar Computation Using the CORDIC Vectoring Kernel

The judicious choice of initial values allows the CORDIC kernel vectoring mode algorithm to directly compute the magnitude $R = \sqrt{x_0^2 + y_0^2}$ and angle $\theta = \text{atan}(y_0/x_0)$.

The input accumulators are initialized to the input coordinate values:

- $x_0 = X$
- $y_0 = Y$

The angle accumulator is initialized to zero:

- $z_0 = 0$

After N iterations, these initial values lead to the following outputs as N approaches $+\infty$:

- $x_N \approx A_N \sqrt{x_0^2 + y_0^2}$
- $z_N \approx \text{atan}(y_0/x_0)$

Other vectoring-kernel-based function approximations are possible via pre- and post-processing and using other initial conditions (see [1,2]).

Example

Suppose that you have some measurements of Cartesian (X,Y) data, normalized to values between [-1, 1], that you want to convert into polar (magnitude, angle) coordinates. Also suppose that you have a 16-bit integer arithmetic unit that can perform add, subtract, shift, and memory operations. With such a device, you could implement the CORDIC vectoring kernel to efficiently compute magnitude and angle from the input (X,Y) coordinate values, without the use of multiplies or large lookup tables.

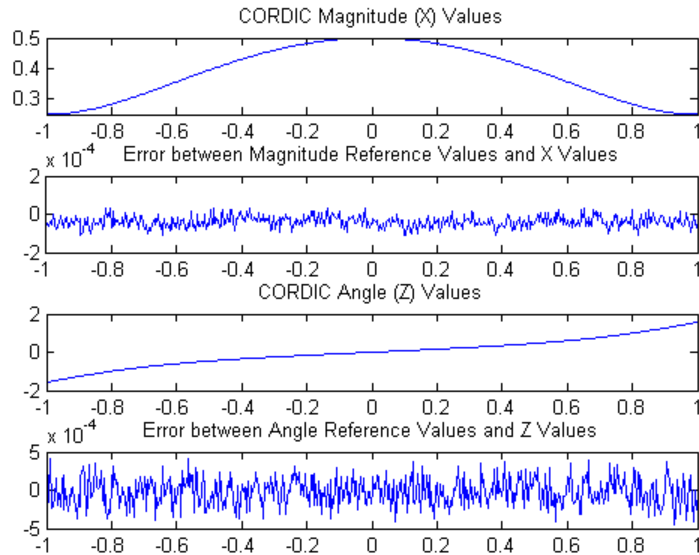
```
sumWL = 16; % CORDIC sum word length
thNorm = -1.0:(2^-8):1.0; % Also using normalized [-1.0, 1.0] angle values
theta = fi(thNorm, 1, sumWL); % Fixed-point angle values (best precision)
z_NT = numerictype(theta); % Data type for Z
xyCPNT = numerictype(1,16,15); % Using normalized X-Y range [-1.0, 1.0]
thetaRadians = pi/2 .* thNorm; % real-world range [-pi/2 pi/2] angle values
inXfix = fi(0.50 .* cos(thetaRadians), xyCPNT); % X coordinate values
inYfix = fi(0.25 .* sin(thetaRadians), xyCPNT); % Y coordinate values

nitters = 13; % Number of CORDIC iterations
inpLUT = fi(atan(2 .^ (-((0:(nitters-1))')))) .* (2/pi), z_NT); % Normalized
z_c2p = fi(zeros(size(theta)), z_NT); % Z array pre-allocation
x_c2p = fi(zeros(size(theta)), xyCPNT); % X array pre-allocation
y_c2p = fi(zeros(size(theta)), xyCPNT); % Y array pre-allocation

for idx = 1:length(inXfix)
    % CORDIC vectoring kernel iterations
    [x_c2p(idx), y_c2p(idx), z_c2p(idx)] = ...
        fidemo.cordic_vectoring_kernel(...
            inXfix(idx), inYfix(idx), fi(0, z_NT), inpLUT, nitters);
end

% Get the Real World Value (RWV) of the CORDIC outputs for comparison
% and plot the error between the (magnitude, angle) values
AnGain = prod(sqrt(1+2.^(-2*(0:(nitters-1))))); % CORDIC gain
x_c2p_RWV = (1/AnGain) .* double(x_c2p); % Magnitude (scaled by CORDIC g
z_c2p_RWV = (pi/2) .* double(z_c2p); % Angles (in radian units)
[thRWV,rRWV] = cart2pol(double(inXfix), double(inYfix)); % MATLAB reference
magnitudeErr = rRWV - x_c2p_RWV;
angleErr = thRWV - z_c2p_RWV;
```

```
figure;
subplot(411);
plot(thNorm, x_c2p_RWV);
axis([-1 1 0.25 0.5]);
title('CORDIC Magnitude (X) Values');
subplot(412);
plot(thNorm, magnitudeErr);
title('Error between Magnitude Reference Values and X Values');
subplot(413);
plot(thNorm, z_c2p_RWV);
title('CORDIC Angle (Z) Values');
subplot(414);
plot(thNorm, angleErr);
title('Error between Angle Reference Values and Z Values');
```



References

- 1** Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.
- 2** Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Set Data Types Using Min/Max Instrumentation

This example shows how to set fixed-point data types by instrumenting MATLAB code for min/max logging and using the tools to propose data types.

The functions you will use are:

- `buildInstrumentedMex` - Build MEX function with instrumentation enabled
- `showInstrumentationResults` - Show instrumentation results
- `clearInstrumentationResults` - Clear instrumentation results

The Unit Under Test

The function that you convert to fixed-point in this example is a second-order direct-form 2 transposed filter. You can substitute your own function in place of this one to reproduce these steps in your own work.

```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
    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
```

For a MATLAB function to be instrumented, it must be suitable for code generation. For information on code generation, see the reference page for `buildInstrumentedMex`. A MATLAB Coder™ license is not required to use `buildInstrumentedMex`.

In this function the variables `y` and `z` are used as both inputs and outputs. This is an important pattern because:

- You can set the data type of `y` and `z` outside the function, thus allowing you to re-use the function for both fixed-point and floating-point types.
- The generated C code will create `y` and `z` as references in the function argument list. For more information about this pattern, see the

documentation under Code Generation from MATLAB > User's Guide > Generating Efficient and Reusable Code > Generating Efficient Code > Eliminating Redundant Copies of Function Inputs.

Run the following code to copy the test function into a temporary directory so this example doesn't interfere with your own work.

```
tempdirObj = fidemo.fiTempdir('fi_instrumentation_fixed_point_filter_demo')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
    'fi_2nd_order_df2t_filter.m'),'.','f');
```

Run the following code to capture current states, and reset the global states.

```
FIPREF_STATE = fipref;
reset(fipref)
```

Data Types Determined by the Requirements of the Design

In this example, the requirements of the design determine the data type of input x . These requirements are signed, 16-bit, and fractional.

```
N = 256;
x = fi(zeros(N,1),1,16,15);
```

The requirements of the design also determine the fixed-point math for a DSP target with a 40-bit accumulator. This example uses floor rounding and wrap overflow to produce efficient generated code.

```
F = fimath('RoundingMethod','Floor',...
    'OverflowAction','Wrap',...
    'ProductMode','KeepLSB',...
    'ProductWordLength',40,...
    'SumMode','KeepLSB',...
    'SumWordLength',40);
```

The following coefficients correspond to a second-order lowpass filter created by

```
[num,den] = butter(2,0.125)
```

The values of the coefficients influence the range of the values that will be assigned to the filter output and states.

```
num = [0.0299545822080925  0.0599091644161849  0.0299545822080925];
den = [1                    -1.4542435862515900  0.5740619150839550];
```

The data type of the coefficients, determined by the requirements of the design, are specified as 16-bit word length and scaled to best-precision. A pattern for creating `fi` objects from constant coefficients is:

1. Cast the coefficients to `fi` objects using the default round-to-nearest and saturate overflow settings, which gives the coefficients better accuracy.
2. Attach `fimath` with floor rounding and wrap overflow settings to control arithmetic, which leads to more efficient C code.

```
b = fi(num,1,16); b.fimath = F;
a = fi(den,1,16); a.fimath = F;
```

Hard-code the filter coefficients into the implementation of this filter by passing them as constants to the `buildInstrumentedMex` command.

```
B = coder.Constant(b);
A = coder.Constant(a);
```

Data Types Determined by the Values of the Coefficients and Inputs

The values of the coefficients and values of the inputs determine the data types of output `y` and state vector `z`. Create them with a scaled double datatype so their values will attain full range and you can identify potential overflows and propose data types.

```
yisd = fi(zeros(N,1),1,16,15,'DataType','ScaledDouble','fimath',F);
zisd = fi(zeros(2,1),1,16,15,'DataType','ScaledDouble','fimath',F);
```

Instrument the MATLAB Function as a Scaled-Double MEX Function

To instrument the MATLAB code, you create a MEX function from the MATLAB function using the `buildInstrumentedMex` command. The inputs to `buildInstrumentedMex` are the same as the inputs to `fiaccel`, but `buildInstrumentedMex` has no `fi`-object restrictions. The output of

`buildInstrumentedMex` is a MEX function with instrumentation inserted, so when the MEX function is run, the simulated minimum and maximum values are recorded for all named variables and intermediate values.

Use the `'-o'` option to name the MEX function that is generated. If you do not use the `'-o'` option, then the MEX function is the name of the MATLAB function with `'_mex'` appended. You can also name the MEX function the same as the MATLAB function, but you need to remember that MEX functions take precedence over MATLAB functions and so changes to the MATLAB function will not run until either the MEX function is re-generated, or the MEX function is deleted and cleared.

```
buildInstrumentedMex fi_2nd_order_df2t_filter ...
    -o filter_scaled_double ...
    -args {B,A,x,yisd,zisd}
```

Test Bench with Chirp Input

The test bench for this system is set up to run chirp and step signals. In general, test benches for systems should cover a wide range of input signals.

The first test bench uses a chirp input. A chirp signal is a good representative input because it covers a wide range of frequencies.

```
t = linspace(0,1,N);           % Time vector from 0 to 1 second
f1 = N/2;                     % Target frequency of chirp set to Nyquist
xchirp = sin(pi*f1*t.^2);      % Linear chirp from 0 to Fs/2 Hz in 1 second
x(:) = xchirp;                 % Cast the chirp to fixed-point
```

Run the Instrumented MEX Function to Record Min/Max Values

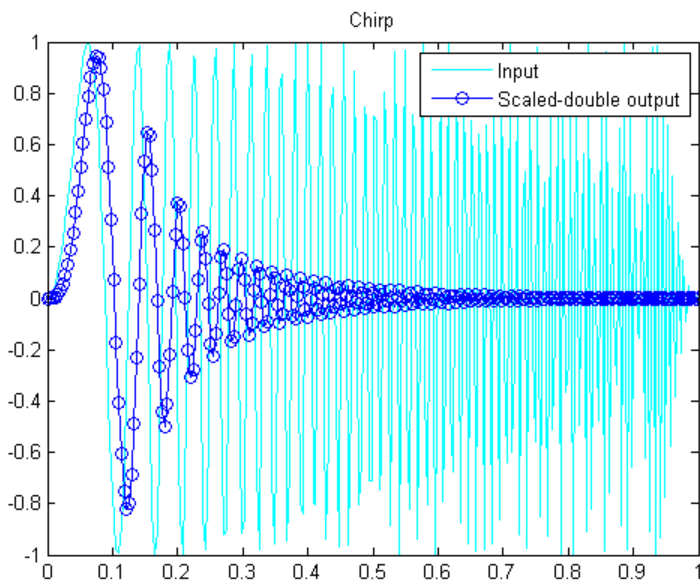
The instrumented MEX function must be run to record minimum and maximum values for that simulation run. Subsequent runs accumulate the instrumentation results until they are cleared with `clearInstrumentationResults`.

Note that the numerator and denominator coefficients were compiled as constants so they are not provided as input to the generated MEX function.

```
ychirp = filter_scaled_double(x,yisd,zisd);
```

The plot of the filtered chirp signal shows the lowpass behavior of the filter with these particular coefficients. Low frequencies are passed through and higher frequencies are attenuated.

```
clf
plot(t,x,'c',t,ychirp,'bo-')
title('Chirp')
legend('Input','Scaled-double output')
figure(gcf); drawnow;
```



Show Instrumentation Results with Proposed Fraction Lengths for Chirp

The `showInstrumentationResults` command displays the code generation report with instrumented values. The input to `showInstrumentationResults` is the name of the instrumented MEX function for which you wish to show results.

This is the list of options to the `showInstrumentationResults` command:

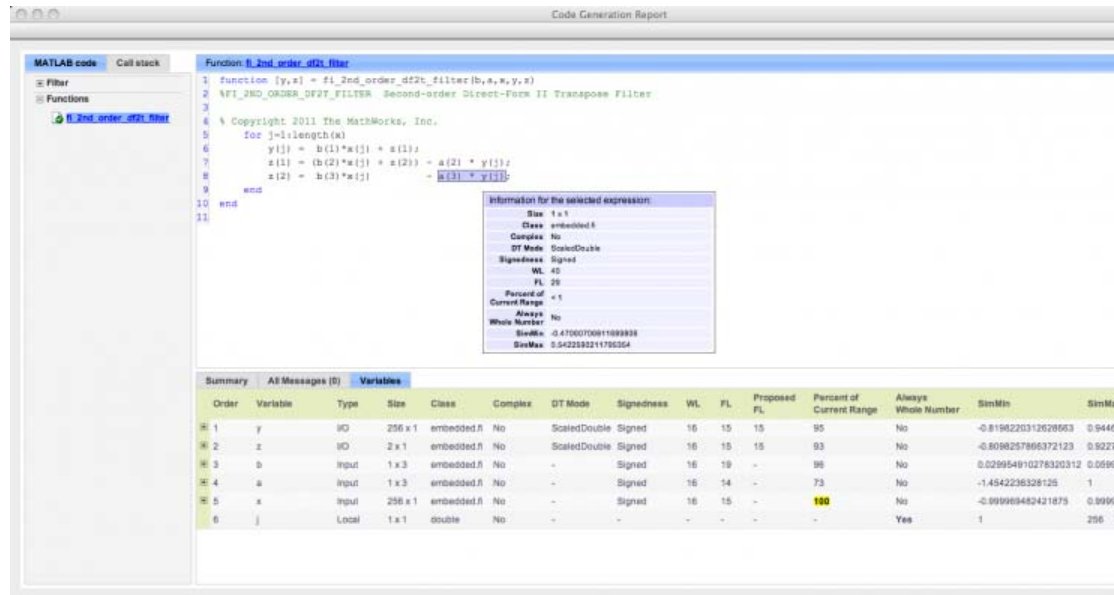
- `-browser` Open the instrumentation results in a system web browser window. Use this option to open multiple reports so you can compare results.
- `-defaulttDT T` Default data type to propose for doubles, where `T` is a `numericType` object, or one of the strings `{remainFloat, double, single, int8, int16, int32, int64, uint8, uint16, uint32, uint64}`. The default is `remainFloat`.
- `-optimizeWholeNumbers` Optimize the word length of variables whose simulation min/max logs indicate that they were always whole numbers.
- `-percentSafetyMargin N` Safety margin for simulation min/max, where `N` represents a percent value.
- `-printable` Create a printable report.
- `-proposeFL` Propose fraction lengths for specified word lengths.
- `-proposeWL` Propose word lengths for specified fraction lengths.

Potential overflows are only displayed for `fi` objects with Scaled Double data type.

This particular design is for a DSP, where the word lengths are fixed, so use the `proposeFL` flag to propose fraction lengths.

```
showInstrumentationResults filter_scaled_double -proposeFL
```

Hover over expressions or variables in the instrumented code generation report to see the simulation minimum and maximum values. In this design, the inputs fall between -1 and +1, and the values of all variables and intermediate results also fall between -1 and +1. This suggests that the data types can all be fractional (fraction length one bit less than the word length). However, this will not always be true for this function for other kinds of inputs and it is important to test many types of inputs before setting final fixed-point data types.



Test Bench with Step Input

The next test bench is run with a step input. A step input is a good representative input because it is often used to characterize the behavior of a system.

```

xstep = [ones(N/2,1);-ones(N/2,1)];
x(:) = xstep;

```

Run the Instrumented MEX Function with Step Input

The instrumentation results are accumulated until they are cleared with `clearInstrumentationResults`.

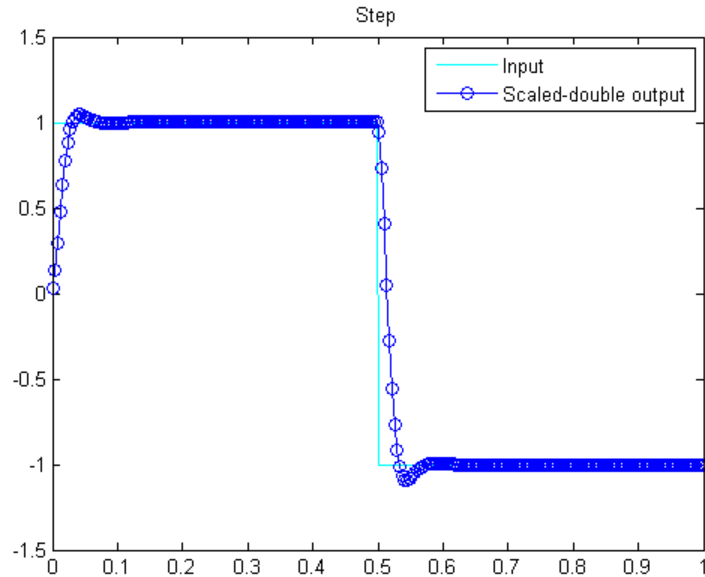
```

ystep = filter_scaled_double(x,yisd,zisd);

clf
plot(t,x,'c',t,ystep,'bo-')
title('Step')
legend('Input','Scaled-double output')

```

```
figure(gcf); drawnow;
```



Show Accumulated Instrumentation Results

Even though the inputs for step and chirp inputs are both full range as indicated by x at 100 percent current range in the instrumented code generation report, the step input causes overflow while the chirp input did not. This is an illustration of the necessity to have many different inputs for your test bench. For the purposes of this example, only two inputs were used, but real test benches should be more thorough.

```
showInstrumentationResults filter_scaled_double -proposeFL
```

MATLAB code

Call stack

Filter

Functions

fi_2nd_order_df2t_filter

Function: fi_2nd_order_df2t_filter

```
1 function [y,z] = fi_2nd_order_df2t_filter(b,a,x,F,x)
2 %FI_2ND_ORDER_DF2T_FILTER Second-order Direct-Form II Transpose Filter
3
4 % Copyright 2011 The MathWorks, Inc.
5 for j=1:length(x)
6     y[j] = b(1)*x[j] + z(1);
7     z(1) = (b(2)*x[j] + z(2)) - a(2) * y[j];
8     z(2) = b(3)*x[j] - a(3) * y[j];
9 end
10 end
11
```

Summary

All Messages (0)

Variables

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Proposed FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	IO	256 x 1	embedded fi	No	ScaledDouble	Signed	16	15	14	100	No	-1.0811287150691327	1.0495
2	z	IO	2 x 1	embedded fi	No	ScaledDouble	Signed	16	15	14	100	No	-1.0611738048108124	1.0150
3	b	Input	1 x 3	embedded fi	No	-	Signed	16	19	-	96	No	0.029954910278320312	0.0599
4	a	Input	1 x 3	embedded fi	No	-	Signed	16	14	-	73	No	-1.4542236328125	1
5	x	Input	256 x 1	embedded fi	No	-	Signed	16	15	-	100	No	-1	0.9999
6	j	Local	1 x 1	double	No	-	-	-	-	-	-	Yes	1	256

Apply Proposed Fixed-Point Properties

To prevent overflow, set proposed fixed-point properties based on the proposed fraction lengths of 14-bits for y and z from the instrumented code generation report.

At this point in the workflow, you use true fixed-point types (as opposed to the scaled double types that were used in the earlier step of determining data types).

```
yi = fi(zeros(N,1),1,16,14,'fimath',F);
zi = fi(zeros(2,1),1,16,14,'fimath',F);
```

Instrument the MATLAB Function as a Fixed-Point MEX Function

Create an instrumented fixed-point MEX function by using fixed-point inputs and the buildInstrumentedMex command.

```
buildInstrumentedMex fi_2nd_order_df2t_filter ...
    -o filter_fixed_point ...
```

```
-args {B,A,x,yi,zi}
```

Validate the Fixed-Point Algorithm

After converting to fixed-point, run the test bench again with fixed-point inputs to validate the design.

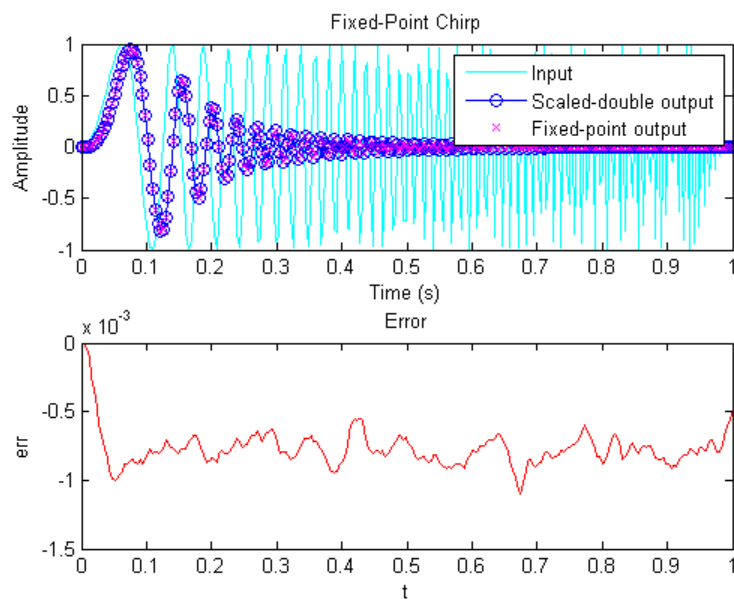
Validate with Chirp Input

Run the fixed-point algorithm with a chirp input to validate the design.

```
x(:) = xchirp;
[y,z] = filter_fixed_point(x,yi,zi);
[ysd,zsd] = filter_scaled_double(x,yisd,zisd);
err = double(y) - double(ysd);
```

Compare the fixed-point outputs to the scaled-double outputs to verify that they meet your design criteria.

```
clf
subplot(211);plot(t,x,'c',t,ysd,'bo-',t,y,'mx')
xlabel('Time (s)');
ylabel('Amplitude')
legend('Input','Scaled-double output','Fixed-point output');
title('Fixed-Point Chirp')
subplot(212);plot(t,err,'r');title('Error');xlabel('t'); ylabel('err');
figure(gcf); drawnow;
```



Inspect the variables and intermediate results to ensure that the min/max values are within range.

```
showInstrumentationResults filter_fixed_point
```

The screenshot shows the MATLAB Coder interface. On the left, the 'MATLAB code' pane displays the function `fi_2nd_order_df2t_filter`. The code is as follows:

```

1 function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
2 %FI_2ND_ORDER_DF2T_FILTER Second-order Direct-Form II Transpose Filter
3
4 % Copyright 2011 The MathWorks, Inc.
5 for j=1:length(x)
6     y[j] = b(1)*x[j] + z(1);
7     z(1) = (b(2)*x[j] + z(2)) - a(2) * y[j];
8     z(2) = b(3)*x[j] - a(3) * y[j];
9 end
10 end
11

```

On the right, the 'Variables' tab of the instrumentation report is displayed. The table below represents the data shown in this tab.

Order	Variable	Type	Size	Class	Complex	Signedness	WL	PL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	IO	256 x 1	embedded.f	No	Signed	16	14	46	No	-8.82061767578125	0.94378
2	z	IO	2 x 1	embedded.f	No	Signed	16	14	47	No	-8.81060791015625	0.82183
3	b	Input	1 x 3	embedded.f	No	Signed	16	19	96	No	0.029954910278320312	0.05990
4	a	Input	1 x 3	embedded.f	No	Signed	16	14	73	No	-1.4542236328125	1
5	x	Input	256 x 1	embedded.f	No	Signed	16	15	100	No	-8.999969482421875	0.99993
6	j	Local	1 x 1	double	No	-	-	-	-	Yes	1	256

Validate with Step Inputs

Run the fixed-point algorithm with a step input to validate the design.

Run the following code to clear the previous instrumentation results to see only the effects of running the step input.

```
clearInstrumentationResults filter_fixed_point
```

Run the step input through the fixed-point filter and compare with the output of the scaled double filter.

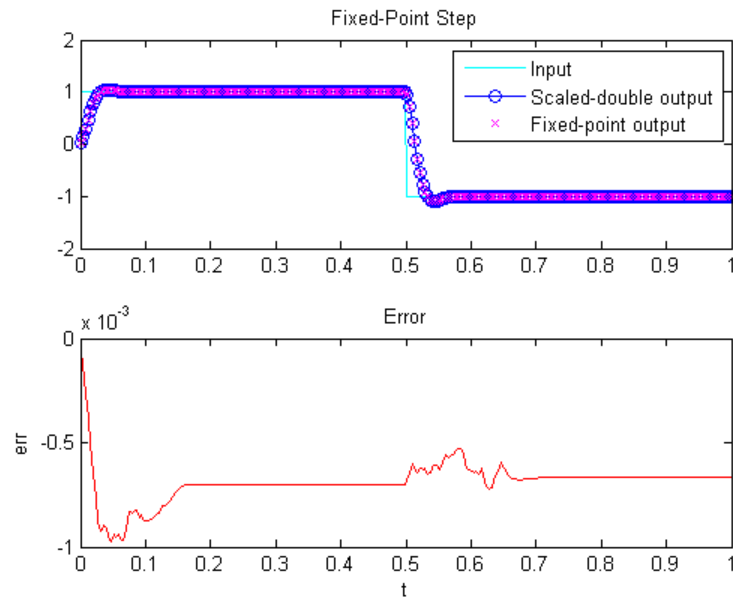
```

x(:) = xstep;
[y,z] = filter_fixed_point(x,yi,zi);
[ysd,zsd] = filter_scaled_double(x,yisd,zisd);
err = double(y) - double(ysd);

```

Plot the fixed-point outputs against the scaled-double outputs to verify that they meet your design criteria.

```
clf
subplot(211);plot(t,x,'c',t,y,'bo-',t,y,'mx')
title('Fixed-Point Step');
legend('Input','Scaled-double output','Fixed-point output')
subplot(212);plot(t,err,'r');title('Error');xlabel('t'); ylabel('err');
figure(gcf); drawnow;
```



Inspect the variables and intermediate results to ensure that the min/max values are within range.

```
showInstrumentationResults filter_fixed_point
```


Code Generation Report

MATLAB code **Call stack**

Function: fi_2nd_order_df2t_filter

```

1 function [y,z] = fi_2nd_order_df2t_filter(b,a,x,s)
2 %FI_2ND_ORDER_DF2T_FILTER Second-order Direct-Form II Transpose Filter
3
4 % Copyright 2011 The MathWorks, Inc.
5 for j=1:length(x)
6     [z(1) z(2)] = b(1)*x(j) + z(1);
7     z(1) = (b(2)*x(j) + z(2)) - a(2) * z(1);
8     z(2) = b(3)*x(j) - a(3) * z(1);
9 end
10 end
11

```

Summary **All Messages (0)** **Variables**

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	IO	256 x 1	embedded:fi	No	Signed	16	14	55	No	-1.09173583964375	1.04481
2	z	IO	2 x 1	embedded:fi	No	Signed	16	14	54	No	-1.061767576125	1.01470
3	b	Input	1 x 3	embedded:fi	No	Signed	16	19	96	No	0.029954910278320312	0.05990
4	a	Input	1 x 3	embedded:fi	No	Signed	16	14	73	No	-1.4542236328125	1
5	s	Input	256 x 1	embedded:fi	No	Signed	16	15	100	No	-1	0.99998
6	j	Local	1 x 1	double	No	-	-	-	-	Yes	1	256

Run the following code to restore the global states.

```

fipref(FIPREF_STATE);
clearInstrumentationResults filter_fixed_point
clearInstrumentationResults filter_scaled_double
clear fi_2nd_order_df2t_filter_fixed_instrumented
clear fi_2nd_order_df2t_filter_float_instrumented

```

Run the following code to delete the temporary directory.

```
tempdirObj.cleanUp;
```

Convert Fast Fourier Transform (FFT) to Fixed Point

This example shows how to convert a textbook version of the Fast Fourier Transform (FFT) algorithm into fixed-point MATLAB code.

Run the following code to copy functions from the Fixed-Point Toolbox™ examples directory into a temporary directory so this example doesn't interfere with your own work.

```
tempdirObj = fidemo.fiTempdir('fi_radix2fft_demo');

copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fideos','+fideo',...
    'fi_m_radix2fft_algorithm1_6_2.m'),'.', 'f');
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fideos',...
    'fi_m_radix2fft_withscaling.m'),'.', 'f');
```

Run the following code to capture current states, and reset the global states.

```
FIPREF_STATE = fipref;
reset(fipref)
```

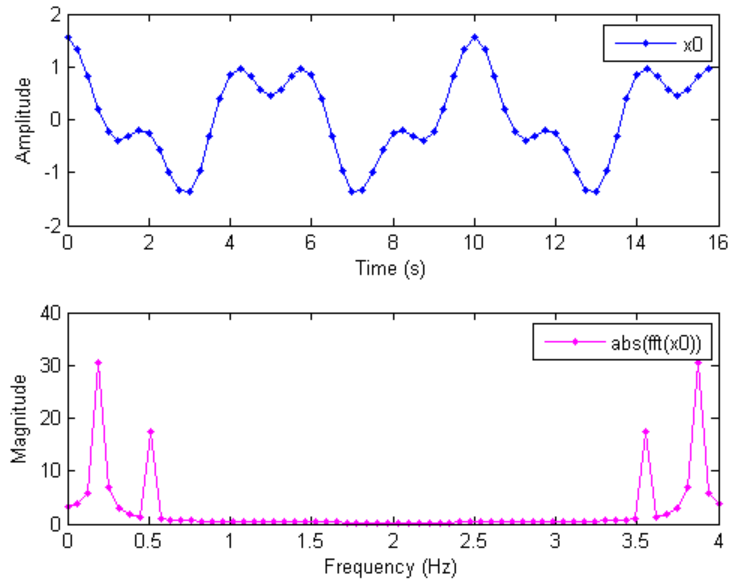
Textbook FFT Algorithm

FFT is a complex-valued linear transformation from the time domain to the frequency domain. For example, if you construct a vector as the sum of two sinusoids and transform it with the FFT, you can see the peaks of the frequencies in the FFT magnitude plot.

```
n = 64; % Number of points
Fs = 4; % Sampling frequency in Hz
t = (0:(n-1))/Fs; % Time vector
f = linspace(0,Fs,n); % Frequency vector
f0 = .2; f1 = .5; % Frequencies, in Hz
x0 = cos(2*pi*f0*t) + 0.55*cos(2*pi*f1*t); % Time-domain signal
x0 = complex(x0); % The textbook algorithm requires
% the input to be complex
y = fft(x0); % Frequency-domain transformation

figure(gcf); clf
subplot(211); plot(t,real(x0),'b.-'); xlabel('Time (s)'); ylabel('Amplitude')
```

```
subplot(212); plot(f,abs(y),'m.-'); xlabel('Frequency (Hz)'); ylabel('Magni
```



The peaks at 0.2 and 0.5 Hz in the frequency plot correspond to the two sinusoids of the time-domain signal at those frequencies.

Note the reflected peaks at 3.5 and 3.8 Hz. When the input to an FFT is real-valued, as it is in this case, then the output y is conjugate-symmetric:

$$y(k) = \text{conj}(y(N - k)).$$

There are many different implementations of the FFT, each having its own costs and benefits. You may find that a different algorithm is better for your application than the one given here. This algorithm is used to provide you with an example of how you might begin your own exploration.

This example uses the decimation-in-time unit-stride FFT shown in Algorithm 1.6.2 on page 45 of the book *Computational Frameworks for the Fast Fourier Transform* by Charles Van Loan (<http://www.mathworks.com/support/books/book1384.html>).

In pseudo-code, the algorithm in the textbook is as follows.

Algorithm 1.6.2. If x is a complex vector of length n and $n = 2^t$, then the following algorithm overwrites x with $F_n x$.

```

 $x = P_n x$ 
 $w = w_n^{(long)}$       (See Van Loan §1.4.11.)
for  $q = 1 : t$ 
     $L = 2^q$ ;  $r = n/L$ ;  $L_s = L/2$ ;
    for  $k = 0 : r - 1$ 
        for  $j = 0 : L_s - 1$ 
             $\tau = w(L_s - 1 + j) \cdot x(kL + j + L_s)$ 
             $x(kL + j + L_s) = x(kL + j) - \tau$ 
             $x(kL + j) = x(kL + j) + \tau$ 
        end
    end
end
end

```

The textbook algorithm uses zero-based indexing. F_n is an n -by- n Fourier-transform matrix, P_n is an n -by- n bit-reversal permutation matrix, and w is a complex vector of twiddle factors. The twiddle factors, w , are complex roots of unity computed by the following algorithm:

```

function w = fi_radix2twiddles(n)
t = log2(n);
if floor(t) ~= t
    error('N must be an exact power of two. ');
end
w = zeros(n-1,1);
k=1;
L=2;
% Equation 1.4.11, p. 34
while L<=n
    theta = 2*pi/L;
    % Algorithm 1.4.1, p. 23
    for j=0:(L/2 - 1)
        w(k) = complex( cos(j*theta), -sin(j*theta) );
        k = k + 1;
    end
    L=L*2;
end

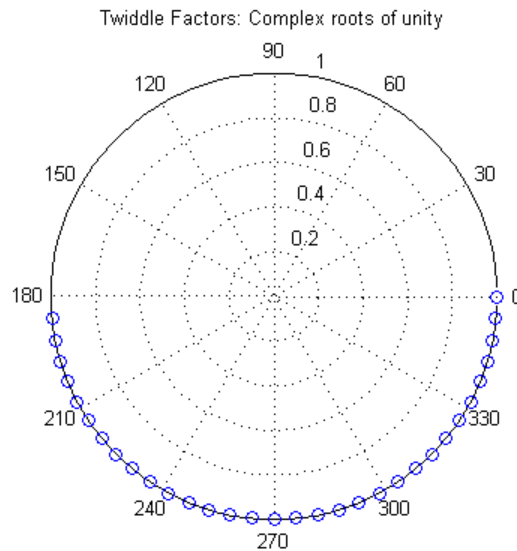
```

```

    L = L*2;
end

figure(gcf);clf
w0 = fidemo.fi_radix2twiddles(n);
polar(angle(w0),abs(w0),'o')
title('Twiddle Factors: Complex roots of unity')

```



Verify Floating-Point Code

To implement the algorithm in MATLAB, you can use the `fidemo.fi_bitreverse` function to bit-reverse the input sequence, and you must add one to the indices to convert them from zero-based to one-based.

```

function x = fi_m_radix2fft_algorithm1_6_2(x, w)
n = length(x); t = log2(n);
x = fidemo.fi_bitreverse(x,n);
for q=1:t
    L = 2^q; r = n/L; L2 = L/2;

```

```

for k=0:(r-1)
    for j=0:(L2-1)
        temp = w(L2-1+j+1) * x(k*L+j+L2+1);
        x(k*L+j+L2+1) = x(k*L+j+1) - temp;
        x(k*L+j+1) = x(k*L+j+1) + temp;
    end
end
end
end

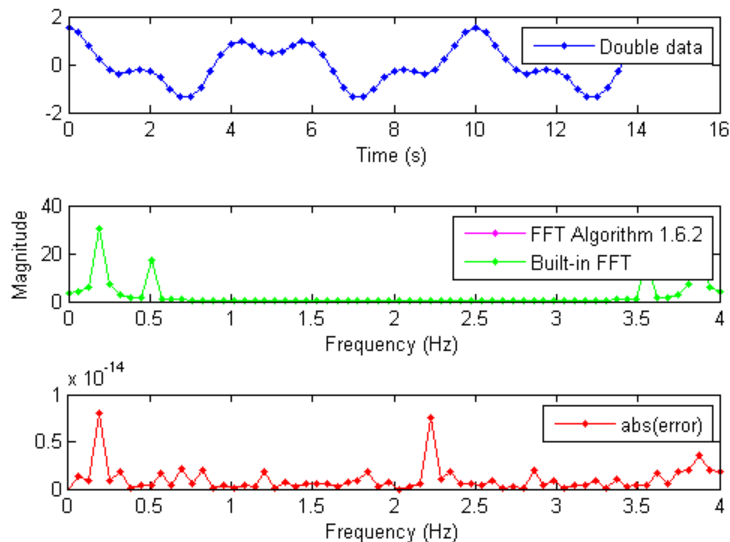
```

To verify that you correctly implemented the algorithm in MATLAB, run a known signal through it and compare the results to the results produced by the MATLAB FFT function.

```
y = fi_m_radix2fft_algorithm1_6_2(x0, w0);
```

```
y0 = fft(x0); % MATLAB's built-in FFT for comparison
```

```
fidemo.fi_fft_demo_plot(real(x0),y,y0,Fs,'Double data', {'FFT Algorithm 1.6.2', 'Built-in FFT'})
```



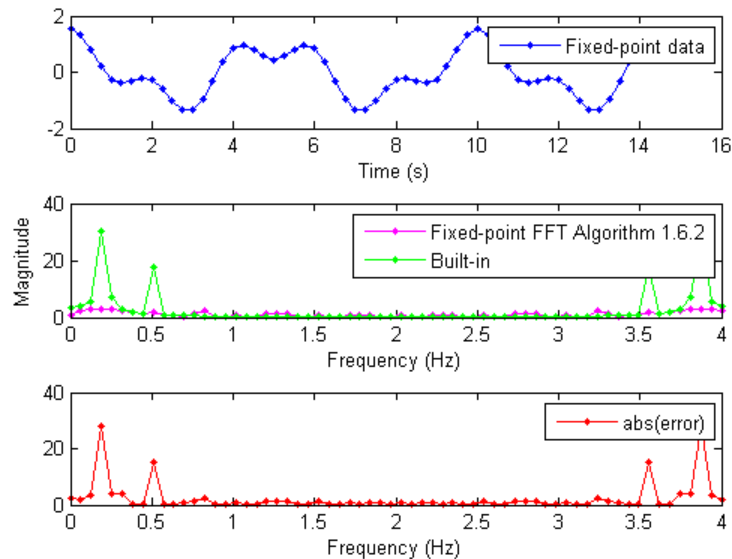
Because the error is within tolerance of the MATLAB built-in FFT function, you know you have correctly implemented the algorithm.

Identify Fixed-Point Issues

Now, try converting the data to fixed-point and see if the algorithm still looks good. In this first pass, you use all the defaults for signed fixed-point data by using the `sfi` constructor.

```
x = sfi(x0); % Convert to signed fixed-point
w = sfi(w0); % Convert to signed fixed-point

% Re-run the same algorithm with the fixed-point inputs
y = fi_m_radix2fft_algorithm1_6_2(x,w);
fidemo.fi_fft_demo_plot(real(x),y,y0,Fs,'Fixed-point data', ...
                        {'Fixed-point FFT Algorithm 1.6.2','Built-in'});
```



Note that the magnitude plot (center) of the fixed-point FFT does not resemble the plot of the built-in FFT. The error (bottom plot) is much larger than what

you would expect to see for round off error, so it is likely that overflow has occurred.

Use Min/Max Instrumentation to Identify Overflows

To instrument the MATLAB code, you create a MEX function from the MATLAB function using the `buildInstrumentedMex` command. The inputs to `buildInstrumentedMex` are the same as the inputs to `fiaccel`, but `buildInstrumentedMex` has no fi-object restrictions. The output of `buildInstrumentedMex` is a MEX function with instrumentation inserted, so when the MEX function is run, the simulated minimum and maximum values are recorded for all named variables and intermediate values.

The `'-o'` option is used to name the MEX function that is generated. If the `'-o'` option is not used, then the MEX function is the name of the MATLAB function with `'_mex'` appended. You can also name the MEX function the same as the MATLAB function, but you need to remember that MEX functions take precedence over MATLAB functions and so changes to the MATLAB function will not run until either the MEX function is re-generated, or the MEX function is deleted and cleared.

Create the input with a scaled double datatype so its values will attain full range and you can identify potential overflows.

```
x_scaled_double = fi(x0,'DataType','ScaledDouble');  
buildInstrumentedMex fi_m_radix2fft_algorithm1_6_2 ...  
    -o fft_instrumented -args {x_scaled_double w}
```

Run the instrumented MEX function to record min/max values.

```
y_scaled_double = fft_instrumented(x_scaled_double,w);
```

Show the instrumentation results.

```
showInstrumentationResults fft_instrumented
```

You can see from the instrumentation results that there were overflows when assigning into the variable `x`.

Code Generation

MATLAB code

Call stack

Filter

Functions

fi_bitreverse

fi_m_radix2fft_algorithm1

Function: fi_m_radix2fft_algorithm1_6_2

```

1 function x = fi_m_radix2fft_algorithm1_6_2(x0, w)
2 %FI_M_RADIX2FFT_ALGORITHM1_6_2 Radix-2 FFT example.
3 % Y = FI_M_RADIX2FFT_ALGORITHM1_6_2(X, W) computes the r
4 % input vector X with twiddle-factors W.
5 %
6 % The length of vector X must be an exact power of two.
7 % Twiddle-factors W are computed via
8 % W = fidemo.fi_radix2twiddles(N)
9 % where N = length(X).
10 %
11 % This version of the algorithm has no scaling before th
12 %
13 % See also FI_RADIX2FFT_DEMO, FI_M_RADIX2FFT_WITHSCALING
14 %
15 % Reference:
16 % Charles Van Loan, Computational Frameworks for the F
17 % Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2
18 if isreal(x0)
19     x = complex(x0,0);
20 else
21     x = x0;
22 end
23 n = length(x); t = log2(n);
24 x = fi_bitreverse(x,n);
25 for q=1:t
26     L = 2^q; r = n/L; L2 = L/2;
27     for k=0:(r-1)
28         for j=0:(L2-1)
29             temp = w(L2-1+j+1) * x(k*L+j+L2+1);
30             x(k*L+j+L2+1) = x(k*L+j+1) - temp;
31             x(k*L+j+1) = x(k*L+j+1) + temp;
32         end
33     end
34 end
35

```

Summary

All Messages (0)

Variables

Order	Variable	Type	Size	Class	Complex	DT Mode
1	x	Output	1 x 64	embedded.fi	Yes	ScaledDouble
2	x0	Input	1 x 64	embedded.fi	No	ScaledDouble
3	w	Input	63 x 1	embedded.fi	Yes	3-175
4	n	Local	1 x 1	double	No	-
5	t	Local	1 x 1	double	No	-
6	r	Local	1 x 1	double	No	-

Modify the Algorithm to Address Fixed-Point Issues

The magnitude of an individual bin in the FFT grows, at most, by a factor of n , where n is the length of the FFT. Hence, by scaling your data by $1/n$, you can prevent overflow from occurring for any input.

When you scale only the input to the first stage of a length- n FFT by $1/n$, you obtain a noise-to-signal ratio proportional to n^2 [Oppenheim & Schaffer 1989, equation 9.101], [Welch 1969].

However, if you scale the input to each of the stages of the FFT by $1/2$, you can obtain an overall scaling of $1/n$ and produce a noise-to-signal ratio proportional to n [Oppenheim & Schaffer 1989, equation 9.105], [Welch 1969].

An efficient way to scale by $1/2$ in fixed-point is to right-shift the data. To do this, you use the bit shift right arithmetic function `bitsra`. After scaling each stage of the FFT, and optimizing the index variable computation, your algorithm becomes:

```
function x = fi_m_radix2fft_withscaling(x, w)
n = length(x); t = log2(n);
x = fidemo.fi_bitreverse(x,n);
% Generate index variables as integer constants so they are not computed in
% the loop.
LL = int32(2.^(1:t)); rr = int32(n./LL); LL2 = int32(LL./2);
for q=1:t
    L = LL(q); r = rr(q); L2 = LL2(q);
    for k=0:(r-1)
        for j=0:(L2-1)
            temp = w(L2-1+j+1) * x(k*L+j+L2+1);
            x(k*L+j+L2+1) = bitsra(x(k*L+j+1) - temp, 1);
            x(k*L+j+1) = bitsra(x(k*L+j+1) + temp, 1);
        end
    end
end
```

Run the scaled algorithm with fixed-point data.

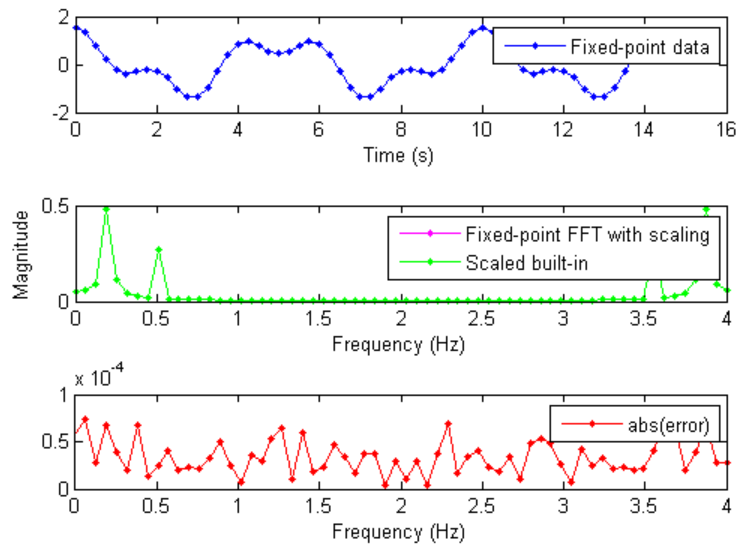
```

x = sfi(x0);
w = sfi(w0);

y = fi_m_radix2fft_withscaling(x,w);

fidemo.fi_fft_demo_plot(real(x), y, y0/n, Fs, 'Fixed-point data', ...
    {'Fixed-point FFT with scaling','Scaled built-in'})

```



You can see that the scaled fixed-point FFT algorithm now matches the built-in FFT to a tolerance that is expected for 16-bit fixed-point data.

References

Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992, <http://www.mathworks.com/support/books/book1384.html>.

Cleve Moler, *Numerical Computing with MATLAB*, SIAM, 2004, Chapter 8 Fourier Analysis,

<http://www.mathworks.com/company/aboutus/founders/clevemoler.html>,
<http://www.mathworks.com/support/books/book7638.html>.

Alan V. Oppenheim and Ronald W. Schaffer, *Discrete-Time Signal Processing*,
Prentice Hall, 1989.

Peter D. Welch, "A Fixed-Point Fast Fourier Transform Error Analysis,"
IEEE Transactions on Audio and Electroacoustics, Vol. AU-17, No. 2, June
1969, pp. 151-157.

Run the following code to restore the global states.

```
fipref(FIPREF_STATE);  
clearInstrumentationResults fft_instrumented  
clear fft_instrumented
```

Run the following code to delete the temporary directory.

```
tempdirObj.cleanup;
```

Detect Limit Cycles in Fixed-Point State-Space Systems

This example shows how to analyze a fixed-point state-space system to detect limit cycles.

The example focuses on detecting large scale limit cycles due to overflow with zero inputs and highlights the conditions that are sufficient to prevent such oscillations.

References:

[1] Richard A. Roberts and Clifford T. Mullis, "Digital Signal Processing", Addison-Wesley, Reading, Massachusetts, 1987, ISBN 0-201-16350-0, Section 9.3.

[2] S. K. Mitra, "Digital Signal Processing: A Computer Based Approach", McGraw-Hill, New York, 1998, ISBN 0-07-042953-7.

Select a State-Space Representation of the System.

We observe that the system is stable by observing that the eigenvalues of the state-transition matrix A have magnitudes less than 1.

```
format
A = [0 1; -.5 1]; B = [0; 1]; C = [1 0]; D = 0;
eig(A)
```

```
ans =

    0.5000 + 0.5000i
    0.5000 - 0.5000i
```

Filter Implementation

```
type(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','fisiso
```

```
function [y,z] = fisisostatespacefilter(A,B,C,D,x,z)
```

```
%FISISOSTATESPACEFILTER Single-input, single-output statespace filter
% [Y,Zf] = FISISOSTATESPACEFILTER(A,B,C,D,X,Zi) filters data X with
% initial conditions Zi with the state-space filter defined by matrices
% A, B, C, D. Output Y and final conditions Zf are returned.

% Copyright 2004-2011 The MathWorks, Inc.
% $Revision: 1.1.6.1 $

y = x;
z(:,2:length(x)+1) = 0;
for k=1:length(x)
    y(k) = C*z(:,k) + D*x(k);
    z(:,k+1) = A*z(:,k) + B*x(k);
end
```

Floating-Point Filter

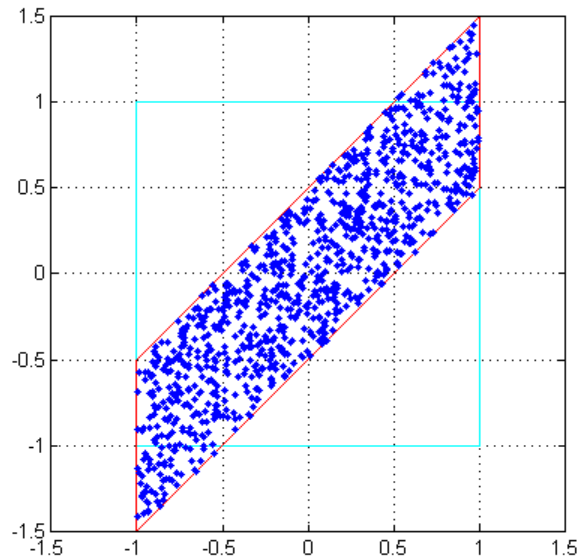
Create a floating-point filter and observe the trajectory of the states.

First, we choose random states within the unit square and observe where they are projected after one step of being multiplied by the state-transition matrix A.

```
rng('default');
clf
x1 = [-1 1 1 -1 -1];
y1 = [-1 -1 1 1 -1];
plot(x1,y1,'c')
axis([-1.5 1.5 -1.5 1.5]); axis square; grid;
hold on

% Plot the projection of the square
p = A*[x1;y1];
plot(p(1,:),p(2:,:), 'r')

r = 2*rand(2,1000)-1;
pr = A*r;
plot(pr(1,:),pr(2:,:), 'r')
```

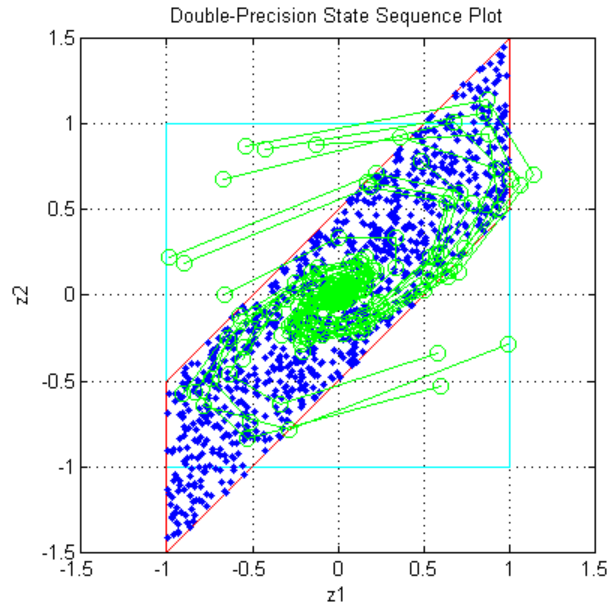


Random Initial States Followed Through Time

Drive the filter with a random initial state, normalized to be inside the unit square, with the input all zero, and run the filter.

Note that some of the states wander outside the unit square, and that they eventually wind down to the zero state at the origin, $z=[0;0]$.

```
x = zeros(10,1);
zi = [0;0];
q = quantizer([16 15]);
for k=1:20
    y = x;
    zi(:) = randquant(q,size(A,1),1);
    [y,zf] = fidemo.fisisostatespacefilter(A,B,C,D,x,zi);
    plot(zf(1,:), zf(2,:), 'go-', 'markersize',8);
end
title('Double-Precision State Sequence Plot');
xlabel('z1'); ylabel('z2')
```



State Trajectory

Because the eigenvalues are less than one in magnitude, the system is stable, and all initial states wind down to the origin with zero input. However, the eigenvalues don't tell the whole story about the trajectory of the states, as in this example, where the states were projected outward first, before they start to contract.

The singular values of A give us a better indication of the overall state trajectory. The largest singular value is about 1.46, which indicates that states aligned with the corresponding singular vector will be projected away from the origin.

`svd(A)`

`ans =`


```
1.4604
0.3424
```

Fixed-Point Filter Creation

Create a fixed-point filter and check for limit cycles.

The MATLAB code for the filter remains the same. It becomes a fixed-point filter because we drive it with fixed-point inputs.

For the sake of illustrating overflow oscillation, we are choosing product and sum data types that will overflow.

```
rng('default');
F = fimath('OverflowAction','Wrap',...
          'ProductMode','SpecifyPrecision',...
          'ProductWordLength',16,'ProductFractionLength',15,...
          'SumMode','SpecifyPrecision',...
          'SumWordLength',16,'SumFractionLength',15);
```

```
A = fi(A,'fimath',F)
B = fi(B,'fimath',F)
C = fi(C,'fimath',F)
D = fi(D,'fimath',F)
```

```
A =
```

```
0    1.0000
-0.5000    1.0000
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 14
```

```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: SpecifyPrecision
```

```

        ProductWordLength: 16
    ProductFractionLength: 15
        SumMode: SpecifyPrecision
        SumWordLength: 16
    SumFractionLength: 15
        CastBeforeSum: true

B =

    0
    1

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
    FractionLength: 14

        RoundingMethod: Nearest
    OverflowAction: Wrap
        ProductMode: SpecifyPrecision
    ProductWordLength: 16
    ProductFractionLength: 15
        SumMode: SpecifyPrecision
        SumWordLength: 16
    SumFractionLength: 15
        CastBeforeSum: true

C =

    1    0

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
    FractionLength: 14

        RoundingMethod: Nearest
    OverflowAction: Wrap
        ProductMode: SpecifyPrecision
    ProductWordLength: 16

```

```

ProductFractionLength: 15
    SumMode: SpecifyPrecision
    SumWordLength: 16
    SumFractionLength: 15
    CastBeforeSum: true

D =

    0

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 15

    RoundingMethod: Nearest
    OverflowAction: Wrap
    ProductMode: SpecifyPrecision
    ProductWordLength: 16
    ProductFractionLength: 15
    SumMode: SpecifyPrecision
    SumWordLength: 16
    SumFractionLength: 15
    CastBeforeSum: true

```

Plot the Projection of the Square in Fixed-Point

Again, we choose random states within the unit square and observe where they are projected after one step of being multiplied by the state-transition matrix A . The difference is that this time matrix A is fixed-point.

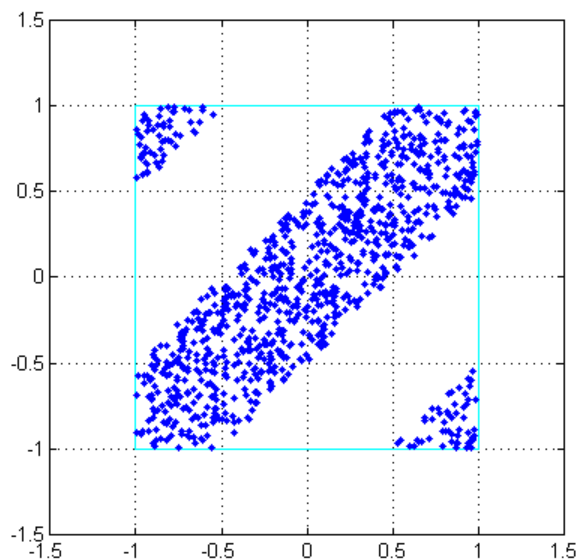
Note that the triangles that projected out of the square before in floating-point, are now wrapped back into the interior of the square.

```

clf
r = 2*rand(2,1000)-1;
pr = A*r;
plot([-1 1 1 -1 -1],[-1 -1 1 1 -1],'c')
axis([-1.5 1.5 -1.5 1.5]); axis square; grid;
hold on

```

```
plot(pr(1,:),pr(2,:),'.')
```

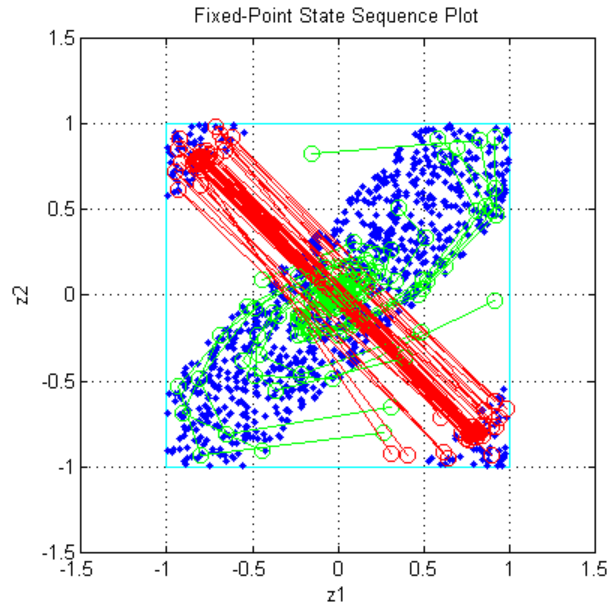


Execute the Fixed-Point Filter.

The only difference between this and the previous code is that we are driving it with fixed-point data types.

```
x = fi(zeros(10,1),1,16,15,'fimath',F);
zi = fi([0;0],1,16,15,'fimath',F);
q = assignmentquantizer(zi);
e = double(eps(zi));
rng('default');
for k=1:20
    y = x;
    zi(:) = randquant(q,size(A,1),1);
    [y,zf] = fidemo.fisisostatespacefilter(A,B,C,D,x,zi);
    if abs(double(zf(end)))>0.5, c='ro-'; else, c='go-'; end
    plot(zf(1,:), zf(2,:),c,'markersize',8);
end
```

```
title('Fixed-Point State Sequence Plot');
xlabel('z1'); ylabel('z2')
```



Trying this for other randomly chosen initial states illustrates that once a state enters one of the triangular regions, then it is projected into the other triangular region, and back and forth, and never escapes.

Sufficient Conditions for Preventing Overflow Limit Cycles

There are two sufficient conditions to prevent overflow limit cycles in a system:

- the system is stable i.e., $\text{abs}(\text{eig}(A)) < 1$,
- the matrix A is normal i.e., $A^*A = A A^*$.

Note that for the current representation, the second condition does not hold.

Apply Similarity Transform to Create a Normal A

We now apply a similarity transformation to the original system that will create a normal state-transition matrix A2.

```
T = [-2 0;-1 1];  
Tinv = [-.5 0;-.5 1];  
A2 = Tinv*A*T; B2 = Tinv*B; C2 = C*T; D2 = D;
```

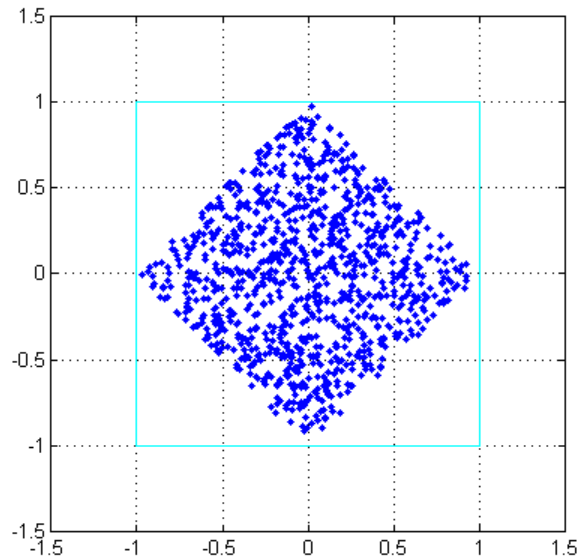
Similarity transformations preserve eigenvalues, as a result of which the system transfer function of the transformed system remains same as before. However, the transformed state transformation matrix A2 is normal.

Check for Limit Cycles on the Transformed System.

Plot the Projection of the Square of the Normal-Form System

Now the projection of random initial states inside the unit square all contract uniformly. This is the result of the state transition matrix A2 being normal. The states are also rotated by 90 degrees counterclockwise.

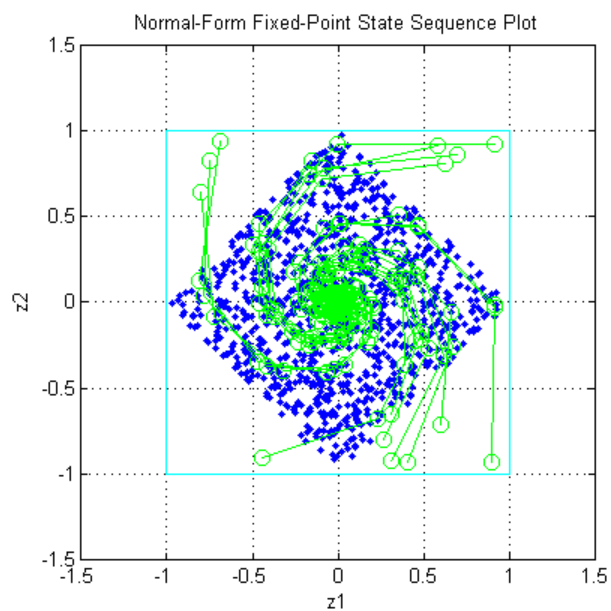
```
clf  
r = 2*rand(2,1000)-1;  
pr = A2*r;  
plot([-1 1 1 -1 -1],[-1 -1 1 1 -1],'c')  
axis([-1.5 1.5 -1.5 1.5]); axis square; grid;  
hold on  
plot(pr(1,:),pr(2,:),'.')
```



Plot the State Sequence

Plotting the state sequences again for the same initial states as before we see that the outputs now spiral towards the origin.

```
x = fi(zeros(10,1),1,16,15,'fimath',F);
zi = fi([0;0],1,16,15,'fimath',F);
q = assignmentquantizer(zi);
e = double(eps(zi));
rng('default');
for k=1:20
    y = x;
    zi(:) = randquant(q,size(A,1),1);
    [y,zf] = fidemo.fisisostatespacefilter(A2,B2,C2,D2,x,zi);
    if abs(double(zf(end)))>0.5, c='ro-'; else, c='go-'; end
    plot(zf(1,:), zf(2,:),c,'markersize',8);
end
title('Normal-Form Fixed-Point State Sequence Plot');
xlabel('z1'); ylabel('z2')
```



Trying this for other randomly chosen initial states illustrates that there is no region from which the filter is unable to recover.

Compute Quantization Error

This example shows how to compute and compare the statistics of the signal quantization error when using various rounding methods.

First, a random signal is created that spans the range of the quantizer.

Next, the signal is quantized, respectively, with rounding methods 'fix', 'floor', 'ceil', 'nearest', and 'convergent', and the statistics of the signal are estimated.

The theoretical probability density function of the quantization error will be computed with ERRPDF, the theoretical mean of the quantization error will be computed with ERRMEAN, and the theoretical variance of the quantization error will be computed with ERRVAR.

Uniformly Distributed Random Signal

First we create a uniformly distributed random signal that spans the domain -1 to 1 of the fixed-point quantizers that we will look at.

```
q = quantizer([8 7]);
r = realmax(q);
u = r*(2*rand(50000,1) - 1);          % Uniformly distributed (-1,1)
xi=linspace(-2*eps(q),2*eps(q),256);
```

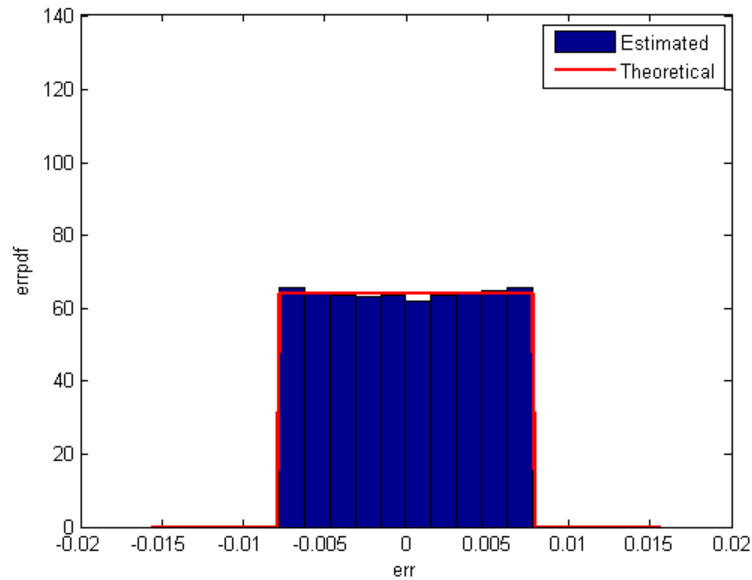
Fix: Round Towards Zero.

Notice that with 'fix' rounding, the probability density function is twice as wide as the others. For this reason, the variance is four times that of the others.

```
q = quantizer('fix',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t = errvar(q);
% Theoretical variance = eps(q)^2 / 3
% Theoretical mean     = 0
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
```

```
Estimated    error variance (dB) = -46.8586
```

```
Theoretical error variance (dB) = -46.9154
Estimated   mean = 7.788e-06
Theoretical mean = 0
```



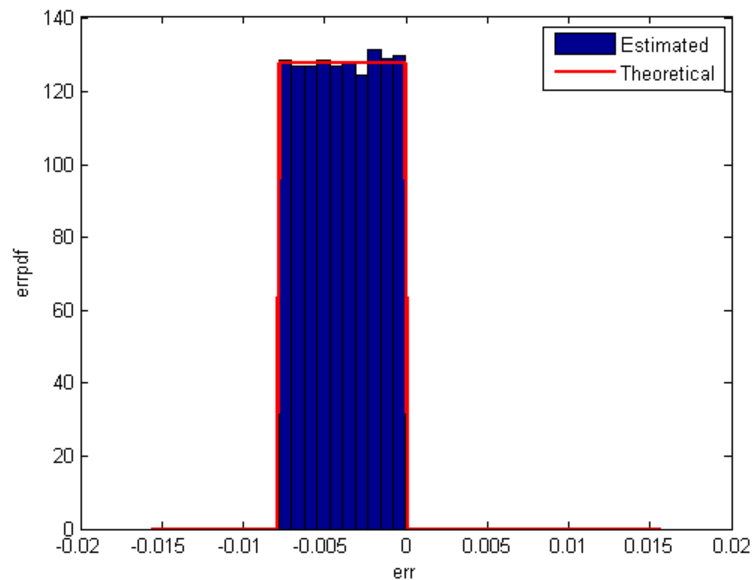
Floor: Round Towards Minus Infinity.

Floor rounding is often called truncation when used with integers and fixed-point numbers that are represented in two's complement. It is the most common rounding mode of DSP processors because it requires no hardware to implement. Floor does not produce quantized values that are as close to the true values as ROUND will, but it has the same variance, and small signals that vary in sign will be detected, whereas in ROUND they will be lost.

```
q = quantizer('floor',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t = errvar(q);
% Theoretical variance = eps(q)^2 / 12
```

```
% Theoretical mean      = -eps(q)/2
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
```

```
Estimated   error variance (dB) = -52.9148
Theoretical error variance (dB) = -52.936
Estimated   mean = -0.0038956
Theoretical mean = -0.0039063
```



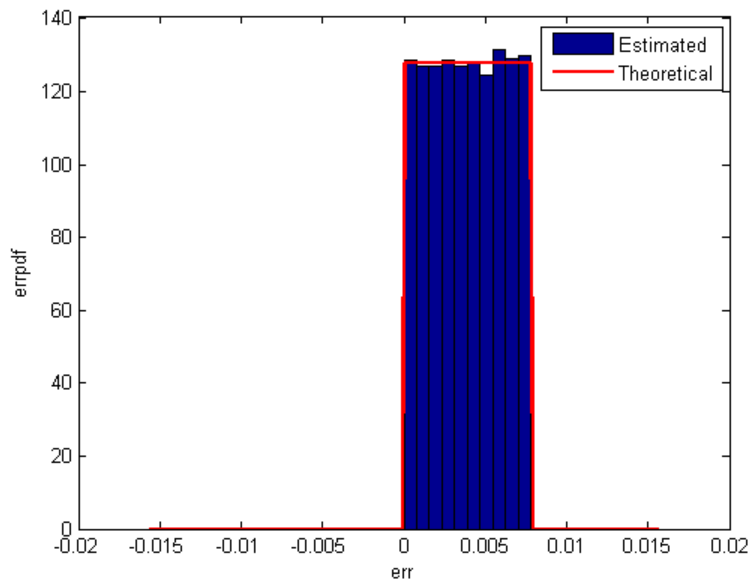
Ceil: Round Towards Plus Infinity.

```
q = quantizer('ceil',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t = errvar(q);
% Theoretical variance = eps(q)^2 / 12
% Theoretical mean      = eps(q)/2
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
```

```

Estimated   error variance (dB) = -52.9148
Theoretical error variance (dB) = -52.936
Estimated   mean = 0.0039169
Theoretical mean = 0.0039063

```



Round: Round to Nearest. In a Tie, Round to Largest Magnitude.

Round is more accurate than floor, but all values smaller than $\text{eps}(q)$ get rounded to zero and so are lost.

```

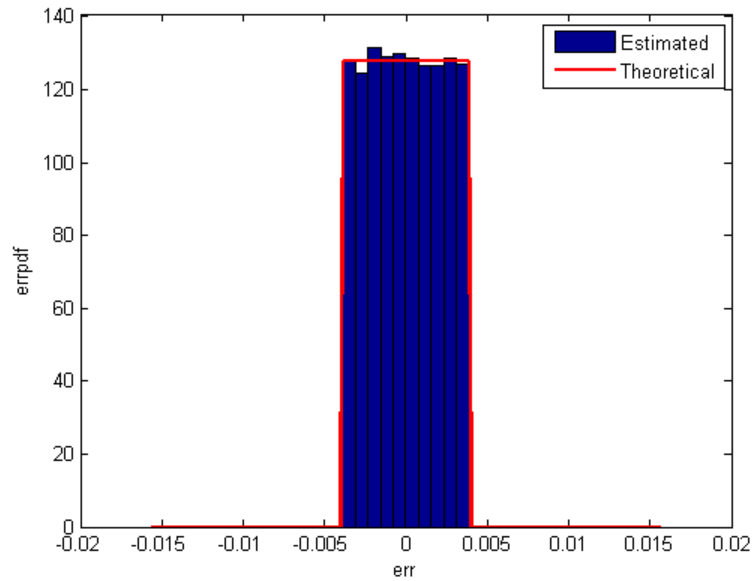
q = quantizer('nearest',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t = errvar(q);
% Theoretical variance =  $\text{eps}(q)^2 / 12$ 
% Theoretical mean = 0
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)

```

```

Estimated   error variance (dB) = -52.9579
Theoretical error variance (dB) = -52.936
Estimated   mean = -2.212e-06
Theoretical mean = 0

```



Convergent: Round to Nearest. In a Tie, Round to Even.

Convergent rounding eliminates the bias introduced by ordinary "round" caused by always rounding the tie in the same direction.

```

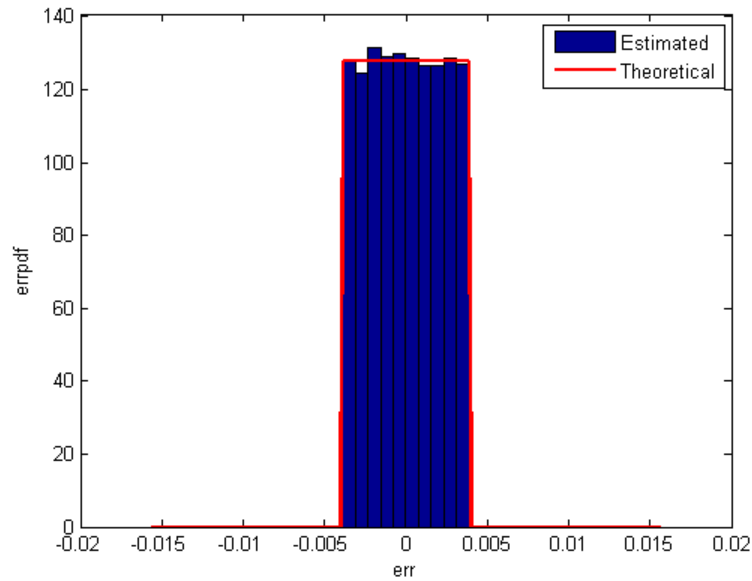
q = quantizer('convergent',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t = errvar(q);
% Theoretical variance = eps(q)^2 / 12
% Theoretical mean = 0
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)

```

```

Estimated   error variance (dB) = -52.9579
Theoretical error variance (dB) = -52.936
Estimated   mean = -2.212e-06
Theoretical mean = 0

```



Comparison of Nearest vs. Convergent

The error probability density function for convergent rounding is difficult to distinguish from that of round-to-nearest by looking at the plot.

The error p.d.f. of convergent is

$f(\text{err}) = 1/\text{eps}(q)$, for $-\text{eps}(q)/2 \leq \text{err} \leq \text{eps}(q)/2$, and 0 otherwise

while the error p.d.f. of round is

$f(\text{err}) = 1/\text{eps}(q)$, for $-\text{eps}(q)/2 < \text{err} \leq \text{eps}(q)/2$, and 0 otherwise

Note that the error p.d.f. of convergent is symmetric, while round is slightly biased towards the positive.

The only difference is the direction of rounding in a tie.

```
x=[ -3.5:3.5]';
[x convergent(x) nearest(x)]
```

```
ans =
```

```

-3.5000   -4.0000   -3.0000
-2.5000   -2.0000   -2.0000
-1.5000   -2.0000   -1.0000
-0.5000         0         0
 0.5000         0    1.0000
 1.5000    2.0000    2.0000
 2.5000    2.0000    3.0000
 3.5000    4.0000    4.0000
```

Plot Helper Function

The helper function that was used to generate the plots in this example is listed below.

```
type(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','qerrorr
```

```
function qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
%QERRORDEMO PLOT Plot function for QERRORDEMO.
%   QERRORDEMO PLOT(Q,F_T,XI,MU_T,V_T,ERR) produces the plot and display us
%   the example function QERRORDEMO, where Q is the quantizer whos attribu
%   being analyzed; F_T is the theoretical quantization error probability
%   density function for quantizer Q computed by ERRPDF; XI is the domain
%   values being evaluated by ERRPDF; MU_T is the theoretical quantization
%   error mean of quantizer Q computed by ERRMEAN; V_T is the theoretical
%   quantization error variance of quantizer Q computed by ERRVAR; and ERR
%   is the error generated by quantizing a random signal by quantizer Q.
%
%   See QERRORDEMO for examples of use.
%
%   Copyright 1999-2012 The MathWorks, Inc.
```

```
v=10*log10(var(err));
disp(['Estimated   error variance (dB) = ',num2str(v)]);
disp(['Theoretical error variance (dB) = ',num2str(10*log10(v_t))]);
disp(['Estimated   mean = ',num2str(mean(err))]);
disp(['Theoretical mean = ',num2str(mu_t)]);
[n,c]=hist(err);
figure(gcf)
bar(c,n/(length(err)*(c(2)-c(1))), 'hist');
line(xi,f_t,'linewidth',2,'color','r');
% Set the ylim uniformly on all plots
set(gca,'ylim',[0 max(errpdf(quantizer(q.format,'nearest'),xi)*1.1)])
legend('Estimated','Theoretical')
xlabel('err'); ylabel('errpdf')
```


Normalize Data for Lookup Tables

This example shows how to normalize data for use in lookup tables.

Lookup tables are a very efficient way to write computationally-intense functions for fixed-point embedded devices. For example, you can efficiently implement logarithm, sine, cosine, tangent, and square-root using lookup tables. You normalize the inputs to these functions to produce a smaller lookup table, and then you scale the outputs by the normalization factor. This example shows how to implement the normalization function that is used in examples [Implement Fixed-Point Square Root Using Lookup Table](#) and [Implement Fixed-Point Log2 Using Lookup Table](#).

Setup

To assure that this example does not change your preferences or settings, this code stores the original state, and you will restore it at the end.

```
originalFormat = get(0, 'format'); format long g
originalWarningState = warning('off','fixed:fi:underflow');
originalFiprefState = fipref; reset(fipref)
```

Function to Normalize Unsigned Data

This algorithm normalizes unsigned data with 8-bit words. Given input $u > 0$, the output x is normalized such that

$$u = x * 2^n$$

where $1 \leq x < 2$ and n is an integer. Note that n may be positive, negative, or zero.

Function `fi_normalize_unsigned_8_bit_word` looks at the 8 most-significant-bits of the input at a time, and left shifts the bits until the most-significant bit is a 1. The number of bits to shift for each 8-bit word is read from the number-of-leading-zeros lookup table, `NLZLUT`.

```
function [x,n] = fi_normalize_unsigned_8_bit_word(u) %#codegen
    assert(isscalar(u),'Input must be scalar');
    assert(all(u>0),'Input must be positive.');
```

```

assert(isfi(u) && isfixed(u), 'Input must be a fi object with fixed-point');
u = removefimath(u);
NLZLUT = number_of_leading_zeros_look_up_table();
word_length = u.WordLength;
u_fraction_length = u.FractionLength;
B = 8;
leftshifts=int8(0);
% Reinterpret the input as an unsigned integer.
T_unsigned_integer = numerictype(0, word_length, 0);
v = reinterpretcast(u,T_unsigned_integer);
F = fimath('OverflowAction','Wrap',...
          'RoundingMethod','Floor',...
          'SumMode','KeepLSB',...
          'SumWordLength',v.WordLength);
v = setfimath(v,F);
% Unroll the loop in generated code so there will be no branching.
for k = coder.unroll(1:ceil(word_length/B))
    % For each iteration, see how many leading zeros are in the high
    % byte of V, and shift them out to the left. Continue with the
    % shifted V for as many bytes as it has.
    %
    % The index is the high byte of the input plus 1 to make it a
    % one-based index.
    index = int32(bitsra(v, word_length - B) + uint8(1));
    % Index into the number-of-leading-zeros lookup table. This lookup
    % table takes in a byte and returns the number of leading zeros in
    % binary representation.
    shiftamount = NLZLUT(index);
    % Left-shift out all the leading zeros in the high byte.
    v = bitsll(v,shiftamount);
    % Update the total number of left-shifts
    leftshifts = leftshifts+shiftamount;
end
% The input has been left-shifted so the most-significant-bit is a 1.
% Reinterpret the output as unsigned with one integer bit, so
% that 1 <= x < 2.
T_x = numerictype(0,word_length,word_length-1);
x = reinterpretcast(v, T_x);
x = removefimath(x);
% Let Q = int(u). Then u = Q*2^(-u_fraction_length),

```

```
% and x = Q*2^leftshifts * 2^(1-word_length). Therefore,
% u = x*2^n, where n is defined as:
n = word_length - u_fraction_length - leftshifts - 1;
end
```

Number-of-Leading-Zeros Lookup Table

Function `number_of_leading_zeros_look_up_table` is used by `fi_normalize_unsigned_8_bit_word` and returns a table of the number of leading zero bits in an 8-bit word.

The first element of NLZLUT is 8 and corresponds to $u=0$. In 8-bit value $u = 00000000_2$, where subscript 2 indicates base-2, there are 8 leading zero bits.

The second element of NLZLUT is 7 and corresponds to $u=1$. There are 7 leading zero bits in 8-bit value $u = 00000001_2$.

And so forth, until the last element of NLZLUT is 0 and corresponds to $u=255$. There are 0 leading zero bits in the 8-bit value $u=11111111_2$.

The NLZLUT table was generated by:

```
>> B = 8; % Number of bits in a byte
>> NLZLUT = int8(B-ceil(log2((1:2^B))))

function NLZLUT = number_of_leading_zeros_look_up_table()
% B = 8; % Number of bits in a byte
% NLZLUT = int8(B-ceil(log2((1:2^B))))
NLZLUT = int8([8 7 6 6 5 5 5 5 ...
4 4 4 4 4 4 4 4 ...
3 3 3 3 3 3 3 3 ...
3 3 3 3 3 3 3 3 ...
2 2 2 2 2 2 2 2 ...
2 2 2 2 2 2 2 2 ...
2 2 2 2 2 2 2 2 ...
2 2 2 2 2 2 2 2 ...
1 1 1 1 1 1 1 1 ...
1 1 1 1 1 1 1 1 ...
1 1 1 1 1 1 1 1 ...
1 1 1 1 1 1 1 1 ...])
```


The high byte is 0 = 00000000_2. Add 1 to make an index out of it: $\text{index} = 0 + 1 = 1$. The number-of-leading-zeros lookup table at index 1 indicates that there are 8 leading zeros: $\text{NLZLUT}(1) = 8$. Left shift by this many bits.

High byte	Low byte	
01001101	00000000	Left-shifted by 8 bits.

Iterate once more to remove the leading zeros from the next byte.

The high byte is 77 = 01001101_2. Add 1 to make an index out of it: $\text{index} = 77 + 1 = 78$. The number-of-leading-zeros lookup table at index 78 indicates that there is 1 leading zero: $\text{NLZLUT}(78) = 1$. Left shift by this many bits.

High byte	Low byte	
100110100	00000000	Left-shifted by 1 additional bit, for a total of 9.

Reinterpret these bits as unsigned fixed-point with 15 fractional bits.

$x = 1.001101000000000_2 = 1.203125$

The value for n is the word-length of u , minus the fraction length of u , minus the number of left shifts, minus 1.

$n = 16 - 8 - 9 - 1 = -2$.

And so your result is:

$[x, n] = \text{fi_normalize_unsigned_8_bit_word}(u)$

$x =$

1.203125

DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 16
FractionLength: 15

$n =$

-2

Comparing binary values, you can see that `x` has the same bits as `u`, left-shifted by 9 bits.

```
binary_representation_of_u = bin(u)
binary_representation_of_x = bin(x)
```

```
binary_representation_of_u =
```

```
0000000001001101
```

```
binary_representation_of_x =
```

```
1001101000000000
```

Cleanup

Restore original state.

```
set(0, 'format', originalFormat);
warning(originalWarningState);
fipref(originalFiprefState);
```

Implement Fixed-Point Log2 Using Lookup Table

This example shows how to implement fixed-point log2 using a lookup table. Lookup tables generate efficient code for embedded devices.

Setup

To assure that this example does not change your preferences or settings, this code stores the original state, and you will restore it at the end.

```
originalFormat = get(0, 'format'); format long g
originalWarningState = warning('off','fixed:fi:underflow');
originalFiprefState = fipref; reset(fipref)
```

Log2 Implementation

The log2 algorithm is summarized here.

- 1** Declare the number of bits in a byte, B, as a constant. In this example, B=8.
- 2** Use function `fi_normalize_unsigned_8_bit_word()` described in example Normalize Data for Lookup Tables to normalize the input $u > 0$ such that $u = x * 2^n$ and $1 \leq x < 2$.
- 3** Extract the upper B-bits of x. Let x_B denote the upper B-bits of x.
- 4** Generate lookup table, LOG2LUT, such that the integer $i = \text{uint8}(x_B) - 2^{(B-1)} + 1$ is used as an index to LOG2LUT so that $\log_2(x_B)$ can be evaluated by looking up the index $\log_2(x_B) = \text{LOG2LUT}(i)$.
- 5** Use the remainder, $r = x - x_B$, interpreted as a fraction, to linearly interpolate between $\text{LOG2LUT}(i)$ and the next value in the table $\text{LOG2LUT}(i+1)$. The remainder, r, is created by extracting the lower $w - B$ bits of x, where w denotes the word length of x. It is interpreted as a fraction by using function `reinterpretcast()`.
- 6** Finally, compute the output using the lookup table and linear interpolation:

$$\begin{aligned} \log_2(u) &= \log_2(x * 2^n) \\ &= n + \log_2(x) \end{aligned}$$

```

        = n + LOG2LUT( i ) + r * ( LOG2LUT( i+1 ) - LOG2LUT( i ) )

function y = fi_log2lookup_8_bit_word(u) %#codegen
    % Load the lookup table
    LOG2LUT = log2_lookup_table();
    % Remove fimath from the input to insulate this function from math
    % settings declared outside this function.
    u = removefimath(u);
    % Declare the output
    y = eml.nullcopy(fi(zeros(size(u)), numerictype(LOG2LUT), fimath(LOG2LUT)));
    B = 8; % Number of bits in a byte
    w = u.WordLength;
    for k = 1:prod(size(u))
        assert(u(k)>0, 'Input must be positive. ');
        % Normalize the input such that u = x * 2^n and 1 <= x < 2
        [x,n] = fi_normalize_unsigned_8_bit_word(u(k));
        % Extract the high byte of x
        high_byte = uint8( storedInteger(bitsliceget(x, w, w - B + 1)) );
        % Convert the high byte into an index for LOG2LUT
        i = high_byte - 2^(B-1) + 1;
        % Interpolate between points.
        % The upper byte was used for the index into LOG2LUT
        % The remaining bits make up the fraction between points.
        T_unsigned_fraction = numerictype(0, w-B, w-B);
        r = reinterpretcast(bitsliceget(x,w-B,1), T_unsigned_fraction);
        y(k) = n + LOG2LUT(i) + ...
            r*(LOG2LUT(i+1) - LOG2LUT(i)) ;
    end
    % Remove fimath from the output to insulate the caller from math settings
    % declared inside this function.
    y = removefimath(y);
end

```

Log2 Lookup Table

Function `log2_lookup_table` loads the lookup table of `log2` values. You can create the table by running:

```

B = 8;
log2_table = log2((2^(B-1) : 2^B) / 2^(B - 1))

```



```

function LOG2LUT = log2_lookup_table()
    B = 8; % Number of bits in a byte
    % log2_table = log2((2^(B-1) : 2^(B)) / 2^(B - 1))
    log2_table = [0.000000000000000 0.011227255423254 0.022367813028454
                  0.044394119358453 0.055282435501190 0.066089190457773
                  0.087462841250339 0.098032082960527 0.108524456778169
                  0.129283016944966 0.139551352398794 0.149747119504682
                  0.169925001442312 0.179909090014934 0.189824558880017
                  0.209453365628950 0.219168520462162 0.228818690495881
                  0.247927513443586 0.257387842692652 0.266786540694901
                  0.285402218862248 0.294620748891627 0.303780748177103
                  0.321928094887362 0.330916878114617 0.339850002884625
                  0.357552004618084 0.366322214245816 0.375039431346925
                  0.392317422778760 0.400879436282184 0.409390936137702
                  0.426264754702098 0.434628227636725 0.442943495848728
                  0.459431618637297 0.467605550082997 0.475733430966398
                  0.491853096329675 0.499845887083205 0.507794640198696
                  0.523561956057013 0.531381460516312 0.539158811108031
                  0.554588851677637 0.562242424221073 0.569855608330948
                  0.584962500721156 0.592457037268080 0.599912842187128
                  0.614709844115208 0.622051819456376 0.629356620079610
                  0.643856189774725 0.651051691178929 0.658211482751795
                  0.672425341971496 0.679480099505446 0.686500527183218
                  0.700439718141092 0.707359132080883 0.714245517666123
                  0.727920454563199 0.734709620225838 0.741466986401147
                  0.754887502163469 0.761551232444479 0.768184324776926
                  0.781359713524660 0.787902559391432 0.794415866350106
                  0.807354922057604 0.813781191217037 0.820178962415188
                  0.832890014164742 0.839203788096944 0.845490050944375
                  0.857980995127572 0.864186144654280 0.870364719583405
                  0.882643049361841 0.888743248898259 0.894817763307943
                  0.906890595608518 0.912889336229962 0.918863237274595
                  0.930737337562886 0.936637939002571 0.942514505339240
                  0.954196310386875 0.960001932068081 0.965784284662087
                  0.977279923499916 0.982993574694310 0.988684686772166
                  1.000000000000000];

    % Cast to fixed point with the most accurate rounding method
    WL = 4*B; % Word length

```

```
FL = 2*B; % Fraction length
LOG2LUT = fi(log2_table,1,WL,FL,'RoundingMethod','Nearest');
% Set fimath for the most efficient math operations
F = fimath('OverflowAction','Wrap',...
          'RoundingMethod','Floor',...
          'SumMode','SpecifyPrecision',...
          'SumWordLength',WL,...
          'SumFractionLength',FL,...
          'ProductMode','SpecifyPrecision',...
          'ProductWordLength',WL,...
          'ProductFractionLength',2*FL);
LOG2LUT = setfimath(LOG2LUT,F);
end
```

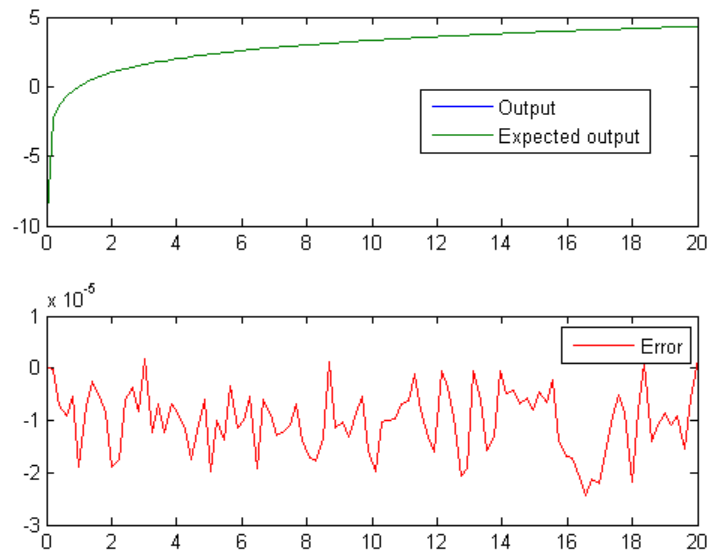
Example

```
u = fi(linspace(0.001,20,100));

y = fi_log2lookup_8_bit_word(u);

y_expected = log2(double(u));
%%3
clf
subplot(211)
plot(u,y,u,y_expected)
legend('Output','Expected output','Location','Best')

subplot(212)
plot(u,double(y)-y_expected,'r')
legend('Error')
figure(gcf)
```



Cleanup

Restore original state.

```
set(0, 'format', originalFormat);  
warning(originalWarningState);  
fipref(originalFiprefState);
```

Implement Fixed-Point Square Root Using Lookup Table

This example shows how to implement fixed-point square root using a lookup table. Lookup tables generate efficient code for embedded devices.

Setup

To assure that this example does not change your preferences or settings, this code stores the original state, and you will restore it at the end.

```
originalFormat = get(0, 'format'); format long g
originalWarningState = warning('off','fixed:fi:underflow');
originalFiprefState = fipref; reset(fipref)
```

Square Root Implementation

The square root algorithm is summarized here.

- 1** Declare the number of bits in a byte, B , as a constant. In this example, $B=8$.
- 2** Use function `fi_normalize_unsigned_8_bit_word()` described in example Normalize Data for Lookup Tables to normalize the input $u>0$ such that $u = x * 2^n$, $0.5 \leq x < 2$, and n is even.
- 3** Extract the upper B -bits of x . Let x_B denote the upper B -bits of x .
- 4** Generate lookup table, `SQRTLUT`, such that the integer $i = \text{uint8}(x_B) - 2^{(B-2)} + 1$ is used as an index to `SQRTLUT` so that `sqrt(x_B)` can be evaluated by looking up the index `sqrt(x_B) = SQRTLUT(i)`.
- 5** Use the remainder, $r = x - x_B$, interpreted as a fraction, to linearly interpolate between `SQRTLUT(i)` and the next value in the table `SQRTLUT(i+1)`. The remainder, r , is created by extracting the lower $w - B$ bits of x , where w denotes the word-length of x . It is interpreted as a fraction by using function `reinterpretcast()`.
- 6** Finally, compute the output using the lookup table and linear interpolation:

```
sqrt( u ) = sqrt( x * 2^n )
           = sqrt(x) * 2^(n/2)
```

$$= (\text{SQRTLUT}(i) + r * (\text{SQRTLUT}(i+1) - \text{SQRTLUT}(i))) * 2^n$$

```

function y = fi_sqrtlookup_8_bit_word(u) %#codegen
    % Load the lookup table
    SQRTLUT = sqrt_lookup_table();
    % Remove fimath from the input to insulate this function from math
    % settings declared outside this function.
    u = removefimath(u);
    % Declare the output
    y = coder.nullcopy(fi(zeros(size(u)), numerictype(SQRTLUT), fimath(SQRTLUT)));
    B = 8; % Number of bits in a byte
    w = u.WordLength;
    for k = 1:prod(size(u))
        assert(u(k)>=0,'Input must be non-negative.');
```

$$\text{if } u(k)=0$$

```

            y(k)=0;
        else
            % Normalize the input such that u = x * 2^n and 0.5 <= x < 2
            [x,n] = fi_normalize_unsigned_8_bit_word(u(k));
            isodd = int8(storedInteger(bitand(fi(1,1,8,0),fi(n))));
            x = bitsra(x,isodd);
            n = n + isodd;
            % Extract the high byte of x
            high_byte = uint8( storedInteger(bitsliceget(x, w, w - B + 1)) );
            % Convert the high byte into an index for SQRTLUT
            i = high_byte - 2^(B-2) + 1;
            % The upper byte was used for the index into SQRTLUT.
            % The remainder, r, interpreted as a fraction, is used to
            % linearly interpolate between points.
            T_unsigned_fraction = numerictype(0, w-B, w-B);
            r = reinterpretcast(bitsliceget(x,w-B,1), T_unsigned_fraction);
            y(k) = bitshift((SQRTLUT(i) + r*(SQRTLUT(i+1) - SQRTLUT(i))),...
                            bitsra(n,1));
        end
    end
    % Remove fimath from the output to insulate the caller from math settings
    % declared inside this function.
    y = removefimath(y);
end
```

Square Root Lookup Table

Function `sqrt_lookup_table` loads the lookup table of square-root values.
You can create the table by running:

```
sqrt_table = sqrt( (2^(B-2):2^(B))/2^(B-1) );

function SQRTLUT = sqrt_lookup_table()
    B = 8; % Number of bits in a byte
    % sqrt_table = sqrt( (2^(B-2):2^(B))/2^(B-1) )
    sqrt_table = [0.707106781186548    0.712609640686961    0.718070330817254
                  0.728868986855663    0.734208757779421    0.739509972887452
                  0.750000000000000    0.755190373349661    0.760345316287277
                  0.770551750371122    0.775604602874429    0.780624749799800
                  0.790569415042095    0.795495128834866    0.800390529679106
                  0.810092587300983    0.814900300650331    0.819679815537750
                  0.829156197588850    0.833854004007896    0.838525491562421
                  0.847791247890659    0.852386356061616    0.856956825050130
                  0.866025403784439    0.870524267324007    0.875000000000000
                  0.883883476483184    0.888291900221993    0.892678553567856
                  0.901387818865997    0.905711046636840    0.910013736160065
                  0.918558653543692    0.922801441264588    0.927024810886958
                  0.935414346693485    0.939581023648307    0.943729304408844
                  0.951971638232989    0.956066158798647    0.960143218483576
                  0.968245836551854    0.972271824131503    0.976281209488332
                  0.984250984251476    0.988211768802619    0.992156741649222
                  1.000000000000000    1.003898650263063    1.007782218537319
                  1.015504800579495    1.019344151893756    1.023169096484056
                  1.030776406404415    1.034559084827928    1.038327982864759
                  1.045825033167594    1.049553476484167    1.053268721647045
                  1.060660171779821    1.064336647870400    1.068000468164691
                  1.075290658380328    1.078917281352004    1.082531754730548
                  1.089724735885168    1.093303480283494    1.096870548424015
                  1.103970108290981    1.107502821666834    1.111024302164449
                  1.118033988749895    1.121522402807898    1.125000000000000
                  1.131923142267177    1.135368882786559    1.138804197393037
                  1.145643923738960    1.149048519428140    1.152443057161611
                  1.159202311936963    1.162567202358642    1.165922381636102
                  1.172603939955857    1.175930482639174    1.179247641507075
                  1.185854122563142    1.189143599402528    1.192424001771182]
```

```

1.198957880828180    1.202211503854459    1.205456345124119
1.211919964354082    1.215138880951474    1.218349293101120
1.224744871391589    1.227930169024281    1.231107225224513
1.237436867076458    1.240589577579950    1.243734296383275
1.250000000000000    1.253121103485214    1.256234452640111
1.262438117295260    1.265528545707287    1.268611445636527
1.274754878398196    1.277815518766305    1.280868845744950
1.286953767623375    1.289985465034393    1.293010054098575
1.299038105676658    1.302041665999979    1.305038313613819
1.311011060212689    1.313987252601790    1.316956719106592
1.322875655532295    1.325825214724777    1.328768226591831
1.334634781503914    1.337558409939543    1.340475661845451
1.346291201783626    1.349189571557681    1.352081728298996
1.357847561400027    1.360721316067327    1.363589014329464
1.369306393762915    1.372156150006259    1.375000000000000
1.380670127148408    1.383496476323666    1.386317063301177
1.391941090707505    1.394744600276337    1.397542485937369
1.403121520040228    1.405902734900249    1.408678458698081
1.414213562373095];
% Cast to fixed point with the most accurate rounding method
WL = 4*B; % Word length
FL = 2*B; % Fraction length
SQRTLUT = fi(sqrt_table, 1, WL, FL, 'RoundingMethod','Nearest');
% Set fimath for the most efficient math operations
F = fimath('OverflowAction','Wrap',...
    'RoundingMethod','Floor',...
    'SumMode','KeepLSB',...
    'SumWordLength',WL,...
    'ProductMode','KeepLSB',...
    'ProductWordLength',WL);
SQRTLUT = setfimath(SQRTLUT, F);
end

```

Example

```

u = fi(linspace(0,128,1000),0,16,12);

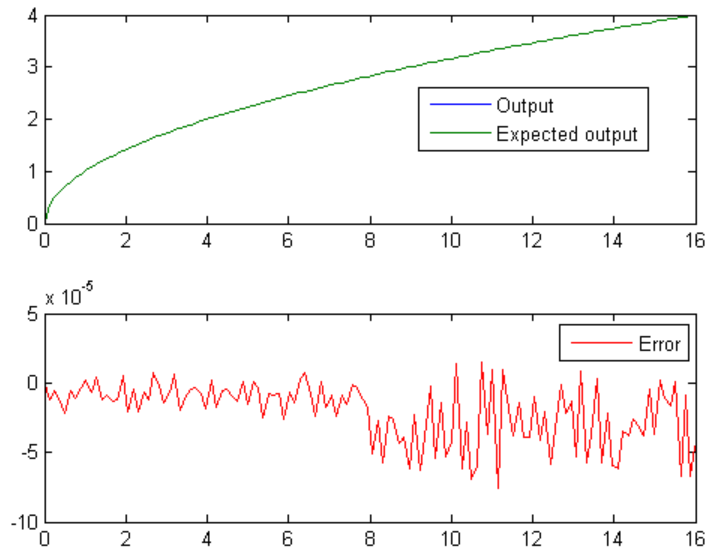
y = fi_sqrtlookup_8_bit_word(u);

y_expected = sqrt(double(u));

```

```
clf
subplot(211)
plot(u,y,u,y_expected)
legend('Output','Expected output','Location','Best')

subplot(212)
plot(u,double(y)-y_expected,'r')
legend('Error')
figure(gcf)
```



Cleanup

Restore original state.

```
set(0, 'format', originalFormat);
warning(originalWarningState);
fipref(originalFiprefState);
```


Set Fixed-Point Math Attributes

This example shows how to set fixed point math attributes in MATLAB code.

You can control fixed-point math attributes for assignment, addition, subtraction, and multiplication using the `fimath` object. You can attach a `fimath` object to a `fi` object using `setfimath`. You can remove a `fimath` object from a `fi` object using `removefimath`.

You can generate C code from the examples if you have MATLAB Coder™ software.

Set and Remove Fixed Point Math Attributes

You can write functions that control their own fixed-point math attributes without being affected by `globalfimath` and `fimath` objects attached to input variables. You can also return from functions with no `fimath` attached to output variables. This gives you local control over fixed-point math settings without interfering with the settings in other functions.

MATLAB Code

```
function y = user_written_sum(u)
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB',...
        'SumWordLength',32);
    u = setfimath(u,F);
    y = fi(0,true,32,get(u,'FractionLength'),F);
    % Algorithm
    for i=1:length(u)
        y(:) = y + u(i);
    end
    % Cleanup
    y = removefimath(y);
end
```

Output has no Attached FIMATH

When you run the code, the `fimath` controls the arithmetic inside the function, but the return value has no attached `fimath`. This is due to the use of `setfimath` and `removefimath` inside the function `user_written_sum`.

```
>> u = fi(1:10,true,16,11);
>> y = user_written_sum(u)

y =
    55
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 32
      FractionLength: 11
```

Generated C Code

If you have MATLAB Coder software, you can generate C code using the following commands.

```
>> u = fi(1:10,true,16,11);
>> codegen user_written_sum -args {u} -config:lib -launchreport
```

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
int32_T user_written_sum(const int16_T u[10])
{
    int32_T y;
    int32_T i;
    /* Setup */
    y = 0;
    /* Algorithm */
    for (i = 0; i < 10; i++) {
        y += u[i];
    }
    /* Cleanup */
    return y;
}
```

Mismatched FIMATH

When you operate on `fi` objects, their `fimath` properties must be equal, or you get an error.

```
>> A = fi(pi,'ProductMode','KeepLSB');
>> B = fi(2,'ProductMode','SpecifyPrecision');
>> C = A * B
```

```
Error using embedded.fi/mtimes
The embedded.fimath of both operands must be equal.
```

To avoid this error, you can remove `fimath` from one of the variables in the expression. In this example, the `fimath` is removed from `B` in the context of the expression without modifying `B` itself, and the product is computed using the `fimath` attached to `A`.

```
>> C = A * removefimath(B)
```

```
C =
```

```
6.283203125
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 26

RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: KeepLSB
ProductWordLength: 32
SumMode: FullPrecision
```

Changing FIMATH on Temporary Variables

If you have variables with no attached `fimath`, but you want to control a particular operation, then you can attach a `fimath` in the context of the expression without modifying the variables.

For example, the product is computed with the `fimath` defined by `F`.

```
>> F = fimath('ProductMode','KeepLSB','OverflowAction','Wrap','RoundingMeth
```

```
>> A = fi(pi);
>> B = fi(2);
>> C = A * setfimath(B,F)

C =

    6.2832

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 26

    RoundingMethod: Floor
    OverflowAction: Wrap
    ProductMode: KeepLSB
    ProductWordLength: 32
    SumMode: FullPrecision
    MaxSumWordLength: 128
```

Note that variable B is not changed.

```
>> B

B =

    2

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 13
```

Removing FIMATH Conflict in a Loop

You can compute products and sums to match the accumulator of a DSP with floor rounding and wrap overflow, and use nearest rounding and saturate overflow on the output. To avoid mismatched fimath errors, you can remove the fimath on the output variable when it is used in a computation with the other variables.

MATLAB Code

In this example, the products are 32-bits, and the accumulator is 40-bits, keeping the least-significant-bits with floor rounding and wrap overflow like C's native integer rules. The output uses nearest rounding and saturate overflow.

```
function [y,z] = setfimath_removefimath_in_a_loop(b,a,x,z)
    % Setup
    F_floor = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'ProductMode','KeepLSB',...
        'ProductWordLength',32,...
        'SumMode','KeepLSB',...
        'SumWordLength',40);
    F_nearest = fimath('RoundingMethod','Nearest',...
        'OverflowAction','Wrap');
    % Set fimaths that are local to this function
    b = setfimath(b,F_floor);
    a = setfimath(a,F_floor);
    x = setfimath(x,F_floor);
    z = setfimath(z,F_floor);
    % Create y with nearest rounding
    y = coder.nullcopy(fi(zeros(size(x)),true,16,14,F_nearest));
    % Algorithm
    for j=1:length(x)
        % Nearest assignment into y
        y(j) = b(1)*x(j) + z(1);
        % Remove y's fimath conflict with other fimaths
        z(1) = (b(2)*x(j) + z(2)) - a(2) * removefimath(y(j));
        z(2) = b(3)*x(j) - a(3) * removefimath(y(j));
    end
    % Cleanup: Remove fimath from outputs
    y = removefimath(y);
    z = removefimath(z);
end
```

Code Generation Instructions

If you have MATLAB Coder software, you can generate C code with the specified hardware characteristics using the following commands.

```
N = 256;
t = 1:N;
xstep = [ones(N/2,1); -ones(N/2,1)];
num = [0.0299545822080925    0.0599091644161849    0.0299545822080925];
den = [1                    -1.4542435862515900    0.5740619150839550];

b = fi(num,true,16);
a = fi(den,true,16);
x = fi(xstep,true,16,15);
zi = fi(zeros(2,1),true,16,14);

B = coder.Constant(b);
A = coder.Constant(a);

config_obj = coder.config('lib');
config_obj.GenerateReport = true;
config_obj.LaunchReport = true;
config_obj.TargetLang = 'C';
config_obj.GenerateComments = true;
config_obj.GenCodeOnly = true;
config_obj.HardwareImplementation.ProdBitPerChar=8;
config_obj.HardwareImplementation.ProdBitPerShort=16;
config_obj.HardwareImplementation.ProdBitPerInt=32;
config_obj.HardwareImplementation.ProdBitPerLong=40;

codegen -config config_obj setfimath_removefimath_in_a_loop -args {B,A,x,zi}
```

Generated C Code

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
void setfimath_removefimath_in_a_loop(const int16_T x[256], int16_T z[2],
    int16_T y[256])
{
    int32_T j;
```

```

int40_T i0;
int16_T b_y;

/* Setup */
/* Set fimaths that are local to this function */
/* Create y with nearest rounding */
/* Algorithm */
for (j = 0; j < 256; j++) {
    /* Nearest assignment into y */
    i0 = 15705 * x[j] + ((int40_T)z[0] << 20);
    b_y = (int16_T)((int32_T)(i0 >> 20) + ((i0 & 524288L) != 0L));

    /* Remove y's fimath conflict with other fimaths */
    z[0] = (int16_T)(((31410 * x[j] + ((int40_T)z[1] << 20)) - ((int40_T)(-
        * b_y) << 6)) >> 20);
    z[1] = (int16_T)((15705 * x[j] - ((int40_T)(9405 * b_y) << 6)) >> 20);
    y[j] = b_y;
}

/* Cleanup: Remove fimath from outputs */
}

```

Polymorphic Code

You can write MATLAB code that can be used for both floating-point and fixed-point types using `setfimath` and `removefimath`.

```

function y = user_written_function(u)
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB');
    u = setfimath(u,F);
    % Algorithm
    y = u + u;
    % Cleanup
    y = removefimath(y);
end

```

Fixed Point Inputs

When the function is called with fixed-point inputs, then `fimath` `F` is used for the arithmetic, and the output has no attached `fimath`.

```
>> u = fi(pi/8,true,16,15,'RoundingMethod','Convergent');  
>> y = user_written_function(u)
```

```
y =
```

```
0.785400390625
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 32  
FractionLength: 15
```

Generated C Code for Fixed Point

If you have MATLAB Coder software, you can generate C code using the following commands.

```
>> u = fi(pi/8,true,16,15,'RoundingMethod','Convergent');  
>> codegen user_written_function -args {u} -config:lib -launchreport
```

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
int32_T user_written_function(int16_T u)  
{  
    /* Setup */  
    /* Algorithm */  
    /* Cleanup */  
    return u + u;  
}
```

Double Inputs

Since `setfimath` and `removefimath` are pass-through for floating-point types, the `user_written_function` example works with floating-point types, too.

```
function y = user_written_function(u)
```



```

% Setup
F = fimath('RoundingMethod','Floor',...
    'OverflowAction','Wrap',...
    'SumMode','KeepLSB');
u = setfimath(u,F);
% Algorithm
y = u + u;
% Cleanup
y = removefimath(y);
end

```

Generated C Code for Double

When compiled with floating-point input, you get the following generated C code.

```

>> codegen user_written_function -args {0} -config:lib -launchreport

real_T user_written_function(real_T u)
{
    return u + u;
}

```

Where the `real_T` type is defined as a double:

```
typedef double real_T;
```

More Polymorphic Code

This function is written so that the output is created to be the same type as the input, so both floating-point and fixed-point can be used with it.

```

function y = user_written_sum_polymorphic(u)
% Setup
F = fimath('RoundingMethod','Floor',...
    'OverflowAction','Wrap',...
    'SumMode','KeepLSB',...
    'SumWordLength',32);

    u = setfimath(u,F);

```

```
    if isfi(u)
        y = fi(0,true,32,get(u,'FractionLength'),F);
    else
        y = zeros(1,1,class(u));
    end

    % Algorithm
    for i=1:length(u)
        y(:) = y + u(i);
    end

    % Cleanup
    y = removefimath(y);

end
```

Fixed Point Generated C Code

If you have MATLAB Coder software, you can generate fixed-point C code using the following commands.

```
>> u = fi(1:10,true,16,11);
>> codegen user_written_sum_polymorphic -args {u} -config:lib -launchreport
```

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
int32_T user_written_sum_polymorphic(const int16_T u[10])
{
    int32_T y;
    int32_T i;

    /* Setup */
    y = 0;

    /* Algorithm */
    for (i = 0; i < 10; i++) {
        y += u[i];
    }
}
```

```

    /* Cleanup */
    return y;
}

```

Floating Point Generated C Code

If you have MATLAB Coder software, you can generate floating-point C code using the following commands.

```

>> u = 1:10;
>> codegen user_written_sum_polymorphic -args {u} -config:lib -launchreport

```

```

real_T user_written_sum_polymorphic(const real_T u[10])
{
    real_T y;
    int32_T i;

    /* Setup */
    y = 0.0;

    /* Algorithm */
    for (i = 0; i < 10; i++) {
        y += u[i];
    }

    /* Cleanup */
    return y;
}

```

Where the `real_T` type is defined as a double:

```

typedef double real_T;

```

SETFIMATH on Integer Types

Following the established pattern of treating built-in integers like `fi` objects, `setfimath` converts integer input to the equivalent `fi` with attached `fimath`.

```

>> u = int8(5);
>> codegen user_written_u_plus_u -args {u} -config:lib -launchreport

function y = user_written_u_plus_u(u)

```

```
% Setup
F = fimath('RoundingMethod','Floor',...
    'OverflowAction','Wrap',...
    'SumMode','KeepLSB',...
    'SumWordLength',32);
u = setfimath(u,F);
% Algorithm
y = u + u;
% Cleanup
y = removefimath(y);
end
```

The output type was specified by the `fimath` to be 32-bit.

```
int32_T user_written_u_plus_u(int8_T u)
{
    /* Setup */
    /* Algorithm */
    /* Cleanup */
    return u + u;
}
```

Working with fimath Objects

- “fimath Object Construction” on page 4-2
- “fimath Object Properties” on page 4-6
- “fimath Properties Usage for Fixed-Point Arithmetic” on page 4-11
- “fimath for Rounding and Overflow Modes” on page 4-20
- “fimath for Sharing Arithmetic Rules” on page 4-22
- “fimath ProductMode and SumMode” on page 4-25

fimath Object Construction

In this section...
“fimath Object Syntaxes” on page 4-2
“Building fimath Object Constructors in a GUI” on page 4-4

fimath Object Syntaxes

The arithmetic attributes of a `fi` object are defined by a local `fimath` object, which is attached to that `fi` object. If a `fi` object has no local `fimath`, the following default `fimath` values are used:

```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

You can create `fimath` objects in Fixed-Point Toolbox software in one of two ways:

- You can use the `fimath` constructor function to create new `fimath` objects.
- You can use the `fimath` constructor function to copy an existing `fimath` object.

To get started, type

```
F = fimath
```

to create a `fimath` object.

```
F =
```

```
RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

To copy a `fimath` object, simply use assignment as in the following example:

```
F = fimath;  
G = F;  
isequal(F,G)  
  
ans =  
  
    1
```

The syntax

```
F = fimath(...'PropertyName',PropertyValue...)
```

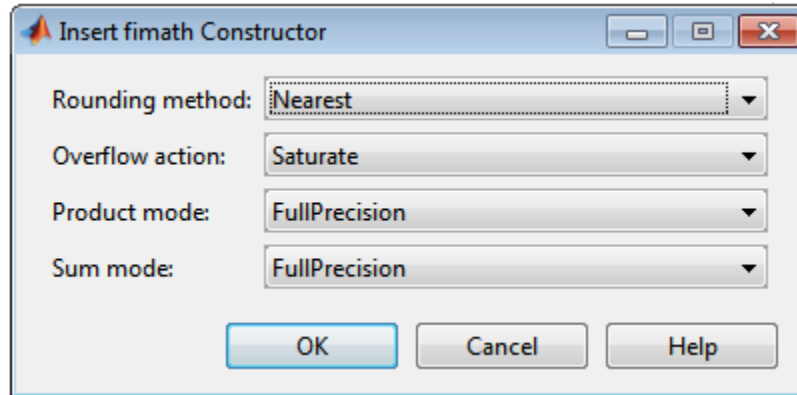
allows you to set properties for a `fimath` object at object creation with property name/property value pairs. Refer to “Setting fimath Properties at Object Creation” on page 4-7.

Building fimath Object Constructors in a GUI

When you are working with files in MATLAB, you can build your `fimath` object constructors using the **Insert fimath Constructor** dialog box. After specifying the properties of the `fimath` object in the dialog box, you can insert the prepopulated `fimath` object constructor string at a specific location in your file.

For example, to create a `fimath` object that uses convergent rounding and wraps on overflow, perform the following steps:

- 1 Open the **Insert fimath Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert fimath Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the `fimath` object:
 - **Rounding method** = Floor
 - **Overflow action** = Wrap
 - **Product mode** = FullPrecision
 - **Sum mode** = FullPrecision



- 3 To insert the `fimath` object constructor string in your file, place your cursor at the desired location in the file. Then click **OK** on the **Insert fimath Constructor** dialog box. Clicking **OK** closes the **Insert fimath Constructor** dialog box and automatically populates the `fimath` object constructor string in your file:

```
6 F = fimath('RoundMode', 'Floor', ...
7         'OverflowMode', 'Wrap', ...
8         'ProductMode', 'FullPrecision', ...
9         'MaxProductWordLength', 128, ...
10        'SumMode', 'FullPrecision', ...
11        'MaxSumWordLength', 128, ...
12        'CastBeforeSum', true)
```

fimath Object Properties

In this section...
“Math, Rounding, and Overflow Properties” on page 4-6
“Setting fimath Object Properties” on page 4-7

Math, Rounding, and Overflow Properties

You can always write to the following properties of `fimath` objects:

Property	Description
<code>CastBeforeSum</code>	Whether both operands are cast to the sum data type before addition
<code>MaxProductWordLength</code>	Maximum allowable word length for the product data type
<code>MaxSumWordLength</code>	Maximum allowable word length for the sum data type
<code>OverflowAction</code>	Action to take on overflow
<code>ProductBias</code>	Bias of the product data type
<code>ProductFixedExponent</code>	Fixed exponent of the product data type
<code>ProductFractionLength</code>	Fraction length, in bits, of the product data type
<code>ProductMode</code>	Defines how the product data type is determined
<code>ProductSlope</code>	Slope of the product data type
<code>ProductSlopeAdjustmentFactor</code>	Slope adjustment factor of the product data type
<code>ProductWordLength</code>	Word length, in bits, of the product data type
<code>RoundingMethod</code>	Rounding method

Property	Description
SumBias	Bias of the sum data type
SumFixedExponent	Fixed exponent of the sum data type
SumFractionLength	Fraction length, in bits, of the sum data type
SumMode	Defines how the sum data type is determined
SumSlope	Slope of the sum data type
SumSlopeAdjustmentFactor	Slope adjustment factor of the sum data type
SumWordLength	Word length, in bits, of the sum data type

For details about these properties, refer to the “fi Object Properties” on page 2-17. To learn how to specify properties for `fimath` objects in Fixed-Point Toolbox software, refer to “Setting fimath Object Properties” on page 4-7.

Setting fimath Object Properties

- “Setting fimath Properties at Object Creation” on page 4-7
- “Using Direct Property Referencing with fimath” on page 4-8
- “Setting fimath Properties in the Model Explorer” on page 4-8

Setting fimath Properties at Object Creation

You can set properties of `fimath` objects at the time of object creation by including properties after the arguments of the `fimath` constructor function.

For example, to set the overflow action to `Saturate` and the rounding method to `Convergent`,

```
F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent')
```

```
F =
```

```
RoundingMethod: Convergent
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
```

Using Direct Property Referencing with fimath

You can reference directly into a property for setting or retrieving `fimath` object property values using MATLAB structure-like referencing. You do so by using a period to index into a property by name.

For example, to get the `RoundingMethod` of `F`,

```
F.RoundingMethod
```

```
ans =
```

```
Convergent
```

To set the `OverflowAction` of `F`,

```
F.OverflowAction = 'Wrap'
```

```
F =
```

```
RoundingMethod: Convergent
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

Setting fimath Properties in the Model Explorer

You can view and change the properties for any `fimath` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View > Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The following figure shows the Model Explorer when you define the following fimath objects in the MATLAB workspace:

```
F = fimath
```

```
F =
```

```

RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision

```

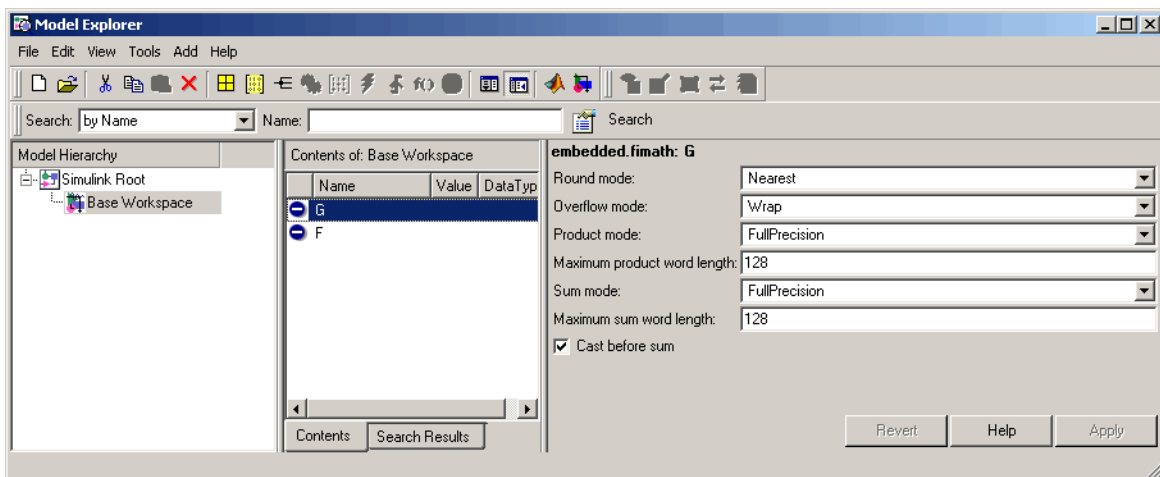
```
G = fimath('OverflowAction','Wrap')
```

```
G =
```

```

RoundingMethod: Nearest
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision

```



Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a fimath object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

For more information on working with the Model Explorer, see the following sections of the Fixed-Point Toolbox documentation:

- “Specifying Fixed-Point Parameters in the Model Explorer” on page 8-77
- “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-81

fimath Properties Usage for Fixed-Point Arithmetic

In this section...

“fimath Rules for Fixed-Point Arithmetic” on page 4-11

“Binary-Point Arithmetic” on page 4-13

“[Slope Bias] Arithmetic” on page 4-17

fimath Rules for Fixed-Point Arithmetic

fimath properties define the rules for performing arithmetic operations on `fi` objects. The fimath properties that govern fixed-point arithmetic operations can come from a local fimath object or the fimath default values.

To determine whether a `fi` object has a local fimath object, use the `isfimathlocal` function.

The following sections discuss how `fi` objects with local fimath objects interact with `fi` objects without local fimath.

Binary Operations

In binary fixed-point operations such as $c = a + b$, the following rules apply:

- If both `a` and `b` have no local fimath, the operation uses default fimath values to perform the fixed-point arithmetic. The output `fi` object `c` also has no local fimath.
- If either `a` or `b` has a local fimath object, the operation uses that fimath object to perform the fixed-point arithmetic. The output `fi` object `c` has the same local fimath object as the input.

Unary Operations

In unary fixed-point operations such as $b = \text{abs}(a)$, the following rules apply:

- If `a` has no local fimath, the operation uses default fimath values to perform the fixed-point arithmetic. The output `fi` object `b` has no local fimath.

- If `a` has a local `fimath` object, the operation uses that `fimath` object to perform the fixed-point arithmetic. The output `fi` object `b` has the same local `fimath` object as the input `a`.

When you specify a `fimath` object in the function call of a unary fixed-point operation, the operation uses the `fimath` object you specify to perform the fixed-point arithmetic. For example, when you use a syntax such as `b = abs(a,F)` or `b = sqrt(a,F)`, the `abs` and `sqrt` operations use the `fimath` object `F` to compute intermediate quantities. The output `fi` object `b` always has no local `fimath`.

Concatenation Operations

In fixed-point concatenation operations such as `c = [a b]`, `c = [a;b]` and `c = bitconcat(a,b)`, the following rule applies:

- The `fimath` properties of the left-most `fi` object in the operation determine the `fimath` properties of the output `fi` object `c`.

For example, consider the following scenarios for the operation `d = [a b c]`:

- If `a` is a `fi` object with no local `fimath`, the output `fi` object `d` also has no local `fimath`.
- If `a` has a local `fimath` object, the output `fi` object `d` has the same local `fimath` object.
- If `a` is not a `fi` object, the output `fi` object `d` inherits the `fimath` properties of the next left-most `fi` object. For example, if `b` is a `fi` object with a local `fimath` object, the output `fi` object `d` has the same local `fimath` object as the input `fi` object `b`.

fimath Object Operations: `add`, `mpy`, `sub`

The output of the `fimath` object operations `add`, `mpy`, and `sub` always have no local `fimath`. The operations use the `fimath` object you specify in the function call, but the output `fi` object never has a local `fimath` object.

MATLAB Function Block Operations

Fixed-point operations performed with the MATLAB Function block use the same rules as fixed-point operations performed in MATLAB.

All input signals to the MATLAB Function block that you treat as **fi** objects associate with whatever you specify for the **MATLAB Function block fimath** parameter. When you set this parameter to **Same as MATLAB**, your **fi** objects do not have local **fimath**. When you set the **MATLAB Function block fimath** parameter to **Specify other**, you can define your own set of **fimath** properties for all **fi** objects in the MATLAB Function block to associate with. You can choose to treat only fixed-point input signals as **fi** objects or both fixed-point and integer input signals as **fi** objects. See “Using **fimath** Objects in MATLAB Function Blocks” on page 8-79.

Binary-Point Arithmetic

The **fimath** object encapsulates the math properties of Fixed-Point Toolbox software.

fi objects only have a local **fimath** object when you explicitly specify **fimath** properties in the **fi** constructor. When you use the **sfi** or **ufi** constructor or do not specify any **fimath** properties in the **fi** constructor, the resulting **fi** object does not have any local **fimath** and uses default **fimath** values.

```
a = fi(pi)
```

```
a =  
    3.1416
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 16  
        FractionLength: 13
```

```
a.fimath  
isfimathlocal(a)
```

```
ans =
```

```
        RoundingMethod: Nearest  
        OverflowAction: Saturate  
        ProductMode: FullPrecision  
        SumMode: FullPrecision
```

```
ans =  
    0
```

To perform arithmetic with `+`, `-`, `.*`, or `*` on two `fi` operands with local `fimath` objects, the local `fimath` objects must be identical. If one of the `fi` operands does not have a local `fimath`, the `fimath` properties of the two operands need not be identical. See “`fimath` Rules for Fixed-Point Arithmetic” on page 4-11 for more information.

```
a = fi(pi);  
b = fi(8);  
isequal(a.fimath, b.fimath)
```

```
ans =  
  
    1
```

```
a + b
```

```
ans =  
  
11.1416
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 19  
        FractionLength: 13
```

To perform arithmetic with `+`, `-`, `.*`, or `*`, two `fi` operands must also have the same data type. For example, you can perform addition on two `fi` objects with data type `double`, but not on an object with data type `double` and one with data type `single`:

```
a = fi(3, 'DataType', 'double')  
  
a =  
  
    3
```

```

        DataTypeMode: Double

b = fi(27, 'DataType', 'double')

b =

    27
    
```

```

        DataTypeMode: Double

a + b

ans =

    30
    
```

```

        DataTypeMode: Double

c = fi(12, 'DataType', 'single')

c =

    12
    
```

```

        DataTypeMode: Single

a + c
??? Math operations are not allowed on FI objects with
different data types.
    
```

Fixed-point `fi` object operands do not have to have the same scaling. You can perform binary math operations on a `fi` object with a fixed-point data type and a `fi` object with a scaled doubles data type. In this sense, the scaled double data type acts as a fixed-point data type:

```

a = fi(pi)

a =

    3.1416
    
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13

b = fi(magic(2), ...
'DataTypeMode', 'Scaled double: binary point scaling')

b =

     1     3
     4     2

        DataTypeMode: Scaled double: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 12

a + b

ans =

     4.1416     6.1416
     7.1416     5.1416

        DataTypeMode: Scaled double: binary point scaling
        Signedness: Signed
        WordLength: 18
        FractionLength: 13
```

Use the `divide` function to perform division with doubles, singles, or binary point-only scaling `fi` objects.

[Slope Bias] Arithmetic

Fixed-Point Toolbox software supports fixed-point arithmetic using the local `fimath` object or default `fimath` for all binary point-only signals. The toolbox also supports arithmetic for [Slope Bias] signals with the following restrictions:

- [Slope Bias] signals must be real.
- You must set the `SumMode` and `ProductMode` properties of the governing `fimath` to 'SpecifyPrecision' for sum and multiply operations, respectively.
- You must set the `CastBeforeSum` property of the governing `fimath` to 'true'.
- Fixed-Point Toolbox does not support the `divide` function for [Slope Bias] signals.

```
f = fimath('SumMode', 'SpecifyPrecision', ...
          'SumFractionLength', 16)
```

```
f =
```

```

RoundingMethod: Nearest
OverflowAction: Saturate
  ProductMode: FullPrecision
    SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true
```

```
a = fi(pi, 'fimath', f)
```

```
a =
```

```
3.1416
```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
```

```
FractionLength: 13

RoundingMethod: Nearest
OverflowAction: Saturate
    ProductMode: FullPrecision
        SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true

b = fi(22, true, 16, 2^-8, 3, 'fimath', f)

b =

    22

    DataTypeMode: Fixed-point: slope and bias scaling
        Signedness: Signed
        WordLength: 16
        Slope: 0.00390625
        Bias: 3

    RoundingMethod: Nearest
    OverflowAction: Saturate
        ProductMode: FullPrecision
            SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true

a + b

ans =

    25.1416

    DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 16
```

```
RoundingMethod: Nearest
OverflowAction: Saturate
    ProductMode: FullPrecision
        SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true
```

Setting the `SumMode` and `ProductMode` properties to `SpecifyPrecision` are mutually exclusive except when performing the `*` operation between matrices. In this case, you must set both the `SumMode` and `ProductMode` properties to `SpecifyPrecision` for [Slope Bias] signals. Doing so is necessary because the `*` operation performs both sum and multiply operations to calculate the result.

fimath for Rounding and Overflow Modes

Only rounding methods and overflow actions set prior to an operation with `fi` objects affect the outcome of those operations. Once you create a `fi` object in MATLAB, changing its rounding or overflow settings does not affect its value. For example, consider the `fi` objects `a` and `b`:

```
p = fipref('NumberDisplay', 'RealWorldValue',...  
          'NumericTypeDisplay', 'none', 'FimathDisplay', 'none');  
T = numericitytype('WordLength',8,'FractionLength',7);  
F = fimath('RoundingMethod','Floor','OverflowAction','Wrap');  
a = fi(1,T,F)
```

```
a =  
  
    -1
```

```
b = fi(1,T)
```

```
b =  
  
    0.9922
```

Because you create `a` with a `fimath` object `F` that has `OverflowAction` set to `Wrap`, the value of `a` wraps to `-1`. Conversely, because you create `b` with the default `OverflowAction` value of `Saturate`, its value saturates to `0.9922`.

Now, assign the `fimath` object `F` to `b`:

```
b.fimath = F
```

```
b =  
  
    0.9922
```

Because the assignment operation and corresponding overflow and saturation happened when you created `b`, its value does not change when you assign it the new `fimath` object `F`.

Note `fi` objects with no local `fimath` and created from a floating-point value always get constructed with a `RoundingMethod` of `Nearest` and an `OverflowAction` of `Saturate`. To construct `fi` objects with different `RoundingMethod` and `OverflowAction` properties, specify the desired `RoundingMethod` and `OverflowAction` properties in the `fi` constructor.

fimath for Sharing Arithmetic Rules

There are two ways of sharing `fimath` properties in Fixed-Point Toolbox software:

- “Default `fimath` Usage to Share Arithmetic Rules” on page 4-22
- “Local `fimath` Usage to Share Arithmetic Rules” on page 4-22

Sharing `fimath` properties across `fi` objects ensures that the `fi` objects are using the same arithmetic rules and helps you avoid “mismatched `fimath`” errors.

Default `fimath` Usage to Share Arithmetic Rules

You can ensure that your `fi` objects are all using the same `fimath` properties by not specifying any local `fimath`. To assure no local `fimath` is associated with a `fi` object, you can:

- Create a `fi` object using the `fi` constructor without specifying any `fimath` properties in the constructor call. For example:

```
a = fi(pi)
```

- Create a `fi` object using the `sfi` or `ufi` constructor. All `fi` objects created with these constructors have no local `fimath`.

```
b = sfi(pi)
```

- Use `removefimath` to remove a local `fimath` object from an existing `fi` object.

Local `fimath` Usage to Share Arithmetic Rules

You can also use a `fimath` object to define common arithmetic rules that you would like to use for multiple `fi` objects. You can then create your `fi` objects, using the same `fimath` object for each. To do so, you must also create a `numerictype` object to define a common data type and scaling. Refer to “`numerictype` Object Construction” on page 6-2 for more information on `numerictype` objects. The following example shows the creation of a

numeric type object and fimath object, and then uses those objects to create two fi objects with the same numeric type and fimath attributes:

```
T = numericType('WordLength',32,'FractionLength',30)
```

```
T =
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 30

```

```
F = fimath('RoundingMethod','Floor',...
           'OverflowAction','Wrap')
```

```
F =
```

```

        RoundingMethod: Floor
        OverflowAction: Wrap
        ProductMode: FullPrecision
        SumMode: FullPrecision

```

```
a = fi(pi, T, F)
```

```
a =
```

```
-0.8584
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 30

```

```

        RoundingMethod: Floor
        OverflowAction: Wrap
        ProductMode: FullPrecision
        SumMode: FullPrecision

```

```
b = fi(pi/2, T, F)
```

```
b =
```

```
1.5708
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 32  
      FractionLength: 30  
  
      RoundingMethod: Floor  
      OverflowAction: Wrap  
      ProductMode: FullPrecision  
      SumMode: FullPrecision
```

fimath ProductMode and SumMode

In this section...

“Example Setup” on page 4-25

“FullPrecision” on page 4-26

“KeepLSB” on page 4-27

“KeepMSB” on page 4-28

“SpecifyPrecision” on page 4-30

Example Setup

The examples in the sections of this topic show the differences among the four settings of the ProductMode and SumMode properties:

- FullPrecision
- KeepLSB
- KeepMSB
- SpecifyPrecision

To follow along, first set the following preferences:

```
p = fipref;
p.NumericTypeDisplay = 'short';
p.FimathDisplay = 'none';
p.LoggingMode = 'on';
F = fimath('OverflowAction','Wrap',...
    'RoundingMethod','Floor',...
    'CastBeforeSum',false);
warning off
format compact
```

Next, define `fi` objects `a` and `b`. Both have signed 8-bit data types. The fraction length gets chosen automatically for each `fi` object to yield the best possible precision:

```
a = fi(pi, true, 8)
```

```
a =  
    3.1563  
    s8,5  
  
b = fi(exp(1), true, 8)  
b =  
    2.7188  
    s8,5
```

FullPrecision

Now, set ProductMode and SumMode for a and b to FullPrecision and look at some results:

```
F.ProductMode = 'FullPrecision';  
F.SumMode = 'FullPrecision';  
a.fimath = F;  
b.fimath = F;  
a  
a =  
    3.1563    %011.00101  
    s8,5  
  
b  
b =  
    2.7188    %010.10111  
    s8,5  
  
a*b  
ans =  
    8.5811    %001000.1001010011  
    s16,10  
  
a+b  
ans =  
    5.8750    %0101.11100  
    s9,5
```

In FullPrecision mode, the product word length grows to the sum of the word lengths of the operands. In this case, each operand has 8 bits, so the

product word length is 16 bits. The product fraction length is the sum of the fraction lengths of the operands, in this case $5 + 5 = 10$ bits.

The sum word length grows by one most significant bit to accommodate the possibility of a carry bit. The sum fraction length aligns with the fraction lengths of the operands, and all fractional bits are kept for full precision. In this case, both operands have 5 fractional bits, so the sum has 5 fractional bits.

KeepLSB

Now, set ProductMode and SumMode for a and b to KeepLSB and look at some results:

```
F.ProductMode = 'KeepLSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepLSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;

a
a =
    3.1563    %011.00101
    s8,5

b
b =
    2.7188    %010.10111
    s8,5

a*b
ans =
    0.5811    %00.1001010011
    s12,10

a+b
ans =
    5.8750    %0000101.11100
    s12,5
```

In `KeepLSB` mode, you specify the word lengths and the least significant bits of results are automatically kept. This mode models the behavior of integer operations in the C language.

The product fraction length is the sum of the fraction lengths of the operands. In this case, each operand has 5 fractional bits, so the product fraction length is 10 bits. In this mode, all 10 fractional bits are kept. Overflow occurs because the full-precision result requires 6 integer bits, and only 2 integer bits remain in the product.

The sum fraction length aligns with the fraction lengths of the operands, and in this model all least significant bits are kept. In this case, both operands had 5 fractional bits, so the sum has 5 fractional bits. The full-precision result requires 4 integer bits, and 7 integer bits remain in the sum, so no overflow occurs in the sum.

KeepMSB

Now, set `ProductMode` and `SumMode` for `a` and `b` to `KeepMSB` and look at some results:

```
F.ProductMode = 'KeepMSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepMSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563    %011.00101
         s8,5

b
b =
    2.7188    %010.10111
         s8,5

a*b
ans =
    8.5781    %001000.100101
```



```

        s12,6

a+b
ans =
    5.8750    %0101.11100000
        s12,8

```

In **KeepMSB** mode, you specify the word lengths and the most significant bits of sum and product results are automatically kept. This mode models the behavior of many DSP devices where the product and sum are kept in double-wide registers, and the programmer chooses to transfer the most significant bits from the registers to memory after each operation.

The full-precision product requires 6 integer bits, and the fraction length of the product is adjusted to accommodate all 6 integer bits in this mode. No overflow occurs. However, the full-precision product requires 10 fractional bits, and only 6 are available. Therefore, precision is lost.

The full-precision sum requires 4 integer bits, and the fraction length of the sum is adjusted to accommodate all 4 integer bits in this mode. The full-precision sum requires only 5 fractional bits; in this case there are 8, so there is no loss of precision.

This example shows that, in **KeepMSB** mode the fraction length changes regardless of whether or not an overflow occurs. The fraction length is set to the amount needed to represent the product in case both terms use the maximum possible value ($18+18-16=20$ in this example).

```

F = fimath('SumMode','KeepMSB','ProductMode','KeepMSB',...
    'ProductWordLength',16,'SumWordLength',16);
a=fi(100,1,16,-2,'fimath',F);
a*a

ans =

    0

```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16

```

```
FractionLength: -20  
  
RoundingMethod: Nearest  
OverflowAction: Saturate  
ProductMode: KeepMSB  
ProductWordLength: 16  
SumMode: KeepMSB  
SumWordLength: 16  
CastBeforeSum: true
```

SpecifyPrecision

Now set ProductMode and SumMode for a and b to SpecifyPrecision and look at some results:

```
F.ProductMode = 'SpecifyPrecision';  
F.ProductWordLength = 8;  
F.ProductFractionLength = 7;  
F.SumMode = 'SpecifyPrecision';  
F.SumWordLength = 8;  
F.SumFractionLength = 7;  
a.fimath = F;  
b.fimath = F;  
a  
a =  
    3.1563    %011.00101  
    s8,5  
  
b  
b =  
    2.7188    %010.10111  
    s8,5  
  
a*b  
ans =  
    0.5781    %0.1001010  
    s8,7  
  
a+b  
ans =
```

```
-0.1250    %1.1110000  
s8,7
```

In `SpecifyPrecision` mode, you must specify both word length and fraction length for sums and products. This example unwisely uses fractional formats for the products and sums, with 8-bit word lengths and 7-bit fraction lengths.

The full-precision product requires 6 integer bits, and the example specifies only 1, so the product overflows. The full-precision product requires 10 fractional bits, and the example only specifies 7, so there is precision loss in the product.

The full-precision sum requires 4 integer bits, and the example specifies only 1, so the sum overflows. The full-precision sum requires 5 fractional bits, and the example specifies 7, so there is no loss of precision in the sum.

Working with fipref Objects

- “fipref Object Construction” on page 5-2
- “fipref Object Properties” on page 5-3
- “fi Object Display Preferences Using fipref” on page 5-5
- “Underflow and Overflow Logging Using fipref” on page 5-7
- “Data Type Override Preferences Using fipref” on page 5-12

fipref Object Construction

The `fipref` object defines the display and logging attributes for all `fi` objects. You can use the `fipref` constructor function to create a new object.

To get started, type

```
P = fipref
```

to create a default `fipref` object.

```
P =  
    NumberDisplay: 'RealWorldValue'  
    NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'ForceOff'
```

The syntax

```
P = fipref(... 'PropertyName', 'PropertyValue'...)
```

allows you to set properties for a `fipref` object at object creation with property name/property value pairs.

Your `fipref` settings persist throughout your MATLAB session. Use `reset(fipref)` to return to the default settings during your session. Use `savefipref` to save your display preferences for subsequent MATLAB sessions.

fipref Object Properties

In this section...

“Display, Data Type Override, and Logging Properties” on page 5-3
 “fipref Object Properties Setting” on page 5-3

Display, Data Type Override, and Logging Properties

The following properties of `fipref` objects are always writable:

- `FimathDisplay` — Display options for the local `fimath` attributes of a `fi` object
- `DataTypeOverride` — Data type override options
- `LoggingMode` — Logging options for operations performed on `fi` objects
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object
- `NumberDisplay` — Display options for the value of a `fi` object

These properties are described in detail in the “`fi` Object Properties” on page 2-17. To learn how to specify properties for `fipref` objects in Fixed-Point Toolbox software, refer to “fipref Object Properties Setting” on page 5-3.

fipref Object Properties Setting

Setting fipref Properties at Object Creation

You can set properties of `fipref` objects at the time of object creation by including properties after the arguments of the `fipref` constructor function. For example, to set `NumberDisplay` to `bin` and `NumericTypeDisplay` to `short`,

```
P = fipref('NumberDisplay', 'bin', ...
          'NumericTypeDisplay', 'short')
```

```
P =
    NumberDisplay: 'bin'
 NumericTypeDisplay: 'short'
```

```
FimathDisplay: 'full'  
LoggingMode: 'Off'  
DataTypeOverride: 'ForceOff'
```

Using Direct Property Referencing with fipref

You can reference directly into a property for setting or retrieving `fipref` object property values using MATLAB structure-like referencing. You do this by using a period to index into a property by name.

For example, to get the `NumberDisplay` of `P`,

```
P.NumberDisplay
```

```
ans =
```

```
bin
```

To set the `NumericTypeDisplay` of `P`,

```
P.NumericTypeDisplay = 'full'
```

```
P =
```

```
    NumberDisplay: 'bin'  
NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'ForceOff'
```


fi Object Display Preferences Using fipref

You use the `fipref` object to specify three aspects of the display of `fi` objects: the object value, the local `fimath` properties, and the `numericType` properties.

For example, the following code shows the default `fipref` display for a `fi` object with a local `fimath` object:

```
a = fi(pi, 'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap')

a =
    3.1415

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
           WordLength: 16
    FractionLength: 13

    RoundingMethod: Floor
    OverflowAction: Wrap
       ProductMode: FullPrecision
          SumMode: FullPrecision
```

The default `fipref` display for a `fi` object with no local `fimath` is as follows:

```
a = fi(pi)

a =
    3.1416

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
           WordLength: 16
    FractionLength: 13
```

Next, change the `fipref` display properties:

```
P = fipref;
```

```
P.NumberDisplay = 'bin';
P.NumericTypeDisplay = 'short';
P.FimathDisplay = 'none'

P =
    NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
    FimathDisplay: 'none'
    LoggingMode: 'Off'
    DataTypeOverride: 'ForceOff'
a

a =
0110010010000111
    s16,13
```

For more information on the default `fipref` display, see “View Fixed-Point Data”.

Underflow and Overflow Logging Using fipref

In this section...
“Logging Overflows and Underflows as Warnings” on page 5-7
“Accessing Logged Information with Functions” on page 5-9

Logging Overflows and Underflows as Warnings

Overflows and underflows are logged as warnings for all assignment, plus, minus, and multiplication operations when the fipref LoggingMode property is set to on. For example, try the following:

- 1 Create a signed fi object that is a vector of values from 1 to 5, with 8-bit word length and 6-bit fraction length.

```
a = fi(1:5,1,8,6);
```

- 2 Define the fimath object associated with a, and indicate that you will specify the sum and product word and fraction lengths.

```
F = a.fimath;  
F.SumMode = 'SpecifyPrecision';  
F.ProductMode = 'SpecifyPrecision';  
a.fimath = F;
```

- 3 Define the fipref object and turn on overflow and underflow logging.

```
P = fipref;  
P.LoggingMode = 'on';
```

- 4 Suppress the numeric type and fimath displays.

```
P.NumericTypeDisplay = 'none';  
P.FimathDisplay = 'none';
```

- 5 Specify the sum and product word and fraction lengths.

```
a.SumWordLength = 16;  
a.SumFractionLength = 15;
```

```
a.ProductWordLength = 16;  
a.ProductFractionLength = 15;
```

- 6** Warnings are displayed for overflows and underflows in assignment operations. For example, try:

```
a(1) = pi  
Warning: 1 overflow occurred in the fi assignment operation.
```

```
a =
```

```
    1.9844    1.9844    1.9844    1.9844    1.9844
```

```
a(1) = double(eps(a))/10  
Warning: 1 underflow occurred in the fi assignment operation.
```

```
a =
```

```
    0    1.9844    1.9844    1.9844    1.9844
```

- 7** Warnings are displayed for overflows and underflows in addition and subtraction operations. For example, try:

```
a+a  
Warning: 12 overflows occurred in the fi + operation.
```

```
ans =
```

```
    0    1.0000    1.0000    1.0000    1.0000
```

```
a-a
```

```
Warning: 8 overflows occurred in the fi - operation.
```

```
ans =
```

```
    0    0    0    0    0
```

- 8** Warnings are displayed for overflows and underflows in multiplication operations. For example, try:

```
a.*a
Warning: 4 product overflows occurred in the fi .* operation.

ans =

         0    1.0000    1.0000    1.0000    1.0000

a*a'
Warning: 4 product overflows occurred in the fi * operation.
Warning: 3 sum overflows occurred in the fi * operation.

ans =

    1.0000
```

The final example above is a complex multiplication that requires both multiplication and addition operations. The warnings inform you of overflows and underflows in both.

Because overflows and underflows are logged as warnings, you can use the `dbstop` MATLAB function with the syntax

```
dbstop if warning
```

to find the exact lines in a file that are causing overflows or underflows.

Use

```
dbstop if warning fi:underflow
```

to stop only on lines that cause an underflow. Use

```
dbstop if warning fi:overflow
```

to stop only on lines that cause an overflow.

Accessing Logged Information with Functions

When the `fipref` `LoggingMode` property is set to on, you can use the following functions to return logged information about assignment and creation operations to the MATLAB command line:

- `maxlog` — Returns the maximum real-world value
- `minlog` — Returns the minimum value
- `noverflows` — Returns the number of overflows
- `nunderflows` — Returns the number of underflows

`LoggingMode` must be set to on before you perform any operation in order to log information about it. To clear the log, use the function `resetlog`.

For example, consider the following. First turn logging on, then perform operations, and then finally get information about the operations:

```
fipref('LoggingMode','on');  
x = fi([-1.5 eps 0.5], true, 16, 15);  
x(1) = 3.0;  
maxlog(x)
```

```
ans =  
  
1.0000
```

```
minlog(x)
```

```
ans =  
-1
```

```
noverflows(x)
```

```
ans =  
  
2
```

```
nunderflows(x)
```

```
ans =  
  
1
```

Next, reset the log and request the same information again. Note that the functions return empty [], because logging has been reset since the operations were run:

```
resetlog(x)
```

```
maxlog(x)
```

```
ans =
```

```
    []
```

```
minlog(x)
```

```
ans =
```

```
    []
```

```
noverflows(x)
```

```
ans =
```

```
    []
```

```
nunderflows(x)
```

```
ans =
```

```
    []
```

Data Type Override Preferences Using fipref

In this section...
“Overriding the Data Type of fi Objects” on page 5-12
“Data Type Override for Fixed-Point Scaling” on page 5-13

Overriding the Data Type of fi Objects

Use the `fipref` `DataTypeOverride` property to override `fi` objects with singles, doubles, or scaled doubles. Data type override only occurs when the `fi` constructor function is called. Objects that are created while data type override is on have the overridden data type. They maintain that data type when data type override is later turned off. To obtain an object with a data type that is not the override data type, you must create an object when data type override is off:

```
p = fipref('DataTypeOverride', 'TrueDoubles')
```

```
p =
```

```
    NumberDisplay: 'RealWorldValue'  
NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'TrueDoubles'
```

```
a = fi(pi)
```

```
a =
```

```
    3.1416
```

```
    DataTypeMode: Double
```

```
p = fipref('DataTypeOverride', 'ForceOff')
```

```
p =
```



```
        NumberDisplay: 'RealWorldValue'
NumericTypeDisplay: 'full'
        FimathDisplay: 'full'
        LoggingMode: 'Off'
        DataTypeOverride: 'ForceOff'

a

a =

    3.1416

        DataTypeMode: Double

b = fi(pi)

b =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13
```

Tip To reset the `fipref` object to its default values use `reset(fipref)` or `reset(p)`, where `p` is a `fipref` object. This is useful to ensure that data type override and logging are off.

Data Type Override for Fixed-Point Scaling

Choosing the scaling for the fixed-point variables in your algorithms can be difficult. In Fixed-Point Toolbox software, you can use a combination of data type override and min/max logging to help you discover the numerical ranges that your fixed-point data types need to cover. These ranges dictate the appropriate scalings for your fixed-point data types. In general, the procedure is

- 1** Implement your algorithm using fixed-point `fi` objects, using initial “best guesses” for word lengths and scalings.
- 2** Set the `fipref` `DataTypeOverride` property to `ScaledDoubles`, `TrueSingles`, or `TrueDoubles`.
- 3** Set the `fipref` `LoggingMode` property to `on`.
- 4** Use the `maxlog` and `minlog` functions to log the maximum and minimum values achieved by the variables in your algorithm in floating-point mode.
- 5** Set the `fipref` `DataTypeOverride` property to `ForceOff`.
- 6** Use the information obtained in step 4 to set the fixed-point scaling for each variable in your algorithm such that the full numerical range of each variable is representable by its data type and scaling.

A detailed example of this process is shown in the Fixed-Point Toolbox Setting Fixed-Point Data Types Using Min/Max Instrumentation example.

Working with numerictype Objects

- “numerictype Object Construction” on page 6-2
- “numerictype Object Properties” on page 6-7
- “numerictype Structure of Fixed-Point Objects” on page 6-11
- “numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects” on page 6-14

numerictype Object Construction

In this section...
“numerictype Object Syntaxes” on page 6-2
“Example: Construct a numerictype Object with Property Name and Property Value Pairs” on page 6-3
“Example: Copy a numerictype Object” on page 6-4
“Example: Build numerictype Object Constructors in a GUI” on page 6-5

numerictype Object Syntaxes

numerictype objects define the data type and scaling attributes of `fi` objects, as well as Simulink signals and model parameters. You can create numerictype objects in Fixed-Point Toolbox software in one of two ways:

- You can use the `numerictype` constructor function to create a new object.
- You can use the `numerictype` constructor function to copy an existing numerictype object.

To get started, type

```
T = numerictype
```

to create a default numerictype object.

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

To see all of the numerictype object syntaxes, refer to the numerictype constructor function reference page.

The following examples show different ways of constructing `numerictype` objects. For more examples of constructing `numerictype` objects, see the “Examples” on the `numerictype` constructor function reference page.

Example: Construct a `numerictype` Object with Property Name and Property Value Pairs

When you create a `numerictype` object using property name and property value pairs, Fixed-Point Toolbox software first creates a default `numerictype` object, and then, for each property name you specify in the constructor, assigns the corresponding value.

This behavior differs from the behavior that occurs when you use a syntax such as `T = numerictype(s,w)`, where you only specify the property values in the constructor. Using such a syntax results in no default `numerictype` object being created, and the `numerictype` object receives only the assigned property values that are specified in the constructor.

The following example shows how the property name/property value syntax creates a slightly different `numerictype` object than the property values syntax, even when you specify the same property values in both constructors.

To demonstrate this difference, suppose you want to create an unsigned `numerictype` object with a word length of 32 bits.

First, create the `numerictype` object using property name/property value pairs.

```
T1 = numerictype('Signed',0,'WordLength',32)
```

```
T1 =
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 32
        FractionLength: 15
    
```

The numerictype object T1 has the same DataTypeMode and FractionLength as a default numerictype object, but the WordLength and Signed properties are overwritten with the values you specified.

Now, create another unsigned 32 bit numerictype object, but this time specify only property values in the constructor.

```
T2 = numerictype(0,32)
```

```
T2 =
```

```
DataTypeMode: Fixed-point: unspecified scaling  
Signedness: Unsigned  
WordLength: 32
```

Unlike T1, T2 only has the property values you specified. The DataTypeMode of T2 is Fixed-Point: unspecified scaling, so no fraction length is assigned.

fi objects cannot have unspecified numerictype properties. Thus, all unspecified numerictype object properties become specified at the time of fi object creation.

Example: Copy a numerictype Object

To copy a numerictype object, simply use assignment as in the following example:

```
T = numerictype;  
U = T;  
isequal(T,U)
```

```
ans =
```

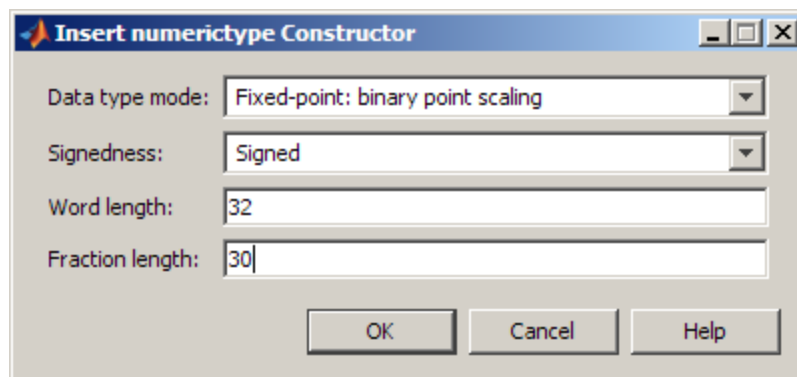
```
1
```

Example: Build numerictype Object Constructors in a GUI

When you are working with files in MATLAB, you can build your `numerictype` object constructors using the **Insert numerictype Constructor** dialog box. After specifying the properties of the `numerictype` object in the dialog box, you can insert the prepopulated `numerictype` object constructor string at a specific location in your file.

For example, to create a signed `numerictype` object with binary-point scaling, a word length of 32 bits and a fraction length of 30 bits, perform the following steps:

- 1 Open the **Insert numerictype Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert numerictype Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the `numerictype` object:
 - **Data type mode** = Fixed-point: binary point scaling
 - **Signedness** = Signed
 - **Word length** = 32
 - **Fraction length** = 30



- 3 To insert the `numerictype` object constructor string in your file, place your cursor at the desired location in the file, and click **OK** on the **Insert**

numerictype Constructor dialog box. Clicking **OK** closes the **Insert numerictype Constructor** dialog box and automatically populates the numerictype object constructor string in your file:

```
5 T = numerictype(1, 32, 30)
```


numerictype Object Properties

In this section...
“Data Type and Scaling Properties” on page 6-7
“Set numerictype Object Properties” on page 6-8

Data Type and Scaling Properties

All properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object become read only after the `fi` object has been created. Any `numerictype` properties of a `fi` object that are unspecified at the time of `fi` object creation are automatically set to their default values. The properties of a `numerictype` object are:

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Signedness` — Signed, unsigned, or auto
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

These properties are described in detail in the “`fi` Object Properties” on page 2-17. To learn how to specify properties for `numerictype` objects in Fixed-Point Toolbox software, refer to “Set `numerictype` Object Properties” on page 6-8.

Set numerictype Object Properties

Setting numerictype Properties at Object Creation

You can set properties of numerictype objects at the time of object creation by including properties after the arguments of the numerictype constructor function.

For example, to set the word length to 32 bits and the fraction length to 30 bits,

```
T = numerictype('WordLength', 32, 'FractionLength', 30)
```

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 32  
      FractionLength: 30
```

Use Direct Property Referencing with numerictype Objects

You can reference directly into a property for setting or retrieving numerictype object property values using MATLAB structure-like referencing. You do this by using a period to index into a property by name.

For example, to get the word length of T,

```
T.WordLength
```

```
ans =
```

```
32
```

To set the fraction length of T,

```
T.FractionLength = 31
```

```
T =
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 31
    
```

Set numericity Properties in the Model Explorer

You can view and change the properties for any numericity object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View > Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The figure below shows the Model Explorer when you define the following numericity objects in the MATLAB workspace:

```
T = numericity
```

```
T =
```

```

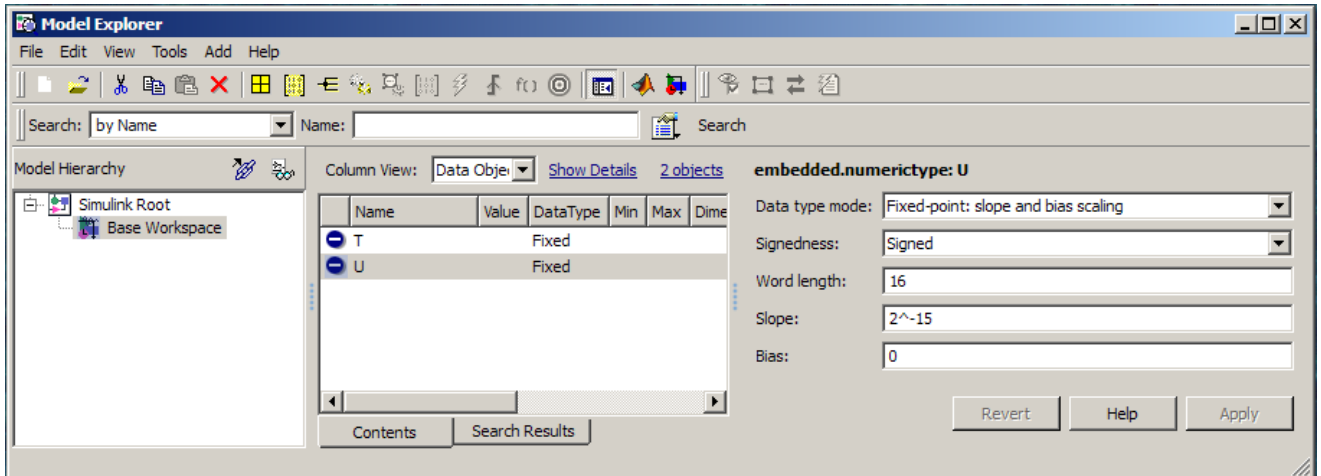
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 15
    
```

```
U = numericity('DataTypeMode', 'Fixed-point: slope and bias')
```

```
U =
```

```

        DataTypeMode: Fixed-point: slope and bias scaling
        Signedness: Signed
        WordLength: 16
        Slope: 2^-15
        Bias: 0
    
```



Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a numerictype object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

numerictype Structure of Fixed-Point Objects

In this section...
“Valid Values for numerictype Structure Properties” on page 6-11
“Properties That Affect the Slope” on page 6-13
“Stored Integer Value and Real World Value” on page 6-13

Valid Values for numerictype Structure Properties

The numerictype object contains all the data type and scaling attributes of a fixed-point object. The numerictype object behaves like any MATLAB structure, except that it only lets you set valid values for defined fields. The following table shows the possible settings of each field of the structure.

Note When you create a `fi` object, any unspecified field of the `numerictype` object reverts to its default value. Thus, if the `DataTypeMode` is set to unspecified `scaling`, it defaults to `binary point scaling` when the `fi` object is created. If the `Signedness` property of the `numerictype` object is set to `Auto`, it defaults to `Signed` when the `fi` object is created.

DataTypeMode	DataType	Scaling	Signedness	Word- Length	Fraction- Length	Slope	Bias
<i>Fixed-point data types</i>							
Fixed-point: binary point scaling	Fixed	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	2 [^] (-fraction length)	Any floating- point number
Fixed-point: slope and bias scaling	Fixed	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating- point number	Any floating- point number

DataTypeMode	DataType	Scaling	Signedness	Word- Length	Fraction- Length	Slope	Bias
Fixed-point: unspecified scaling	Fixed	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
<i>Scaled double data types</i>							
Scaled double: binary point scaling	ScaledDouble	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	$2^{(-\text{fraction length})}$	Any
Scaled double: slope and bias scaling	ScaledDouble	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating- point number	Any floating- point number
Scaled double: unspecified scaling	ScaledDouble	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
<i>Built-in data types</i>							
Double	double	N/A	1 true	64	0	1	0
Single	single	N/A	1 true	32	0	1	0
Boolean	boolean	N/A	0 false	1	0	1	0

You cannot change the numerictype properties of a fi object after fi object creation.

Properties That Affect the Slope

The **Slope** field of the numerictype structure is related to the SlopeAdjustmentFactor and FixedExponent properties by

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The FixedExponent and FractionLength properties are related by

$$\text{fixed exponent} = -\text{fraction length}$$

If you set the SlopeAdjustmentFactor, FixedExponent, or FractionLength property, the **Slope** field is modified.

Stored Integer Value and Real World Value

The numerictype StoredIntegerValue and RealWorldValue properties are related according to

$$\text{real-world value} = \text{stored integer value} \times 2^{-\text{fraction length}}$$

which is equivalent to

$$\begin{aligned} \text{real-world value} = \\ \text{stored integer value} \times (\text{slope adjustment factor} \times 2^{\text{fixed exponent}}) + \text{bias} \end{aligned}$$

If any of these properties is updated, the others are modified accordingly.

numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects

You can use a numerictype object to define common data type and scaling rules that you would like to use for many fi objects. You can then create multiple fi objects, using the same numerictype object for each.

Example 1

In the following example, you create a numerictype object T with word length 32 and fraction length 28. Next, to ensure that your fi objects have the same numerictype attributes, create fi objects a and b using your numerictype object T.

```
format long g
T = numerictype('WordLength',32,'FractionLength',28)

T =

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 28

a = fi(pi,T)

a =

    3.1415926553309

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 28

b = fi(pi/2, T)
```



```
b =

    1.5707963258028

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 28
```

Example 2

In this example, start by creating a `numerictype` object `T` with [Slope Bias] scaling. Next, use that object to create two `fi` objects, `c` and `d` with the same `numerictype` attributes:

```
T = numerictype('Scaling','slopebias','Slope', 2^2, 'Bias', 0)
```

```
T =

    DataTypeMode: Fixed-point: slope and bias scaling
    Signedness: Signed
    WordLength: 16
    Slope: 2^2
    Bias: 0
```

```
c = fi(pi, T)
```

```
c =

    4
```

```
    DataTypeMode: Fixed-point: slope and bias scaling
    Signedness: Signed
    WordLength: 16
    Slope: 2^2
    Bias: 0
```

```
d = fi(pi/2, T)
```

d =

0

DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 2^2
Bias: 0

Working with quantizer Objects

- “Constructing quantizer Objects” on page 7-2
- “quantizer Object Properties” on page 7-3
- “Quantizing Data with quantizer Objects” on page 7-4
- “Transformations for Quantized Data” on page 7-6

Constructing quantizer Objects

You can use quantizer objects to quantize data sets. You can create quantizer objects in Fixed-Point Toolbox software in one of two ways:

- You can use the `quantizer` constructor function to create a new object.
- You can use the `quantizer` constructor function to copy a quantizer object.

To create a quantizer object with default properties, type

```
q = quantizer

q =

        DataMode = fixed
    RoundingMethod = Floor
    OverflowAction = Saturate
        Format = [16 15]
```

To copy a quantizer object, simply use assignment as in the following example:

```
q = quantizer;
r = q;
isequal(q,r)

ans =

     1
```

A listing of all the properties of the quantizer object `q` you just created is displayed along with the associated property values. All property values are set to defaults when you construct a quantizer object this way. See “quantizer Object Properties” on page 7-3 for more details.

quantizer Object Properties

The following properties of quantizer objects are always writable:

- **DataMode** — Type of arithmetic used in quantization
- **Format** — Data format of a quantizer object
- **OverflowAction** — Action to take on overflow
- **RoundingMethod** — Rounding method

See the “fi Object Properties” on page 2-17 for more details about these properties, including their possible values.

For example, to create a fixed-point quantizer object with

- The **Format** property value set to [16,14]
- The **OverflowAction** property value set to 'Saturate'
- The **RoundingMethod** property value set to 'Ceiling'

type

```
q = quantizer('datamode','fixed','format',[16,14],...  
             'OverflowMode','saturate','RoundMode','ceil')
```

You do not have to include quantizer object property names when you set quantizer object property values.

For example, you can create quantizer object `q` from the previous example by typing

```
q = quantizer('fixed',[16,14],'saturate','ceil')
```

Note You do not have to include default property values when you construct a quantizer object. In this example, you could leave out 'fixed' and 'saturate'.

Quantizing Data with quantizer Objects

You construct a `quantizer` object to specify the quantization parameters to use when you quantize data sets. You can use the `quantize` function to quantize data according to a `quantizer` object's specifications.

Once you quantize data with a `quantizer` object, its state values might change.

The following example shows

- How you use `quantize` to quantize data
- How quantization affects `quantizer` object states
- How you reset `quantizer` object states to their default values using `reset`

1 Construct an example data set and a `quantizer` object.

```
format long g
rng('default');
x = randn(100,4);
q = quantizer([16,14]);
```

2 Retrieve the values of the `maxlog` and `noverflows` states.

```
q.maxlog

ans =

    -1.79769313486232e+308

q.noverflows

ans =

     0
```

Note that `maxlog` is equal to `-realmax`, which indicates that the quantizer `q` is in a reset state.

3 Quantize the data set according to the `quantizer` object's specifications.

```
y = quantize(q,x);
```

```
Warning: 626 overflow(s) occurred in the fi quantize operation.
```

4 Check the values of `maxlog` and `noverflows`.

```
q.maxlog
```

```
ans =
```

```
1.99993896484375
```

```
q.noverflows
```

```
ans =
```

```
626
```

Note that the maximum logged value was taken after quantization, that is,
`q.maxlog == max(y)`.

5 Reset the quantizer states and check them.

```
reset(q)
```

```
q.maxlog
```

```
ans =
```

```
-1.79769313486232e+308
```

```
q.noverflows
```

```
ans =
```

```
0
```

Transformations for Quantized Data

You can convert data values from numeric to hexadecimal or binary according to a quantizer object's specifications.

Use

- `num2bin` to convert data to binary
- `num2hex` to convert data to hexadecimal
- `hex2num` to convert hexadecimal data to numeric
- `bin2num` to convert binary data to numeric

For example,

```
q = quantizer([3 2]);  
x = [0.75   -0.25  
      0.50   -0.50  
      0.25   -0.75  
       0     -1 ];  
b = num2bin(q,x)
```

```
b =  
011  
010  
001  
000  
111  
110  
101  
100
```

produces all two's complement fractional representations of 3-bit fixed-point numbers.

Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms

- “Code Acceleration and Code Generation from MATLAB” on page 8-3
- “Requirements for Generating Compiled C Code Files” on page 8-4
- “Functions Supported for Code Acceleration or Generation” on page 8-5
- “Fixed-Point Code Acceleration and Generation Workflow” on page 8-14
- “Set Up Compiler to Generate Compiled C Code Functions” on page 8-15
- “Accelerate Code Using fiaccel” on page 8-16
- “File Infrastructure and Paths Setup” on page 8-21
- “Detect and Debug Code Generation Errors” on page 8-25
- “Set Up C Code Compilation Options” on page 8-28
- “MEX Configuration Dialog Box Options ” on page 8-30
- “Specify Primary Function Input Properties” on page 8-36
- “Best Practices for Accelerating Fixed-Point Code” on page 8-48
- “Create and Use Fixed-Point Code Generation Reports” on page 8-52
- “Generate C Code from Code Containing Global Data” on page 8-57
- “Define Input Properties Programmatically in MATLAB File” on page 8-63
- “Control Run-Time Checks” on page 8-71
- “Generation with MATLAB® Coder™ ” on page 8-74

- “Code Generation with MATLAB Function Block” on page 8-75
- “Generate Fixed-Point FIR Code Using MATLAB Function Block” on page 8-84
- “Fixed-Point FFT Code Example Parameter Values” on page 8-89
- “Accelerate Code for Variable-Size Data” on page 8-92
- “Accelerate C Code for Fixed-Point Mean Value (TBD)” on page 8-101
- “Create Embeddable C Code for Summing Values (TBD)” on page 8-102
- “Propose Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-103
- “Apply Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-113
- “Code Generation Readiness Tool” on page 8-119
- “Check Code Using the Code Generation Readiness Tool” on page 8-125

Code Acceleration and Code Generation from MATLAB

In many cases, you may want your code to run faster and more efficiently. *Code acceleration* provides optimizations for accelerating fixed-point algorithms through MEX file building. In Fixed-Point Toolbox the `fiaccel` function converts your MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.

Code generation creates efficient, production-quality C/C++ code for desktop and embedded applications. There are several ways to use Fixed-Point Toolbox software to generate C/C++ code.

Use...	To...	Requires...	See...
MATLAB Coder™ (codegen) function	Automatically convert MATLAB code to C/C++ code	MATLAB Coder code generation software license	“C Code Generation at the Command Line” in the MATLAB Coder documentation
MATLAB Function	Use MATLAB code in your Simulink models that generate embeddable C/C++ code	Simulink license	in the Simulink documentation

MATLAB code generation supports variable-size arrays and matrices with known upper bounds. To learn more about using variable-size signals, see “What Is Variable-Size Data?” on page 21-2 in the Code Generation for MATLAB documentation.

Requirements for Generating Compiled C Code Files

You use the `fiaccel` function to generate MEX code from a MATLAB algorithm. The algorithm must meet these requirements:

- Must be a MATLAB function, not a script
- Must meet the requirements listed on the `fiaccel` reference page
- Does not call custom C code using any of the following MATLAB Coder constructs:
 - `coder.ceval`
 - `coder.ref`
 - `coder.rref`
 - `coder.wref`

Functions Supported for Code Acceleration or Generation

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

- `fipref` and quantizer objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given `fi` variable after that variable has been created.
- The boolean value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- You can use parallel `for` (`parfor`) loops in code compiled with `fiaccel`, but those loops are treated like regular `for` loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- All general limitations of C/C++ code generated from MATLAB apply. See “MATLAB Language Features Not Supported for C/C++ Code Generation” for more information.

Function	Remarks/Limitations
<code>abs</code>	N/A
<code>add</code>	N/A
<code>all</code>	N/A
<code>any</code>	N/A
<code>bitand</code>	Not supported for slope-bias scaled <code>fi</code> objects.
<code>bitandreduce</code>	N/A

Function	Remarks/Limitations
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	N/A
bitsra	N/A
bitsrl	N/A
bitxor	Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	N/A
ceil	N/A
complex	N/A
conj	N/A

Function	Remarks/Limitations
conv	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordicccexp	Variable-size signals are not supported.
cordicccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
ctranspose	N/A
diag	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
disp	N/A

Function	Remarks/Limitations
divide	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
double	N/A
end	N/A
eps	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, fi single and fi double signals.
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> The default constructor syntax without any input arguments is not supported. If the <code>numerictype</code> is not fully specified, the input to <code>fi</code> must be a constant, a <code>fi</code>, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows <code>fi</code> to autoscale its fraction length based on the known data type of the input. <code>numerictype</code> object information must be available for nonfixed-point Simulink inputs.
filter	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
fimath	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer. Use to create <code>fimath</code> objects in the generated code.
fix	N/A
floor	N/A
ge	Not supported for fixed-point signals with different biases.

Function	Remarks/Limitations
get	The syntax <code>structure = get(o)</code> is not supported.
getlsb	N/A
getmsb	N/A
gt	Not supported for fixed-point signals with different biases.
horzcat	N/A
imag	N/A
int8, int16, int32	N/A
iscolumn	N/A
isempty	N/A
isequal	N/A
isfi	N/A
isfimath	N/A
isfimathlocal	N/A
isfinite	N/A
isinf	N/A
isnan	N/A
isnumeric	N/A
isnumerictype	N/A
isreal	N/A
isrow	N/A
isscalar	N/A
issigned	N/A
isvector	N/A
le	Not supported for fixed-point signals with different biases.
length	N/A
logical	N/A

Function	Remarks/Limitations
lowerbound	N/A
lsb	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	Not supported for fixed-point signals with different biases.
max	N/A
mean	N/A
median	N/A
min	N/A
minus	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mpower	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when the first input, a, is nonscalar. However, when a is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
mpy	When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code> .

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

Function	Remarks/Limitations
power	The exponent input, k , must be constant; that is, its value must be known at compile time.
range	N/A
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterprecast	N/A
repmat	N/A
rescale	N/A
reshape	N/A
round	N/A
sfi	N/A
sign	N/A
single	N/A
size	N/A
sort	N/A
sqrt	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.

Function	Remarks/Limitations
times	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32	N/A
uminus	N/A
uplus	N/A
upperbound	N/A
vertcat	N/A

Fixed-Point Code Acceleration and Generation Workflow

Step	Action	Details
1	Set up your C compiler.	See “Set Up Compiler to Generate Compiled C Code Functions” on page 8-15
2	Set up your file infrastructure.	See “File Infrastructure and Paths Setup” on page 8-21.
3	Make your MATLAB algorithm suitable for code generation	See “Best Practices for Accelerating Fixed-Point Code” on page 8-48.
4	Set compilation options.	See “Set Up C Code Compilation Options” on page 8-28
5	Specify properties of primary function inputs.	See “Specify Primary Function Input Properties” on page 8-36.
6	Run <code>fiaccel</code> with the appropriate command-line options.	See “Recommended Compilation Options for <code>fiaccel</code> ” on page 8-48

Set Up Compiler to Generate Compiled C Code Functions

Set up your C compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference. You must run this command even if you use the default C compiler that comes with MATLAB for Windows® platforms. You can also use `mex` to choose and configure a different C compiler, as described in “What You Need to Build MEX-Files” in the MATLAB External Interfaces documentation.

You can use the following compilers to generate MEX functions with `fiaccl`:

- Lcc-win32 C 2.4.1
- Microsoft® Visual C++® 2008 Express
- Microsoft Visual C++ 2005
- Microsoft Visual C++ 6.0
- Open Watcom C++ 1.7
- GCC

Accelerate Code Using `fiaccel`

In this section...

“Speeding Up Fixed-Point Execution with `fiaccel`” on page 8-16

“Running `fiaccel`” on page 8-16

“Generated Files and Locations” on page 8-17

“Data Type Override Using `fiaccel`” on page 8-20

Speeding Up Fixed-Point Execution with `fiaccel`

You can convert fixed-point MATLAB code to MEX functions using `fiaccel`. The generated MEX functions contain optimizations to automatically accelerate fixed-point algorithms to compiled C/C++ code speed in MATLAB. The `fiaccel` function can greatly increase the execution speed of your algorithms.

Running `fiaccel`

The basic command is:

```
fiaccel M_fcn
```

By default, `fiaccel` performs the following actions:

- Searches for the function `M_fcn` stored in the file `M_fcn.m` as specified in “Compile Path Search Order” on page 8-21.
- Compiles `M_fcn` to MEX code.
- If there are no errors or warnings, generates a platform-specific MEX file in the current folder, using the naming conventions described in “File Naming Conventions” on page 8-51.
- If there are errors, does not generate a MEX file, but produces an error report in a default output folder, as described in “Generated Files and Locations” on page 8-17.
- If there are warnings, but no errors, generates a platform-specific MEX file in the current folder, but does report the warnings.

You can modify this default behavior by specifying one or more compiler options with `fiaccel`, separated by spaces on the command line.

Generated Files and Locations

`fiaccel` generates files in the following locations:

Generates:	In:
Platform-specific MEX files	Current folder
HTML reports (if errors or warnings occur during compilation)	Default output folder: <code>fiaccel/mex/M_fcn_name/html</code>

You can change the name and location of generated files by using the options `-o` and `-d` when you run `fiaccel`.

In this example, you will use the `fiaccel` function to compile different parts of a simple algorithm. By comparing the run times of the two cases, you will see the benefits and best use of the `fiaccel` function.

Example: Comparing Run Times When Accelerating Different Algorithm Parts

The algorithm used throughout this example replicates the functionality of the MATLAB `sum` function, which sums the columns of a matrix. To see the algorithm, type `open fi_matrix_column_sum.m` at the MATLAB command line.

```
function B = fi_matrix_column_sum(A)
% Sum the columns of matrix A.
%#codegen
    [m,n] = size(A);
    w = get(A,'WordLength') + ceil(log2(m));
    f = get(A,'FractionLength');
    B = fi(zeros(1,n),true,w,f);
    for j = 1:n
        for i = 1:m
            B(j) = B(j) + A(i,j);
```

```
        end  
    end
```

Trial 1: Best Performance

The best way to speed up the execution of the algorithm is to compile the entire algorithm using the `fiaccel` function. To evaluate the performance improvement provided by the `fiaccel` function when the entire algorithm is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the entire algorithm using the `fiaccel` function. The MATLAB `tic` and `toc` functions keep track of the run times for each method of execution.

```
% MATLAB  
fipref('NumericTypeDisplay','short');  
A = fi(randn(1000,10));  
tic  
B = fi_matrix_column_sum(A)  
t_matrix_column_sum_m = toc  
  
% fiaccel  
fiaccel fi_matrix_column_sum -args {A} ...  
-I [matlabroot '/toolbox/fixedpoint/fidemos']  
tic  
B = fi_matrix_column_sum_mex(A);  
t_matrix_column_sum_mex = toc
```

Trial 2: Worst Performance

Compiling only the smallest unit of computation using the `fiaccel` function leads to much slower execution. In some cases, the overhead that results from calling the `mex` function inside a nested loop can cause even slower execution than using MATLAB functions alone. To evaluate the performance of the `mex` function when only the smallest unit of computation is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the smallest unit of computation with the `fiaccel` function, leaving the rest of the computations to MATLAB.

```

% MATLAB
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_m = toc

% fiaccel
fiaccel fi_scalar_sum -args {B(1),A(1,1)} ...
-I [matlabroot ' /toolbox/fixedpoint/fidemos']
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum_mex(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_mex = toc

```

Ratio of Times

A comparison of Trial 1 and Trial 2 appears in the following table. Your computer may record different times than the ones the table shows, but the ratios should be approximately the same. There is an extreme difference in ratios between the trial where the entire algorithm was compiled using `fiaccel` (`t_matrix_column_sum_mex.m`) and where only the scalar sum was compiled (`t_scalar_sum_mex.m`). Even the file with no `fiaccel` compilation (`t_matrix_column_sum_m`) did better than when only the smallest unit of computation was compiled using `fiaccel` (`t_scalar_sum_mex`).

X (Overall Performance Rank)	Time	X/Best	X_m/X_mex
Trial 1: Best Performance			
t_matrix_column_sum_m (2)	1.99759	84.4917	84.4917
t_matrix_column_sum_mex (1)	0.0236424	1	
Trial 2: Worst Performance			
t_scalar_sum_m (4)	10.2067	431.71	2.08017
t_scalar_sum_mex (3)	4.90664	207.536	

Data Type Override Using `fiaccel`

Fixed-Point Toolbox software ships with an example of how to generate a MEX function from MATLAB code. The code in the example takes the weighted average of a signal to create a lowpass filter. To run the example in the Help browser select Examples under Fixed-Point Toolbox, and then select Fixed-Point Lowpass Filtering Using MATLAB for Code Generation.

You can specify data type override in this example by typing an extra command at the MATLAB prompt in the “Define Fixed-Point Parameters” section of the example. To turn data type override on, type the following command at the MATLAB prompt after running the `reset(fipref)` command in that section:

```
fipref('DataTypeOverride','TrueDoubles')
```

This command tells Fixed-Point Toolbox software to create all `fi` objects with type `fi double`. When you compile the code using the `fiaccel` command in the “Compile the M-File into a MEX File” section of the example, the resulting MEX-function uses floating-point data.

File Infrastructure and Paths Setup

In this section...

“Compile Path Search Order” on page 8-21

“When to Use the Code Generation Path” on page 8-21

“Add Files to the Code Generation Path” on page 8-22

“Adding Folders to Search Paths” on page 8-22

“Naming Conventions” on page 8-22

Compile Path Search Order

`fiaccel` resolves function calls by searching first on the code generation path and then on the MATLAB path. By default, `fiaccel` tries to compile and generate code for functions it finds on the path unless you explicitly declare the function to be extrinsic. An *extrinsic function* is a function on the MATLAB path that is dispatched to MATLAB software for execution. `fiaccel` does not compile extrinsic functions, but rather dispatches them to MATLAB for execution.

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. Because `fiaccel` searches the code generation path first, a MATLAB file on that path always shadows a MATLAB file of the same name on the MATLAB path.

To override a MATLAB function with a customized version:

- 1 Create each version of the MATLAB function in identically named files.
- 2 Add the MATLAB version to the MATLAB path.
- 3 Add the customized version to the code generation path.

See “Adding Folders to Search Paths” on page 8-22.

Add Files to the Code Generation Path

With `fiaccel`, you can prepend folders and files to the code generation path, as described in “Adding Folders to Search Paths” on page 8-22. By default, the code generation path contains the current folder and the toolbox functions supported for code generation.

Adding Folders to Search Paths

To add folders to:	Do this:
Code generation path	Prepend folders to the code generation path by using the <code>fiaccel -I</code> option.
MATLAB path	Follow the instructions in in the MATLAB Programming documentation.

Naming Conventions

MATLAB enforces naming conventions for functions and generated files.

- “Reserved Prefixes” on page 8-22
- “Reserved Keywords” on page 8-22
- “Conventions for Naming Generated files” on page 8-24

Reserved Prefixes

MATLAB reserves the prefix `eml` for global C functions and variables in generated code. For example, run-time library function names all begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C functions or primary MATLAB functions with the prefix `eml`.

Reserved Keywords

- “C Reserved Keywords” on page 8-23
- “C++ Reserved Keywords” on page 8-23
- “Reserved Keywords for Code Generation” on page 8-24

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains any reserved keywords, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the “C++ Reserved Keywords” on page 8-23.

C Reserved Keywords.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C++ Reserved Keywords.

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t
export	private	throw	

Reserved Keywords for Code Generation.

abs	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen
boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true
cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_SIZE	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMPLES	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T
cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

Conventions for Naming Generated files

MATLAB provides platform-specific extensions for MEX files.

Platform	MEX File Extension
Linux® x86-64	.mexa64
Windows (32-bit)	.mexw32
Windows x64	.mexw64

Detect and Debug Code Generation Errors

In this section...
“Debugging Strategies” on page 8-25
“Error Detection at Design Time” on page 8-26
“Error Detection at Compile Time” on page 8-26

Debugging Strategies

To prepare your algorithms for code generation, MathWorks recommends that you choose a debugging strategy for detecting and correcting violations in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other’s functions. Here are two best practices:

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<div><div>1</div>Verify that your lowest-level (leaf) functions are suitable for code generation.</div> <div><div>2</div>Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function.</div>	<ul style="list-style-type: none">• Efficient• Safe• Easy to isolate syntax violations	Requires application tests that work from the bottom up

Debugging Strategy	What to Do	Pros	Cons
Top-down verification	<ol style="list-style-type: none">1 Declare all functions called by the top-level function to be extrinsic so <code>fiaccl</code> does not compile them.2 Verify that your top-level function is suitable for code generation.3 Work downward in the function hierarchy to:<ol style="list-style-type: none">a. Remove extrinsic declarations one by oneb. Compile and verify each function, ending with the leaf functions.	Lets you retain your top-level tests	<p>Introduces extraneous code that you must remove after code verification, including:</p> <ul style="list-style-type: none">• Extrinsic declarations• Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values.

Error Detection at Design Time

To detect potential issues for MEX file building as you write your MATLAB algorithm, add the `%codegen` directive to the code that you want `fiaccl` to compile. Adding this directive indicates that you intend to generate code from the algorithm and turns on detailed diagnostics during MATLAB code analysis (see in the MATLAB Desktop Tools and Development Environment documentation).

Error Detection at Compile Time

Before you can successfully generate code from a MATLAB algorithm, you must verify that the algorithm does not contain syntax and semantics violations that would cause compile-time errors, as described in “Detect and Debug Code Generation Errors” on page 8-25.

`fiaccel` checks for all potential syntax violations at compile time. When `fiaccel` detects errors or warnings, it automatically produces a code generation report that describes the issues and provides links to the offending code. See “Create and Use Fixed-Point Code Generation Reports” on page 8-52.

If your MATLAB code calls functions on the MATLAB path, `fiaccel` attempts to compile these functions unless you declare them to be extrinsic.

Set Up C Code Compilation Options

In this section...

“C Code Compiler Configuration Object” on page 8-28

“Compilation Options Modification at the Command Line Using Dot Notation” on page 8-28

“How fiaccel Resolves Conflicting Options” on page 8-29

C Code Compiler Configuration Object

For C code generation to a MEX file, MATLAB provides a configuration object `coder.MEXConfig` for fine-tuning the compilation. To set MEX compilation options:

- 1 Define the compiler configuration object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.mexconfig
```

MATLAB displays the list of compiler options and their current values in the command window.

- 2 Modify the compilation options as necessary. See “Compilation Options Modification at the Command Line Using Dot Notation” on page 8-28
- 3 Invoke `fiaccel` with the `-config` option and specify the configuration object as its argument:

```
fiaccel -config comp_cfg myMfile
```

The `-config` option instructs `fiaccel` to convert `myFile.m` to a MEX function, based on the compilation settings in `comp_cfg`.

Compilation Options Modification at the Command Line Using Dot Notation

Use dot notation to modify the value of compilation options, using this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties. For example, to change the maximum size function to inline and the stack size limit for inlined functions during MEX generation, enter this code at the command line:

```
co_cfg = coder.MEXConfig
co_cfg.InlineThreshold = 25;
co_cfg.InlineStackLimit = 4096;
fiaccel -config co_cfg myFun
```

How fiaccel Resolves Conflicting Options

`fiaccel` takes the union of all options, including those specified using configuration objects, so that you can specify options in any order.

MEX Configuration Dialog Box Options

MEX Configuration Dialog Box Options

The following table describes parameters for fine-tuning the behavior of `fiaccel` for converting MATLAB files to MEX:

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Report		
Create code generation report	GenerateReport true, false	Document generated code in an HTML report.
Launch report automatically	LaunchReport true, false	Specify whether to automatically display HTML reports after code generation completes. Note Requires that you enable Create code generation report
Debugging		
Echo expressions without semicolons	EchoExpressions true , false	Specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window.
Enable debug build	EnableDebugging true, false	Compile the generated code in debug mode.
Language and Semantics		
Constant Folding Timeout	ConstantFoldingTimeout <i>integer</i> , 10000	Specify the maximum number of instructions to be executed by the constant folder.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Dynamic memory allocation	DynamicMemoryAllocation 'off' , 'AllVariableSizeArrays'	Enable dynamic memory allocation for variable-size data. By default, dynamic memory allocation is disabled and fiaccl allocates memory statically on the stack. When you select dynamic memory allocation, fiaccl allocates memory for all variable-size data dynamically on the heap. You <i>must</i> use dynamic memory allocation for all unbounded variable-size data.
Enable variable sizing	EnableVariableSizing true , false	Enable support for variable-size arrays.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Extrinsic calls	ExtrinsicCalls true , false	<p>Allow calls to extrinsic functions.</p> <p>When enabled (true), the compiler generates code for the call to a MATLAB function, but does not generate the function's internal code.</p> <p>When disabled (false), the compiler ignores the extrinsic function. Does not generate code for the call to the MATLAB function—as long as the extrinsic function does not affect the output of the caller function. Otherwise, the compiler issues a compiler error.</p>

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Global Data Synchronization Mode	GlobalDataSyncMethod <i>string</i> , SyncAlways , SyncAtEntryAndExits, NoSync	<p>Controls when global data is synchronized with the MATLAB global workspace. By default, (SyncAlways), synchronizes global data at MEX function entry and exit and for all extrinsic calls. This synchronization ensures maximum consistency between MATLAB and generated code. If the extrinsic calls do not affect global data, use this option with the <code>coder.extrinsic -sync:off</code> option to turn off synchronization for these calls.</p> <p>SyncAtEntryAndExits synchronizes global data at MEX function entry and exit only. If only a few extrinsic calls affect global data, use this option with the <code>coder.extrinsic -sync:on</code> option to turn on synchronization for these calls.</p> <p>NoSync disables synchronization. Ensure that your generated code does not interact with MATLAB before disabling synchronization. Otherwise, inconsistencies might occur.</p>

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Saturate on integer overflow	SaturateOnIntegerOverflow true , false	Add checks in the generated code to detect integer overflow or underflow.
Safety (disable for faster MEX)		
Ensure memory integrity	IntegrityChecks true , false	Detects violations of memory integrity in code generated from MATLAB algorithms and stops execution with a diagnostic message. Setting IntegrityChecks to false also disables the run-time stack.
Ensure responsiveness	ResponsivenessChecks true , false	Enables responsiveness checks in code generated from MATLAB algorithms.
Function Inlining and Stack Allocation		
Inline Stack Limit	InlineStackLimit <i>integer</i> , 4000	Specify the stack size limit on inlined functions.
Inline Threshold	InlineThreshold <i>integer</i> , 10	Specify the maximum size of functions to be inlined.
Inline Threshold Max	InlineThresholdMax <i>integer</i> , 200	Specify the maximum size of functions after inlining.
Stack Usage Max	StackUsageMax <i>integer</i> , 200000	Specify the maximum stack usage per application in bytes. Set a limit that is lower than the available stack size. Otherwise, a runtime stack overflow might occur. Overflows are detected and reported by the C compiler, not by <code>fiaccel</code> .

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Optimizations		
Use BLAS library if possible	EnableBLAS true , false	Speed up low-level matrix operations during simulation by calling the Basic Linear Algebra Subprograms (BLAS) library.

See Also

- “Control Run-Time Checks” on page 8-71
- “Variable-Size Data Definition for Code Generation” on page 21-3
- “Generate C Code from Code Containing Global Data” on page 8-57

Specify Primary Function Input Properties

In this section...

“Why You Must Specify Input Properties” on page 8-36

“Properties to Specify” on page 8-36

“Rules for Specifying Properties of Primary Inputs” on page 8-39

“Methods for Defining Properties of Primary Inputs” on page 8-40

“Input Properties Definition by Example at the Command Line” on page 8-40

Why You Must Specify Input Properties

To generate code in a statically typed language, `fiaccel` must determine the properties of all variables in the MATLAB code at compile time. Therefore, you must specify the class, size, and complexity of inputs to the primary function (also known as the *top-level* or *entry-point* function). If your primary function has no input parameters, `fiaccel` can compile your MATLAB algorithm without modification. You do not need to specify properties of inputs to local or external functions called by the primary function. For `fiaccel` requirements, refer to its reference page.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input:

For:	Specify Properties:				
	Class	Size	Complexity	numericity	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Structure inputs*	✓	✓	✓	✓ (if structure field is fixed-point)	✓ (if structure field is fixed-point)
All other inputs	✓	✓	✓		

* When a primary input is a structure, `fiaccl` treats each field as a separate input.

Default Property Values

`fiaccl` assigns the following default values for properties of primary function inputs:

Property	Default
class	double
size	scalar
complexity	real
numericity	No default
fimath	MATLAB default fimath object

Specifying Default Values for Structure Fields. In most cases, `fiaccl` uses defaults when you don't explicitly specify values for properties—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 may need to specify default values for properties of structure fields. For examples, see “Example: Specifying Class and Size of Scalar Structure” on page 8-69 and “Example: Specifying Class and Size of Structure Array” on page 8-70.

Specifying Default fimath Values for MEX Functions. MEX functions generated with `fiaccel` use the MATLAB default `fimath`. The MATLAB factory default `fimath` has the following properties:

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
```

For more information, see “`fimath` Object Construction” on page 4-2.

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time error, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` will rely on the default `fimath` object at compile time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```
fiaccel test
% fiaccel 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

Supported Classes

The following table presents the class names supported by `fiaccl`:

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
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

Follow these rules when specifying the properties of primary inputs:

- For each primary function input whose class is fixed point (`fi`), you must specify the input's `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

You can use any of the following methods to define the properties of primary function inputs:

Method	Pros	Cons
“Input Properties Definition by Example at the Command Line” on page 8-40	<ul style="list-style-type: none">• Easy to use• Does not alter original MATLAB code• Designed for prototyping a function that has a small number of primary inputs	<ul style="list-style-type: none">• Must be specified at the command line every time you invoke <code>fiaccel</code> (unless you use a script)• Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties Programmatically in MATLAB File” on page 8-63	<ul style="list-style-type: none">• Integrated with MATLAB code so you do not need to redefine properties each time you invoke <code>fiaccel</code>• Provides documentation of property specifications in the MATLAB code• Efficient for specifying memory-intensive inputs such as large structures	<ul style="list-style-type: none">• Uses complex syntax

Note To specify the properties of inputs for any given primary function, use one of these methods or the other, but not both.

Input Properties Definition by Example at the Command Line

- “Command Line Option `-args`” on page 8-41
- “Rules for using the `-args` option” on page 8-42
- “Specifying Constant Inputs” on page 8-43

- “Specifying Variable-Size Inputs” on page 8-44

Command Line Option -args

`fiaccel` provides a command-line option `-args` for specifying the properties of primary function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values.

Example: Specifying Properties of Primary Inputs by Example.

Consider a function that adds its two inputs:

```
function y = emcf(u,v) %#codegen
% The directive %#codegen indicates that you
% intend to generate code for this algorithm
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, fixed-point values:

```
fiaccel -o emcfx emcf ...
    -args {fi(0,1,16,15),fi(0,1,16,15)}
```

- 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, fixed-point value:

```
fiaccel -o emcfx emcf ...
    -args {zeros(1,4,'uint16'),fi(0,1,16,15)}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = fi([1;2;3;4],0,8,0)
b = fi([5;6;7;8],0,8,0)
ex = {a,b}
fiaccel -o emcfx emcf -args ex
```

Example: Specifying Properties of Primary Fixed-Point Inputs by Example.

Consider a function that calculates the square root of a fixed-point number:

```
function y = sqrtfi(x) %#codegen  
y = sqrt(x);
```

To specify the properties of the primary fixed-point input `x` by example on the MATLAB command line, follow these steps:

- 1 Define the `numerictype` properties for `x`, as in this example:

```
T = numerictype('WordLength',32,...  
    'FractionLength',23,'Signed',true);
```

- 2 Define the `fimath` properties for `x`, as in this 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 you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

- 4 Compile the function `sqrtfi` using the `fiaccel` command, passing the variable `myeg` as the argument to the `-args` option, as in this example:

```
fiaccel sqrtfi -args myeg;
```

Rules for using the `-args` option

Follow these rules when using the `-args` command-line option to define properties by example:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Specifying Constant Inputs

In cases where you know your primary inputs will not change at run time, you can specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using this command-line option:

```
-args {coder.Constant(constant_input)}
```

This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs. `fiaccel` compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function `identity` which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, type the following command at the MATLAB prompt:

```
fiaccel -o identity_mex identity...
    -args {coder.Constant(fi(0.1,1,16,15))}
```

To run the MATLAB function, supply the constant argument as follows:

```
identity(fi(0.1,1,16,15))
```

You get the following result:

```
ans =

    0.1000
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Example: Specifying a Structure as a Constant Input. Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix, as follows:

```
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 = fi(zeros(p.rows,p.cols),1,16,15) + 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:

```
fiaccel rowcol ...
    -args {fi(0,1,16,15),coder.Constant(tmp)}
```

To run this code, use

```
u = fi(0.5,1,16,15)
y_m = rowcol(u,tmp)

y_mex = rowcol_mex(u)
```

Specifying Variable-Size Inputs

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:

Expression	Description
------------	-------------

Expression	Description
<code>-args {coder.typeof(<i>example_value</i>, <i>size_vector</i>, <i>dim_vector</i>)}</code>	<p>Specifies a variable-size input with:</p> <ul style="list-style-type: none"> • Same class and complexity as <i>example_value</i> • Same size and upper bounds as <i>size_vector</i> <p><i>dim_vector</i> specifies which dimensions are variable. A value of <code>true</code> or <code>one</code> means that the corresponding dimension is variable. A value of <code>false</code> or <code>zero</code> means that the corresponding dimension is fixed.</p>

Example: Specifying a Variable-Size Vector Input.

- 1 Write a function that computes the sum of every *n* elements of a vector *A* and stores them in a vector *B*:

```
function B = nway(A,n) %#codegen
% Compute sum of every N elements of A and put them in B.

coder.extrinsic('error');
Tb = numerictype(1,32,24);
if ((mod(numberofelements(A),n) == 0) && ...
    (n>=1 && n<=numberofelements(A)))
    B = fi(zeros(1,numberofelements(A)/n),Tb);
    k = 1;
    for i = 1 : numberofelements(A)/n
        B(i) = sum(A(k + (0:n-1)));
        k = k + n;
    end
else
    B = fi(zeros(1,0),Tb);
    error('n<=0 or does not divide evenly');
```

end

- 2** Specify the first input `A` as a `fi` object. 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.

```
fiaccel nway ...  
    -args {coder.typeof(fi(0,1,16,15),[1 100],1),0}...  
    -report
```

Note You do not need to explicitly cast these inputs as `double` because `fiaccel` assumes the default properties of inputs are real, double scalars.

- 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(fi(0,1,16,15),[1 100],1)  
fiaccel nway -args {vareg, double(0)}
```

Note For comparison, this command does explicitly cast the inputs to `double`.

Best Practices for Accelerating Fixed-Point Code

In this section...

- “Recommended Compilation Options for `fiaccel`” on page 8-48
- “Build Scripts” on page 8-49
- “Check Code Interactively Using MATLAB Code Analyzer” on page 8-50
- “Separating Your Test Bench from Your Function Code” on page 8-51
- “Preserving Your Code” on page 8-51
- “File Naming Conventions” on page 8-51

Recommended Compilation Options for `fiaccel`

- `-args` – Specify input parameters by example

Use the `-args` option to specify the properties of primary function inputs as a cell array of example values at the same time as you generate code for the MATLAB file with `fiaccel`. The cell array can be a variable or literal array of constant values. The cell array should provide the same number and order of inputs as the primary function.

When you use the `-args` option you are specifying the data types and array dimensions of these parameters, not the values of the variables. For more information, see “Define Input Properties by Example at the Command Line” in the MATLAB Coder documentation.

Note Alternatively, you can use the `assert` function to define properties of primary function inputs directly in your MATLAB file. For more information, see “Define Input Properties Programmatically in MATLAB File” on page 8-63.

- `-report` – Generate code generation report

Use the `-report` option to generate a report in HTML format at code generation time to help you debug your MATLAB code and verify that it

is suitable for code generation. If you do not specify the `-report` option, `fiaccel` generates a report only if build errors or warnings occur.

The code generation report contains the following information:

- Summary of code generation results, including type of target and number of warnings or errors
- Target build log that records build and linking activities
- Links to generated files
- Error and warning messages (if any)

For more information, see `fiaccel`.

Build Scripts

Use build scripts to call `fiaccel` to generate MEX functions from your MATLAB function.

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors. For instance, you can use a build script to clear your workspace before each build and to specify code generation options.

This example shows a build script to run `fiaccel` to process `lms_02.m`:

```
close all;
clear all;
clc;

N = 73113;

fiaccel -report lms_02.m ...
    -args { zeros(N,1) zeros(N,1) }
```

In this example, the following actions occur:

- `close all` deletes all figures whose handles are not hidden. See `close` in the MATLAB Graphics function reference for more information.

- `clear all` removes all variables, functions, and MEX-files from memory, leaving the workspace empty. This command also clears all breakpoints.

Note Remove the `clear all` command from the build scripts if you want to preserve breakpoints for debugging.

- `clc` clears all input and output from the Command Window display, giving you a “clean screen.”
- `N = 73113` sets the value of the variable `N`, which represents the number of samples in each of the two input parameters for the function `lms_02`
- `fiaccel -report lms_02.m -args { zeros(N,1) zeros(N,1) }` calls `fiaccel` to accelerate simulation of the file `lms_02.m` using the following options:
 - `-report` generates a code generation report
 - `-args { zeros(N,1) zeros(N,1) }` specifies the properties of the function inputs as a cell array of example values. In this case, the input parameters are `N`-by-1 vectors of real doubles.

Check Code Interactively Using MATLAB Code Analyzer

The code analyzer checks your code for problems and recommends modifications to maximize performance and maintainability. You can use the code analyzer to check your code continuously in the MATLAB Editor while you work.

To ensure that continuous code checking is enabled:

- 1 From the MATLAB menu, select **File > Preferences > Code Analyzer**.

The list of code analyzer preferences appears.

- 2 Select the **Enable integrated warning and error messages** check box.

Separating Your Test Bench from Your Function Code

Separate your core algorithm from your test bench. Create a separate test script to do all the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results. See the example on the `fiaccel` reference page.

Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention, as described in “File Naming Conventions” on page 8-51. For example, add a 2-digit suffix to the file name for each file in a sequence. Alternatively, use a version control system.

File Naming Conventions

Use a consistent file naming convention to identify different types and versions of your MATLAB files. This approach keeps your files organized and minimizes the risk of overwriting existing files or creating two files with the same name in different folders.

For example, the file naming convention in the Generating MEX Functions getting started tutorial is:

- The suffix `_build` identifies a build script.
- The suffix `_test` identifies a test script.
- A numerical suffix, for example, `_01` identifies the version of a file. These numbers are typically two-digit sequential integers, beginning with 01, 02, 03, and so on.

For example:

- The file `build_01.m` is the first version of the build script for this tutorial.
- The file `test_03.m` is the third version of the test script for this tutorial.

Create and Use Fixed-Point Code Generation Reports

In this section...

“Code Generation Report Creation” on page 8-52

“Code Generation Report Opening” on page 8-53

“Viewing Your MATLAB Code” on page 8-53

“Viewing Variables in the Variables Tab” on page 8-55

“See Also” on page 8-56

Code Generation Report Creation

When you compile your code with the `fiaccel` function or the MATLAB Coder `codegen` function, you can use the `-report` option to generate a code generation report. This report allows you to examine the data types of the variables and expressions in your code.

To see an example of the code generation report generated by the `fiaccel` function, compile `cordic_atan_kernel.m`. This file ships as a part of the Fixed-Point ATAN2 Calculation example. You can open the file by typing the following at the MATLAB command line:

```
open cordic_atan_kernel
```

To compile the `cordic_atan_kernel` file, you must provide inputs `x`, `y`, `N`, and `angleLUT`. This example uses the following input values:

```
x = fi(0.23);
y = x;
N = 12;
Tz = numerictype(1,16,13);
angleLUT = fi(atan(2.^(0:N-1)), 'NumericType', Tz);
```

After you define the input variables in the MATLAB workspace, change your working folder to a local folder and compile the file using `fiaccel`. Use the `-report` option to generate the code generation report:

```
fiaccel cordic_atan_kernel -args {x,y,N,angleLUT} -report
```

Code Generation Report Opening

If the compilation is successful, you receive the following message:

Code generation successful: [View report](#)

Click the **View report** link to view the report.

If the compilation fails, a link to the error report appears:

Code generation failed: [View report](#)

Click the **View report** link to view the error report and debug your code. For more information on working with error reports, see “Code Generation Reports” in the MATLAB Coder documentation.

Viewing Your MATLAB Code

When the code generation report opens, you can hover your cursor over the variables and expressions in your MATLAB code to see their data type information. The code generation report provides color-coded data type information according to the following legend.

Color	Meaning
Green	Data type information is available for the selected variable at this location in the code.
Orange	There is a warning message associated with the selected variable or expression.
Pink	No data type information is available for the selected variable.
Purple	Data type information is available for the selected expression at this location in the code.
Red	There is an error message associated with the selected variable or expression.

Variables in your code that have data type information available appear highlighted in green, as shown in the following figure.

Function: cordic_atan_kernel Callers: Select a function call-site:

```

1 function [z,x,y] = cordic_atan_kernel(y,x,N,angleLUT) %#codegen
2 % Calculate arctangent in range [-pi/2, pi/2] using Vectoring mode CORDIC
3 % algorithm. Both x and y inputs must be real scalar, x must >= 0.
4 % Full precision Fimath is used in all fixed-point operations
5 %
6 % Inputs:
7 % y : y coordinate or imaginary part of the input vector
8 % x : x coordinate or real part of the input vector
9 % N : total number of iterations, must be a non-negative integer
10 % angleLUT : the angle look-up table, has same numerictype as the output
11 %         angle
12 % Output:
13 % z : angle that equals atan2(y,x), in radians
14 %     the output angle range is within [-pi/2, +pi/2]
15 % x : x coordinate of the last vector at the end of the iterations
16 % y : y coordinate of the last vector at the end of the iterations
17 %
18 % Copyright 1984-2010 The MathWorks, Inc.
19 % $Revision: 1.1.6.2 $ $Date: 2010/10/15 14:11:18 $
20
21 % initialization
22 z = angleLUT(1); z(:) = 0; % assume z_{0} is 0 and the same data type as angleLUT
23 for i = 0:N-1,
24     x0 = x;
25     if y < 0 % negative y leads to counter clock-wise rotation
26         x(i) = x0 - bitara(y,i); % x_{i+1} = x_{i} - y_{i}>>i
27         y(i) = y0 + bitara(x,i); % y_{i+1} = y_{i} + x_{i}>>i
28         z(i) = z(i) + atan(2^(-i))
29     else
30         x(i) = x0 + bitara(y,i); % x_{i+1} = x_{i} + y_{i}>>i
31         y(i) = y0 - bitara(x,i); % y_{i+1} = y_{i} - x_{i}>>i
32         z(i) = z(i) - atan(2^(-i))
33     end
34 end
35

```

Information for the selected variable:

Size	1x1
Complex	No
Class	embedded.fi
Signedness	Signed
WL	16
FL	17

Expressions in your code that have data type information available appear highlighted in purple, as the next figure shows.

```

21 % initialization
22 z = angleLUT(1); z(:) = 0; % assume z_{0} is 0 and the same data type as
23 for i = 0:N-1
24     x = ...
25     i = ...
26     y = x_{i} - y_{i} >> i
27     z = y_{i} + x_{i} >> i
28     z = z_{i} + atan(2^{-i})
29     e = ...
30     y = x_{i} + y_{i} >> i
31     y(:) = y - bitsra(x0,i); % y_{i+1} = y_{i} - x_{i} >> i
32     z(:) = z + angleLUT(i+1); % z_{i+1} = z_{i} - atan(2^{-i})
33 end
34 end
35

```




Information for the selected expression:

Size	1 x 1
Complex	No
Class	embedded.fi
Signedness	Signed
WL	16
FL	13

Viewing Variables in the Variables Tab

To see the data type information for all the variables in your file, click the **Variables** tab of the code generation report. You can expand all **fi** and **fimath** objects listed in the **Variables** tab to display the **fimath** properties. When you expand a **fi** object in the **Variables** tab, the report indicates whether the **fi** object has a local **fimath** object or is using default **fimath** values.

The following figure shows the information displayed for a **fi** object that is using default **fimath** values.

Summary	All Messages (0)		Variables						
Order	Variable	Type	Size	Complex	Class	Signedness	WL	FL	
	1	z	Output	1 x 1	No	embedded.fi	Signed	16	13
	Global fimath:								
	Round mode: nearest				Sum mode: FullPrecision				
	Overflow mode: saturate				Maximum sum word length: 128				
	Product mode: FullPrecision				Cast before sum: Yes				
Maximum product word length: 128									
	2	x	Output	1 x 1	No	embedded.fi	Signed	16	17
	3	y	Output	1 x 1	No	embedded.fi	Signed	16	17

You can sort the variables by clicking the column headings in the **Variables** tab. To sort the variables by multiple columns, press the **Shift** key while clicking the column headings.

See Also

For more information about using the code generation report with the `fiaccel` function, see the `fiaccel` reference page.

For information about local and default `fimath`, see “`fimath` Object Construction” on page 4-2.

For information about using the code generation report with the `codegen` function, see “Code Generation Reports” in the MATLAB Coder documentation.

Generate C Code from Code Containing Global Data

In this section...

“Workflow Overview” on page 8-57

“Declaring Global Variables” on page 8-57

“Defining Global Data” on page 8-58

“Synchronizing Global Data with MATLAB” on page 8-59

“Limitations of Using Global Data” on page 8-62

Workflow Overview

To generate MEX functions from MATLAB code that uses global data:

- 1 Declare the variables as global in your code.
- 2 Define and initialize the global data before using it.

For more information, see “Defining Global Data” on page 8-58.

- 3 Compile your code using `fiaccel`.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated code. If there is no interaction between MATLAB and the generated code, it is safe to disable synchronization. Otherwise, you should enable synchronization. For more information, see “Synchronizing Global Data with MATLAB” on page 8-59.

Declaring Global Variables

For code generation, you must declare global variables before using them in your MATLAB code. Consider the `use_globals` function that uses two global variables `AR` and `B`.

```
function y = use_globals()
%#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) = B(1);
y = AR * 2;
```

Defining Global Data

You can define global data either in the MATLAB global workspace or at the command line. If you do not initialize global data at the command line, `fiaccel` looks for the variable in the MATLAB global workspace. If the variable does not exist, `fiaccel` generates an error.

Defining Global Data in the MATLAB Global Workspace

To compile the `use_globals` function described in “Declaring Global Variables” on page 8-57 using `fiaccel`:

- 1 Define the global data in the MATLAB workspace. At the MATLAB prompt, enter:

```
global AR B;
AR = fi(ones(4),1,16,14);
B = fi([1 2 3],1,16,13);
```

- 2 Compile the function to generate a MEX file named `use_globalsx`.

```
fiaccel -o use_globalsx use_globals
```

Defining Global Data at the Command Line

To define global data at the command line, use the `fiaccel -global` option. For example, to compile the `use_globals` function described in “Declaring Global Variables” on page 8-57, specify two global inputs `AR` and `B` at the command line.

```
fiaccel -o use_globalsx ...
    -global {'AR',fi(ones(4)),'B',fi([1 2 3])} use_globals
```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`.

Defining Variable-Sized Global Data. To provide initial values for variable-sized 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:

```
fiaccel foo -globals {'g1',{coder.typeof(0,[2 2],1),[1 1]}}
```

For a detailed explanation of `coder.typeof` syntax, see `coder.typeof`.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The generated code and MATLAB each have their own copies of global data. To ensure consistency, 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.

When to Synchronize Global Data

By default, synchronization between global data in MATLAB and generated code occurs at MEX function entry and exit and for all *extrinsic* calls, which are calls to MATLAB functions on the MATLAB path that `fiaccel` dispatches to MATLAB for execution. This behavior ensures maximum consistency between generated code 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 8-60.

Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Ensure maximum consistency when all extrinsic calls modify global data.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Default behavior.
Ensure maximum consistency when most extrinsic calls modify global data, but a few do not.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Use the <code>coder.extrinsic-sync:off</code> option to turn off synchronization for the extrinsic calls that do not affect global data.
Ensure maximum consistency when most extrinsic calls do not modify global data, but a few do.	At MEX-function entry and exit	Yes. Use the <code>coder.extrinsic-sync:on</code> option to synchronize only the calls that modify global data
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	At MEX-function entry and exit	No.
Communicate between generated code files only. No interaction between global data in MATLAB and generated code.	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 8-59.

You control the synchronization of global data with extrinsic functions using the `coder.extrinsic -sync:on` and `-sync:off` options.

Controlling the Global Data Synchronization Mode from the Command Line.

- 1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.mexconfig
```

- 2 From the command line, set the `GlobalDataSyncMethod` property to `Always`, `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

```
comp_cfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
```

- 3 Use the `comp_cfg` configuration object when compiling your code by specifying it using the `-config` compilation option. For example,

```
fiaccel -config comp_cfg myFile
```

Controlling Synchronization for Extrinsic Function Calls. You can control whether synchronization between global data in MATLAB and generated code occurs before and after you call an extrinsic function. To do so, 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 affect global data, turn off synchronization for these calls using the `-sync:off` option. Turning off synchronization improves performance. For example, if functions `foo1` and `foo2` *do not* affect 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 affect global data, turn on synchronization for these calls using the

-sync:on option. For example, if functions `foo1` and `foo2` *do* affect 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 control the synchronization for specific extrinsic calls. The `-sync:on` option has no effect.

Limitations of Using Global Data

You cannot use global data with

- The `coder.cstructname` function. This function does not support global variables.
- The `coder.varsize` function. Instead, use a `coder.typeof` object to define variable-sized global data as described in “Defining Variable-Sized Global Data” on page 8-59.

Define Input Properties Programmatically in MATLAB File

In this section...

“How to Use `assert`” on page 8-63

“Rules for Using `assert` Function” on page 8-67

“Example: Specifying Properties of Primary Fixed-Point Inputs” on page 8-68

“Example: Specifying Class and Size of Scalar Structure” on page 8-69

“Example: Specifying Class and Size of Structure Array” on page 8-70

How to Use `assert`

You can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

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, 'embedded.fi'));  
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify `numerictype` of Fixed-Point Input” on page 8-66. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-67.

If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify `fi` Class

```
assert ( isfi ( param ) )  
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...  
assert(isfi(U));  
...
```

or

```
...  
assert(isa(U, 'embedded.fi'));  
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify `numerictype` of Fixed-Point Input” on page 8-66. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-67.

Specify Structure Class

```
assert ( isstruct ( param ) )
```


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'));
...
```

Note If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify Any Size

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. For example, to set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...  
assert(all(size(U)== [1]));  
...
```

Specify Real Input

```
assert ( isreal ( param ) )
```

Specifies that the input parameter *param* is real. For example, 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. For example, 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:

```
...  
% Define the numerictype object.
```

```
T = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isequal(numerictype(U),T));
...
```

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *fiparam* to the `fimath` object *F*. For example, to specify the `fimath` property of fixed-point input *U* so that it saturates on integer overflow, use the following code:

```
...
% Define the fimath object.
F = fimath('OverflowAction','Saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

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:

```
...
assert(isa(U,'double') && ~isreal(U) && all(size(U) == [3 3]) && isa(V,'uint16'));
...
```

Rules for Using assert Function

Follow these rules when using the `assert` function to specify the properties of primary function inputs:

- Call `assert` functions at the beginning of the primary function, before any flow-control operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- If you set the class of an input parameter to `fi`:
 - You must also set its `numerictype`, see “Specify `numerictype` of Fixed-Point Input” on page 8-66.
 - You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-67.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of each field in the structure in the order in which you define the fields in the structure definition.

Example: Specifying Properties of Primary Fixed-Point Inputs

In the following example, the primary MATLAB function `emcsqrtfi` takes one fixed-point input: `x`. The code specifies the following properties for this input:

Property	Value
<code>class</code>	<code>fi</code>
<code>numerictype</code>	numerictype object <code>T</code> , as specified in the primary function
<code>fimath</code>	fimath object <code>F</code> , as specified in the primary function
<code>size</code>	scalar (by default)
<code>complexity</code>	real (by default)

```
function y = emcsqrtfi(x)
T = numerictype('WordLength',32,'FractionLength',23,...
    'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
    'SumWordLength',32,'SumFractionLength',23,...
    'ProductMode','SpecifyPrecision',...
    'ProductWordLength',32,'ProductFractionLength',23,...
    'ProductSigned',true);
```

```

        'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numericity(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);

```

Example: Specifying Class and Size of Scalar Structure

Assume you have defined **S** as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',fi(4,true,8,0));
```

This code specifies the class and size of **S** and its fields when passed as an input to your MATLAB function:

```

function y = fcn(S)

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% in the order in which you defined them.
T = numericity('Wordlength', 8,'FractionLength', ...
    0,'signed',true);
assert(isa(S.r,'double'));
assert(isfi(S.i) && isequal(numericity(S.i),T));

y = S;

```

Note The only way to name a field in a structure is to set at least one of its properties. Therefore in the preceding example, an `assert` function specifies that field `S.r` is of type `double`, even though `double` is the default.

Example: Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',...
    {fi(4,1,8,0), fi(5,1,8,0)});
```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

```
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));
T = numerictype('Wordlength', 8, 'FractionLength', ...
    0, 'signed', true);

% 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(isfi(S(1).i) && isequal(numerictype(S(1).i), T));

y = S;
```

Note The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S(1).r` is of type `double`, even though `double` is the default.

Control Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 8-71
“When to Disable Run-Time Checks” on page 8-72
“How to Disable Run-Time Checks” on page 8-72

Types of Run-Time Checks

In simulation, the code generated for your MATLAB functions includes the following run-time checks and external function calls.

- Memory integrity checks

These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

Caution For safety, these checks are enabled by default. Without memory integrity checks, violations will 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 For safety, these checks are enabled by default. Without these checks the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information

about extrinsic functions, see “Declaring MATLAB Functions as Extrinsic Functions” on page 10-12.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower simulation than generating code with the checks disabled. Similarly, extrinsic calls are time consuming and have an adverse effect on performance. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation, with these caveats:

Consider disabling...	Only if...
Memory integrity checks	You are sure that your code is safe and that all array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C .
Extrinsic calls	You are only using extrinsic calls to functions that do not affect application results.

How to Disable Run-Time Checks

To disable run-time checks:

- 1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.MEXConfig
```

- 2 From the command line set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties false, as applicable:

```
comp_cfg.IntegrityChecks = false;  
comp_cfg.ExtrinsicCalls = false;
```



```
comp_cfg.ResponsivenessChecks = false;
```

Generation with MATLAB Coder

MATLAB Coder codegen automatically converts MATLAB code directly to C code. It generates standalone C code that is bit-true to fixed-point MATLAB code. Using Fixed-Point Toolbox and MATLAB Coder software you can generate C code with algorithms containing integer math only (i.e., without any floating-point math).

Code Generation with MATLAB Function Block

In this section...

“Composing MATLAB Language Function in Simulink Model” on page 8-75

“MATLAB Function Block with Data Type Override” on page 8-75

“Fixed-Point Data Types with MATLAB Function Block” on page 8-76

Composing MATLAB Language Function in Simulink Model

The MATLAB Function block lets you compose a MATLAB language function in a Simulink model that generates embeddable code. When you simulate the model or generate code for a target environment, a function in a MATLAB Function block generates efficient C/C++ code. This code meets the strict memory and data type requirements of embedded target environments. In this way, the MATLAB Function blocks bring the power of MATLAB for the embedded environment into Simulink.

For more information about the MATLAB Function block and code generation, refer to the following:

- MATLAB Function block reference page in the Simulink documentation
- in the Simulink documentation
- “Code Generation Workflow” in the MATLAB Coder documentation

MATLAB Function Block with Data Type Override

When you use the MATLAB Function block in a Simulink model that specifies data type override, the block determines the data type override equivalents of the input signal and parameter types. The block then uses these equivalent values to run the simulation. The following table shows how the MATLAB Function block determines the data type override equivalent using

- The data type of the input signal or parameter
- The data type override setting in the Simulink model

Note The MATLAB Function block does not support the Scaled double data type override setting.

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Equivalent
Inherited single	Double	fi double
	Single	fi single
Specified single	Double	Built-in double
	Single	Built-in single
Inherited double	Double	fi double
	Single	fi single
Specified double	Double	Built-in double
	Single	Built-in single
Inherited Fixed	Double	fi double
	Single	fi single
Specified Fixed	Double	fi double
	Single	fi single

For more information about using the MATLAB Function block with data type override, see the following section of the Simulink documentation:

“Using Data Type Override with the MATLAB Function Block”

Fixed-Point Data Types with MATLAB Function Block

Code generation from MATLAB supports a significant number of Fixed-Point Toolbox functions. Refer to “Functions Supported for Code Acceleration or Generation” on page 8-5 for information about which Fixed-Point Toolbox functions are supported.

For more information on working with fixed-point MATLAB Function blocks, see:

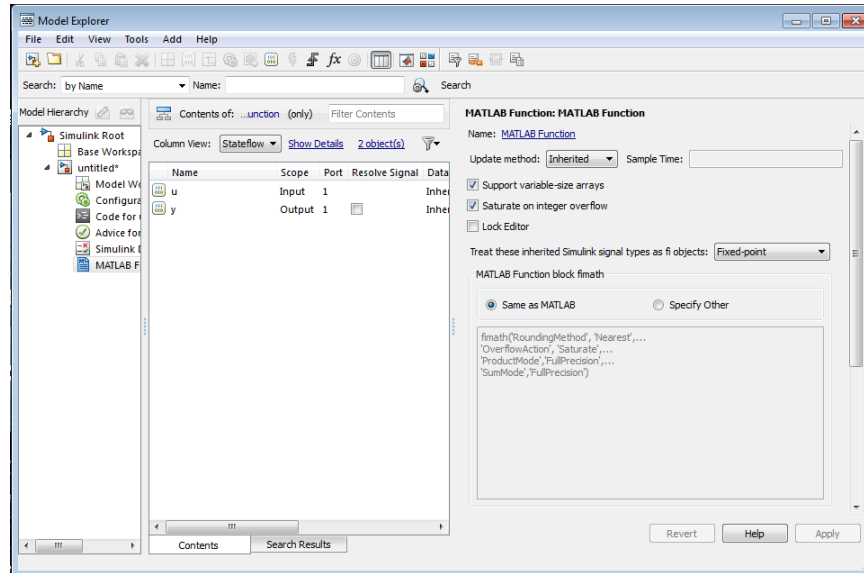
- “Specifying Fixed-Point Parameters in the Model Explorer” on page 8-77
- “Using fimath Objects in MATLAB Function Blocks” on page 8-79
- “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-81

Note To simulate models using fixed-point data types in Simulink, you must have a Simulink Fixed Point license.

Specifying Fixed-Point Parameters in the Model Explorer

You can specify parameters for an MATLAB Function block in a fixed-point model using the Model Explorer. Try the following exercise:

- 1** Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2** Open the Model Explorer by selecting **View > Model Explorer** from your model.
- 3** Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer. Then, select the **MATLAB Function** node. The Model Explorer now appears as shown in the following figure.



The following parameters in the **Dialog** pane apply to MATLAB Function blocks in models that use fixed-point and integer data types:

Treat these inherited Simulink signal types as fi objects

Choose whether to treat inherited fixed-point and integer signals as **fi** objects.

- When you select **Fixed-point**, the MATLAB Function block treats all fixed-point inputs as Fixed-Point Toolbox **fi** objects.
- When you select **Fixed-Point & Integer**, the MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Toolbox **fi** objects.

MATLAB Function block fimath

Specify the **fimath** properties for the block to associate with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as **fi** objects.
- All **fi** and **fimath** objects constructed in the MATLAB Function block.

You can select one of the following options for the **MATLAB Function block `fimath`**:

- **Same as MATLAB** — When you select this option, the block uses the same `fimath` properties as the current default `fimath`. The edit box appears dimmed and displays the current default `fimath` in read-only form.
- **Specify other** — When you select this option, you can specify your own `fimath` object in the edit box.

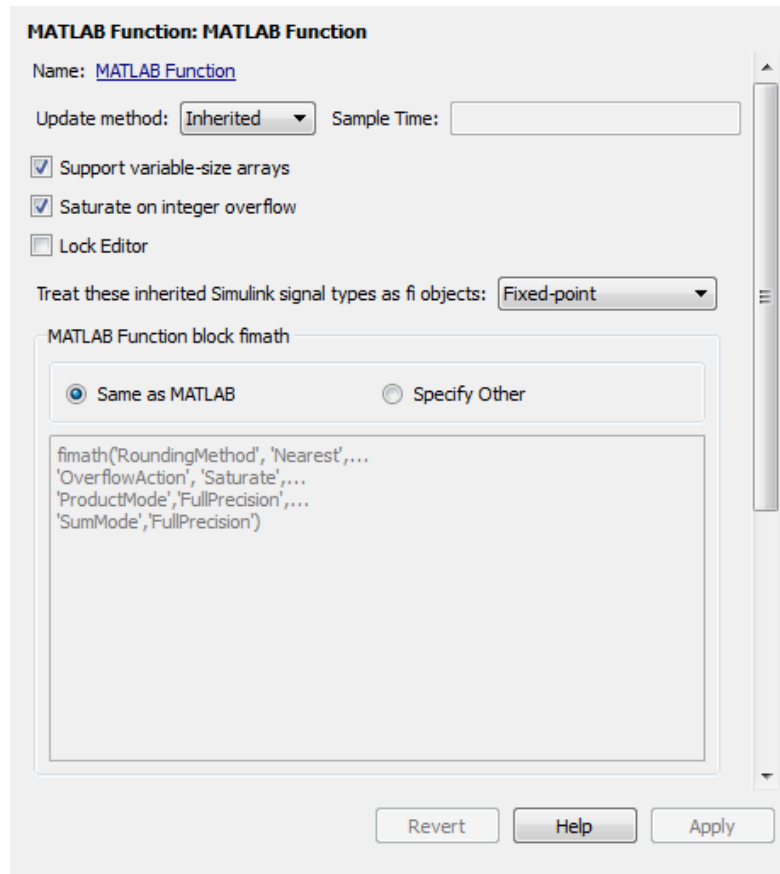
For more information on these parameters, see “Using `fimath` Objects in MATLAB Function Blocks” on page 8-79.

Using `fimath` Objects in MATLAB Function Blocks

The **MATLAB Function block `fimath`** parameter enables you to specify one set of `fimath` object properties for the MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the MATLAB Function block.

You can set these parameters on the following dialog box, which you can access through either the Model Explorer or the “Ports and Data Manager”.



- To access this pane through the Model Explorer:
 - Select **View > Model Explorer** from your model menu.
 - Then, select the MATLAB Function block from the Model Hierarchy pane on the left side of the Model Explorer.
- To access this pane through the Ports and Data Manager, select **Tools > Edit Data/Ports** from the MATLAB Editor menu.

When you select **Same as MATLAB** for the **MATLAB Function block fimath**, the MATLAB Function block uses the current default fimath. The current default fimath appears dimmed and in read-only form in the edit box.

When you select **Specify other** the block allows you to specify your own `fimath` object in the edit box. You can do so in one of two ways:

- Constructing the `fimath` object inside the edit box.
- Constructing the `fimath` object in the MATLAB or model workspace and then entering its variable name in the edit box.

Note If you use this option and plan to share your model with others, make sure you define the variable in the model workspace. See “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-81 for more information on sharing models.

The Fixed-Point Toolbox `isfimathlocal` function supports code generation for MATLAB.

Sharing Models with Fixed-Point MATLAB Function Blocks

When you collaborate with a coworker, you can share a fixed-point model using the MATLAB Function block. To share a model, make sure that you move any variables you define in the MATLAB workspace, including `fimath` objects, to the model workspace. For example, try the following:

- 1 Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2 Define a `fimath` object in the MATLAB workspace that you want to use for any Simulink fixed-point signal entering the MATLAB Function block as an input:

```
F = fimath('RoundingMethod','Floor','OverflowAction','Wrap',...
          'ProductMode','KeepLSB','ProductWordLength',32,...
          'SumMode','KeepLSB','SumWordLength',32)
```

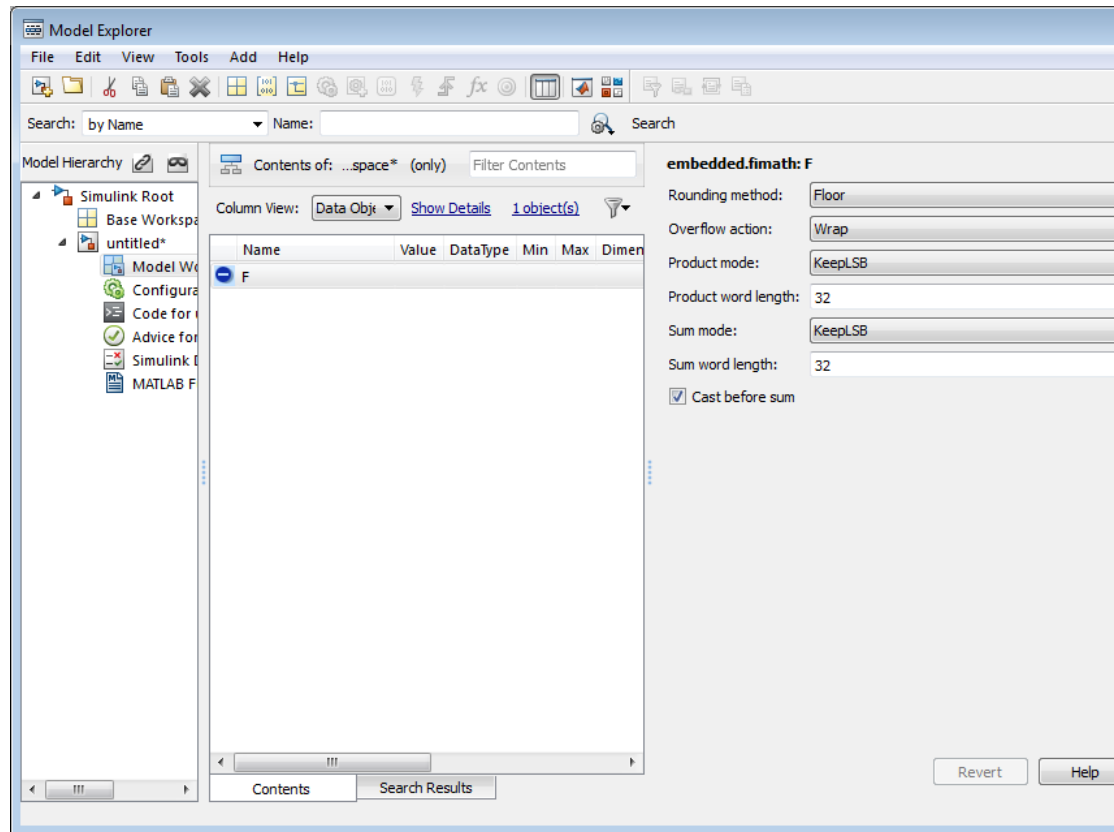
```
F =
    RoundingMethod: Floor
    OverflowAction: Wrap
    ProductMode: KeepLSB
ProductWordLength: 32
    SumMode: KeepLSB
```

```
SumWordLength: 32  
CastBeforeSum: true
```

- 3 Open the Model Explorer by selecting **View > Model Explorer** from your model.
- 4 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer, and select the **MATLAB Function** node.
- 5 Select **Specify other** for the **MATLAB Function block fimath** parameter and enter the variable **F** into the edit box on the **Dialog** pane. Click **Apply** to save your changes.

You have now defined the **fimath** properties to be associated with all Simulink fixed-point input signals and all **fi** and **fimath** objects constructed within the block.

- 6 Select the **Base Workspace** node in the **Model Hierarchy** pane. You can see the variable **F** that you have defined in the MATLAB workspace listed in the **Contents** pane. If you send this model to a coworker, that coworker must first define that same variable in the MATLAB workspace to get the same results.
- 7 Cut the variable **F** from the base workspace, and paste it into the model workspace listed under the node for your model, in this case, **untitled***. The Model Explorer now appears as shown in the following figure.



You can now email your model to a coworker. Because you included the required variables in the workspace of the model itself, your coworker can simply run the model and get the correct results. Receiving and running the model does not require any extra steps.

Generate Fixed-Point FIR Code Using MATLAB Function Block

In this section...

- “Program the MATLAB Function Block” on page 8-84
- “Prepare the Inputs” on page 8-85
- “Create the Model” on page 8-85
- “Define the fimath Object Using the Model Explorer” on page 8-87
- “Run the Simulation” on page 8-88

Program the MATLAB Function Block

The following example shows how to create a fixed-point, lowpass, direct form FIR filter in Simulink. To create the FIR filter, you use Fixed-Point Toolbox software and the MATLAB Function block. In this example, you perform the following tasks in the sequence shown:

- 1** Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2** Save your model as `cgen_fi.mdl`.
- 3** Double-click the MATLAB Function block in your model to open the MATLAB Function Block Editor. Type or copy and paste the following MATLAB code, including comments, into the Editor:

```
function [yout,zf] = dffirdemo(b, x, zi)
%codegen_fi doc model example
%Initialize the output signal yout and the final conditions zf
Ty = numerictype(1,12,8);
yout = fi(zeros(size(x)), 'numerictype', Ty);
zf = zi;

% FIR filter code
for k=1:length(x);
    % Update the states: z = [x(k);z(1:end-1)]
    zf(:) = [x(k);zf(1:end-1)];
```

```

    % Form the output:  $y(k) = b*z$ 
    yout(k) = b*zf;
end

% Plot the outputs only in simulation.
% This does not generate C code.
coder.extrinsic('figure');
coder.extrinsic('subplot');
coder.extrinsic('plot');
coder.extrinsic('title');
coder.extrinsic('grid');
figure;
subplot(211);plot(x); title('Noisy Signal');grid;
subplot(212);plot(yout); title('Filtered Signal');grid;

```

Prepare the Inputs

Define the filter coefficients b , noise x , and initial conditions z_i by typing the following code at the MATLAB command line:

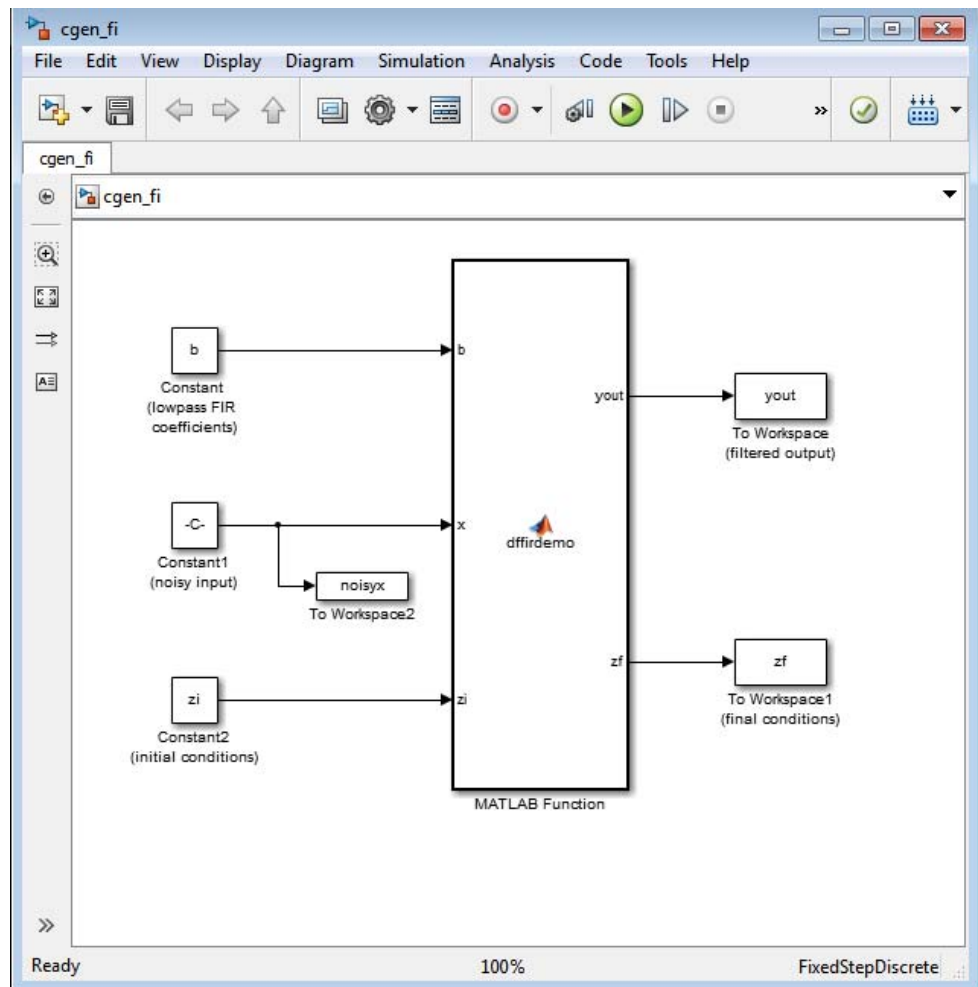
```

b=fidemo.fi_fir_coefficients;
load mtlb
x = mtlb;
n = length(x);
noise = sin(2*pi*2140*(0:n-1)'./Fs);
x = x + noise;
zi = zeros(length(b),1);

```

Create the Model

- 1 Add blocks to your model to create the following system.



- 2 Set the block parameters in the model to these “Fixed-Point FFT Code Example Parameter Values” on page 8-89.
- 3 From the model menu, select **Simulation > Model Configuration Parameters** and set the following parameters.

Parameter	Value
Stop time	0
Type	Fixed-step
Solver	discrete (no continuous states)

Click **Apply** to save your changes.

Define the fimath Object Using the Model Explorer

- 1 Open the Model Explorer for the model.
- 2 Click the **cgen_fi > MATLAB Function** node in the **Model Hierarchy** pane. The dialog box for the MATLAB Function block appears in the **Dialog** pane of the Model Explorer.
- 3 Select **Specify other** for the **MATLAB Function block fimath** parameter on the MATLAB Function block dialog box. You can then create the following fimath object in the edit box:

```
fimath('RoundingMethod','Floor','OverflowAction','Wrap',...
      'ProductMode','KeepLSB','ProductWordLength',32,...
      'SumMode','KeepLSB','SumWordLength',32)
```

The fimath object you define here is associated with fixed-point inputs to the MATLAB Function block as well as the fi object you construct within the block.

By selecting **Specify other** for the **MATLAB Function block fimath**, you ensure that your model always uses the fimath properties you specified.

Run the Simulation

- 1** Run the simulation by selecting your model and typing **Ctrl+T**. While the simulation is running, information outputs to the MATLAB command line. You can look at the plots of the noisy signal and the filtered signal.
- 2** Next, build embeddable C code for your model by selecting the model and typing **Ctrl+B**. While the code is building, information outputs to the MATLAB command line. A folder called `coder_fi_grt_rtw` is created in your current working folder.
- 3** Navigate to `coder_fi_grt_rtw > coder_fi.c`. In this file, you can see the code generated from your model. Search for the following comment in your code:

```
/* coder_fi doc model example */
```

This search brings you to the beginning of the section of the code that your MATLAB Function block generated.

Fixed-Point FFT Code Example Parameter Values

Block	Parameter	Value
Constant	Constant value	b
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^{-12}
	Bias	0
Constant1	Constant value	x+noise
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	1
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^{-8}
	Bias	0

Block	Parameter	Value
Constant2	Constant value	zi
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^-8
	Bias	0
To Workspace	Variable name	yout
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

Block	Parameter	Value
To Workspace1	Variable name	zf
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected
To Workspace2	Variable name	noisyx
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

Accelerate Code for Variable-Size Data

In this section...

“Disable Support for Variable-Size Data” on page 8-92

“Control Dynamic Memory Allocation” on page 8-93

“Accelerate Code for MATLAB Functions with Variable-Size Data” on page 8-94

“Accelerate Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-96

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. 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 exceeds a configurable threshold.

Disable Support for Variable-Size Data

By default, for MEX and C/C++ code acceleration, support for variable-size data is enabled. You modify variable sizing settings at the command line.

- 1 Create a configuration object for code generation.

```
cfg = coder.mexconfig;
```

- 2 Set the EnableVariableSizing option:

```
cfg.EnableVariableSizing = false;
```

- 3 Using the -config option, pass the configuration object to fiaccel :

```
fiaccel -config cfg foo
```

Control Dynamic Memory Allocation

By default, dynamic memory allocation is enabled for variable-size arrays whose size exceeds a configurable threshold. If you disable support for variable-size data, you also disable dynamic memory allocation. You can modify dynamic memory allocation settings at the command line.

- 1 Create a configuration object for code acceleration. For example, for a MEX function:

```
mexcfg = coder.mexconfig;
```

- 2 Set the DynamicMemoryAllocation option:

Setting	Action
mexcfg.DynamicMemoryAllocation='Off';	Dynamic memory allocation is disabled. All variable-size data is allocated statically on the stack.
mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';	Dynamic memory allocation is enabled for all variable-size arrays. All variable-size data is allocated dynamically on the heap.
mexcfg.DynamicMemoryAllocation='Threshold';	Dynamic memory allocation is enabled for all variable-size arrays whose size (in bytes) is greater than or equal to the value specified using the Dynamic memory allocation threshold parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

3 Optionally, if you set Dynamic memory allocation to 'Threshold', configure Dynamic memory allocation threshold to fine tune memory allocation.

4 Using the `-config` option, pass the configuration object to `fiaccl`:

```
fiaccl -config mexcfg foo
```

Accelerate Code for MATLAB Functions with Variable-Size Data

Here is a basic workflow that generates MEX code.

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 `fiaccl`. Use the following command-line options:

- `-args {coder.typeof...}` if you have variable-size inputs
- `-report` to generate a code generation report

For example:

```
fiaccl -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, `fiaccel` detects variables and structure fields that change size after you define them, and reports these occurrences as errors. In addition, `fiaccel` performs a runtime 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 21-23.

5 Fix upper bounds errors

Cause:	How To Fix:	For More Information:
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See “Specifying Upper Bounds for Variable-Size Data” on page 21-6 and “Diagnosing and Fixing Size Mismatch Errors” on page 21-23.
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 8-93

6 Generate C/C++ code using the `fiaccel` function.

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

- “About the MATLAB Function `uniquetol`” on page 8-96
- “Step 1: Add Compilation Directive for Code Generation” on page 8-96
- “Step 2: Address Issues Detected by the Code Analyzer” on page 8-97
- “Step 3: Generate MEX Code” on page 8-97
- “Step 4: Fix the Size Mismatch Error” on page 8-99

About the MATLAB Function `uniquetol`

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

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

Step 1: Add Compilation Directive for Code Generation

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

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



```

        k = i;
    end
end

```

Step 2: Address Issues Detected by the Code Analyzer

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

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

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

Step 3: Generate MEX Code

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

- 1 Generate a MEX function for `uniquetol`:

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

What do these command-line options mean?

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

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

For more information, see “Specify Variable-Size Inputs at the Command Line”.

- The second argument, `coder.typeof(0)`, defines input `tol` as a real double scalar.

The `-report` option instructs `fiaccel` to generate a code generation report, even if no errors or warnings occur.

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

Executing this command generates a compiler error:

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

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

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

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

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

Step 4: Fix the Size Mismatch Error

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

- 1 Add this statement to the `uniquetol` function:

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

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

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

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

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

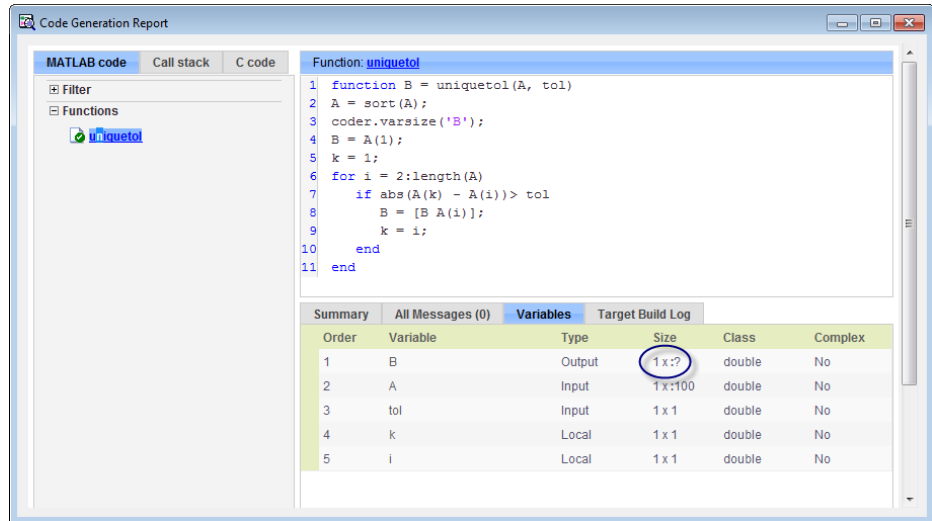
The function `coder.varsize` declares every instance of B in `uniquetol` to be variable sized.

- 2 Generate code again using the same command:

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

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

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



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

Accelerate C Code for Fixed-Point Mean Value (TBD)

TBD

Create Embeddable C Code for Summing Values (TBD)

TBD

Propose Fixed-Point Data Types in a MATLAB Coder Project

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Toolbox
- C compiler

For a list of supported compilers, see
http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler” “Set Up Compiler to Generate Compiled C Code Functions” on page 8-15.

For instructions on installing MathWorks® products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

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


```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	fun_with_matlab.m	Entry-point MATLAB function
Test file	fun_with_matlab_test.m	MATLAB script that tests fun_with_matlab.m

The fun_with_matlab Function

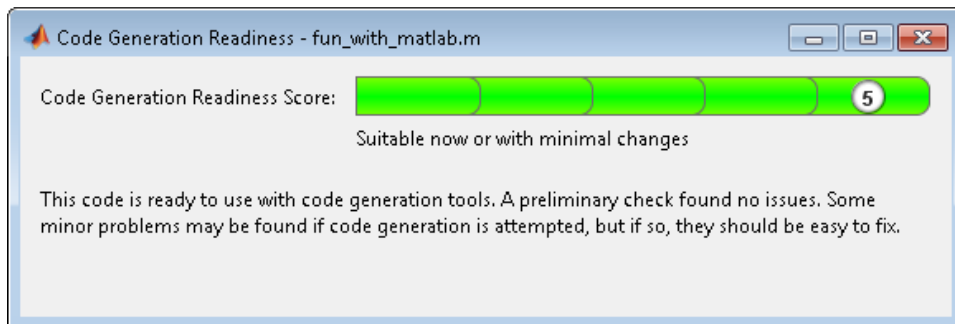
```
function y = fun_with_matlab(x)
    persistent z
    if isempty(z)
        z = zeros(2,1);
    end
    % [b,a] = butter(2, 0.25)
    b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
    a = [
        1, -0.942809041582063, 0.333333333333333];

    y = zeros(size(x));
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i) - a(3) * y(i);
    end
end
```

Check Code Generation Readiness

In the current working folder, right-click the fun_with_matlab.m function. From the context menu, select Check Code Generation Readiness.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the fun_with_matlab.m function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis” “Detect and Debug Code Generation Errors” on page 8-25.

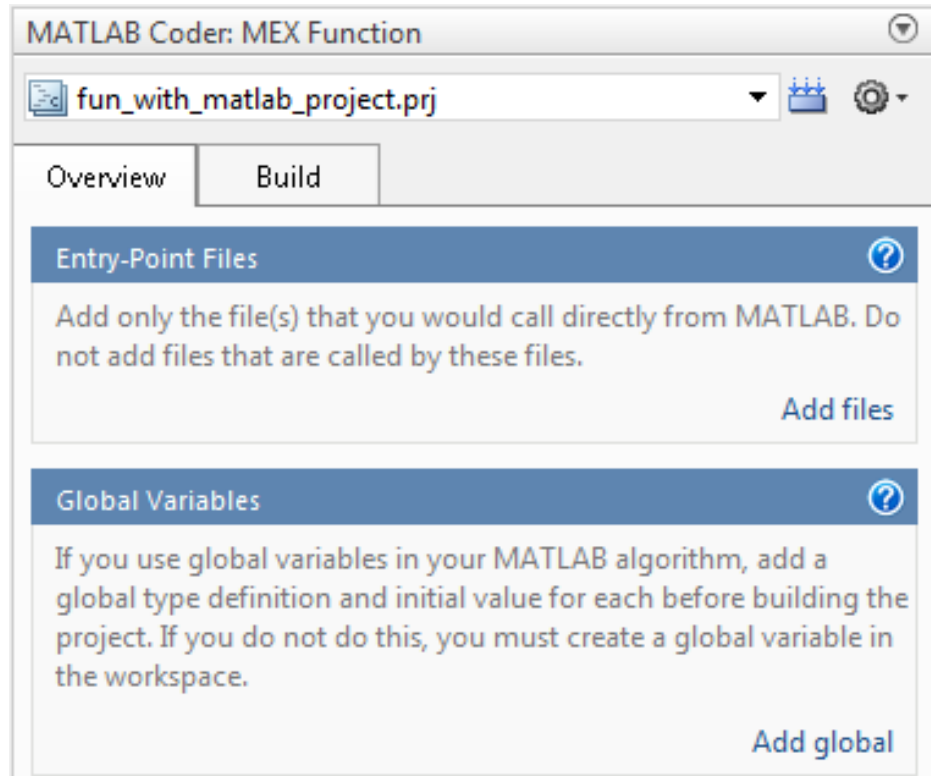
Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this tutorial.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_matlab_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

About the `fun_with_matlab_test` Script

The test script runs the `fun_with_matlab` function with three input signals: `chirp`, `step`, and `impulse`. The script then plots the results.

Contents of `fun_with_matlab_test`

```
% fun_with_matlab_test
%
% Define representative inputs
N = 256;                % Number of points
t = linspace(0,1,N);    % Time vector from 0 to 1 second
```

```

f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1)=1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i=1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

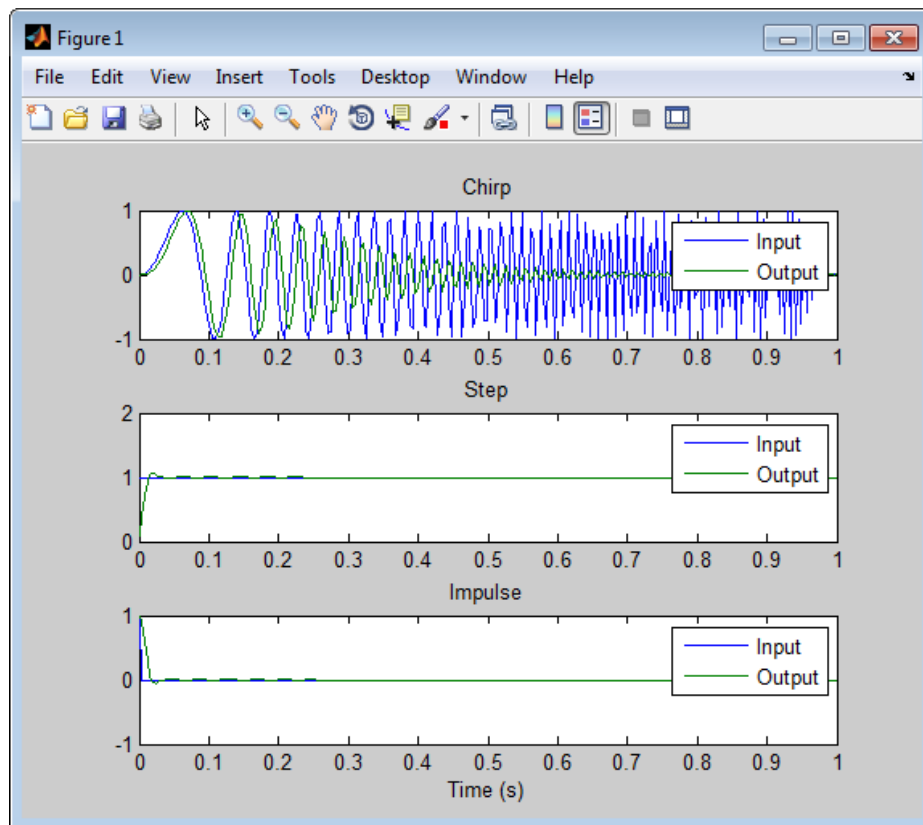
% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i=1:size(x,1)
    subplot(size(x,1),1,i);
    plot(t,x(i,:),t,y(i,:));
    title(titles{i})
    legend('Input','Output');
end
xlabel('Time (s)')
figure(gcf)

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

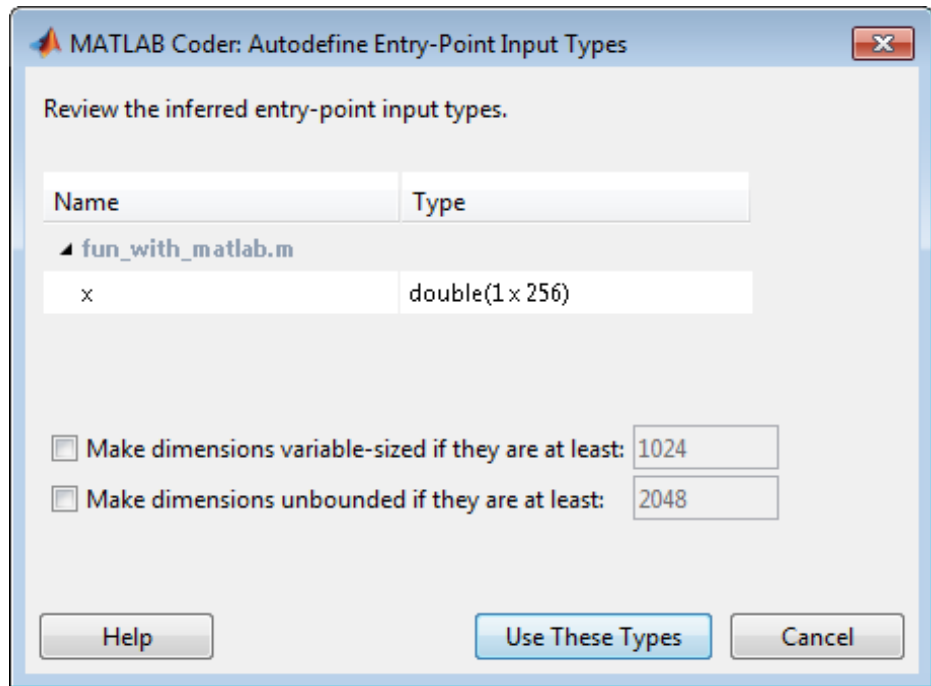
Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Entry-Point Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

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



MATLAB Coder determines the input types from the test file and then displays them.



- 3 In the Autodefine Entry-Point Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

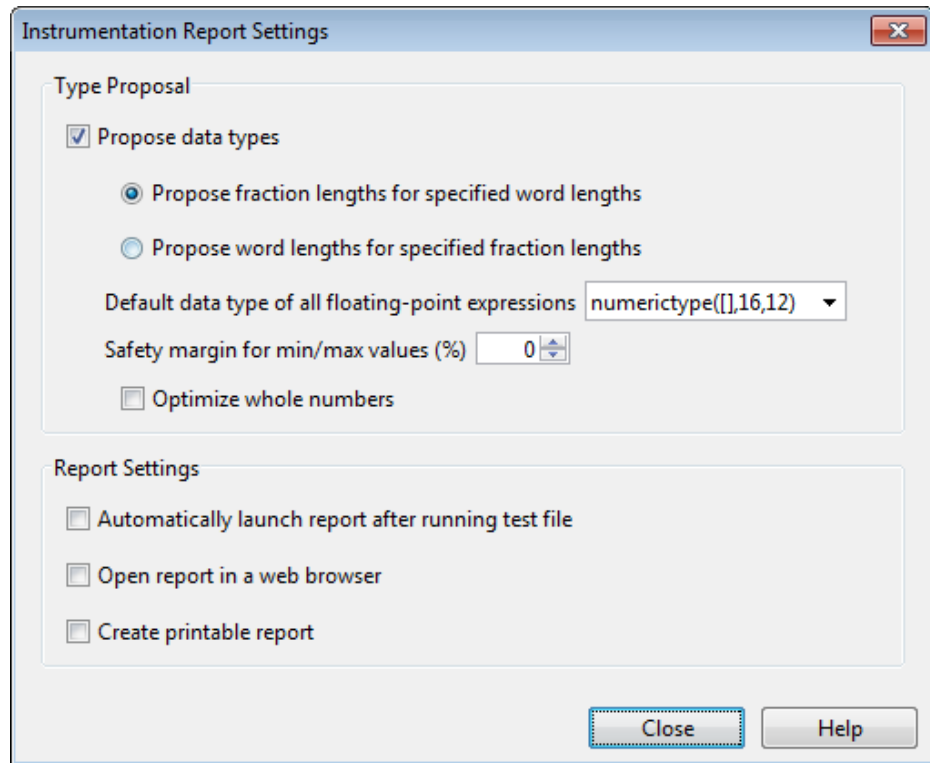
Build Instrumented MEX Function

- 1 In the project, click the **Build** tab.
- 2 On the **Build** tab, set the **Output type** to Instrumented MEX Function.
- 3 Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function `fun_with_matlab_mex` in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of all variables in your MATLAB code.

View Data Type Proposal Settings

- 1 On the **Show Instrumentation Results** pane, click the **Data type proposal and report settings** link.



This example uses the default data type proposal settings which propose fraction lengths for the specified word lengths. Because the MATLAB code is floating-point, the word length is specified by the **Default data type of all floating-point expressions** field. You can specify the `numerictype` signedness, word length and fraction length. Specifying `[]` for signedness instructs MATLAB Coder to choose the appropriate signedness based on simulation values. The default word length is 16. The default fraction length is 12.

For more information, see “Modify Data Type Proposal Settings”.

2 Close the dialog box.

Run Simulation

1 On the **Run Simulation** pane, verify that the test file is set to `fun_with_matlab_test` and that **Redirect entry-point calls to MEX function** is selected. That way, each call to `fun_with_matlab` is replaced with a call to the instrumented MEX function `fun_with_matlab_mex`.

2 On the **Run Simulation** pane, click **Run**.

The `fun_with_matlab_test` file runs and calls `fun_with_matlab_mex`. The outputs of the filters are displayed as before.

View Code Generation Report

1 On the **Show Instrumentation Results** pane, click **View Report**.

2 In the **Code Generation Report**, click the **Variables** tab.

The report displays the simulation minimum and maximum values and the proposed data types.

The screenshot shows the MATLAB Code Generation Report window. The **Variables** tab is selected, displaying a table of variables and their proposed data types. A tooltip for variable `b` is visible, showing its size (1x3), class (double), and proposed signedness (Unsigned).

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax
1	y	Output	1 x 256	double	No	Signed	16	14	No	-0.0696817930434206	1.0553496057969345
2	x	Input	1 x 256	double	No	Signed	16	14	No	-0.9999756307053946	1
3	z	Persistent	2 x 1	double	No	Signed	16	15	No	-0.8907046852192462	0.957718532859117
4	b	Local	1 x 3	double	No	Unsigned	16	18	No	0.0976310729378175	0.195262145875635
5	a	Local	1 x 3	double	No	Signed	16	14	No	-0.942809041582063	1
6	i	Local	1 x 1	double	No	Unsigned	16	0	Yes	1	256

MATLAB Coder proposes data types with word length of 16 and fraction length optimized to avoid overflows.

Next Steps

To learn how to apply the proposed data types to your entry-point MATLAB function and verify that the fixed-point version of your algorithm is functionally equivalent to your original MATLAB algorithm, see “Apply Fixed-Point Data Types” “Apply Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-113.

Apply Fixed-Point Data Types in a MATLAB Coder Project

This example shows you how to write a fixed-point version of your entry-point function using the data types proposed in “Propose Fixed-Point Data Types” “Propose Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-103.

You will learn how to:

- Use the proposed data types to create a fixed-point version of your entry-point function.
- Update your test file to call the fixed-point entry-point function.
- Verify that the fixed-point function is functionally equivalent to the original MATLAB algorithm.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Toolbox
- C compiler

For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Before generating C code, you must set up the C compiler. See “Setting Up the C/C++ Compiler” “Set Up Compiler to Generate Compiled C Code Functions” on page 8-15.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

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

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

- 3 Copy the following files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>
Function code	<code>fun_with_fi.m</code>	Entry-point MATLAB function — fixed-point version of <code>fun_with_matlab</code> that uses data types proposed in “Propose Fixed-Point Data Types”“Propose Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-103
Test file	<code>fun_with_fi_test.m</code>	MATLAB script that runs both <code>fun_with_matlab</code> and <code>fun_with_fi</code> and compares the results

The `fun_with_fi` Function

The `fun_with_fi` is a fixed-point version of the `fun_with_matlab` function that uses the data types proposed in “Propose Fixed-Point Data Types”“Propose Fixed-Point Data Types in a MATLAB® Coder™ Project” on page 8-103.

Variable	Proposed Signedness	Proposed Word Length	Proposed Fraction Length
y	Signed	16	14
x	Signed	16	14
z	Signed	16	15
a	Unsigned	16	18
b	Signed	16	14
i	Unsigned	16	0

For example, in `fun_with_matlab`, variable `y` is defined as `y = zeros(size(x));`. In `fun_with_fi`, to specify that it is a signed fixed-point data type with a word length of 16 and a fraction length of 14, `y = fi(zeros(size(x)),1,16,14,'OverflowAction','Wrap','RoundingMethod','Floor')`

For more information, see `fi`.

Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this tutorial.
- 2 On the **MATLAB Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_matlab_project.prj`.

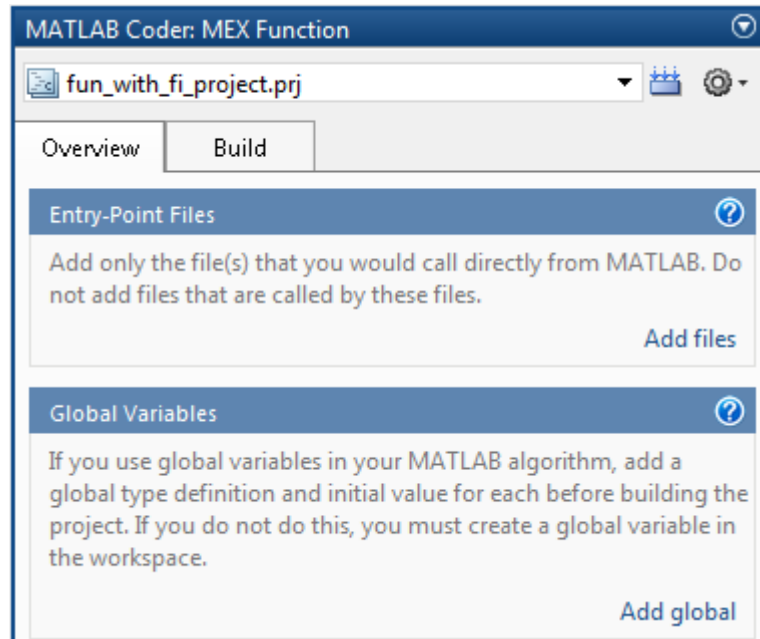
Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_fi_project.prj
```

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_fi_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_fi.m`, and then click **OK** to add the file to the project.

Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Entry-Point Input Types dialog box, add `fun_with_fi_test` as a test file, and then click **Run**.

The test file runs and plots the outputs of the filter. MATLAB Coder determines the input types from the test file and then displays them.

- 3 In the Autodefine Entry-Point Input Types dialog box, click **Use These Types** to accept the autodefined input type.

MATLAB Coder sets the type of `x` to `double(1x256)`.

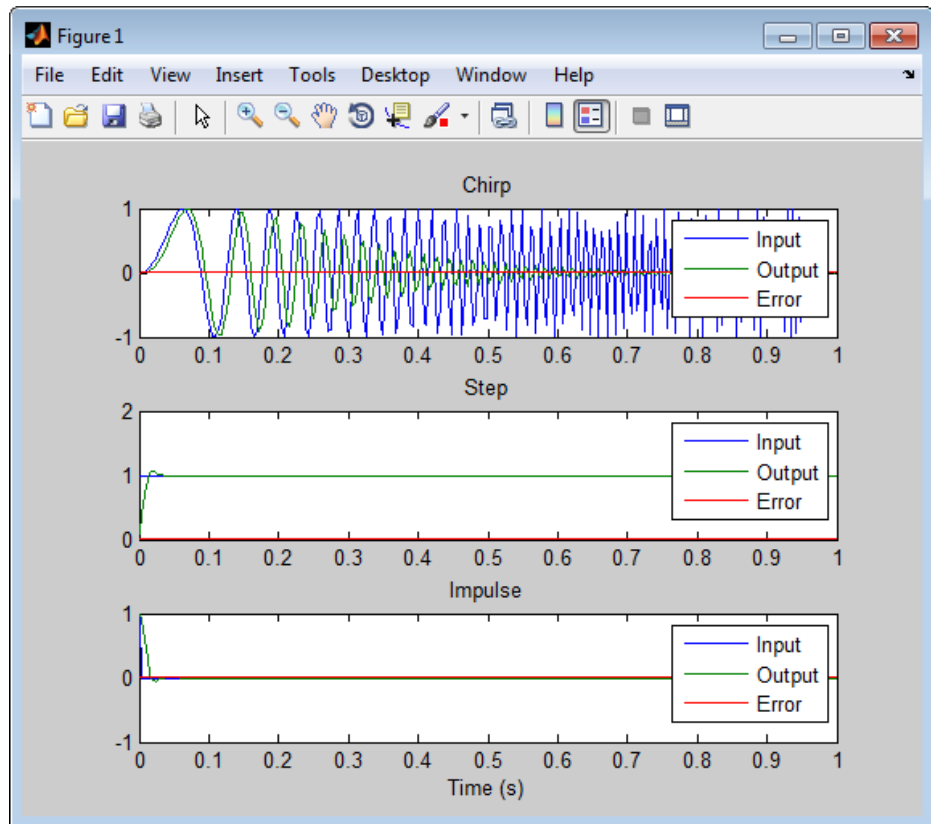
The `fun_with_fi_test` Script

The `fun_with_fi_test` script runs the original floating-point MATLAB algorithm, `fun_with_matlab`, then runs the fixed-point version of the algorithm, `fun_with_fi`. The script then plots the outputs for the floating-point and fixed-point algorithms and the difference in results.

Run Simulation

- 1** In the project, click the **Build** tab.
- 2** On the **Verification** pane, verify that the test file is set to `fun_with_fi_test`. Clear **Rebuild MEX function** and **Redirect entry-point calls to MEX function** so that the test file calls the MATLAB versions of the original and fixed-point algorithms.
- 3** On **Verification** pane, click **Run**.

The `fun_with_fi_test` file runs. The test file runs the original MATLAB algorithm and the fixed-point version, and plots the difference in their outputs.



- 4 Optionally, zoom in on each plot in turn to view the error (difference between the two versions of the algorithm). In this example, the errors are very small, on the order of 10^{-3} . If the error is unacceptably large, refine the fixed-point data types.

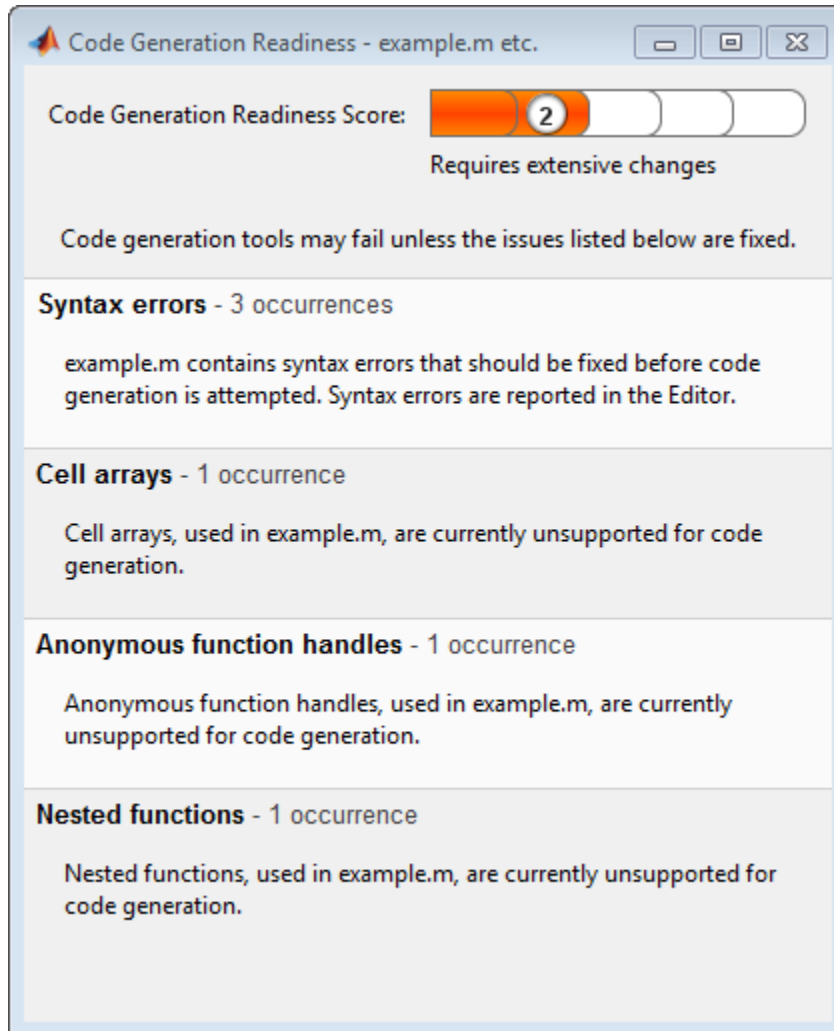
Code Generation Readiness Tool

In this section...
“What Information Does the Code Generation Readiness Tool Provide?” on page 8-119
“Summary Tab” on page 8-120
“Code Structure Tab” on page 8-121
“See Also” on page 8-124

What Information Does the Code Generation Readiness Tool Provide?

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code suitable for code generation. The tool might not detect all code generation issues. Under certain circumstances, it might report false errors. Because the tool might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.

Summary Tab

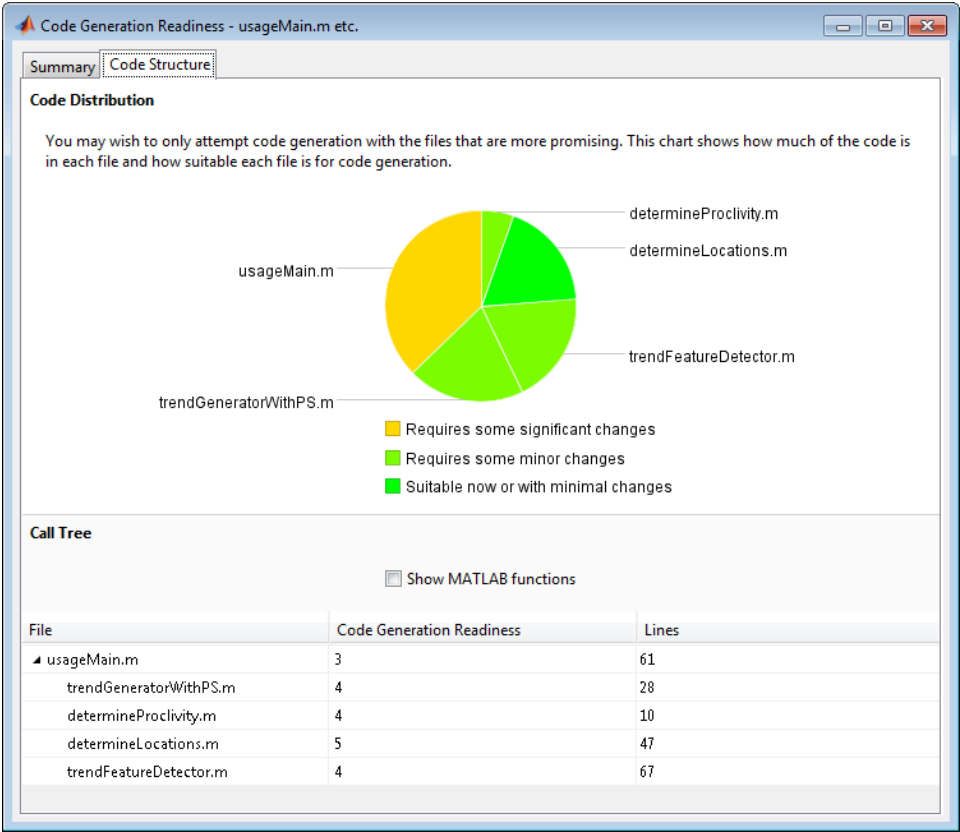


The **Summary** tab provides a **Code Generation Readiness Score** which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected any code generation issues; the code is ready to use with no or minimal changes.

On this tab, the tool also provides information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. Use the code analyzer to learn more about the issues and how to fix them.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as recursion, cell arrays, nested functions, and function handles.
- Unsupported data types.

Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab provides information about the relative size of each file and how suitable each file is for code generation.

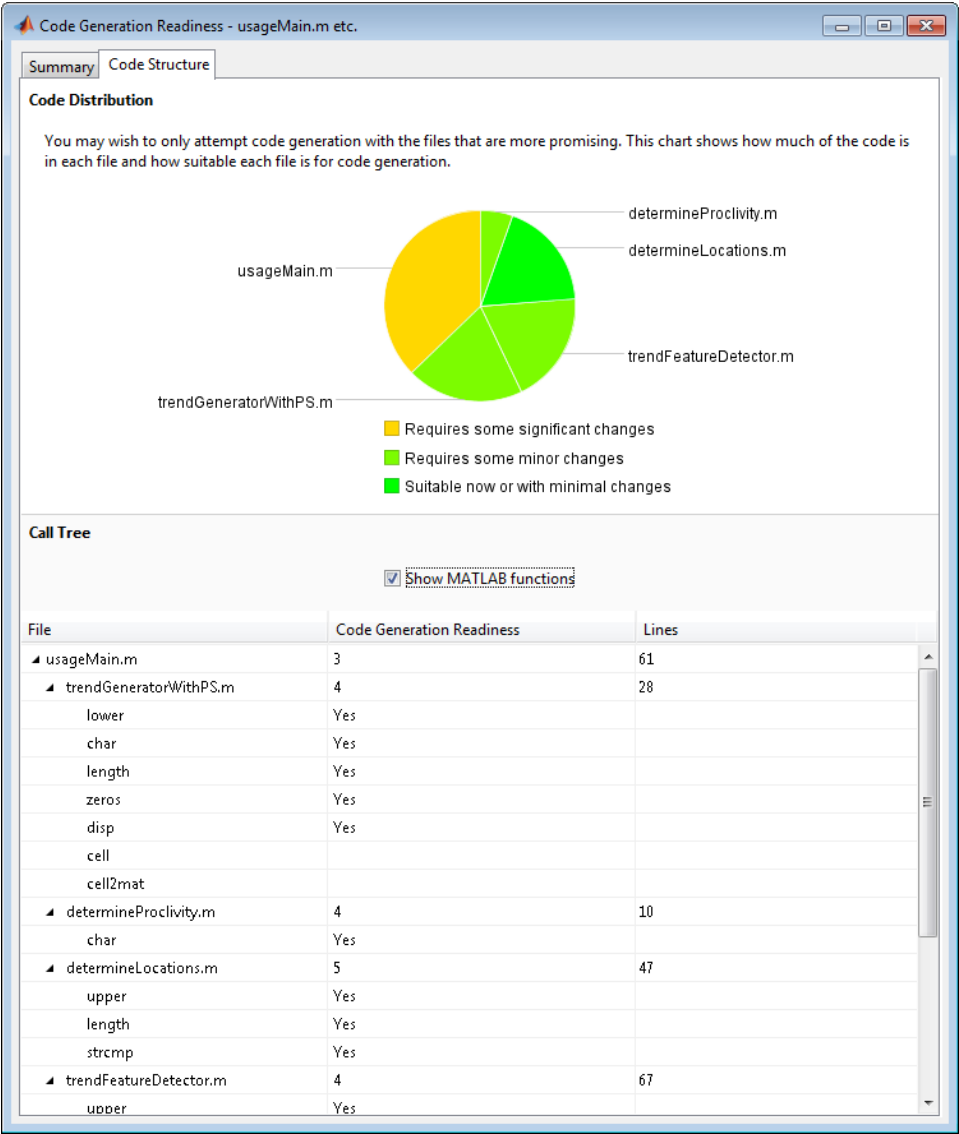
Code Distribution

The **Code Distribution** pane provides a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. This information is useful during the planning phase of a project for estimation and scheduling purposes. If the report indicates that there are multiple files not yet suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane provides information on the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected any code generation issues; the code is ready to use with no or minimal changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions. If you select **Show MATLAB Functions**, the report also lists all the MATLAB functions called by your function code. For each of these MATLAB functions, if the function is supported for code generation, the report sets **Code Generation Readiness** to Yes.



See Also

- “Check Code Using the Code Generation Readiness Tool”
“Check Code Using the Code Generation Readiness Tool” on page 8-125
“Check Code Using the Code Generation Readiness Tool”

Check Code Using the Code Generation Readiness Tool

Run Code Generation Readiness Tool at the Command Line

- 1 Navigate to the folder that contains the file that you want to check for code generation readiness.
- 2 At the MATLAB command prompt, enter:

```
coder.screener('filename')
```

The **Code Generation Readiness** tool opens for the file named `filename`, provides a code generation readiness score, and lists issues that must be fixed prior to code generation.

Run the 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 and provides a code generation readiness score and lists issues that must be fixed prior to code generation.

See Also

- “Code Generation Readiness Tool” on page 8-119

Interoperability with Other Products

- “fi Objects with Simulink” on page 9-2
- “fi Objects with DSP System Toolbox ” on page 9-7
- “Ways to Generate Code” on page 9-12

fi Objects with Simulink

In this section...
“Reading Fixed-Point Data from the Workspace” on page 9-2
“Writing Fixed-Point Data to the Workspace” on page 9-2
“Setting the Value and Data Type of Block Parameters” on page 9-6
“Logging Fixed-Point Signals” on page 9-6
“Accessing Fixed-Point Block Data During Simulation” on page 9-6

Reading Fixed-Point Data from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model via the From Workspace block. To do so, the data must be in a structure format with a `fi` object in the `values` field. In array format, the From Workspace block only accepts real, double-precision data.

To read in `fi` data, the **Interpolate data** parameter of the From Workspace block must not be selected, and the **Form output after final data value by** parameter must be set to anything other than Extrapolation.

Writing Fixed-Point Data to the Workspace

You can write fixed-point output from a model to the MATLAB workspace via the To Workspace block in either array or structure format. Fixed-point data written by a To Workspace block to the workspace in structure format can be read back into a Simulink model in structure format by a From Workspace block.

Note To write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

For example, you can use the following code to create a structure in the MATLAB workspace with a `fi` object in the `values` field. You can then use the From Workspace block to bring the data into a Simulink model.

```
a = fi([sin(0:10)' sin(10:-1:0)'])
```

```
a =
```

```

         0   -0.5440
    0.8415   0.4121
    0.9093   0.9893
    0.1411   0.6570
   -0.7568  -0.2794
   -0.9589  -0.9589
   -0.2794  -0.7568
    0.6570   0.1411
    0.9893   0.9093
    0.4121   0.8415
   -0.5440         0

```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 15

```

```
s.signals.values = a
```

```
s =
```

```
    signals: [1x1 struct]
```

```
s.signals.dimensions = 2
```

```
s =
```

```
    signals: [1x1 struct]
```

```
s.time = [0:10]'
```

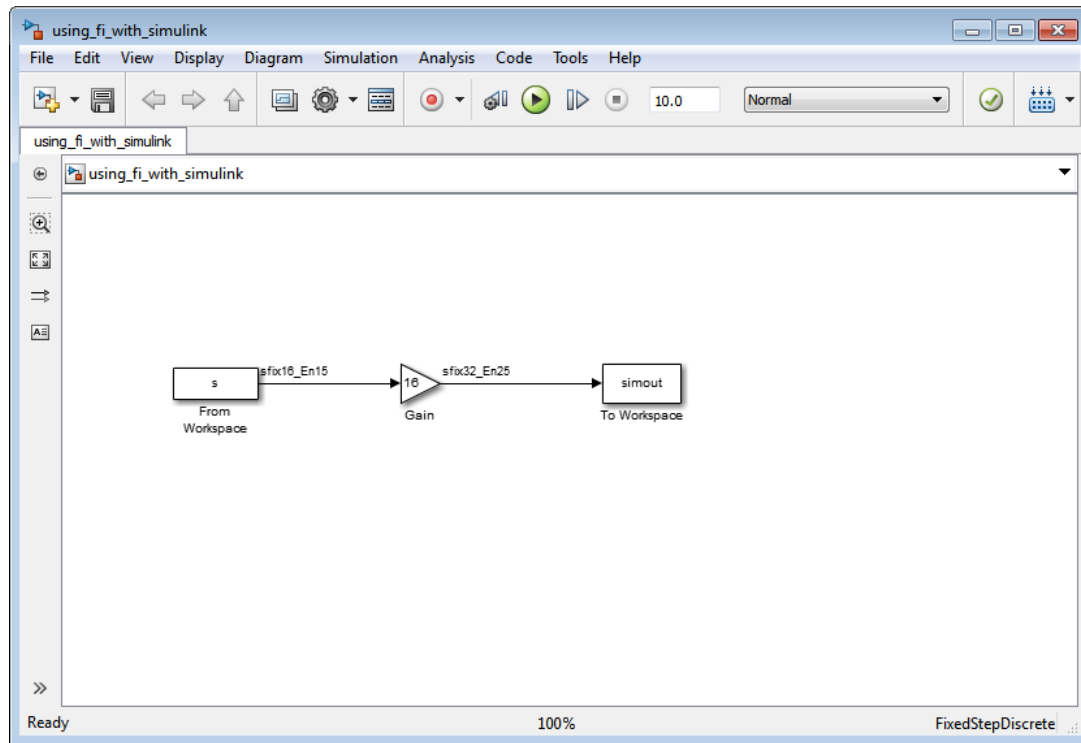
```
s =  
  
    signals: [1x1 struct]  
      time: [11x1 double]
```

The From Workspace block in the following model has the **fi** structure **s** in the **Data** parameter.

Remember, to write fixed-point data to the MATLAB workspace as a **fi** object, select the **Log fixed-point data as a fi object** check box on the To Workspace block dialog. Otherwise, fixed-point data is converted to **double** and written to the workspace as **double**.

In the model, the following parameters in the **Solver** pane of the **Model Configuration Parameters** dialog have the indicated settings:

- **Start time** — 0.0
- **Stop time** — 10.0
- **Type** — Fixed-step
- **Solver** — Discrete (no continuous states)
- **Fixed step size (fundamental sample time)** — 1.0



The To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` structure.

```
simout.signals.values
```

```
ans =
```

```

         0   -8.7041
    13.4634    6.5938
    14.5488   15.8296
     2.2578   10.5117
   -12.1089   -4.4707
   -15.3428  -15.3428
    -4.4707  -12.1089
    10.5117    2.2578
    15.8296   14.5488
```

```

        6.5938    13.4634
    -8.7041         0
    
```

```

DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 25
    
```

Setting the Value and Data Type of Block Parameters

You can use Fixed-Point Toolbox expressions to specify the value and data type of block parameters in Simulink. Refer to “Block Support for Data and Numeric Signal Types” in the Simulink documentation for more information.

Logging Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as `fi` objects. To enable signal logging for a signal, select the **Log signal data** option in the signal’s **Signal Properties** dialog box. For more information, refer to “Export Signal Data Using Signal Logging” in the Simulink documentation.

When you log signals from a referenced model or Stateflow® chart in your model, the word lengths of `fi` objects may be larger than you expect. The word lengths of fixed-point signals in referenced models and Stateflow charts are logged as the next largest data storage container size.

Accessing Fixed-Point Block Data During Simulation

Simulink provides an application program interface (API) that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to develop MATLAB programs capable of accessing block data while a simulation is running or to access the data from the MATLAB command line. Fixed-point signal information is returned to you via this API as `fi` objects. For more information on the API, refer to “Accessing Block Data During Simulation” in the Simulink documentation.

fi Objects with DSP System Toolbox

In this section...

“Reading Fixed-Point Signals from the Workspace” on page 9-7

“Writing Fixed-Point Signals to the Workspace” on page 9-7

“fi Objects with dfilt Objects” on page 9-11

Reading Fixed-Point Signals from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model using the Signal From Workspace and Triggered Signal From Workspace blocks from DSP System Toolbox™ software. Enter the name of the defined **fi** variable in the **Signal** parameter of the Signal From Workspace or Triggered Signal From Workspace block.

Writing Fixed-Point Signals to the Workspace

Fixed-point output from a model can be written to the MATLAB workspace via the Signal To Workspace or Triggered To Workspace block from the blockset. The fixed-point data is always written as a 2-D or 3-D array.

Note To write fixed-point data to the MATLAB workspace as a **fi** object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace or Triggered To Workspace block dialog. Otherwise, fixed-point data is converted to **double** and written to the workspace as **double**.

For example, you can use the following code to create a `fi` object in the MATLAB workspace. You can then use the Signal From Workspace block to bring the data into a Simulink model.

```
a = fi([sin(0:10)' sin(10:-1:0)'])
```

```
a =
```

```

      0   -0.5440
    0.8415   0.4121
    0.9093   0.9893
    0.1411   0.6570
   -0.7568  -0.2794
   -0.9589  -0.9589
   -0.2794  -0.7568
    0.6570   0.1411
    0.9893   0.9093
    0.4121   0.8415
   -0.5440      0

```

```

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15

```

The Signal From Workspace block in the following model has these settings:

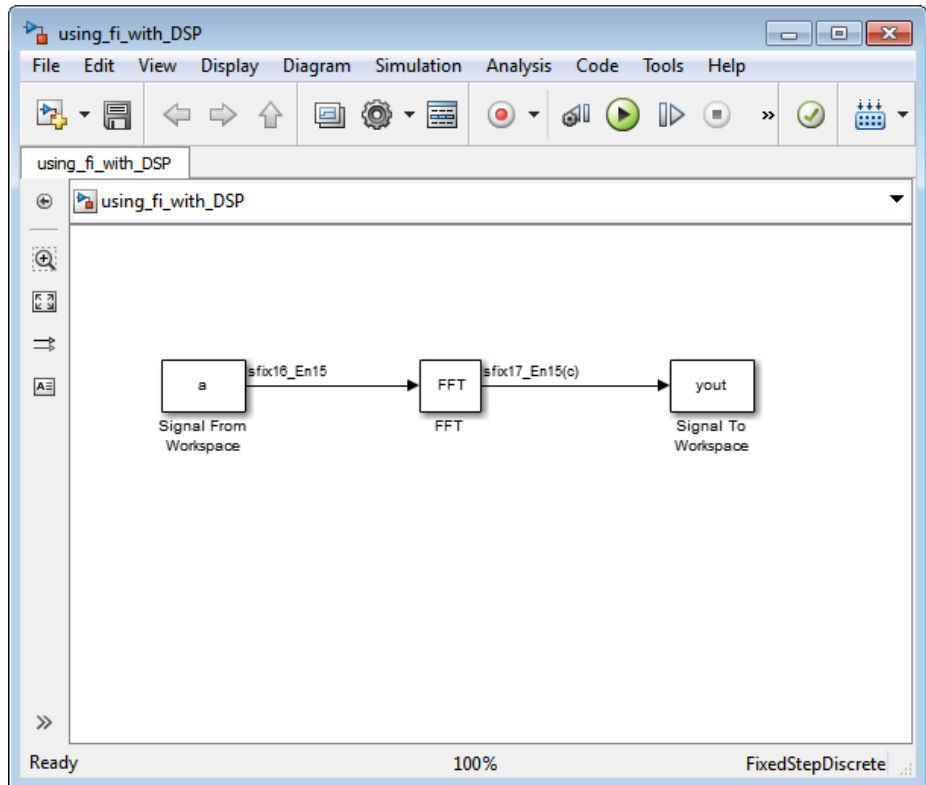
- **Signal** — a
- **Sample time** — 1
- **Samples per frame** — 2
- **Form output after final data value by** — Setting to zero

The following parameters in the **Solver** pane of the **Model Configuration Parameters** dialog have these settings:

- **Start time** — 0.0

- **Stop time** — 10.0
- **Type** — Fixed-step
- **Solver** — Discrete (no continuous states)
- **Fixed step size (fundamental sample time)** — 1.0

Remember, to write fixed-point data to the MATLAB workspace as a **fi** object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace block dialog. Otherwise, fixed-point data is converted to double and written to the workspace as double.



The Signal To Workspace block writes the result of the simulation to the MATLAB workspace as a **fi** object.

yout =

(:, :, 1) =

0.8415	-0.1319
-0.8415	-0.9561

(:, :, 2) =

1.0504	1.6463
0.7682	0.3324

(:, :, 3) =

-1.7157	-1.2383
0.2021	0.6795

(:, :, 4) =

0.3776	-0.6157
-0.9364	-0.8979

(:, :, 5) =

1.4015	1.7508
0.5772	0.0678

(:, :, 6) =

-0.5440	0
-0.5440	0


```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 17
FractionLength: 15
```

fi Objects with dfilt Objects

When the `Arithmetic` property is set to `'fixed'`, you can use an existing `fi` object as the input, states, or coefficients of a `dfilt` object in DSP System Toolbox software. Also, fixed-point filters in the toolbox return `fi` objects as outputs. Refer to the DSP System Toolbox software documentation for more information.

Ways to Generate Code

There are several ways to use Fixed-Point Toolbox software to generate code:

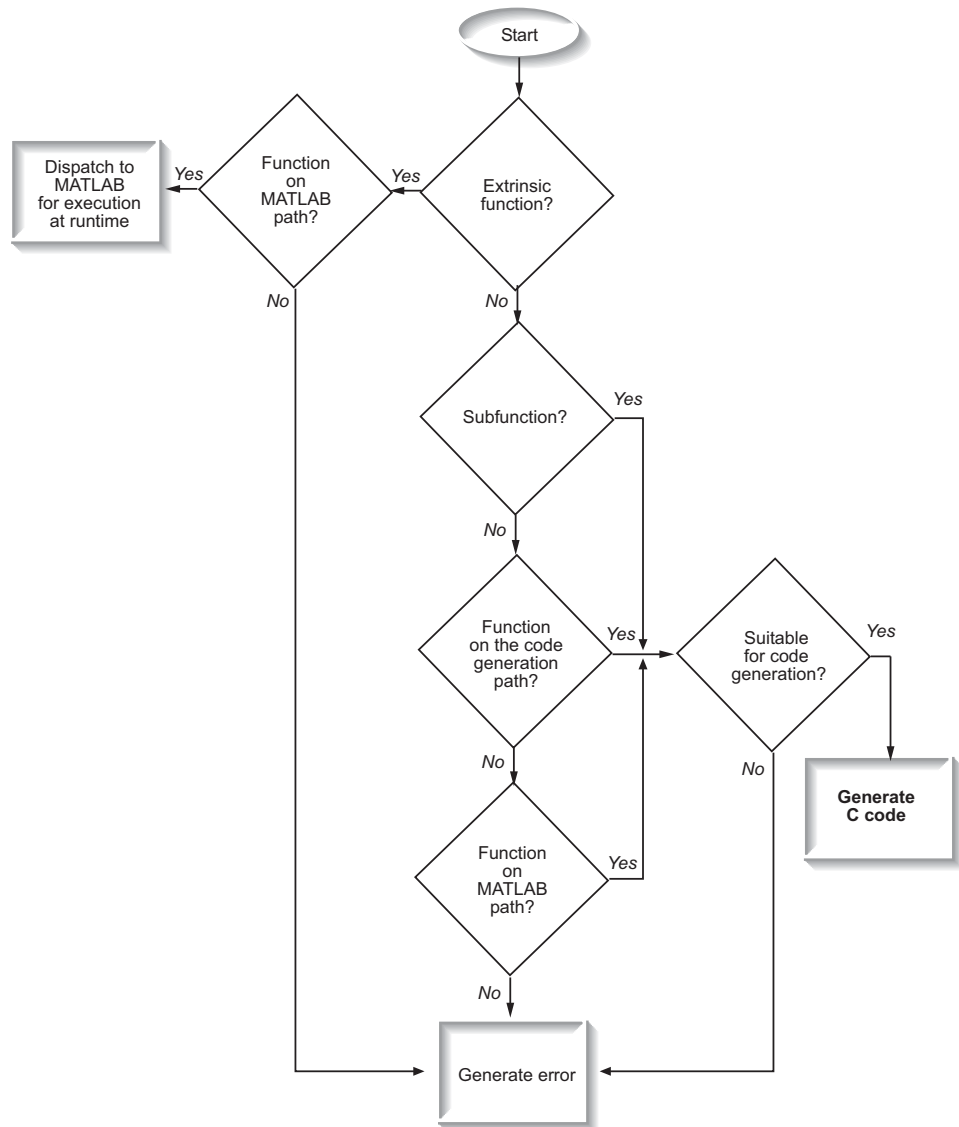
- The Fixed-Point Toolbox `fiaccel` function converts your fixed-point MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.
- The MATLAB Coder `codegen` function automatically converts MATLAB code to C/C++ code. Using the MATLAB Coder software allows you to accelerate your MATLAB code that uses Fixed-Point Toolbox software. To use the `codegen` function with Fixed-Point Toolbox software, you also need to have a MATLAB Coder license. For more information, see “C Code Generation at the Command Line” in the MATLAB Coder documentation.
- The MATLAB Function block allows you to use MATLAB code in your Simulink models that generate embeddable C/C++ code. To use the MATLAB Function block with Fixed-Point Toolbox software, you also need a Simulink license. For more information on the MATLAB Function block, see the Simulink documentation.

Calling Functions for Code Generation

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

Resolution of Function Calls in MATLAB Generated Code

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:



Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path
See “Compile Path Search Order” on page 10-4.
- Attempts to compile all functions unless the code generation software determines that it should not compile them or you explicitly declare them to be extrinsic.

An extrinsic function is a function on the MATLAB path that the compiler dispatches to MATLAB software for execution because the target language does not support the function. MATLAB does not generate code for extrinsic functions. You declare functions to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions” on page 10-12 “Declaring MATLAB Functions as Extrinsic Functions”.

The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, it automatically calls out to MATLAB for these functions. For standalone code generation, it does not generate code for these visualization functions. This capability removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of Files Types on Code Generation Path” on page 10-6

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1 Code generation path

MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.

2 MATLAB path

If the function is not on the code generation path, MATLAB searches this path.

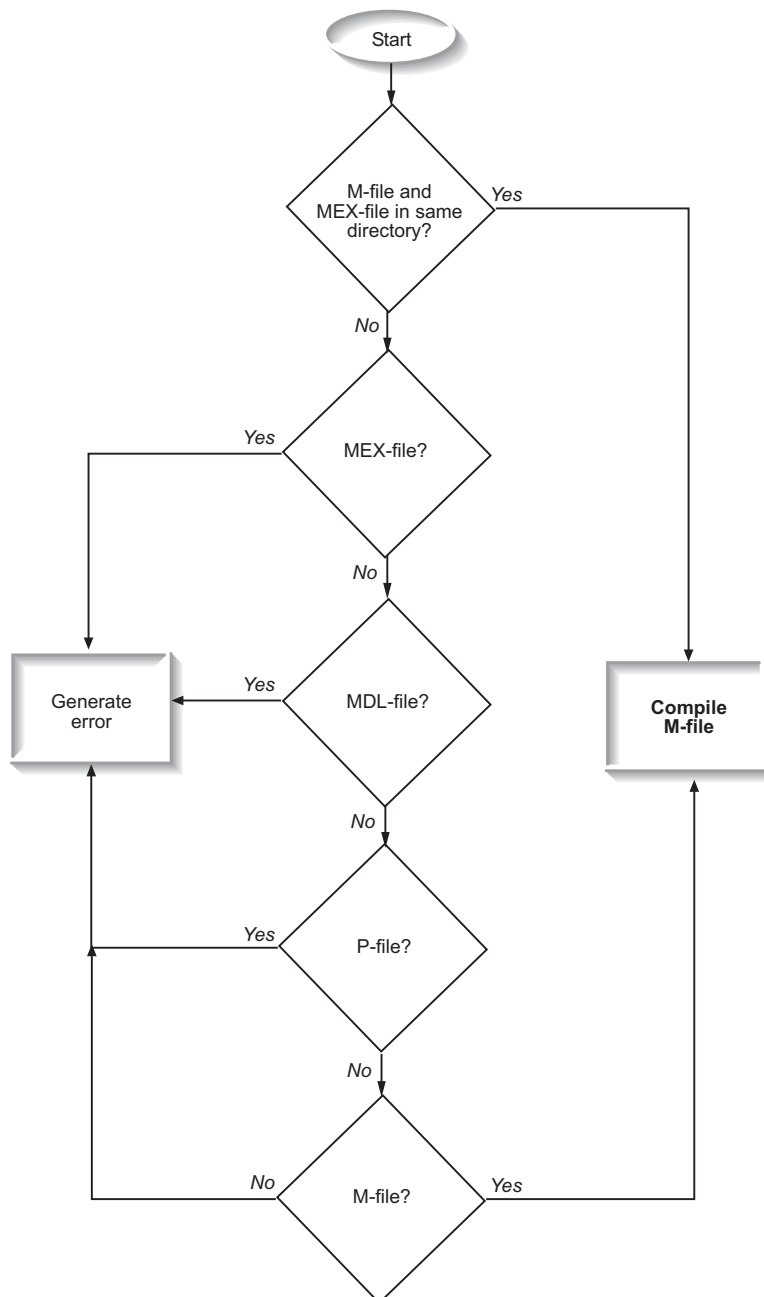
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path always shadows a file of the same name on the MATLAB path.

Resolution of Files Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:



Compilation Directive `%#codegen`

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

Call Local Functions

Local functions are functions defined in the body of MATLAB function. They work the same way for code generation as they do when executing your algorithm in the MATLAB environment.

The following example illustrates how to define and call a local function `mean`:

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

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

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals, '-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

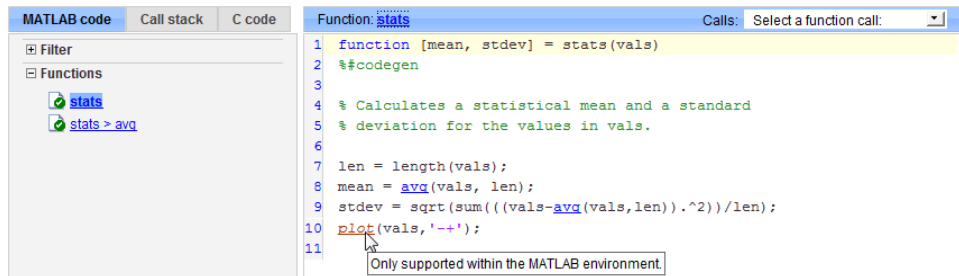
Call Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions Supported for Code Generation — Alphabetical List” on page 20-2.

Call MATLAB Functions

The code generation software attempts to generate code for all functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, it automatically calls out to MATLAB for these functions. For standalone code generation, it does not generate code for them.

For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot` and then run the generated MEX function, the code generation software dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.



For unsupported functions other than common visualization functions, you must declare the functions (like `pause`) to be extrinsic (see “Resolution of Function Calls in MATLAB Generated Code” on page 10-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 10-16).

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

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 10-12).

- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 10-16).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

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

Declaring Extrinsic Functions

The following code declares the MATLAB patch function `extrinsic` in the local function `create_plot`:

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
% and displays the triangle.
```

```
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
```

```
function create_plot(a, b, color)
%Declare patch and axis as extrinsic
```

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

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

The code generation software detects that `axis` is not supported for code generation and automatically treats it as an extrinsic function. The compiler does not generate code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

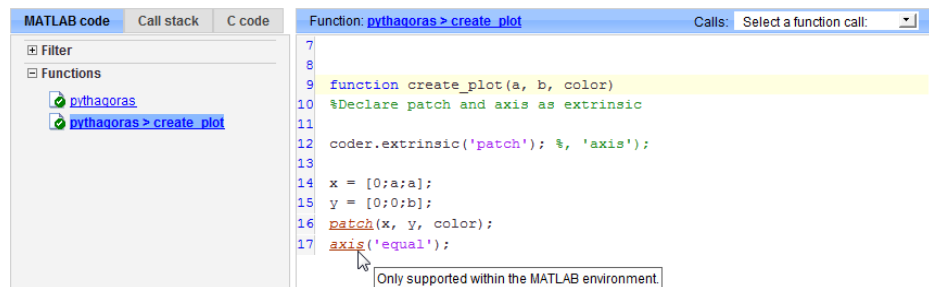
To test the function, follow these steps:

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

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

- 2 Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

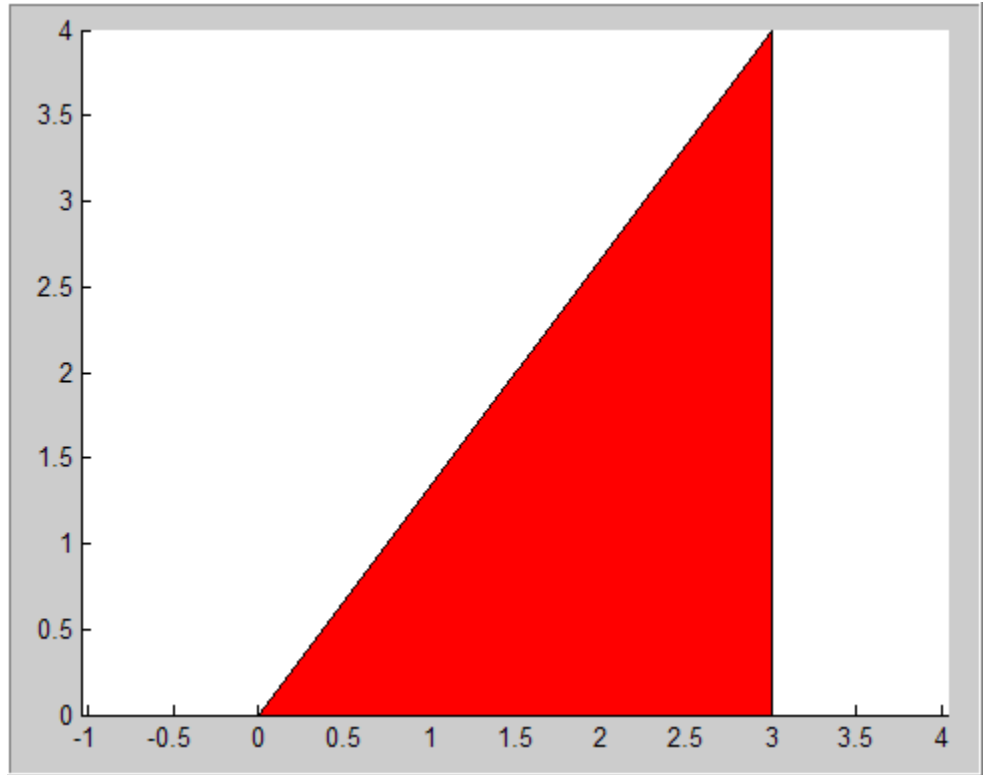
The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.



- 3 Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the `coder.extrinsic` Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that produce no output — such as `pause` — during simulation, without generating unnecessary code (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 10-16).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with `mxArrays`” on page 10-17).
- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 10-15).

- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 10-15). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 10-16).

Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` are treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = mymin(N, D);

function y = mymin(a,b)
coder.extrinsic('min');
y = min(a,b);
```

Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the local function `mymin`.

- Call the MATLAB function using `feval`, as described in “Calling MATLAB Functions Using `feval`” on page 10-16.

Calling MATLAB Functions Using `feval`

The function `feval` is automatically interpreted as an extrinsic function during code generation. Therefore, you can use `feval` to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

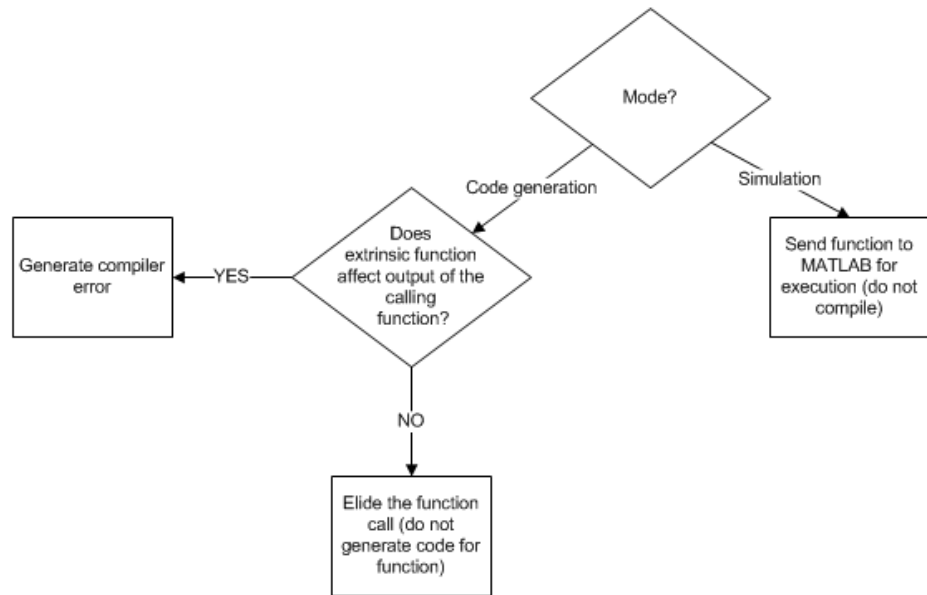
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not compiled — which has the same effect as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

How MATLAB Resolves Extrinsic Functions During Simulation

MATLAB resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, MATLAB generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with `mxArrays`” on page 10-17). Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, MATLAB issues a compiler error.

Working with `mxArrays`

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables
- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxAArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting `mxAArrays` to Known Types” on page 10-18.

Converting `mxAArrays` to Known Types

To convert an `mxAArray` to a known type, assign the `mxAArray` to a variable whose type is defined. At run time, the `mxAArray` is converted to the type of the variable assigned to it. However, if the data in the `mxAArray` 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 `mxAArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxAArrays` to another MATLAB function — in this case, `min` — you cannot assign the `mxAArray` returned by `min` to the output `y`.

If you run this function `foo` in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

Function output 'y' cannot be of MATLAB type.

To fix this problem, define `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller or write to the caller's workspace do not work during code generation. Such functions include:
 - `dbstack`
 - `evalin`
 - `assignin`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code may produce unpredictable results if your extrinsic function performs any of the following actions at run time:
 - Change folders
 - Change the MATLAB path
 - Delete or add MATLAB files
 - Change warning states
 - Change MATLAB preferences
 - Change Simulink parameters

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Code Generation for MATLAB Classes

- “MATLAB Classes Definition for Code Generation” on page 11-2
- “Classes That Support Code Generation” on page 11-9
- “Memory Allocation Requirements” on page 11-10
- “Generate Code for MATLAB Value Classes” on page 11-11
- “Generate Code for MATLAB Handle Classes and System Objects” on page 11-17
- “MATLAB Classes in Code Generation Reports” on page 11-20
- “Troubleshooting Issues with MATLAB Classes” on page 11-23

MATLAB Classes Definition for Code Generation

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

What's Different	More Information
Class must be in a single file. Because of this limitation, there is no code generation support for a class definition that uses an @-folder.	"Creating a Single, Self-Contained Class Definition File"
Restricted set of language features.	"Language Limitations" on page 11-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 11-4
Definition of class properties.	"Defining Class Properties for Code Generation" on page 11-5
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 11-17
Calls to base class constructor.	"Calls to Base Class Constructor" on page 11-6

Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events
- Listeners
- Arrays of objects
- Recursive data structures
 - Linked lists

- Trees
- Graphs
- Overloadable operators `subsref`, `subsassign`, and `subsindex`

In MATLAB, classes can define their own versions of the `subsref`, `subsassign`, and `subsindex` methods. Code generation does not support classes that have their own definitions of these methods.

- The empty method

In MATLAB, all classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.

- The following MATLAB handle class methods:

- `addlistener`
- `delete`
- `eq`
- `findobj`
- `findprop`
- `ge`
- `gt`
- `isvalid`
- `le`
- `lt`
- `ne`
- `notify`

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

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

```
codegen ClassNameA
```

- If an entry-point MATLAB function has an input or output that is a MATLAB class, you cannot generate code for this function.

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

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

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

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

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

- You cannot use `coder.extrinsic` or `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the `coder.ceval` function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you normally would when running your code in the MATLAB environment:

- If a class has a property of handle type, set the property in the class constructor. For System objects, you can also use the `setupImpl` method.
- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of any class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generation software requires a complete assignment to each variable. Similarly, before using any properties, you must explicitly define the class, size, and complexity of all properties.

- Initial values:
 - If the property has no explicit initial value, the code generation software assumes that it is undefined at the beginning of the constructor. The code generation software does not assign an empty matrix as the default.
 - If the property has no initial value and the code generation software cannot determine that the property is assigned on all paths prior to first use, the software generates a compilation error.
 - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

For example, for a nontunable property, you can use the following assignment:

```
mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
```

You cannot use the following partial assignments:

```
mySystemObject.nonTunableProperty.fieldA = a;
mySystemObject.nonTunableProperty.fieldB = b;
```

- If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.
- `coder.versizecoder.versizecoder.versize` is not supported for any class properties.
- MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.targetcoder.targetcoder.target` in MATLAB class property initialization, `coder.target` is always `''`.

Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must be before any `for`, `if`, `return`, `switch` or `while` statements.

For example, if you define a class B based on class A:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
        end
    end
end
```

```

        end
        obj = obj@A(a,b);
    end

    end
end

```

Because the class definition for B uses an if statement before calling the base class constructor for A, you cannot generate code for function callB:

```

function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
end

```

However, you can generate code for callB if you define class B as:

```

classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end

    end
end

function [a,b] = getaandb(varargin)
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end

```


Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 11-11
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 11-17

For more information, see:

- “Classes in the MATLAB Language”
- “MATLAB Classes Definition for Code Generation” on page 11-2

Memory Allocation Requirements

When you create a handle object, you must assign the object to a persistent variable or to a property of another MATLAB object that must also be a persistent variable. The assignment must be in an `if - isempty` clause. After assignment, you can copy the object to a local variable, pass it to or return it from another function. For more information, see “Generate Code for MATLAB Handle Classes and System Objects” on page 11-17.

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

- 1 In a writable folder, create a MATLAB value class, Shape. Save the code as Shape.m.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out = obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    methods(Abstract = true)
        getarea(obj);
    end
    methods(Static)
        function d = distanceBetweenShapes(shape1,shape2)
            xDist = abs(shape1.centerX - shape2.centerX);
            yDist = abs(shape1.centerY - shape2.centerY);
            d = sqrt(xDist^2 + yDist^2);
        end
    end
end
```

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

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

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

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

- 4** Write a function that uses this class.

```
function [TotalArea, Distance] = use_shape
%#codegen
s = Square(2,1,2);
r = Rhombus(3,4,7,10);
TotalArea = s.area + r.area;
Distance = Shape.distanceBetweenShapes(s,r);
```

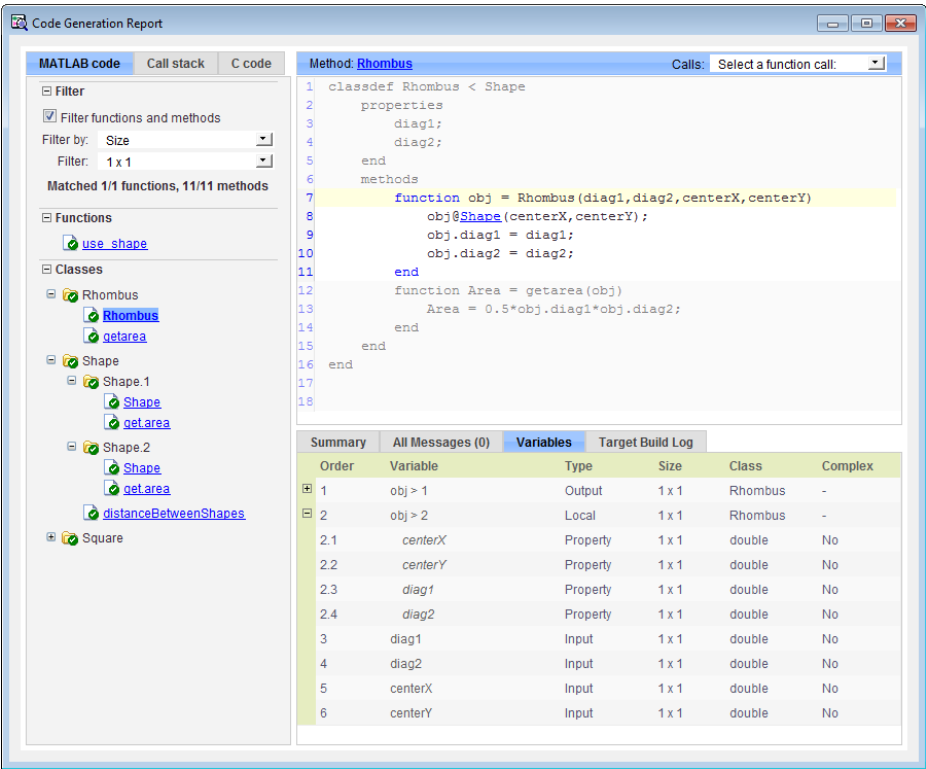
- 5** Generate a static library for `use_shape` and generate a code generation report.

```
codegen -config:lib -report use_shape
```

`codegen` generates a C static library with the default name, `use_shape`, and supporting files in the default folder, `codegen/lib/use_shape`.

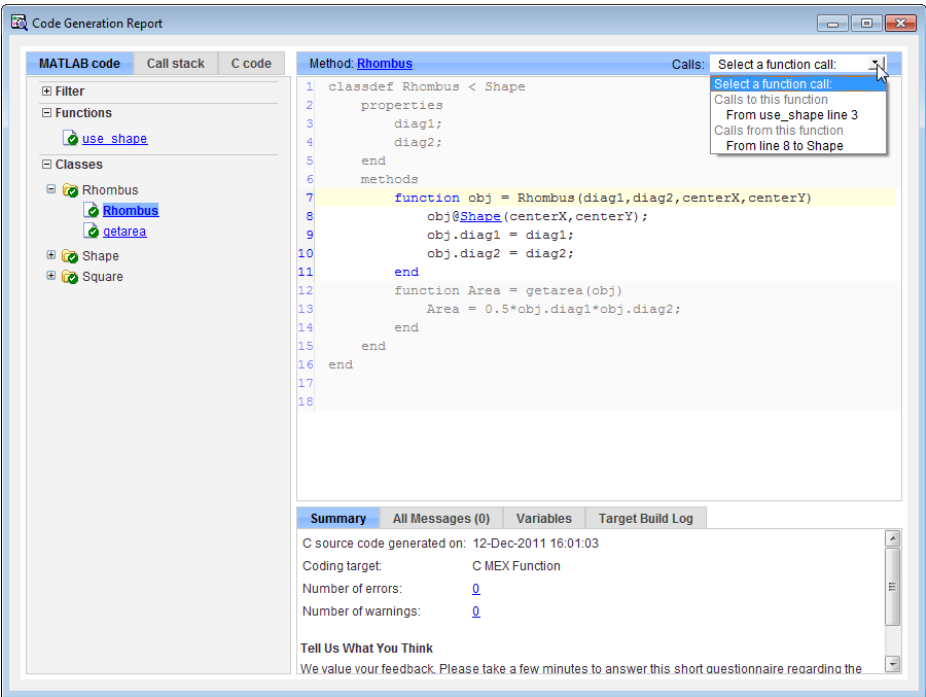
- 6** Click the *View report* link.
- 7** In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of all the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties, as shown for `obj>1`. To view the complete list of properties, expand the list as shown for `obj>2`.

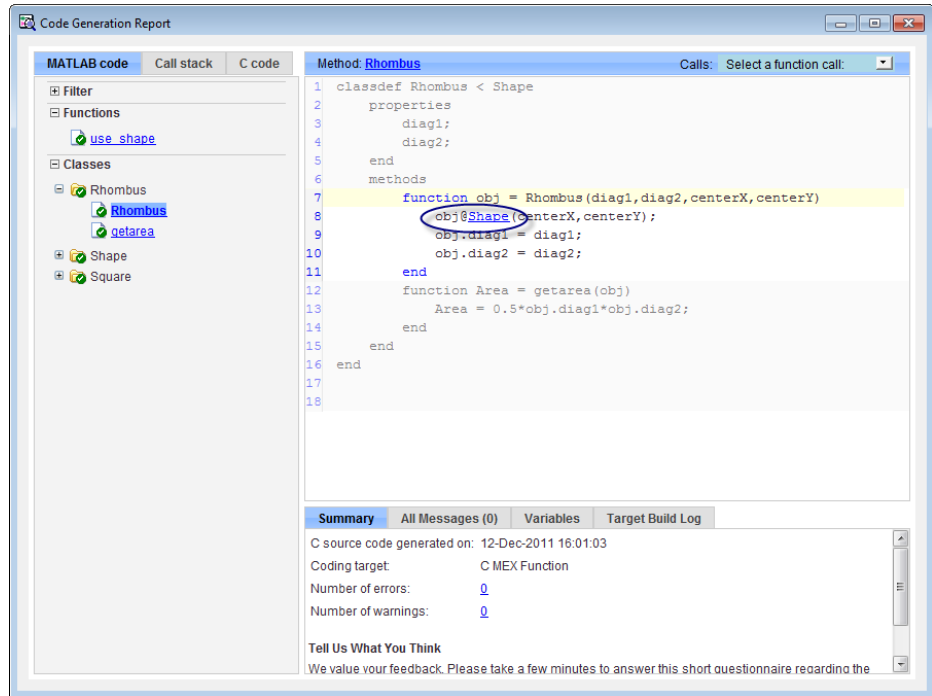


8 At the top right side of the report, expand the **Calls** list.

The **Calls** list shows that there is a call to the Rhombus constructor from use_shape and that this constructor calls the Shape constructor.



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



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

Generate Code for MATLAB Handle Classes and System Objects

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

- 1 In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access=protected)
        % stepImpl method is called by the step method
        function y = stepImpl(~,x)
            y = x+1;
        end
    end
end
```

- 2 Write a function that uses this System object.

```
function y = testAddOne(x)
%#codegen
    persistent p;
    if isempty(p)
        p = AddOne();
    end
    y = p.step(x);
end
```

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

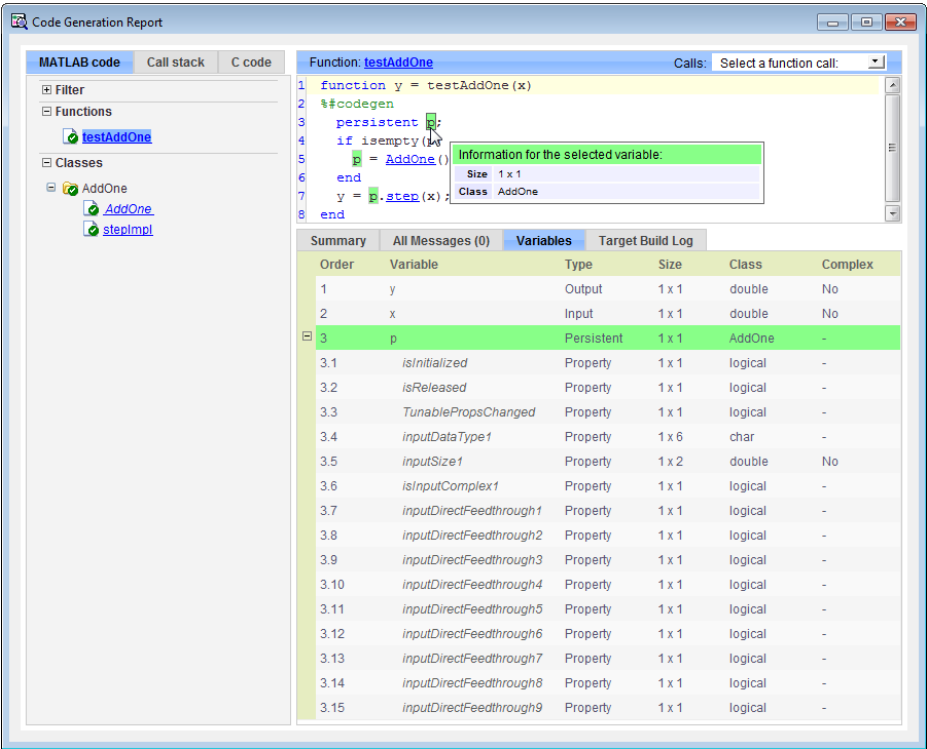
- 3 Generate a MEX function for this code.

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

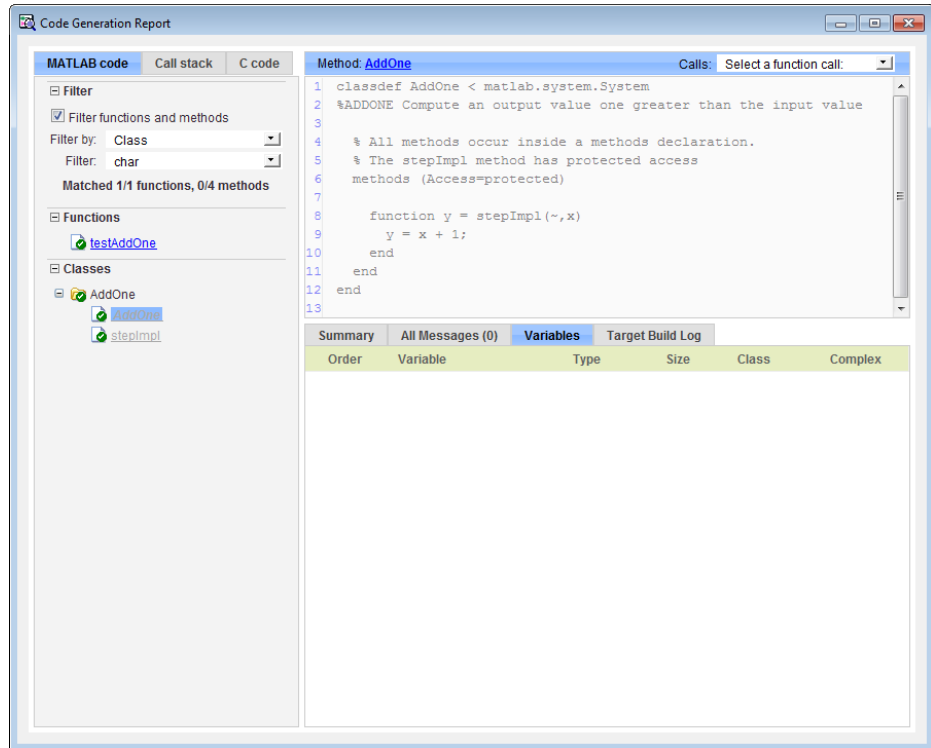
The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur. The `-args` option specifies that the `testAddOne` function takes one scalar double input.

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

- 4 Click the *View report* link.
- 5 In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.



- 6 To view the class definition, on the **Classes** panel, click `AddOne`.



MATLAB Classes in Code Generation Reports

What Reports Tell You About Classes

Code generation reports:

- Provide a hierarchical tree of the classes used in your MATLAB code.
- Display a list of methods for each class in the MATLAB code tab.
- Display the objects used in your MATLAB code together with their properties on the **Variables** tab.
- Provide a filter so that you can sort methods by class, size, and complexity.
- List the set of calls from and to the selected method in the **Calls** list.

How Classes Appear in Code Generation Reports

In the MATLAB Code Tab

The report displays an alphabetical hierarchical list of the classes used in the your MATLAB code. For each class, you can:

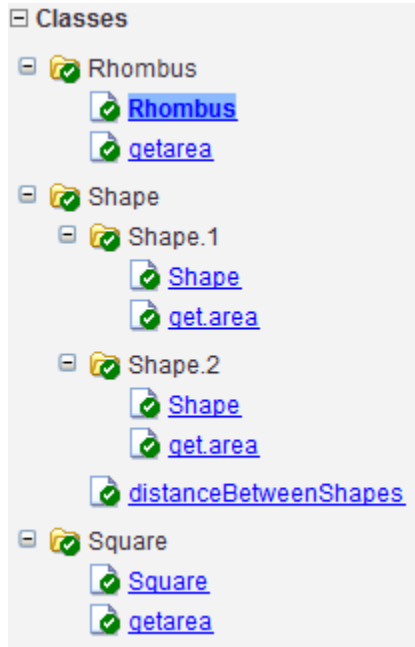
- Expand the class information to view the class methods.
- View a class method by clicking its name. The report displays the methods in the context of the full class definition.
- Filter the methods by size, complexity, and class by using the **Filter functions and methods** option.

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

Specializations. If the same class is specialized into multiple different classes, the report differentiates the specializations by grouping each one under a single node in the tree. The report associates the class definition functions and static methods with the primary node. It associates the instance-specific methods with the corresponding specialized node.

For example, consider a base class, `Shape` that has two specialized subclasses, `Rhombus` and `Square`. The `Shape` class has an abstract method, `getarea`,

and a static method, `distanceBetweenShapes`. The code generation report, displays a node for the specialized `Rhombus` and `Square` classes with their constructors and `getarea` method. It displays a node for the `Shape` class and its associated static method, `distanceBetweenShapes`, and two instances of the `Shape` class, `Shape1` and `Shape2`.



Packages. If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see “Packages Create Namespaces”.

In the Variables Tab

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

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a

MATLAB object with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

How to Generate a Code Generation Report

Add the `-report` option to your `codegen` command (requires a MATLAB Coder license)

Troubleshooting Issues with MATLAB Classes

Class *c*lass does not have a property with name *name*

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
    properties
        myprop
    end
    methods
        function this = MyClass
            this.myprop = MyClass2;
        end
        function y = mymethod(this)
            y = this.myprop;
        end
    end
end
```

```
classdef MyClass2 < handle
    properties
        aa
    end
end
```

You cannot generate code for function `foo`.

```
function foo

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

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

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.

Workaround

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo

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

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

Defining Data for Code Generation

- “Data Definition for Code Generation” on page 12-2
- “Code Generation for Complex Data” on page 12-4
- “Code Generation for Characters” on page 12-6

Data Definition for Code Generation

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

Data	What's Different	More Information
Complex numbers	<ul style="list-style-type: none">Complexity of variables must be set at time of assignment and before first useExpressions containing a complex number or variable always evaluate to a complex result, even if the result is zero <div>Note Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</div>	“Code Generation for Complex Data” on page 12-4
Characters	Restricted to 8 bits of precision	“Code Generation for Characters” on page 12-6

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

Code Generation for Complex Data

In this section...

“Restrictions When Defining Complex Variables” on page 12-4

“Expressions Containing Complex Operands Yield Complex Results” on page 12-5

Restrictions When Defining Complex Variables

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

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

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

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

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

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

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

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

Expressions Containing Complex Operands Yield Complex Results

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

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

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

There are two exceptions to this behavior:

- Functions that take complex arguments, but produce real results

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

- Functions that take real arguments, but produce complex results:

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

Code Generation for Characters

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

Defining Functions for Code Generation

- “Specify Variable Numbers of Arguments” on page 13-2
- “Supported Index Expressions” on page 13-3
- “Apply Operations to a Variable Number of Arguments” on page 13-4
- “Implement Wrapper Functions” on page 13-7
- “Pass Property/Value Pairs” on page 13-8
- “Variable Length Argument Lists for Code Generation” on page 13-10

Specify Variable Numbers of Arguments

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

Common applications of `varargin` and `varargout` for code generation are to:

- “Apply Operations to a Variable Number of Arguments” on page 13-4
- “Implement Wrapper Functions” on page 13-7
- “Pass Property/Value Pairs” on page 13-8

Code generation relies on loop unrolling to produce simple and efficient code for `varargin` and `varargout`. This technique permits most common uses of `varargin` and `varargout`, but not all (see “Variable Length Argument Lists for Code Generation” on page 13-10). The following sections explain how to code effectively using these constructs.

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

Supported Index Expressions

In MATLAB, `varargin` and `varargout` are cell arrays. Generated code does not support cell arrays, but does allow you to use the most common syntax — curly braces `{}` — for indexing into `varargin` and `varargout` arrays, as in this example:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i};
end
```

You can use the following index expressions. The *exp* arguments must be constant expressions or depend on a loop index variable.

Expression		Description
varargin (read only)	varargin{exp}	Read the value of element <i>exp</i>
	varargin{exp1: exp2}	Read the values of elements <i>exp1</i> through <i>exp2</i>
	varargin{:}	Read the values of all elements
varargout (read and write)	varargout{exp}	Read or write the value of element <i>exp</i>

Note The use of `()` is not supported for indexing into `varargin` and `varargout` arrays.

Apply Operations to a Variable Number of Arguments

You can use `varargin` and `varargout` in `for`-loops to apply operations to a variable number of arguments. To index into `varargin` and `varargout` arrays in generated code, the value of the loop index variable must be known at compile time. Therefore, during code generation, the compiler attempts to automatically unroll these `for`-loops. Unrolling eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. For example, the following function automatically unrolls its `for`-loop in the generated code:

```
%#codegen
function [cmLen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmLen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

When to Force Loop Unrolling

To automatically unroll `for`-loops containing `varargin` and `varargout` expressions, the relationship between the loop index expression and the index variable must be determined at compile time.

In the following example, the function `fcn` cannot detect a logical relationship between the index expression `j` and the index variable `i`:

```
%#codegen
function [x,y,z] = fcn(a,b,c)

[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;
for i = 1:length(varargin)
    j = j+1;
    varargout{j} = varargin{j};
```


end

As a result, the function does not unroll the loop and generates a compilation error:

Nonconstant expression or empty matrix.
This expression must be constant because
its value determines the size or class of some expression.

To fix the problem, you can force loop unrolling by wrapping the loop header in the function `coder.unrollcoder.unrollcoder.unroll`, as follows:

```
%#codegen
function [x,y,z] = fcn(a,b,c)
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    j = 0;
    for i = coder.unroll(1:length(varargin))
        j = j + 1;
        varargout{j} = varargin{j};
    end;
```

Using Variable Numbers of Arguments in a for-Loop

The following example multiplies a variable number of input dimensions in inches by 2.54 to convert them to centimeters:

```
%#codegen
function [cmLen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmLen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

Key Points About the Example

- `varargin` and `varargout` appear in the local function `inch_2_cm`, not in the top-level function `conv_2_metric`.
- The index into `varargin` and `varargout` is a for-loop variable

For more information, see “Variable Length Argument Lists for Code Generation” on page 13-10.

Implement Wrapper Functions

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

Passing Variable Numbers of Arguments from One Function to Another

The following example passes a variable number of inputs to different optimization functions, based on a specified input method:

```
%#codegen
function answer = fcn(method,a,b,c)
answer = optimize(method,a,b,c);

function answer = optimize(method,varargin)
    if strcmp(method,'simple')
        answer = simple_optimization(varargin{:});
    else
        answer = complex_optimization(varargin{:});
    end
    ...
```

Key Points About the Example

- You can use `{:}` to read all elements of `varargin` and pass them to another function.
- You can mix variable and fixed numbers of arguments.

For more information, see “Variable Length Argument Lists for Code Generation” on page 13-10.

Pass Property/Value Pairs

You can use `varargin` to pass property/value pairs in functions. However, for code generation, you must take precautions to avoid type mismatch errors when evaluating `varargin` array elements in a `for`-loop:

If	Do This:
You assign <code>varargin</code> array elements to local variables in the <code>for</code> -loop	Verify that for all pairs, the size, type, and complexity are the same for each property and the same for each value
Properties or values have different sizes, types, or complexity	Do not assign <code>varargin</code> array elements to local variables in a <code>for</code> -loop; reference the elements directly

For example, in the following function `test1`, the sizes of the property strings and numeric values are not the same in each pair:

```
%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        name = varargin{i};
        value = varargin{i+1};
        switch name
            case 'size'
                v = set_size(v, value);
            case 'rgb'
                v = set_color(v, value);
            otherwise
            end
        end
    end
end
```

...

Generated code determines the size, type, and complexity of a local variable based on its first assignment. In this example, the first assignments occur in the first iteration of the for-loop:

- Defines local variable `name` with size equal to 4
- Defines local variable `value` with a size of scalar

However, in the second iteration, the size of the property string changes to 3 and the size of the numeric value changes to a vector, resulting in a type mismatch error. To avoid such errors, reference `varargin` array values directly, not through local variables, as highlighted in this code:

```
%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        switch varargin{i}
            case 'size'
                v = set_size(v, varargin{i+1});
            case 'rgb'
                v = set_color(v, varargin{i+1});
            otherwise
                %
            end
        end
    end
end
...
```

Variable Length Argument Lists for Code Generation

Do not use varargin or varargout in top-level functions

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

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

For example, the following code generates compilation errors:

```
%#codegen
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

To fix the problem, write a top-level function that specifies a fixed number of inputs and outputs and then call `inch_2_cm` as an external function or local function, as in this example:

```
%#codegen
function [cmL, cmW, cmH] = conv_2_metric(inL, inW, inH)
[cmL, cmW, cmH] = inch_2_cm(inL, inW, inH);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

Use curly braces {} to index into the argument list

For code generation, you can use curly braces {}, but not parentheses (), to index into varargin and varargout arrays. For more information, see “Supported Index Expressions” on page 13-3.

Verify that indices can be computed at compile time

If you use an expression to index into `varargin` or `varargout`, make sure that the value of the expression can be computed at compile time. For examples, see “Apply Operations to a Variable Number of Arguments” on page 13-4.

Do not write to `varargin`

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

Defining MATLAB Variables for C/C++ Code Generation

- “Variables Definition for Code Generation” on page 14-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 14-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 14-7
- “Reassignment of Variable Properties” on page 14-9
- “Define and Initialize Persistent Variables” on page 14-10
- “Reuse the Same Variable with Different Properties” on page 14-11
- “Avoid Overflows in for-Loops” on page 14-16
- “Supported Variable Types” on page 14-18

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 14-3.

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

In this section...
“Define Variables By Assignment Before Using Them” on page 14-3
“Use Caution When Reassigning Variables” on page 14-6
“Use Type Cast Operators in Variable Definitions” on page 14-6
“Define Matrices Before Assigning Indexed Variables” on page 14-6

Define Variables By Assignment Before Using Them

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

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

Define properties this way so that the variable is defined on all execution paths during C/C++ code generation (see Defining a Variable for Multiple Execution Paths on page 14-4).

The data that you assign to a variable can be a scalar, matrix, or structure. If your variable is a structure, define the properties of each field explicitly (see Defining All Fields in a Structure on page 14-5).

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in “Eliminate Redundant Copies of Variables in Generated Code” on page 14-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see “Define and Initialize Persistent Variables” on page 14-10.

Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

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

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

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

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

Defining All Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Try to use s
use(s);
...
```

Here, the first part of the `if` statement uses only the field `a`, and the `else` clause uses fields `a` and `b`. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation” “Structure Definition for Code Generation” on page 19-2 “Structure Definition for Code Generation”.

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

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

Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 14-9.

Use Type Cast Operators in Variable Definitions

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

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

Define Matrices Before Assigning Indexed Variables

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

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

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

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 21-32.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

“When Redundant Copies Occur” on page 14-7

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 14-7

“Defining Uninitialized Variables” on page 14-8

When Redundant Copies Occur

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

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopycoder.nullcopycoder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 14-7.

How to Eliminate Redundant Copies by Defining Uninitialized Variables

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

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

What happens if you access uninitialized data?

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

Defining Uninitialized Variables

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

```
function X = fcn %#codegen

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

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

```
function X = fcn2 %#codegen

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


Reassignment of Variable Properties

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

Dynamically sized variables

A variable can hold values that have the same class and complexity but different sizes. If the size of the initial assignment is not constant, the variable is dynamically sized in generated code. For more information, see “Variable-Size Data” “Variable-Size Data” “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 14-11.

Define and Initialize Persistent Variables

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

```
persistent PROD_X;
```

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

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

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

Reuse the Same Variable with Different Properties

In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 14-11

“When You Cannot Reuse Variables” on page 14-12

“Limitations of Variable Reuse” on page 14-14

When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if MATLAB can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report (see “Viewing Variables in Your MATLAB Code” “Create and Use Fixed-Point Code Generation Reports” on page 8-52 “Viewing Variables in Your MATLAB Code”).

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

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

To compile this example and see how MATLAB renames the reused variable `t`, see Variable Reuse in an `if` Statement on page 14-12.

When You Cannot Reuse Variables

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

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

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

Variable Reuse in an if Statement

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

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

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

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

2 Compile example1.

For example, to generate a MEX function, enter:

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

Note codegen requires a MATLAB Coder license.

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

3 Open the code generation report.

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

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

```
% First time t is used to hold a scalar double value.
```

```
t = mean(mean(u)) / numel(u);
```

```
u = u - t;
```



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

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

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

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



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

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

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

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

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

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

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

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

- Persistent variables.
- Global variables.
- Variables passed to C code using `coder.ref`, `coder.rref`, `coder.wref`.
- Variables whose size is set using `coder.varsize`.
- Variables whose names are controlled using `coder.cstructname`.
- The index variable of a `for`-loop when it is used inside the loop body.

- The block outputs of a MATLAB Function block in a Simulink model.
- Chart-owned variables of a MATLAB function in a Stateflow chart.

Avoid Overflows in for-Loops

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

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

Loop conditions causing the error	Workaround
<ul style="list-style-type: none">• The loop counter increments by 1• The end value equals the maximum value of the integer type• The loop is not covering the full range of the integer type	<p>Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:</p> <pre>N=intmax('int16') for k=N-10:N</pre> <p>with:</p> <pre>for k=1:10</pre>
<ul style="list-style-type: none">• The loop counter decrements by 1• The end value equals the minimum value of the integer type• The loop is not covering the full range of the integer type	<p>Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:</p> <pre>N=intmin('int32') for k=N+10:-1:N</pre> <p>with:</p> <pre>for k=10:-1:1</pre>

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> • The loop counter increments or decrements by 1 • The start value equals the minimum or maximum value of the integer type • The end value equals the maximum or minimum value of the integer type <p>The loop covers the full range of the integer type.</p>	<p>Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double. For example, rewrite:</p> <pre> M= intmin('int16'); N= intmax('int16'); for k=M:N % Loop body end to M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N) % Loop body end </pre>
<ul style="list-style-type: none"> • The loop counter increments or decrements by a value not equal to 1 • On last loop iteration, the loop variable value is not equal to the end value <hr/> <p>Note The software error checking might be too conservative and report the possibility of an infinite under these circumstances even though an infinite loop would never occur.</p> <hr/>	<p>Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.</p>

Supported Variable Types

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

Type	Description
char	Character array (string)
complex	Complex data. Cast function takes real and imaginary components
double	Double-precision floating point
int8, int16, int32	Signed integer
logical	Boolean true or false
single	Single-precision floating point
struct	Structure
uint8, uint16, uint32	Unsigned integer
Fixed-point	See “Fixed-Point Data Types” on page 1-2.

Design Considerations for C/C++ Code Generation

- “When to Generate Code from MATLAB Algorithms” on page 15-2
- “Which Code Generation Feature to Use” on page 15-4
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 15-6
- “MATLAB Code Design Considerations for Code Generation” on page 15-7
- “Expected Differences in Behavior After Compiling MATLAB Code” on page 15-9
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 15-13

When to Generate Code from MATLAB Algorithms

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

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

When Not to Generate Code from MATLAB Algorithms

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

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

To:	Use:
Deploy web-based or Windows applications	<ul style="list-style-type: none">• MATLAB Builder NE• MATLAB Builder JA
Interface C code with MATLAB	MATLAB mex function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	codegen function	MATLAB Coder	Try this in “MEX Function Generation at the Command Line”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder user interface	MATLAB Coder	Try this in “C Code Generation Using the Project Interface”.
	codegen function	MATLAB Coder	Try this in “C Code Generation at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder user interface	MATLAB Coder	See “Accelerate MATLAB Algorithms”.
	codegen function	MATLAB Coder	
Integrate MATLAB code into Simulink	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code”.
Speed up fixed-point MATLAB code	fiaccel function	Fixed-Point Toolbox	Learn more in “Code Acceleration and Code Generation from MATLAB” on page 8-3.
Integrate custom C code into MATLAB and generate efficient, readable code	codegen function	MATLAB Coder	Learn more in “Custom C/C++ Code Integration”.

To...	Use...	Required Product	To Explore Further...
Integrate custom C code into code generated from MATLAB	<code>coder.ceval</code> function	MATLAB Coder	Learn more in <code>coder.ceval</code> .
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder™	Learn more at www.mathworks.com/products/slhdlcoder .

Prerequisites for C/C++ Code Generation from MATLAB

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

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

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

- Data types

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

- Array sizing

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

- Memory

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

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get best speed performance, 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. The default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms is not a good compiler for performance.
- Consider disabling run-time checks.

By default, 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” “Data Definition Basics” “Data Definition Basics”
- “Variable-Size Data” “Variable-Size Data” “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” “Bounded Versus Unbounded Variable-Size Data” on page 21-4 “Bounded Versus Unbounded Variable-Size Data”
- “Control Dynamic Memory Allocation” “Control Dynamic Memory Allocation” on page 8-93
- “Control Run-Time Checks” “Control Run-Time Checks” on page 8-71

Expected Differences in Behavior After Compiling MATLAB Code

In this section...

“Why Are There Differences?” on page 15-9

“Character Size” on page 15-9

“Order of Evaluation in Expressions” on page 15-9

“Termination Behavior” on page 15-10

“Size of Variable-Size N-D Arrays” on page 15-10

“Size of Empty Arrays” on page 15-11

“Floating-Point Numerical Results” on page 15-11

“NaN and Infinity Patterns” on page 15-12

“Code Generation Target” on page 15-12

“MATLAB Class Initial Values” on page 15-12

“Variable-Size Support for Code Generation” on page 15-12

Why Are There Differences?

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

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Characters” on page 12-6.

Order of Evaluation in Expressions

Generated code does not enforce order of evaluation in expressions. For most expressions, order of evaluation is not significant. However, for expressions

with side effects, the generated code may produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that:

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

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

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

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

as

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

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

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they have no side effects. As a result, the generated code may terminate even though the corresponding MATLAB code does not.

Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code,

but always returns [4 2] in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 21-29.

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 21-30.

Floating-Point Numerical Results

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

When computer hardware uses extended precision registers

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

For certain advanced library functions

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

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

For implementation of BLAS library functions

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

NaN and Infinity Patterns

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

Code Generation Target

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

MATLAB Class Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation”“Defining Class Properties for Code Generation” on page 11-5“Defining Class Properties for Code Generation”.

Variable-Size Support for Code Generation

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

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

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

MATLAB supports the following language features in generated code:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Variable-Size Data Definition for Code Generation” on page 21-3)
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 21-32)
- Complex numbers (see “Code Generation for Complex Data” on page 12-4)
- Numeric classes (see “Supported Variable Types” on page 14-18)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB” on page 8-3)
- Program control statements `if`, `switch`, `for`, and `while`
- All arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see “Define and Initialize Persistent Variables” on page 14-10)
- Global variables (see “Specifying Global Variable Type and Initial Value in a Project”).
- Structures
- Characters (see “Code Generation for Characters” on page 12-6)
- Function handles
- Frames (see “Add Frame-Based Signals”)
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions
- MATLAB classes

- Ability to call functions (see “Resolution of Function Calls in MATLAB Generated Code” on page 10-2)

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

MATLAB does not support the following features in generated code:

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

Code Generation for Enumerated Data

- “Enumerated Data Definition for Code Generation” on page 16-2
- “Enumerated Types Supported for Code Generation” on page 16-3
- “When to Use Enumerated Data for Code Generation” on page 16-6
- “Generate Code for Enumerated Data from MATLAB Algorithms” on page 16-7
- “Generate Code for Enumerated Data from MATLAB Function Blocks” on page 16-9
- “Define Enumerated Data for Code Generation” on page 16-10
- “Instantiate Enumerated Types for Code Generation” on page 16-12
- “Operations on Enumerated Data Allowed for Code Generation” on page 16-13
- “Include Enumerated Data in Control Flow Statements” on page 16-16
- “Customize Enumerated Types Based on int32” on page 16-22
- “Customize Enumerated Types Based on Simulink.IntEnumType” on page 16-28
- “Control Names of Enumerated Type Values in Generated Code” on page 16-29
- “Change and Reload Enumerated Data Types” on page 16-31
- “Restrictions on Use of Enumerated Data in `for`-Loops” on page 16-32
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 16-33

Enumerated Data Definition for Code Generation

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

What's Different	More Information
Supports integer-based enumerated types only	"Enumerated Types Supported for Code Generation" on page 16-3
Name of each enumerated data type must be unique	"Naming Enumerated Types for Code Generation" on page 16-11
Each enumerated data type must be defined in a separate file on the MATLAB path	"Define Enumerated Data for Code Generation" on page 16-10 and "How to Generate Code for Enumerated Data" on page 16-7
Restricted set of operations	"Operations on Enumerated Data Allowed for Code Generation" on page 16-13
Restricted use in for-loops	"Restrictions on Use of Enumerated Data in for-Loops" on page 16-32

What's Different	More Information
Supports integer-based enumerated types only	"Enumerated Types Supported in MATLAB Function Blocks"
Each enumerated data type must be defined in a separate file on the MATLAB path	"Define Enumerated Data Types for MATLAB Function Blocks"
Restricted set of operations	"Operations on Enumerated Data"
Restricted use in for-loops	"Restrictions on Use of Enumerated Data in for-Loops" on page 16-32

Enumerated Types Supported for Code Generation

Enumerated Type Based on int32

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

Syntax

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

Example

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

In this example, the statement `classdef(Enumeration) PrimaryColors < int32` means that the enumerated type `PrimaryColors` is based on the built-in type `int32`. As such, `PrimaryColors` inherits the characteristics of the `int32` type, as well as defining its own unique characteristics. For example, `PrimaryColors` is restricted to three enumerated values:

Enumerated Value	Enumerated Name	Underlying Numeric Value
Red(1)	Red	1
Blue(2)	Blue	2
Yellow(4)	Yellow	4

How to Use

Define enumerated data in MATLAB code and compile the source file. For example, to generate C/C++ code from your MATLAB source, you can use

codegen, as described in “Generate Code for Enumerated Data from MATLAB Algorithms” on page 16-7.

Note codegen requires a MATLAB Coder license.

Enumerated Type Based on Simulink.IntEnumType

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

Syntax

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

Example

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

How to Use

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

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

See “About Simulink Enumerations” for more information about enumerated types based on `Simulink.IntEnumType`

When to Use Enumerated Data for Code Generation

You can use enumerated types to represent program states and to control program logic, especially when you need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow more robust error checking than integers or strings

For example, if you mistype the name of an element in the enumerated type, you get a compile-time error that the element does not belong to the set of allowable values.

- Produce more efficient code than strings

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

Generate Code for Enumerated Data from MATLAB Algorithms

Step	Action	How?
1	Define an enumerated data type that inherits from <code>int32</code> .	See “Define Enumerated Data for Code Generation” on page 16-10.
2	Instantiate the enumerated type in your MATLAB algorithm.	See “Instantiate Enumerated Types for Code Generation” on page 16-12.
3	Compile the function with <code>codegen</code> .	See “How to Generate Code for Enumerated Data” on page 16-7.

This workflow requires a MATLAB Coder license.

How to Generate Code for Enumerated Data

Use the command `codegen` to generate MEX, C, or C++ code from the MATLAB algorithm that contains the enumerated data (requires a MATLAB Coder license). Each enumerated data type must be defined on the MATLAB path in a separate file as a class derived from the built-in type `int32`. See “Define Enumerated Data for Code Generation” on page 16-10.

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

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

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

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

You should get the following result:

```
ans =  
    RED
```


Generate Code for Enumerated Data from MATLAB Function Blocks

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

This workflow requires the following licenses:

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

Define Enumerated Data for Code Generation

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

- 1 Create a class definition file.

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

- 2 Enter the class definition as follows:

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

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

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

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

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

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

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

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

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

4 Save the file on the MATLAB path.

The name of the file must match the name of the enumerated data type. The match is case sensitive.

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

For examples of enumerated data type definitions, see “Define Enumerated Data for Code Generation” on page 16-10.

Naming Enumerated Types for Code Generation

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

For example, you cannot name an enumerated data type `mode` because MATLAB for code generation provides a toolbox function of the same name.

For a list of toolbox functions supported for code generation, see “Functions Supported for Code Generation — Alphabetical List” “Functions Supported for Code Generation — Alphabetical List” on page 20-2 “Functions Supported for Code Generation — Alphabetical List”.

Instantiate Enumerated Types for Code Generation

To instantiate an enumerated type for code generation from MATLAB algorithms, use dot notation to specify *ClassName.EnumName*. For an example, see “Include Enumerated Data in Control Flow Statements” on page 16-16.

Operations on Enumerated Data Allowed for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples are based on the definitions of the enumeration type LEDcolor described in “Class Definition: LEDcolor”“Class Definition: LEDcolor” on page 16-16.

Assignment Operator, =

Example	Result
<pre>xon = LEDcolor.GREEN xoff = LEDcolor.RED</pre>	<pre>xon = GREEN xoff = RED</pre>

Relational Operators, < > <= >= == ~=

Example	Result
<pre>xon == xoff</pre>	<pre>ans = 0</pre>
<pre>xon <= xoff</pre>	<pre>ans = 1</pre>
<pre>xon > xoff</pre>	<pre>ans = 0</pre>

Cast Operation

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

Indexing Operation

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

Control Flow Statements: if, switch, while

Statement	Example	Executable Example
if	<pre> if state == sysMode.ON led = LEDcolor.GREEN; else led = LEDcolor.RED; end </pre>	“if Statement with Enumerated Data Types” on page 16-16
switch	<pre> switch button case VCRButton.Stop state = VCRState.Stop; case VCRButton.PlayOrPause state = VCRState.Play; case VCRButton.Next state = VCRState.Forward; case VCRButton.Previous state = VCRState.Rewind; otherwise state = VCRState.Stop; end </pre>	“switch Statement with Enumerated Data Types” on page 16-17
while	<pre> while state ~= State.Ready switch state case State.Standby initialize(); state = State.Boot; case State.Boot boot(); state = State.Ready; end end end </pre>	“while Statement with Enumerated Data Types” on page 16-20

Include Enumerated Data in Control Flow Statements

The following control statements work with enumerated operands in generated code. However, there are restrictions (see “Restrictions on Use of Enumerated Data in for-Loops” on page 16-32).

if Statement with Enumerated Data Types

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

Class Definition: sysMode

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

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

Class Definition: LEDcolor

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

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

MATLAB Function: displayState

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


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

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

Build and Test a MEX Function for displayState

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

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

- 2 Test the function. For example,

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

You should get the following result:

```
ans =

    RED
```

switch Statement with Enumerated Data Types

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

Class Definition: VCRState

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
        Play(2),
        Forward(3),
```

```
        Rewind(4)
    end
end
```

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

Class Definition: VCRButton

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

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

MATLAB Function: VCR

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

```
function s = VCR(button)
    %#codegen

    persistent state

    if isempty(state)
        state = VCRState.Stop;
    end

    switch state
        case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}
            state = handleDefault(button);
        case VCRState.Play
            switch button
```

```

        case VCRButton.PlayOrPause, state = VCRState.Pause;
        otherwise, state = handleDefault(button);
    end
case VCRState.Pause
    switch button
        case VCRButton.PlayOrPause, state = VCRState.Play;
        otherwise, state = handleDefault(button);
    end
end
s = state;

function state = handleDefault(button)
switch button
    case VCRButton.Stop, state = VCRState.Stop;
    case VCRButton.PlayOrPause, state = VCRState.Play;
    case VCRButton.Next, state = VCRState.Forward;
    case VCRButton.Previous, state = VCRState.Rewind;
    otherwise, state = VCRState.Stop;
end
end

```

Build and Test a MEX Function for VCR

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

```
codegen -args {VCRButton.Stop} VCR
```

- 2 Test the function. For example,

```
s = VCR(VCRButton.Stop)
```

You should get the following result:

```
s =

    Stop
```

while Statement with Enumerated Data Types

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

Class Definition: `State`

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

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

MATLAB Function: `Setup`

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

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

    state = initState;

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

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

```
s = state;

function initialize()
% Perform initialization.

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

Build and Test a MEX Executable for Setup

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

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

- 2 Test the function. For example,

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

You should get the following result:

```
s =

    Ready
```

Customize Enumerated Types Based on int32

About Customizing Enumerated Types

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

Enumerated class definitions can include an optional methods section. You can override the following methods to customize the behavior of an enumerated type. To override a method, include a customized version of the method in the methods section in the enumerated class definition. If you do not want to override the inherited methods, omit the methods section.

Method	Description	Default Value Returned or Specified	When to Use
<code>addClassNameToEnumNames</code>	Specifies whether the class name becomes a prefix in the generated code.	<code>true</code> — prefix is used	If you do not want the class name to become a prefix in the generated code, override this method to set the return value to <code>false</code> . See “Control Names of Enumerated Type Values in Generated Code” on page 16-29.
<code>getDefaultValue</code>	Returns the default enumerated value.	<code>''</code>	If you want the default value for the enumerated type to be something other than the first value listed in the enumerated class definition, override this method to specify a default value. See “Specify a Default Enumerated Value” on page 16-24.

Method	Description	Default Value Returned or Specified	When to Use
getHeaderFile	Specifies the file in which the enumerated class is defined for code generation.	''	If you want to use an enumerated class definition that is specified in a custom header file, override this method to return the path to this header file. In this case, the code generation software does not generate the class definition. See “Specify a Header File” on page 16-25

Specify a Default Enumerated Value

The code generation software and related generated code use an enumerated data type’s default value when you provide no other initial value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumerated class definition. To specify a different default value, add your own `getDefaultValue` method to the methods section. The following code shows a shell for the `getDefaultValue` method:

```
function retVal = getDefaultValue()  
% GETDEFAULTVALUE Returns the default enumerated value.  
% This value must be an instance of the enumerated class.  
% If this method is not defined, the first enumerated value is used.  
    retVal = ThisClass.EnumName;  
end
```

To customize this method, provide a value for `ThisClass.EnumName` that specifies the desired default. `ThisClass` must be the name of the class within

which the method exists. EnumName must be the name of an enumerated value defined in that class. For example:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
    methods (Static)
        function y = getDefaultValue()
            y = LEDcolor.RED;
        end
    end
end
```

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

Specify a Header File

To prevent the declaration of an enumerated type from being embedded in the generated code, allowing you to provide the declaration in an external file, include the following method in the enumerated class's methods section:

```
function y = getHeaderFile()
% GETHEADERFILE File where type is defined for generated code.
% If specified, this file is #included where required in the code.
% Otherwise, the type is written out in the generated code.
y = 'filename';
end
```

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

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

The getHeaderFile method does not create the declaration file itself. You must provide a file of the specified name that declares the enumerated data

type. The file can also contain definitions of enumerated types that you do not use in your MATLAB code.

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

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

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

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

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

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

- 3** Generate a library for the function displayState that takes one input of enumerated data type sysMode.

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

codegen generates a C static library with the default name, displayState, and supporting files in the default folder, codegen/lib/displayState.

- 4** Click the *View Report* link.

- 5** In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The header file contains a `#include` statement for the external header file.

```
#include "my_LEDcolor.h"
```

It does not include a declaration for the enumerated class.

Customize Enumerated Types Based on Simulink.IntEnumType

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

Control Names of Enumerated Type Values in Generated Code

This example shows how to control the name of enumerated type values in code generated by MATLAB Coder. (Requires a MATLAB Coder license.) The example uses the enumerated data type definitions and function `displayState` described in “Include Enumerated Data in Control Flow Statements” on page 16-16.

- 1 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

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

`codegen` generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 2 Click the *View Report* link.

- 3 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The report displays the header file containing the enumerated data type definition.

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

The enumerated value names include the class name prefix `LEDcolor_`.

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

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

```
        RED(2),  
    end  
  
    methods(Static)  
        function y=addClassNameToEnumNames()  
            y=false;  
        end  
    end  
end
```

5 Clear existing class instances:

```
clear classes
```

6 Generate code again.

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

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

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

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

For more information, see:

- `codegen`
- “Include Enumerated Data in Control Flow Statements” on page 16-16 for a description of the example function `displayState` and its enumerated type definitions

Change and Reload Enumerated Data Types

You can change the definition of an enumerated data type by editing and saving the file that contains the definition. You do not need to inform MATLAB that a class definition has changed. MATLAB automatically reads the modified definition when you save the file. However, the class definition changes do not take full effect if any class instances (enumerated values) exist that reflect the previous class definition. Such instances might exist in the base workspace or might be cached. The following table explains options for removing instances of an enumerated data type from the base workspace and cache.

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

Restrictions on Use of Enumerated Data in for-Loops

Do not use enumerated data as the loop counter variable in for-loops

To iterate over a range of enumerated data with consecutive values, you can cast the enumerated data to `int32` in the loop counter.

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

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

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

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


Toolbox Functions That Support Enumerated Types for Code Generation

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

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

- squeeze

Code Generation for Function Handles

- “Function Handles Definition for Code Generation” on page 17-2
- “Define and Pass Function Handles for Code Generation” on page 17-3
- “Define and Pass Function Handles for Code Acceleration” on page 17-5
- “Function Handle Limitations for Code Generation” on page 17-7

Function Handles Definition for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently. You can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported for code generation (see “Functions Supported for Code Generation — Alphabetical List” “Functions Supported for Code Generation — Alphabetical List” on page 20-2 “Functions Supported for Code Generation — Alphabetical List”)

Note You cannot define handles that reference extrinsic MATLAB functions.

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

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in “Function Handle Limitations for Code Generation” on page 17-7

Define and Pass Function Handles for Code Generation

The following code example shows how to define and call function handles for code generation. You can copy the example to a MATLAB Function block in Simulink or MATLAB function in Stateflow. To convert this function to a MEX function using `codegen`, uncomment the two calls to the `assert` function, highlighted below:

```
function addval(m)
%#codegen

% Define class and size of primary input m
% Uncomment next two lines to build MEX function with codegen
% assert(isa(m,'double'));
% assert(all (size(m) == [3 3]));

% Pass function handle to addone
% to add one to each element of m
m = map(@addone, m);
disp(m);

% Pass function handle to addtwo
% to add two to each element of m
m = map(@addtwo, m);
disp(m);

function y = map(f,m)
    y = m;
    for i = 1:numel(y)
        y(i) = f(y(i));
    end

function y = addone(u)
    y = u + 1;

function y = addtwo(u)
    y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed

by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

If you have MATLAB Coder, you can use the function `codegen` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

- 1** At the MATLAB command prompt, issue this command:

```
codegen addval
```

- 2** Define and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = zeros(3)
```

- 3** Execute the function by typing this command:

```
addval(m)
```

You should see the following result:

```
0    0    0
0    0    0
0    0    0

1    1    1
1    1    1
1    1    1

3    3    3
3    3    3
3    3    3
```

For more information, see “MEX Function Generation at the Command Line”.

Define and Pass Function Handles for Code Acceleration

The following code example shows how to define and call function handles for code acceleration.

```
function [y1, y2] = addval(m)
%#codegen

disp(m);

% Pass function handle to addone
% to add one to each element of m
y1 = map(@addone, m);
disp(y1);

% Pass function handle to addtwo
% to add two to each element of m
y2 = map(@addtwo, m);
disp(y2);

function y = map(f,m)
    y = m;
    for i = 1:numel(y)
        y(i) = f(y(i));
    end

function y = addone(u)
    y = u + 1;

function y = addtwo(u)
    y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

You can use the function `fiaccel` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

- 1** At the MATLAB command prompt, iDefine and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = fi(magic(3));
```

- 2** Use fiaccel to compile the function to a MEX executable:

```
fiaccel addval -args {m}
```

- 3** Execute the function by typing this command:

```
[y1, y2] = addval_mex(m);
```

```
8      1      6
3      5      7
4      9      2
```

```
        DataTypeMode: Fixed-point: binary point scaling
```

```
        Signedness: Signed
```

```
        WordLength: 16
```

```
        FractionLength: 11
```

```
9      2      7
4      6      8
5     10      3
```

```
        DataTypeMode: Fixed-point: binary point scaling
```

```
        Signedness: Signed
```

```
        WordLength: 16
```

```
        FractionLength: 11
```

```
10     3      8
5      7      9
6     11      4
```

```
        DataTypeMode: Fixed-point: binary point scaling
```

```
        Signedness: Signed
```

```
        WordLength: 16
```

```
        FractionLength: 11
```


Function Handle Limitations for Code Generation

Function handles must be scalar values.

You cannot store function handles in matrices or structures.

You cannot use the same bound variable to reference different function handles.

After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example

```
%Incorrect code
...
x = @plus;
x = @minus;
...
```

This code produces a compilation error.

You cannot pass function handles to or from extrinsic functions.

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions. For more information, see “Declaring MATLAB Functions as Extrinsic Functions” on page 10-12

You cannot pass function handles to or from primary functions.

You cannot pass function handles as inputs to or outputs from primary functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as primary inputs. `plotFcn` attempts to call the function referenced by

the `fhandle` with the input data and plot the results. However, this code generates a compilation error, indicating that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of primary inputs.

You cannot view function handles from the debugger

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

Generating Efficient and Reusable Code

- “Unroll for-Loops” on page 18-2
- “Inline Functions” on page 18-3
- “Eliminate Redundant Copies of Function Inputs” on page 18-4
- “Generate Reusable Code” on page 18-6

Unroll for-Loops

Unrolling for-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. By unrolling short loops with known bounds at compile time, MATLAB generates highly optimized code with no branches.

You can also force loop unrolling for individual functions by wrapping the loop header in a `coder.unroll` function. For more information, see `coder.unroll``coder.unroll``coder.unroll`.

Inline Functions

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

Eliminate Redundant Copies of Function Inputs

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

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

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, MATLAB passes the variable by reference in the generated code instead of redundantly copying the input to a temporary variable. For example, input A above is passed by reference in the generated code because it also acts as an output for function `foo`:

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

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

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

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

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

```
...
/* Function Definitions */
real_T foo2(real_T A, real_T B)
```

```
{  
    return A * B;  
}  
...
```

Generate Reusable Code

With MATLAB, you can generate reusable code in the following ways:

- Write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or MATLAB function library.
- Compile external functions on the MATLAB path and integrate them into generated C code for embedded targets.

See “Resolution of Function Calls in MATLAB Generated Code” on page 10-2.

Common applications include:

- Overriding generated library function with a custom implementation
- Implementing a reusable library on top of standard library functions that can be used with Simulink
- Swapping between different implementations of the same function

Code Generation for MATLAB Structures

- “Structure Definition for Code Generation” on page 19-2
- “Structure Operations Allowed for Code Generation” on page 19-3
- “Define Scalar Structures for Code Generation” on page 19-4
- “Define Arrays of Structures for Code Generation” on page 19-7
- “Make Structures Persistent” on page 19-9
- “Index Substructures and Fields” on page 19-10
- “Assign Values to Structures and Fields” on page 19-12
- “Pass Large Structures as Input Parameters” on page 19-13

Structure Definition for Code Generation

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

What's Different	More Information
Use a restricted set of operations.	"Structure Operations Allowed for Code Generation" on page 19-3
Observe restrictions on properties and values of scalar structures.	"Define Scalar Structures for Code Generation" on page 19-4
Make structures uniform in arrays.	"Define Arrays of Structures for Code Generation" on page 19-7
Reference structure fields individually during indexing.	"Index Substructures and Fields" on page 19-10
Avoid type mismatch when assigning values to structures and fields.	"Assign Values to Structures and Fields" on page 19-12

What's Different	More Information
Use a restricted set of operations.	"Structure Operations Allowed for Code Generation" on page 19-3
Observe restrictions on properties and values of scalar structures.	"Define Scalar Structures for Code Generation" on page 19-4
Make structures uniform in arrays.	"Define Arrays of Structures for Code Generation" on page 19-7
Reference structure fields individually during indexing.	"Index Substructures and Fields"
Avoid type mismatch when assigning values to structures and fields.	"Assign Values to Structures and Fields"

Structure Operations Allowed for Code Generation

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

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

Define Scalar Structures for Code Generation

In this section...

“Restrictions When Using `struct`” on page 19-4

“Restrictions When Defining Scalar Structures by Assignment” on page 19-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 19-4

“Restriction on Adding New Fields After First Use” on page 19-5

Restrictions When Using `struct`

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

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

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

When you create a structure, you must add fields in the same order on each control flow path. For example, the following code generates a compiler

error because it adds the fields of structure `x` in a different order in each `if` statement clause:

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

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

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

Restriction on Adding New Fields After First Use

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

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

For example, consider this code:

```
...
x.c = 10; % Defines structure and creates field c
```

```
y = x; % Reads from structure
x.d = 20; % Generates an error
...
```

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

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

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

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

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

Define Arrays of Structures for Code Generation

In this section...

“Ensuring Consistency of Fields” on page 19-7

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 19-7

“Defining an Array of Structures Using Concatenation” on page 19-8

Ensuring Consistency of Fields

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

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

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

- 1 Create a scalar structure, as described in “Define Scalar Structures for Code Generation” on page 19-4.
- 2 Call repmat, passing the scalar structure and the dimensions of the array.
- 3 Assign values to each structure using standard array indexing and structure dot notation.

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

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

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

Defining an Array of Structures Using Concatenation

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

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

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


Make Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the `isempty` statement, as described in “Define and Initialize Persistent Variables” on page 14-10.

For example, the following function defines structure `X` to be persistent and initializes its fields `a` and `b`:

```
function f(u) %#codegen
persistent X

if isempty(X)
    X.a = 1;
    X.b = 2;
end
```

Index Substructures and Fields

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

Reference substructure field values individually using dot notation

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

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

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

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

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

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

Reference field values individually in structure arrays

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

```
...
```

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

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

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

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 19-7 for more information.

Do not reference fields dynamically

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

Assign Values to Structures and Fields

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

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

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

Do not use field values as constants

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

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

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

Do not assign mxArray's to structures

You cannot assign mxArray's to structure elements; convert mxArray's to known types before code generation (see “Working with mxArray's” on page 10-17).

Pass Large Structures as Input Parameters

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

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

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

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


Functions Supported for Code Generation

- “Functions Supported for Code Generation — Alphabetical List” on page 20-2
- “Functions Supported for Code Generation — Categorical List” on page 20-75

Functions Supported for Code Generation – Alphabetical List

You can generate efficient C/C++ code for a subset of MATLAB and toolbox functions that you call from MATLAB code. In generated code, each supported function has the same name, arguments, and functionality as its MATLAB or toolbox counterparts. However, to generate code for these functions, you must adhere to certain limitations when calling them from your MATLAB source code. These limitations appear in the list below.

To find supported functions by MATLAB category or toolbox, see “Functions Supported for Code Generation — Categorical List”“Functions Supported for Code Generation — Categorical List” on page 20-75“Functions Supported for Code Generation — Categorical List”.

Note For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB” on page 8-3.

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

Function	Product	Remarks/Limitations
acosh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Toolbox	—
all	MATLAB	—
all	Fixed-Point Toolbox	—
and	MATLAB	—
angle	MATLAB	—
any	MATLAB	—
any	Fixed-Point Toolbox	—
asec	MATLAB	—
asecd	MATLAB	—
asech	MATLAB	—

Function	Product	Remarks/Limitations
asin	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
asind	MATLAB	—
asinh	MATLAB	—
assert	MATLAB	<ul style="list-style-type: none"> Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time.
atan	MATLAB	—
atan2	MATLAB	—
atan2d	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
barthannwin	Signal Processing Toolbox™	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>bartlett</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
besselap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter order must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bi2de	Communications System Toolbox™	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
bin2dec	MATLAB	<ul style="list-style-type: none"> Does not match MATLAB when the input is empty.
bitand	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitand	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	Fixed-Point Toolbox	—

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

Function	Product	Remarks/Limitations
bitror	Fixed-Point Toolbox	—
bitset	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitset	Fixed-Point Toolbox	—
bitshift	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitshift	Fixed-Point Toolbox	—
bitsliceget	Fixed-Point Toolbox	—
bitsll	Fixed-Point Toolbox	—
bitsra	Fixed-Point Toolbox	—
bitsrl	Fixed-Point Toolbox	—
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitxor	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
blackman	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
blackmanharris	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
blanks	MATLAB	—
blkdiag	MATLAB	—
bohmanwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
bsxfun	MATLAB	—

Function	Product	Remarks/Limitations
buttap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter order must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
butter	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
buttord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
bwlookup	Image Processing Toolbox™	<ul style="list-style-type: none"> For best results, specify an input image of class <code>logical</code>.
bwmorph	Image Processing Toolbox	<ul style="list-style-type: none"> The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cast	MATLAB	—
cat	MATLAB	—
ceil	MATLAB	—

Function	Product	Remarks/Limitations
<code>ceil</code>	Fixed-Point Toolbox	—
<code>cfirpm</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>char</code>	MATLAB	—
<code>cheb1ap</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.
<code>cheb1ord</code>	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
cheb2ap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
cheb2ord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
chebwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
cheby1	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>cheby2</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
chol	MATLAB	<ul style="list-style-type: none"> When there are two output arguments, either make the input matrix variable-size in both dimensions, or, if the input matrix must be fixed size, copy the input matrix to a variable-size matrix before calling chol. <pre> coder.varsize('B'); B = A; [B,p] = chol(B); </pre>
circshift	MATLAB	—
class	MATLAB	—
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Toolbox	—
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Toolbox	—
conv	MATLAB	—
conv	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is always computed using the SumMode property of the governing fimath.

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
<code>conv2</code>	MATLAB	—
<code>convergent</code>	Fixed-Point Toolbox	—
<code>convn</code>	MATLAB	—
<code>cordicabs</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicangle</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicatan2</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordiccart2pol</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicccexp</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicccos</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicpol2cart</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicrotate</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicsin</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
<code>cordicsincos</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.

Function	Product	Remarks/Limitations
corrcoef	MATLAB	<ul style="list-style-type: none"> Row-vector input is only supported when the first two inputs are vectors and nonscalar.
cos	MATLAB	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	<ul style="list-style-type: none"> If supplied, dim must be a constant.
csc	MATLAB	—
cscd	MATLAB	—
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Toolbox	—
cumprod	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to double first.
cumsum	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to double first.
cumtrapz	MATLAB	—

Function	Product	Remarks/Limitations
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Requires DSP System Toolbox license to generate code. Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>
de2bi	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
deal	MATLAB	—
deblank	MATLAB	<ul style="list-style-type: none"> Supports only inputs from the <code>char</code> class. Input values must be in the range 0-127.

Function	Product	Remarks/Limitations
dec2bin	MATLAB	<ul style="list-style-type: none"> • If input <code>d</code> is <code>double</code>, <code>d</code> must be less than 2^{52}. • If input <code>d</code> is <code>single</code>, <code>d</code> must be less than 2^{23}. • Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 52 for <code>double</code>, 23 for <code>single</code>, 16 for <code>char</code>, 32 for <code>int32</code>, 16 for <code>int16</code>, and so on.
dec2hex	MATLAB	<ul style="list-style-type: none"> • If input <code>d</code> is <code>double</code>, <code>d</code> must be less than 2^{52}. • If input <code>d</code> is <code>single</code>, <code>d</code> must be less than 2^{23}. • Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 13 for <code>double</code>, 6 for <code>single</code>, 4 for <code>char</code>, 8 for <code>int32</code>, 4 for <code>int16</code>, and so on.
deconv	MATLAB	—
de12	MATLAB	—
det	MATLAB	—
detrend	MATLAB	<ul style="list-style-type: none"> • If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> ▪ Be real ▪ Be sorted in ascending order ▪ Restrict elements to integers in the interval $[1, n-2]$, where <code>n</code> is the number of elements in a column of input argument <code>X</code>, or the number of elements in <code>X</code> when <code>X</code> is a row vector

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> ▪ Contain all unique values
diag	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
diag	Fixed-Point Toolbox	<ul style="list-style-type: none"> • If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
diff	MATLAB	<ul style="list-style-type: none"> • If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.
disp	Fixed-Point Toolbox	—
divide	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Complex and imaginary divisors are not supported. • The syntax <code>T.divide(a,b)</code> is not supported.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Toolbox	—
downsample	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs.

Function	Product	Remarks/Limitations
dpss	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
eig	MATLAB	<ul style="list-style-type: none"> QZ algorithm used in all cases, whereas MATLAB might use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors, and the eigenvalues in D might not be in the same order as in MATLAB. With one input, $[V,D] = \text{eig}(A)$, the results will be similar to those obtained using $[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ in MATLAB, except that for code generation, the columns of V are normalized. Options <code>'balance'</code>, <code>'nobalance'</code> are not supported for the standard eigenvalue problem, and <code>'chol'</code> is not supported for the symmetric generalized eigenvalue problem. Outputs are always of complex type.

Function	Product	Remarks/Limitations
ellip	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
ellipap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
ellipke	MATLAB	—
ellipord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
end	Fixed-Point Toolbox	—
epipolarLine	Computer Vision System Toolbox™	—
eps	MATLAB	—
eps	Fixed-Point Toolbox	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	MATLAB	—

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

Function	Product	Remarks/Limitations
fft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftshift	MATLAB	—
fi	Fixed-Point Toolbox	<ul style="list-style-type: none"> Use to create a fixed-point constant or variable. The default constructor syntax without any input arguments is not supported. The syntax <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>. Works for all input values when complete <code>numerictype</code> information of the <code>fi</code> object is provided. Works only for constant input values (value of input must be known at compile time) when complete <code>numerictype</code> information of the <code>fi</code> object is not specified. <code>numerictype</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
filter2	MATLAB	—

Function	Product	Remarks/Limitations
<code>filtfilt</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>fimath</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the <code>fimath</code> object defined in the MATLAB Function dialog in the Model Explorer. Use to create <code>fimath</code> objects in generated code.

Function	Product	Remarks/Limitations
find	MATLAB	<ul style="list-style-type: none"> Issues an error if a variable-sized input becomes a row vector at run time. <hr/> <p>Note This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.
fir1	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
<code>fir2</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>fircls</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none">Requires DSP System Toolbox license to generate code.
fircls1	Signal Processing Toolbox	<ul style="list-style-type: none">Does not support variable-size inputs.All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">Requires DSP System Toolbox license to generate code.
firls	Signal Processing Toolbox	<ul style="list-style-type: none">Does not support variable-size inputs.All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>firpm</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
firpmord	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.
firrcos	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>fix</code>	MATLAB	—
<code>fix</code>	Fixed-Point Toolbox	—
<code>flattopwin</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>flipdim</code>	MATLAB	—
<code>fliplr</code>	MATLAB	—
<code>flipud</code>	MATLAB	—
<code>floor</code>	MATLAB	—
<code>floor</code>	Fixed-Point Toolbox	—
<code>freqspace</code>	MATLAB	—

Function	Product	Remarks/Limitations
freqz	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”. Requires DSP System Toolbox license to generate code.
fspecial	Image Processing Toolbox	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
full	MATLAB	—
fzero	MATLAB	<ul style="list-style-type: none"> The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument. Supports up to three output arguments. Does not support the fourth output argument (the output structure). Only supports the TolX and FunValCheck fields of an options input structure. Ignores all other options in an options input structure. You cannot use the optimset function to create the options structure. Create this structure directly, for example, <pre>opt.TolX = tol; opt.FunValCheck = 'on';</pre> The input structure field names must match exactly.
gamma	MATLAB	—
gammainc	MATLAB	—
gammainv	MATLAB	—

Function	Product	Remarks/Limitations
gaussfir	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
gausswin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccl</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
get	Fixed-Point Toolbox	<ul style="list-style-type: none"> The syntax <code>structure = get(o)</code> is not supported.
getlsb	Fixed-Point Toolbox	—
getmsb	Fixed-Point Toolbox	—
gradient	MATLAB	—
gt	MATLAB	—
gt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
hadamard	MATLAB	—
hamming	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see</p>

Function	Product	Remarks/Limitations
		<p>“Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>hankel</code>	MATLAB	—
<code>hann</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>hex2dec</code>	MATLAB	—
<code>hex2num</code>	MATLAB	<ul style="list-style-type: none"> For <code>n = hex2num(S)</code>, <code>size(S,2) <= length(num2hex(0))</code>
<code>hilb</code>	MATLAB	—
<code>hist</code>	MATLAB	<ul style="list-style-type: none"> Histogram bar plotting not supported; call with at least one output argument. If supplied, the second argument <code>x</code> must be a scalar constant. Inputs must be real.

Function	Product	Remarks/Limitations
histc	MATLAB	<ul style="list-style-type: none"> The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.
horzcat	Fixed-Point Toolbox	—
hypot	MATLAB	—
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
idivide	MATLAB	<ul style="list-style-type: none"> For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> Length of input vector must be a power of 2. Output of <code>ifft</code> block is always complex. Does not support the 'symmetric' option.

Function	Product	Remarks/Limitations
ifft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2. Does not support the 'symmetric' option.
ifftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2. Does not support the 'symmetric' option.
ifftshift	MATLAB	—
imag	MATLAB	—
imag	Fixed-Point Toolbox	—
ind2sub	MATLAB	<ul style="list-style-type: none"> The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.
inf	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
int8, int16, int32	MATLAB	—
int8, int16, int32	Fixed-Point Toolbox	—
integralImage	Computer Vision System Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> Supports only linear and nearest interpolation methods. Does not handle evenly spaced X indices separately. X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.

Function	Product	Remarks/Limitations
interp2	MATLAB	<ul style="list-style-type: none"> • Supports only $5 \leq \text{nargin} \leq 7$. • XI and YI must be the same size. • Supports only 'linear' and 'nearest' methods. • For best performance, supply X and Y as vectors. • When the X or Y inputs are not vectors, interp2 references only the first row of X and first column of Y. Supports "plaid" input for X and Y but does not verify that the input data is "plaid". • X and Y must contain monotonically increasing values. If your application provides monotonically decreasing values, first use flip1r and flipud to change X, Y, and Z to monotonically increasing form before calling interp2.
intersect	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ Inputs must be row vectors. ▪ If a vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, for example zeros(1,0), to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, outputs ia and ib are always column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output c is 0-by-0.

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. Complex inputs must be single or double.
intfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
intmax	MATLAB	—
intmin	MATLAB	—
inv	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	MATLAB	—
ipermute	MATLAB	—
isa	MATLAB	—
iscell	MATLAB	—
ischar	MATLAB	—

Function	Product	Remarks/Limitations
iscolumn	MATLAB	—
iscolumn	Fixed-Point Toolbox	—
isdeployed	MATLAB Compiler	<ul style="list-style-type: none"> • Returns true and false as appropriate for MEX and SIM targets • Returns false for all other targets
isempty	MATLAB	—
isempty	Fixed-Point Toolbox	—
isEpipoleInImage	Computer Vision System Toolbox	—
isequal	MATLAB	—
isequal	Fixed-Point Toolbox	—
isequaln	MATLAB	—
isfi	Fixed-Point Toolbox	—
isfield	MATLAB	<ul style="list-style-type: none"> • Does not support cell input for second argument
isfimath	Fixed-Point Toolbox	—
isfimathlocal	Fixed-Point Toolbox	—
isfinite	MATLAB	—
isfinite	Fixed-Point Toolbox	—
isfloat	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
isinteger	MATLAB	—
isletter	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127
islogical	MATLAB	—
ismatrix	MATLAB	—
ismcc	MATLAB Compiler	<ul style="list-style-type: none"> Returns true and false as appropriate for MEX and SIM targets. Returns false for all other targets.
ismember	MATLAB	<ul style="list-style-type: none"> The second input, S, must be sorted in ascending order. Complex inputs must be single or double.
isnan	MATLAB	—
isnan	Fixed-Point Toolbox	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Toolbox	—
isnumerictype	Fixed-Point Toolbox	—
isprime	MATLAB	<ul style="list-style-type: none"> For double precision input, the maximum value of A is $2^{32} - 1$. For single precision input, the maximum value of A is $2^{24} - 1$.
isreal	MATLAB	—
isreal	Fixed-Point Toolbox	—
isrow	MATLAB	—
isrow	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
isscalar	MATLAB	—
isscalar	Fixed-Point Toolbox	—
assigned	Fixed-Point Toolbox	—
issorted	MATLAB	—
isspace	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127
issparse	MATLAB	—
isstrprop	MATLAB	<ul style="list-style-type: none"> Supports only inputs from char and integer classes. Input values must be in the range 0-127.
isstruct	MATLAB	—
istrellis	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
isvector	MATLAB	—
isvector	Fixed-Point Toolbox	—
kaiser	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>kaiserord</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Computation performed at run time. Requires DSP System Toolbox license to generate code.
<code>kron</code>	MATLAB	—
<code>label2rgb</code>	Image Processing Toolbox	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. <code>map</code> must be an <code>n-by-3</code>, <code>double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning. If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
<code>lcm</code>	MATLAB	—
<code>ldivide</code>	MATLAB	—
<code>le</code>	MATLAB	—

Function	Product	Remarks/Limitations
le	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Toolbox	—
levinson	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
lineToBorderPoints	Computer Vision System Toolbox	—

Function	Product	Remarks/Limitations
linsolve	MATLAB	<ul style="list-style-type: none"> • The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: <ul style="list-style-type: none"> ▪ UT ▪ LT ▪ UHESS = true (the TRANSA can be either true or false) ▪ SYM = true and POSDEF = true <p>All other options are equivalent to using <code>mldivide</code>.</p>
linspace	MATLAB	—
log	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN in generated code when the input value <code>x</code> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Toolbox	—
logspace	MATLAB	—
lower	MATLAB	<ul style="list-style-type: none"> • Supports only char inputs. • Input values must be in the range 0-127.

Function	Product	Remarks/Limitations
lowerbound	Fixed-Point Toolbox	—
lsb	Fixed-Point Toolbox	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lt	MATLAB	—
lt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
lu	MATLAB	—
magic	MATLAB	—
matchFeatures	Computer Vision System Toolbox	—
max	MATLAB	—
max	Fixed-Point Toolbox	—
maxflat	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
mean	MATLAB	—
mean	Fixed-Point Toolbox	—
median	MATLAB	—
median	Fixed-Point Toolbox	—
meshgrid	MATLAB	—
min	MATLAB	—
min	Fixed-Point Toolbox	—
minus	MATLAB	—
minus	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mldivide	MATLAB	—
mod	MATLAB	<ul style="list-style-type: none"> Performs all arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
mode	MATLAB	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array)
mpower	MATLAB	—

Function	Product	Remarks/Limitations
mpower	Fixed-Point Toolbox	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
mpy	Fixed-Point Toolbox	<ul style="list-style-type: none"> When you provide complex inputs to the <code>mpy</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>0n</code>.
mrdivide	MATLAB	—
mrdivide	Fixed-Point Toolbox	—
mtimes	MATLAB	—

Function	Product	Remarks/Limitations
mtimes	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
NaN or nan	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
nargchk	MATLAB	<ul style="list-style-type: none"> Output structure does not include stack information.
nargin	MATLAB	—

Function	Product	Remarks/Limitations
nargout	MATLAB	<ul style="list-style-type: none"> For a function with no output arguments, returns 1 if called without a terminating semicolon. <hr/> <p>Note This behavior also affects extrinsic calls with no terminating semicolon. <code>nargout</code> is 1 for the called function in MATLAB.</p> <hr/>
nargoutchk	MATLAB	<ul style="list-style-type: none"> Output structure does not include stack information.
nchoosek	MATLAB	—
ndgrid	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Toolbox	—
ne	MATLAB	—
ne	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
nearest	Fixed-Point Toolbox	—
nextpow2	MATLAB	—
nnz	MATLAB	—
nonzeros	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—

Function	Product	Remarks/Limitations
<code>null</code>	MATLAB	<ul style="list-style-type: none"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
<code>num2hex</code>	MATLAB	—
<code>numberofelements</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code (works the same as <code>numel</code> for <code>fi</code> objects in generated code).
<code>numel</code>	MATLAB	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code, rather than always returning 1.
<code>numerictype</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numerictype</code> object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a non-fixed-point signal. • Use to create <code>numerictype</code> objects in the generated code.
<code>nuttallwin</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see</p>

Function	Product	Remarks/Limitations
		<p>“Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
ones	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> Might return a different basis than MATLAB
parzenwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
pascal	MATLAB	—
permute	MATLAB	—
permute	Fixed-Point Toolbox	—
pi	MATLAB	—

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

Function	Product	Remarks/Limitations
prod	MATLAB	—
qr	MATLAB	—
quad2d	MATLAB	<ul style="list-style-type: none"> Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.
quadgk	MATLAB	—
quatconj	Aerospace Toolbox	—
quatdivide	Aerospace Toolbox	—
quatinv	Aerospace Toolbox	—
quatmod	Aerospace Toolbox	—
quatmultiply	Aerospace Toolbox	—
quatnorm	Aerospace Toolbox	—
quatnormalize	Aerospace Toolbox	—
rand	MATLAB	—
randi	MATLAB	—
randn	MATLAB	—
randperm	MATLAB	—
range	Fixed-Point Toolbox	—
rank	MATLAB	—

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

Function	Product	Remarks/Limitations
rectwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
reinterpretcast	Fixed-Point Toolbox	—
rem	MATLAB	<ul style="list-style-type: none"> Performs all arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
repmat	MATLAB	—
repmat	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
resample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
rescale	Fixed-Point Toolbox	—
reshape	MATLAB	—
reshape	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
rng	MATLAB	<ul style="list-style-type: none"> For library and executable code generation targets, and for MEX targets when extrinsic calls are disabled, supports only the 'default' input and these generator inputs: <ul style="list-style-type: none"> 'twister' 'v4' 'v5normal' For these targets, the output of s=rng in the generated code differs from the MATLAB output. You cannot return the output of s=rng from the generated code and pass it to rng in MATLAB. For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by rng.
roots	MATLAB	<ul style="list-style-type: none"> Output is always variable size Output is always complex Roots may not be in the same order as MATLAB Roots of poorly conditioned polynomials may not match MATLAB
rosser	MATLAB	—
rot90	MATLAB	—
round	MATLAB	—
round	Fixed-Point Toolbox	—
rsf2csf	MATLAB	—
schur	MATLAB	Might sometimes return a different Schur decomposition in generated code than in MATLAB.

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

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> When rows is specified, outputs ia and ib are always column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output c is 0-by-0. Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. Complex inputs must be single or double.
sfi	Fixed-Point Toolbox	—
sgolay	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for codegen, use coder.Constant. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for fiaccel, use coder.Constant. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
shiftdim	MATLAB	Second argument must be a constant.
sign	MATLAB	—
sign	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
sin	MATLAB	—
sind	MATLAB	—
single	MATLAB	—
single	Fixed-Point Toolbox	—
sinh	MATLAB	—
size	MATLAB	—
size	Fixed-Point Toolbox	—
sort	MATLAB	—
sort	Fixed-Point Toolbox	—
sortrows	MATLAB	—
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time. • Requires DSP System Toolbox license to generate code.
sph2cart	MATLAB	—
squeeze	MATLAB	—
sqrt	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in $\text{complex}(x)$.
sqrt	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sqrtm	MATLAB	—

Function	Product	Remarks/Limitations
std	MATLAB	—
storedInteger	Fixed-Point Toolbox	—
storedIntegerToDouble	Fixed-Point Toolbox	—
str2func	MATLAB	<ul style="list-style-type: none"> String must be constant/known at compile time
strcmp	MATLAB	<ul style="list-style-type: none"> Arguments must be computable at compile time.
strcmpi	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.
strjust	MATLAB	—
strncmp	MATLAB	—
strncmpi	MATLAB	<ul style="list-style-type: none"> Input values from the char class must be in the range 0-127.
strtok	MATLAB	—
strtrim	MATLAB	<ul style="list-style-type: none"> Supports only inputs from the char class. Input values must be in the range 0-127.
struct	MATLAB	—
structfun	MATLAB	<ul style="list-style-type: none"> Does not support the ErrorHandler option. The number of outputs must be less than or equal to three.
sub	Fixed-Point Toolbox	—
sub2ind	MATLAB	<ul style="list-style-type: none"> The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.
subsasgn	Fixed-Point Toolbox	—
subspace	MATLAB	—

Function	Product	Remarks/Limitations
subsref	Fixed-Point Toolbox	—
sum	MATLAB	—
sum	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
svd	MATLAB	Uses a different SVD implementation than MATLAB. As the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
swapbytes	MATLAB	Inheritance of the class of the input to <code>swapbytes</code> in a MATLAB Function block is supported only when the class of the input is <code>double</code> . For non- <code>double</code> inputs, the input port data types must be specified, not inherited.
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
taylorwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p>

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
times	MATLAB	—
times	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. When you provide complex inputs to the <code>times</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
toeplitz	MATLAB	—
trace	MATLAB	—
trapz	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Toolbox	—
triang	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
<code>tril</code>	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
<code>tril</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
<code>triu</code>	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
<code>triu</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
<code>true</code>	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
<code>tukeywin</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Function	Product	Remarks/Limitations
		<p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.
<code>typecast</code>	MATLAB	<ul style="list-style-type: none">• Value of string input argument <code>type</code> must be lower case• You might receive a size error when you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks. To avoid this error, specify the block’s input port data types explicitly.
<code>ufi</code>	Fixed-Point Toolbox	—
<code>uint8, uint16, uint32</code>	MATLAB	—
<code>uint8, uint16, uint32</code>	Fixed-Point Toolbox	—
<code>uminus</code>	MATLAB	—
<code>uminus</code>	Fixed-Point Toolbox	—

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

Function	Product	Remarks/Limitations
unwrap	MATLAB	<ul style="list-style-type: none"> Row vector input is only supported when the first two inputs are vectors and nonscalar Performs all arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors
upfirdn	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code
uplus	MATLAB	—
uplus	Fixed-Point Toolbox	—
upper	MATLAB	<ul style="list-style-type: none"> Supports only char inputs. Input values must be in the range 0-127.

Function	Product	Remarks/Limitations
upperbound	Fixed-Point Toolbox	—
upsample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code>. For example, <code>assert(n<10)</code>
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Toolbox	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Does not support the case where <code>A</code> is a matrix Does not support partial (abbreviated) strings of biased, unbiased, coeff, or none Computation performed at run time. Requires DSP System Toolbox license to generate code
xor	MATLAB	—

Function	Product	Remarks/Limitations
yulewalk	Signal Processing Toolbox	<ul style="list-style-type: none">Does not support variable-size inputs.If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p> <p>Specifying constants</p> <p>To specify a constant input for <code>fiaccel</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line” on page 22-12.</p> <ul style="list-style-type: none">Requires DSP System Toolbox license to generate code.
zeros	MATLAB	<ul style="list-style-type: none">Dimensions must be real, nonnegative, integers.
zp2tf	MATLAB	—

Functions Supported for Code Generation — Categorical List

In this section...

“Aerospace Toolbox Functions” on page 20-76

“Arithmetic Operator Functions” on page 20-76

“Bit-Wise Operation Functions” on page 20-77

“Casting Functions” on page 20-77

“Communications System Toolbox Functions” on page 20-78

“Complex Number Functions” on page 20-78

“Computer Vision System Toolbox Functions” on page 20-79

“Data Type Functions” on page 20-80

“Derivative and Integral Functions” on page 20-80

“Discrete Math Functions” on page 20-81

“Error Handling Functions” on page 20-81

“Exponential Functions” on page 20-81

“Filtering and Convolution Functions” on page 20-82

“Fixed-Point Toolbox Functions” on page 20-82

“Histogram Functions” on page 20-91

“Image Processing Toolbox Functions” on page 20-91

“Input and Output Functions” on page 20-92

“Interpolation and Computational Geometry” on page 20-92

“Linear Algebra” on page 20-92

“Logical Operator Functions” on page 20-93

“MATLAB Compiler Functions” on page 20-93

“Matrix and Array Functions” on page 20-94

“Nonlinear Numerical Methods” on page 20-98

“Polynomial Functions” on page 20-98

In this section...
“Relational Operator Functions” on page 20-98
“Rounding and Remainder Functions” on page 20-99
“Set Functions” on page 20-99
“Signal Processing Functions in MATLAB” on page 20-100
“Signal Processing Toolbox Functions” on page 20-100
“Special Values” on page 20-105
“Specialized Math” on page 20-105
“Statistical Functions” on page 20-106
“String Functions” on page 20-106
“Structure Functions” on page 20-107
“Trigonometric Functions” on page 20-107

Aerospace Toolbox Functions

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

Arithmetic Operator Functions

See Arithmetic Operators for detailed descriptions of the following operator equivalent functions.

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

Bit-Wise Operation Functions

Function	Description
swapbytes	Swap byte ordering

Casting Functions

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

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

Communications System Toolbox Functions

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

Complex Number Functions

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

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

Computer Vision System Toolbox Functions

Function	Description
<code>epipolarLine</code>	Compute epipolar lines for stereo images
<code>estimateFundamentalMatrix</code>	Estimate fundamental matrix from corresponding points in stereo image
<code>estimateUncalibratedRectification</code>	Uncalibrated stereo rectification
<code>extractFeatures</code>	Extract interest point descriptors
<code>integralImage</code>	Compute integral image
<code>isEpipoleInImage</code>	Determine whether image contains epipole
<code>vision.KalmanFilter</code>	Kalman filter for object tracking

Function	Description
lineToBorderPoints	Intersection points of lines in image and image border
matchFeatures	Find matching image features

Data Type Functions

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

Derivative and Integral Functions

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

Discrete Math Functions

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

Error Handling Functions

Function	Description
assert	Generate error when condition is violated
error	Display message and abort function

Exponential Functions

Function	Description
exp	Exponential
expm	Matrix exponential
expm1	Compute $\exp(x) - 1$ accurately for small values of x
factorial	Factorial function
log	Natural logarithm
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
log10	Common (base 10) logarithm
log1p	Compute $\log(1+x)$ accurately for small values of x

Function	Description
nextpow2	Next higher power of 2
nthroot	Real nth root of real numbers
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Filtering and Convolution Functions

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

Fixed-Point Toolbox Functions

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

- `fipref` and quantizer objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given `fi` variable after that variable has been created.
- The boolean value of the `DataTypeMode` and `DataType` properties are not supported.

- For all SumMode property settings other than FullPrecision, the CastBeforeSum property must be set to true.
- The numel function returns the number of elements of `fi` objects in the generated code.
- You can use parallel for (`parfor`) loops in code compiled with `fiaccl`, but those loops are treated like regular `for` loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- All general limitations of C/C++ code generated from MATLAB apply. See “MATLAB Language Features Not Supported for C/C++ Code Generation” for more information.

Function	Remarks/Limitations
<code>abs</code>	N/A
<code>add</code>	N/A
<code>all</code>	N/A
<code>any</code>	N/A
<code>bitand</code>	Not supported for slope-bias scaled <code>fi</code> objects.
<code>bitandreduce</code>	N/A
<code>bitcmp</code>	N/A
<code>bitconcat</code>	N/A
<code>bitget</code>	N/A
<code>bitor</code>	Not supported for slope-bias scaled <code>fi</code> objects.
<code>bitorreduce</code>	N/A
<code>bitreplicate</code>	N/A
<code>bitrol</code>	N/A
<code>bitror</code>	N/A
<code>bitset</code>	N/A
<code>bitshift</code>	N/A

Function	Remarks/Limitations
bitsliceget	N/A
bitsll	N/A
bitsra	N/A
bitsrl	N/A
bitxor	Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	N/A
ceil	N/A
complex	N/A
conj	N/A
conv	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordicccexp	Variable-size signals are not supported.
cordicccos	Variable-size signals are not supported.

Function	Remarks/Limitations
<code>cordicpol2cart</code>	Variable-size signals are not supported.
<code>cordicrotate</code>	Variable-size signals are not supported.
<code>cordicsin</code>	Variable-size signals are not supported.
<code>cordicsincos</code>	Variable-size signals are not supported.
<code>ctranspose</code>	N/A
<code>diag</code>	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
<code>disp</code>	N/A
<code>divide</code>	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
<code>double</code>	N/A
<code>end</code>	N/A
<code>eps</code>	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
<code>eq</code>	Not supported for fixed-point signals with different biases.
<code>fi</code>	<ul style="list-style-type: none"> The default constructor syntax without any input arguments is not supported. If the <code>numericType</code> is not fully specified, the input to <code>fi</code> must be a constant, a <code>fi</code>, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows <code>fi</code> to autoscale its fraction length based on the known data type of the input. <code>numericType</code> object information must be available for nonfixed-point Simulink inputs.

Function	Remarks/Limitations
<code>filter</code>	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
<code>fimath</code>	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer. Use to create <code>fimath</code> objects in the generated code.
<code>fix</code>	N/A
<code>floor</code>	N/A
<code>ge</code>	Not supported for fixed-point signals with different biases.
<code>get</code>	The syntax <code>structure = get(o)</code> is not supported.
<code>getlsb</code>	N/A
<code>getmsb</code>	N/A
<code>gt</code>	Not supported for fixed-point signals with different biases.
<code>horzcat</code>	N/A
<code>imag</code>	N/A
<code>int8, int16, int32</code>	N/A
<code>iscolumn</code>	N/A
<code>isempty</code>	N/A
<code>isequal</code>	N/A
<code>isfi</code>	N/A
<code>isfimath</code>	N/A
<code>isfimathlocal</code>	N/A
<code>isfinite</code>	N/A
<code>isinf</code>	N/A
<code>isnan</code>	N/A
<code>isnumeric</code>	N/A

Function	Remarks/Limitations
<code>isnumericity</code>	N/A
<code>isreal</code>	N/A
<code>isrow</code>	N/A
<code>isscalar</code>	N/A
<code>issigned</code>	N/A
<code>isvector</code>	N/A
<code>le</code>	Not supported for fixed-point signals with different biases.
<code>length</code>	N/A
<code>logical</code>	N/A
<code>lowerbound</code>	N/A
<code>lsb</code>	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
<code>lt</code>	Not supported for fixed-point signals with different biases.
<code>max</code>	N/A
<code>mean</code>	N/A
<code>median</code>	N/A
<code>min</code>	N/A
<code>minus</code>	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.

Function	Remarks/Limitations
mpower	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when the first input, a, is nonscalar. However, when a is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
mpy	When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code> .
mrdivide	N/A
mtimes	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both

Function	Remarks/Limitations
	inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	numberofelements and numel both work the same as MATLAB numel for fi objects in the generated code.
numerictype	<ul style="list-style-type: none"> Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a numerictype object that is populated with the signal's data type and scaling information. Returns the data type when the input is a nonfixed-point signal. Use to create numerictype objects in generated code.
permute	N/A
plus	Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
pow2	N/A
power	The exponent input, k , must be constant; that is, its value must be known at compile time.
range	N/A
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterpretcast	N/A
repmat	N/A
rescale	N/A
reshape	N/A

Function	Remarks/Limitations
round	N/A
sfi	N/A
sign	N/A
single	N/A
size	N/A
sort	N/A
sqrt	<ul style="list-style-type: none"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
times	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, k , must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32	N/A

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

Histogram Functions

Function	Description
hist	Non-graphical histogram
histc	Histogram count

Image Processing Toolbox Functions

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

Function	Remarks/Limitations
bwlookup	For best results, specify an input image of class <code>logical</code> .
bwmorph	The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code> .
fspecial	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
label2rgb	Referring to the standard syntax: <code>RGB = label2rgb(L, map, zerocolor, order)</code> <ul style="list-style-type: none"> Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. <code>map</code> must be an <code>n</code>-by-3, <code>double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function.

Function	Remarks/Limitations
	<ul style="list-style-type: none">• If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning.• If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.

Input and Output Functions

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

Interpolation and Computational Geometry

Function	Description
<code>cart2pol</code>	Transform Cartesian coordinates to polar or cylindrical
<code>cart2sph</code>	Transform Cartesian coordinates to spherical
<code>interp1</code>	1-D data interpolation (table lookup)
<code>interp2</code>	2-D data interpolation (table lookup)
<code>meshgrid</code>	Generate X and Y arrays for 3-D plots
<code>pol2cart</code>	Transform polar or cylindrical coordinates to Cartesian
<code>sph2cart</code>	Transform spherical coordinates to Cartesian

Linear Algebra

Function	Description
<code>linsolve</code>	Solve linear system of equations
<code>null</code>	Null space

Function	Description
orth	Range space of matrix
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtn	Matrix square root

Logical Operator Functions

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

MATLAB Compiler Functions

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

Matrix and Array Functions

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

Function	Description
<code>flipud</code>	Flip matrix up to down
<code>full</code>	Convert sparse matrix to full matrix
<code>hadamard</code>	Hadamard matrix
<code>hankel</code>	Hankel matrix
<code>hilb</code>	Hilbert matrix
<code>ind2sub</code>	Subscripts from linear index
<code>inv</code>	Inverse of a square matrix
<code>invhilb</code>	Inverse of Hilbert matrix
<code>ipermute</code>	Inverse permute dimensions of array
<code>iscolumn</code>	True if input is a column vector
<code>isempty</code>	Determine whether array is empty
<code>isequal</code>	Test arrays for equality
<code>isequaln</code>	Test arrays for equality, treating NaNs as equal
<code>isfinite</code>	Detect finite elements of an array
<code>isfloat</code>	Determine if input is floating-point array
<code>isinf</code>	Detect infinite elements of an array
<code>isinteger</code>	Determine if input is integer array
<code>islogical</code>	Determine if input is logical array
<code>ismatrix</code>	True if input is a matrix
<code>isnan</code>	Detect NaN elements of an array
<code>isrow</code>	True if input is a row vector
<code>issparse</code>	Determine whether input is sparse
<code>isvector</code>	Determine whether input is vector
<code>kron</code>	Kronecker tensor product
<code>length</code>	Return the length of a matrix
<code>linspace</code>	Generate linearly spaced vectors

Function	Description
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndgrid	Generate arrays for N-D functions and interpolation
ndims	Number of dimensions
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
norm	Vector and matrix norms
normest	2-norm estimate
numel	Number of elements in array or subscripted array
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed random numbers
randperm	Random permutation
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array

Function	Description
reshape	Reshape one array into the dimensions of another
rng	Control random number generation
rosser	Classic symmetric eigenvalue test problem
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Nonlinear Numerical Methods

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

Polynomial Functions

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

Relational Operator Functions

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

Rounding and Remainder Functions

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

Set Functions

Function	Description
<code>intersect</code>	Find set intersection of two vectors
<code>ismember</code>	Array elements that are members of set
<code>issorted</code>	Determine whether set elements are in sorted order
<code>setdiff</code>	Find set difference of two vectors
<code>setxor</code>	Find set exclusive OR of two vectors
<code>union</code>	Find set union of two vectors
<code>unique</code>	Find unique elements of vector

Signal Processing Functions in MATLAB

Function	Description
chol	Cholesky factorization
conv	Convolution and polynomial multiplication
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
filter	Filter a data sequence using a digital filter that works for both real and complex inputs
freqspace	Frequency spacing for frequency response
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse discrete Fourier transform shift
svd	Singular value decomposition
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

Signal Processing Toolbox Functions

All of these functions require a DSP System Toolbox license to generate code. These functions do not support variable-size inputs, you must define the size and type of the function inputs. For more information, see “Specifying Inputs in Code Generation from MATLAB ”.

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

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

Function	Remarks/Limitations
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
dct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	All inputs must be constants. Expressions or variables are allowed if their values do not change.
flattpwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”.
gaussfir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
gausswin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hamming	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hann	All inputs must be constant. Expressions or variables are allowed if their values do not change.
idct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
intfilt	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiser	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiserord	—
levinson	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
maxflat	All inputs must be constant. Expressions or variables are allowed if their values do not change.
nutallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	<ul style="list-style-type: none"> Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. Variable-size inputs are not supported.
upsample	Either declare input n as constant, or use the <code>assert</code> function in the calling function to set upper bounds for n. For example, <code>assert(n<10)</code>
xcorr	—
yulewalk	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Special Values

Symbol	Description
eps	Floating-point relative accuracy
inf	IEEE® arithmetic representation for positive infinity
intmax	Largest possible value of specified integer type
intmin	Smallest possible value of specified integer type
NaN or nan	Not a number
pi	Ratio of the circumference to the diameter for a circle
realmax	Largest positive floating-point number
realmin	Smallest positive floating-point number

Specialized Math

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

Statistical Functions

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

String Functions

Function	Description
bin2dec	Convert binary number string to decimal number
bitmax	Maximum double-precision floating-point integer
blanks	Create string of blank characters
char	Create character array (string)
deblank	Strip trailing blanks from end of string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
ischar	True for character array (string)
isletter	Array elements that are alphabetic letters
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
lower	Convert string to lowercase
num2hex	Convert singles and doubles to IEEE hexadecimal strings

Function	Description
strcmp	Compare strings (case sensitive)
strncmpi	Compare strings (case insensitive)
strjust	Justify character array
strncmp	Compare first n characters of strings (case sensitive)
strncmpi	Compare first n characters of strings (case insensitive)
strtok	Selected parts of string
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

Structure Functions

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

Trigonometric Functions

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees

Function	Description
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atan2d	Four-quadrant inverse tangent, result in degrees
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine
cosd	Cosine; result in degrees
cosh	Hyperbolic cosine
cot	Cotangent; result in radians
cotd	Cotangent; result in degrees
coth	Hyperbolic cotangent
csc	Cosecant; result in radians
cscd	Cosecant; result in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant; result in radians
secd	Secant; result in degrees
sech	Hyperbolic secant
sin	Sine
sind	Sine; result in degrees

Function	Description
sinh	Hyperbolic sine
tan	Tangent
tand	Tangent; result in degrees
tanh	Hyperbolic tangent

Code Generation for Variable-Size Data

- “What Is Variable-Size Data?” on page 21-2
- “Variable-Size Data Definition for Code Generation” on page 21-3
- “Bounded Versus Unbounded Variable-Size Data” on page 21-4
- “Control Memory Allocation of Variable-Size Data” on page 21-5
- “Specify Variable-Size Data Without Dynamic Memory Allocation” on page 21-6
- “Variable-Size Data in Code Generation Reports” on page 21-10
- “Define Variable-Size Data for Code Generation” on page 21-12
- “C Code Interface for Arrays” on page 21-19
- “Troubleshooting Issues with Variable-Size Data” on page 21-23
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 21-27
- “Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation” on page 21-35

What Is Variable-Size Data?

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

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

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

Variable-Size Data Definition for Code Generation

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

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

For fixed-size data, the shape is the same as the size returned in MATLAB.

For example, if `size(A) == [4 5]`, the shape of variable A is 4 x 5.

For variable-size data, the shape can be abstract. That is, one or more dimensions can be unknown (such as 4x? or ?x?).

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

For more information, see “Bounded Versus Unbounded Variable-Size Data” on page 21-4

Bounded Versus Unbounded Variable-Size Data

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

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

You can control the memory allocation of variable-size data. For more information, see “Control Memory Allocation of Variable-Size Data” on page 21-5.

Control Memory Allocation of Variable-Size Data

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

Dynamic memory allocation is an expensive operation; the performance cost may be too high for small data sets. If you use small variable-size data sets or data that does not change size at run time, disable dynamic memory allocation. See “Control Dynamic Memory Allocation” “Control Dynamic Memory Allocation” on page 8-93.

You can control memory allocation globally for your application by modifying the dynamic memory allocation threshold. See “Generate Code for a MATLAB Function That Expands a Vector in a Loop”. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 21-6.

Specify Variable-Size Data Without Dynamic Memory Allocation

In this section...
“Fixing Upper Bounds Errors” on page 21-6
“Specifying Upper Bounds for Variable-Size Data” on page 21-6

Fixing Upper Bounds Errors

If MATLAB cannot determine or compute the upper bound, you must specify an upper bound. See “Specifying Upper Bounds for Variable-Size Data” on page 21-6 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 21-25

Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 21-6
- “Specifying Upper Bounds on the Command Line for Variable-Size Inputs” on page 21-6
- “Specifying Unknown Upper Bounds for Variable-Size Inputs” on page 21-7
- “Specifying Upper Bounds for Local Variable-Size Data” on page 21-7
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 21-8

When to Specify Upper Bounds for Variable-Size Data

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

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

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

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

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

Specifying Unknown Upper Bounds for Variable-Size Inputs

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

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

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

Specifying Upper Bounds for Local Variable-Size Data

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

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

Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data. Use the `assert` function with relational operators to constrain the value of variables that specify the dimensions of variable-size data. For example:

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

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

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

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

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

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

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

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

Using a Matrix Constructor with Nonconstant Dimensions

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

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

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

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

Variable-Size Data in Code Generation Reports

In this section...
“What Reports Tell You About Size” on page 21-10
“How Size Appears in Code Generation Reports” on page 21-11
“How to Generate a Code Generation Report” on page 21-11

What Reports Tell You About Size

Code generation reports:

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

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

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

How Size Appears in Code Generation Reports

:? means variable size, unknown upper bound

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

No colon prefix (:) means fixed size

:100 means variable size, upper bound = 100

Variable	Type	Size
y	Output	1 x 10*

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

How to Generate a Code Generation Report

Add the -report option to your codegen command.

Add the -report option to your fiaccel command.

Define Variable-Size Data for Code Generation

In this section...
“When to Define Variable-Size Data Explicitly” on page 21-12
“Using a Matrix Constructor with Nonconstant Dimensions” on page 21-13
“Inferring Variable Size from Multiple Assignments” on page 21-13
“Defining Variable-Size Data Explicitly Using coder.varsize” on page 21-14

When to Define Variable-Size Data Explicitly

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

Method	See
Assign the data from a variable-size matrix constructor such as <ul style="list-style-type: none">oneszerosrepmat	“Using a Matrix Constructor with Nonconstant Dimensions” on page 21-13
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Inferring Variable Size from Multiple Assignments” on page 21-13
Define all instances of a variable to be variable sized	“Defining Variable-Size Data Explicitly Using coder.varsize” on page 21-14

Using a Matrix Constructor with Nonconstant Dimensions

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

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

Inferring Variable Size from Multiple Assignments

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

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

Inferring Upper Bounds from Multiple Definitions with Different Shapes

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

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

- First dimension is fixed at size 3

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

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

Variable	Type	Size
y	Output	3 x :4 x :5

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

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

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

Variable	Type	Size
y	Output	3 x :? x :?

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

Use the function `coder.varsize` to define one or more variables or structure fields as variable-size data. Optionally, you can also specify which dimensions vary along with their upper bounds (see “Specifying Which Dimensions Vary” on page 21-15). For example:

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

```
coder.varsize('B', [64 64]);
```
- Define `B` as a variable-size matrix:

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

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

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

Specifying Which Dimensions Vary

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

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

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size; dimensions that correspond to ones or `true` vary in size. `coder.varsize` usually treats dimensions of size 1 as fixed (see “Defining Variable-Size Matrices with Singleton Dimensions” on page 21-16).

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

Allowing a Variable to Grow After Defining Fixed Dimensions

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

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

```
else
    Y = zeros(5,5);
end
```

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

Defining Variable-Size Matrices with Singleton Dimensions

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

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

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

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

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

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

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

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

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

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

Defining Variable-Size Structure Fields

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

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

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

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

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

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

Here are other examples:

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

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

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

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

C Code Interface for Arrays

In this section...

“C Code Interface for Statically Allocated Arrays” on page 21-19

“C Code Interface for Dynamically Allocated Arrays” on page 21-20

“Utility Functions for Creating emxArray Data Structures” on page 21-21

C Code Interface for Statically Allocated Arrays

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

For example, consider the MATLAB function `uniquetol`:

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

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

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

In the generated code, the function declaration is:

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

There are two pieces of information about A:

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

C Code Interface for Dynamically Allocated Arrays

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

emxArray Structure Definition

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

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

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

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

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


C Code Interface for Structure Fields

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

Utility Functions for Creating `emxArray` Data Structures

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

Function	Arguments	Description
emxArray_<baseTypeName> *emxCreateWrapper_<baseTypeName> (...)	*data num_rows num_cols	Creates a new 2-dimensional emxArray, but does not allocate it on the heap. Instead uses memory provided by the user and sets canFreeData to false so it never inadvertently free user memory, such as the stack.
emxArray_<baseTypeName> *emxCreateWrapperND_<baseTypeName> (...)	*data numDimensions *size	Same as emxCreateWrapper, except it creates a new N-dimensional emxArray.
emxArray_<baseTypeName> *emxCreate_<baseTypeName> (...)	num_rows num_cols	Creates a new two-dimensional emxArray on the heap, initialized to zero. All data elements have the data type specified by <i>baseTypeName</i> .
emxArray_<baseTypeName> *emxCreateND_<baseTypeName> (...)	numDimensions *size	Same as emxCreate, except it creates a new N-dimensional emxArray on the heap.
emxArray_<baseTypeName> *emxDestroyArray_<baseTypeName> (...)	*emxArray	Frees dynamic memory allocated by *emxCreate and *emxCreateND functions.

Troubleshooting Issues with Variable-Size Data

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 21-23

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 21-25

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

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

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

Compiling this function produces this error:

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

There are several ways to fix this error:

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

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

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

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

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

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

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

Empty Matrix Reshaped to Match Variable-Size Specification

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

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

```

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

```

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

Performing Binary Operations on Fixed and Variable-Size Operands

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

```

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

```

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

```

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

```

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

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

There are several ways to fix the problem:

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

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

To fix the problem, 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 21-27

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 21-29

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 21-30

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 21-31

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 21-32

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 21-34

Incompatibility with MATLAB for Scalar Expansion

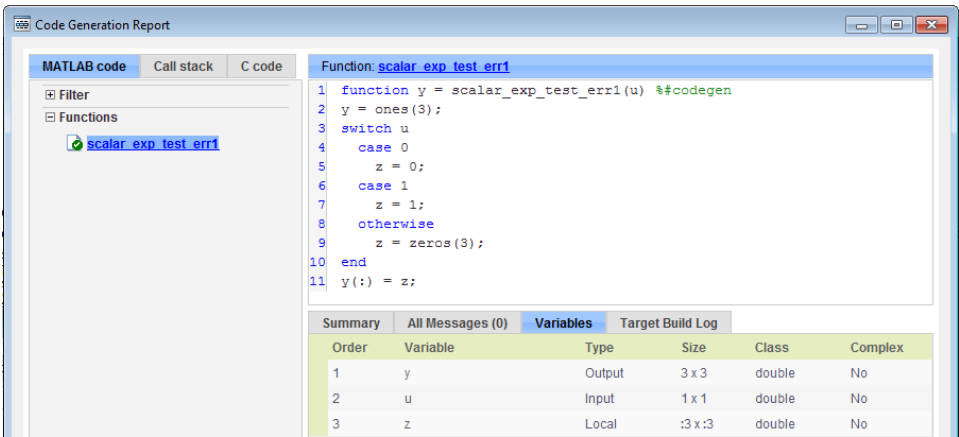
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size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

During code generation, the standard MATLAB scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Instead, it generates a size mismatch error at run time for MEX functions. For non-MEX builds, there is no run-time error checking; the generated code will have unspecified behavior.

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

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

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



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

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

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


Workaround

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

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

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

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

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

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

Workarounds

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

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

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

- Rewrite `size(A)`:

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

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

Incompatibility with MATLAB in Determining Size of Empty Arrays

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

For example, consider the following code:

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

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

Workaround

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

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

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

Incompatibility with MATLAB in Vector-Vector Indexing

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

In this situation, if the code generation software detects that both `A` and `B` are vectors at compile time, it applies the special rule and gives the same result as MATLAB. However, if either `A` or `B` is a variable-size matrix (has shape `?x?`) at compile time, the code generation software applies only the general indexing rule. Then, if both `A` and `B` become vectors at run time, the code generation software reports a run-time error when you run the MEX function. For non-MEX builds, there is no run-time error checking; the generated code

will have unspecified behavior. It is best practice to generate and test a MEX function before generating C code.

Workaround

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

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

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

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

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

- Initialization of the following style:

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

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

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

```
M=zeros(1,10);
```

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

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

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

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

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

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

The following limitations apply to matrix indexing operations for code generation:

- Initialization of the following style:

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

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

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

```
M=zeros(1,10);
```

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

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

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

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

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

Dynamic Memory Allocation Not Supported for MATLAB Function Blocks

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

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation

In this section...

“Common Restrictions” on page 21-35

“Toolbox Functions with Variable Sizing Restrictions” on page 21-36

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Variable Sizing Restrictions” on page 21-36.

Variable-length vector restriction

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

Automatic dimension restriction

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

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

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

Array-to-vector restriction

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

Array-to-scalar restriction

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

Toolbox Functions with Variable Sizing Restrictions

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

Function	Restrictions with Variable-Size Data
all	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
any	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
bsxfun	<ul style="list-style-type: none">• Dimensions expand only where one input array or the other has a fixed length of 1.
cat	<ul style="list-style-type: none">• Dimension argument must be a constant.• An error occurs if variable-size inputs are empty at run time.

Function	Restrictions with Variable-Size Data
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 21-35. • Input vectors must have the same orientation, either both row vectors or both column vectors.
cov	<ul style="list-style-type: none"> • For <code>cov(X)</code>, see “Array-to-vector restriction” on page 21-36.
cross	<ul style="list-style-type: none"> • Variable-size array inputs that become vectors at run time must have the same orientation.
deconv	<ul style="list-style-type: none"> • For both arguments, see “Variable-length vector restriction” on page 21-35.
detrend	<ul style="list-style-type: none"> • For first argument for row vectors only, see “Array-to-vector restriction” on page 21-36 .
diag	<ul style="list-style-type: none"> • See “Array-to-vector restriction” on page 21-36 .
diff	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35. • Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, <code>diff(x,2,1)</code> works but <code>diff(x,5,1)</code> generates a run-time error.
fft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35.

Function	Restrictions with Variable-Size Data
filter	<ul style="list-style-type: none"> For first and second arguments, see “Variable-length vector restriction” on page 21-35. See “Automatic dimension restriction” on page 21-35.
hist	<ul style="list-style-type: none"> For second argument, see “Variable-length vector restriction” on page 21-35. For second input argument, see “Array-to-scalar restriction” on page 21-36.
histc	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 21-35.
ifft	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 21-35.
ind2sub	<ul style="list-style-type: none"> First input (the size vector input) must be fixed size.
interp1	<ul style="list-style-type: none"> For the Y input and xi input, see “Array-to-vector restriction” on page 21-36. Y input can become a column vector dynamically. A run-time error occurs if Y input is not a variable-length vector and becomes a row vector at run time.
ipermute	<ul style="list-style-type: none"> Order input must be fixed size.
issorted	<ul style="list-style-type: none"> For optional rows input, see “Variable-length vector restriction” on page 21-35.

Function	Restrictions with Variable-Size Data
<code>magic</code>	<ul style="list-style-type: none">• Argument must be a constant.• Output can be fixed-size matrices only.
<code>max</code>	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.
<code>mean</code>	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
<code>median</code>	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
<code>min</code>	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.
<code>mode</code>	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
<code>mtimes</code>	<ul style="list-style-type: none"> When an input is variable sized, MATLAB determines whether to generate code for a general matrix*matrix multiplication or a scalar*matrix multiplication, based on whether one of the arguments is a fixed-size scalar. If neither argument is a fixed-size scalar, the inner dimensions of the two arguments must agree even if a variable-size matrix input happens to be a scalar at run time.
<code>nchoosek</code>	<ul style="list-style-type: none"> Inputs must be fixed sized. Second input must be a constant for static allocation. If you enable dynamic allocation, second input can be a variable. You cannot create a variable-size array by passing in a variable <code>k</code> unless you enable dynamic allocation.
<code>permute</code>	<ul style="list-style-type: none"> Order input must be fixed size.
<code>planerot</code>	<ul style="list-style-type: none"> Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.
<code>poly</code>	<ul style="list-style-type: none"> See “Variable-length vector restriction” on page 21-35.
<code>polyfit</code>	<ul style="list-style-type: none"> For first and second arguments, see “Variable-length vector restriction” on page 21-35.

Function	Restrictions with Variable-Size Data
<code>prod</code>	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
<code>rand</code>	<ul style="list-style-type: none"> • For an upper-bounded variable <code>N</code>, <code>rand(1,N)</code> produces a variable-length vector of <code>1x:M</code> where <code>M</code> is the upper bound on <code>N</code>. • For an upper-bounded variable <code>N</code>, <code>rand([1,N])</code> may produce a variable-length vector of <code>:1x:M</code> where <code>M</code> is the upper bound on <code>N</code>.
<code>randn</code>	<ul style="list-style-type: none"> • For an upper-bounded variable <code>N</code>, <code>randn(1,N)</code> produces a variable-length vector of <code>1x:M</code> where <code>M</code> is the upper bound on <code>N</code>. • For an upper-bounded variable <code>N</code>, <code>randn([1,N])</code> may produce a variable-length vector of <code>:1x:M</code> where <code>M</code> is the upper bound on <code>N</code>.
<code>reshape</code>	<ul style="list-style-type: none"> • When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.
<code>roots</code>	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 21-35.

Function	Restrictions with Variable-Size Data
shiftdim	<ul style="list-style-type: none"> • If you do not supply the second argument, the number of shifts is determined at compilation time by the upper bounds of the dimension sizes. Consequently, at run time the number of shifts is always constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must always have the same number of dimensions when you supply a positive number of shifts.
std	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
sub2ind	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
sum	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
trapz	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 21-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
typecast	<ul style="list-style-type: none">• See “Variable-length vector restriction” on page 21-35 on first argument.
var	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 21-35.• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Primary Functions

Primary Function Input Specification

In this section...

- “When to Specify Input Properties” on page 22-2
- “Why You Must Specify Input Properties” on page 22-2
- “Properties to Specify” on page 22-3
- “Rules for Specifying Properties of Primary Inputs” on page 22-8
- “Methods for Defining Properties of Primary Inputs” on page 22-8
- “Define Input Properties by Example at the Command Line” on page 22-9
- “Specify Constant Inputs at the Command Line” on page 22-12
- “Specify Variable-Size Inputs at the Command Line” on page 22-14

When to Specify Input Properties

If you supply a test bench for your MATLAB algorithm, you do not need to manually specify the primary function inputs. The HDL Coder software uses the test bench to infer the data types.

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB CoderHDL Coder Fixed-Point Toolbox must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, MATLAB CoderHDL CoderFixed-Point Toolbox 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 CoderHDL CoderFixed-Point Toolbox. If your primary function has no input parameters, MATLAB CoderHDL CoderFixed-Point Toolbox 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.

If you use the tilde (~) character to specify unused function inputs in an HDL Coder project, and you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For...	Specify properties...				
	Class	Size	Complexity	numericity	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 generation software treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition: <ul style="list-style-type: none"> • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify numericity, and fimath. 				
All other inputs	✓	✓	✓		

For...	Specify properties...				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
All other inputs	✓	✓	✓		

The following data types are not supported for primary function inputs, although you can use them within the primary function:

- structure
- matrix

Variable-size data is not supported in the test bench or the primary function.

Default Property Values

MATLAB CoderHDL CoderFixed-Point Toolbox 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

Property	Default
class	double
size	scalar
complexity	real

Property	Default
numerictype	No default
fimath	hdlfimath

Specifying Default Values for Structure Fields. In most cases, when you don't explicitly specify values for properties, MATLAB CoderHDL CoderFixed-Point Toolbox 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 22-25 and “Specifying Class and Size of Structure Array” on page 22-26.

Specifying Default fimath Values for MEX Functions. MEX functions generated with MATLAB CoderFixed-Point Toolbox use the default `fimath` value in effect at compile time. If you do not specify a default `fimath` value, MATLAB CoderFixed-Point Toolbox 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” on page 4-22.

When running MEX functions that depend on the default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time warning, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` relies on the default `fimath` object in effect at compile

time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```
codegen test
% codegen generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex

ans =

      0

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

Now create a local MATLAB `fimath` value. so you no longer use the default setting:

```
F = fimath('RoundingMethod','Floor');
```

Finally, clear the MEX function from memory and rerun it:

```
clear test_mex
test_mex
```

The mismatch is detected and causes an error:

```
??? This function was generated with a different default
fimath than the current default.
```

```
Error in ==> test_mex
```

Supported Classes

The following table presents the class names supported by MATLAB CoderHDL CoderFixed-Point Toolbox.

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
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

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
single	Single-precision floating-point or fixed-point number array

Class Name	Description
double	Double-precision floating-point or fixed-point number array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules.

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (fi), you must specify the input `numericType` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
	<ul style="list-style-type: none">• If you are working in a MATLAB CoderHDL Coder project, easy to use• Does not alter original MATLAB code• MATLAB CoderHDL Coder saves the definitions in the project file	<ul style="list-style-type: none">• Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties by Example at the Command Line” on page 22-9	<ul style="list-style-type: none">• Easy to use• Does not alter original MATLAB code	<ul style="list-style-type: none">• Must be specified at the command line every time you invoke <code>codegen</code> (unless you use a script)

Method	Advantages	Disadvantages
<div> <div> Note If you define input properties programmatically in the MATLAB file, you cannot use this method </div> </div>	<ul style="list-style-type: none"> Designed for prototyping a function that has a small number of primary inputs 	<ul style="list-style-type: none"> Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties Programmatically in the MATLAB File” on page 22-16	<ul style="list-style-type: none"> Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB CoderHDL Coder Provides documentation of property specifications in the MATLAB code Efficient for specifying memory-intensive inputs such as large structures 	<ul style="list-style-type: none"> Uses complex syntax MATLAB CoderHDL 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 22-10
- “Rules for Using the -args Option” on page 22-10
- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 22-10
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 22-11

Command Line Option -args

The `codegen` function provides a command-line option `-args` for specifying the properties of primary (entry-point) function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you generate code for the MATLAB function with `codegen`. If you have a test function or script that calls the entry-point MATLAB function with the required types, you can use `coder.getArgTypes` to determine the types of the function inputs. `coder.getArgTypes` returns a cell array of `coder.Type` objects that you can pass to `codegen` using the `-args` option. For more information, see the `coder.getArgTypes` function reference information.

See “Specifying General Properties of Primary Inputs” on page 22-24 for `codegen`.

Rules for Using the -args Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Note If you specify an empty cell array with the `-args` option, `codegen` interprets this to mean that the function takes no inputs; a compile-time error occurs if the function does have inputs.

Specifying Properties of Primary Inputs by Example at the Command Line

Consider a MATLAB function that adds its two inputs:

```
function y = mcf(u,v)
%#codegen
```

```
y = u + v;
```

The following examples show how to specify different properties of the primary inputs *u* and *v* by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

```
codegen mcf -args {0,0}
```

- Use a literal cell array of constants to specify that input *u* is an unsigned 16-bit, 1-by-4 vector and input *v* is a scalar double:

```
codegen mcf -args {zeros(1,4,'uint16'),0}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = uint8([1;2;3;4])
b = uint8([5;6;7;8])
ex = {a,b}
codegen mcf -args ex
```

Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line

To generate a MEX function or C/C++ code for fixed-point MATLAB code, you must install Fixed-Point Toolbox 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 on the MATLAB command line, follow these steps:

- 1 Define the `numericType` properties for *x*, as in this example:

```
T = numericType('WordLength',32,...
```

```
'FractionLength',23,...  
'Signed',true);
```

- 2** Define the `fimath` properties for `x`, as in this 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 you just defined, as in this 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, as in this example:

```
codegen sqrtfi -args myeg;
```

Specify Constant Inputs at the Command Line

In cases where you know your primary inputs will not change at run time, it is more efficient to specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using the `-args` command-line option with a `coder.Constant` object, as in this example:

```
-args {coder.Constant(constant_input)}
```

This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs

`codegen` compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function

signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function `identity` which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, at the MATLAB prompt, type the following command:

```
codegen identity -args {coder.Constant(42)}
```

To run the MATLAB function, supply the constant argument:

```
identity(42)
```

You get the following result:

```
ans =
```

```
    42
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Specifying a Structure as a Constant Input

Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = zeros(p.rows,p.cols) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
codegen rowcol -args {0,coder.Constant(tmp)}
```

Specify Variable-Size Inputs at the Command Line

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. You can define inputs to have one or more variable-size dimensions — and specify their upper bounds — using the `-args` option and `coder.typeof` function:

```
-args {coder.typeof(example_value, size_vector, variable_dims)}
```

Specifies a variable-size input with:

- Same class and complexity as *example_value*
- Same size and upper bounds as *size_vector*
- Variable dimensions specified by *variable_dims*

When you enable dynamic memory allocation, you can specify `Inf` in the size vector for dimensions with unknown upper bounds at compile time.

When *variable_dims* is a scalar, it is applied to all the dimensions, with the following exceptions:

- If the dimension is 1 or 0, which are fixed.
- If the dimension is unbounded, which is always variable size.

For more information, see `coder.typeof` and `.`

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}

```

Define Input Properties Programmatically in the MATLAB File

With MATLAB Coder, you use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

In this section...

- “How to Use `assert` with MATLAB® Coder™” on page 22-16
- “Rules for Using `assert` Function” on page 22-23
- “Specifying General Properties of Primary Inputs” on page 22-24
- “Specifying Properties of Primary Fixed-Point Inputs” on page 22-25
- “Specifying Class and Size of Scalar Structure” on page 22-25
- “Specifying Class and Size of Structure Array” on page 22-26

How to Use `assert` with MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

You must use one of the following methods when specifying input properties using the `assert` function. Use the exact syntax that is provided; do not modify it.

- “Specify Any Class” on page 22-17
- “Specify `fi` Class” on page 22-17
- “Specify Structure Class” on page 22-18
- “Specify Fixed Size” on page 22-18
- “Specify Scalar Size” on page 22-19
- “Specify Upper Bounds for Variable-Size Inputs” on page 22-19
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 22-19
- “Specify Size of Individual Dimensions” on page 22-20
- “Specify Real Input” on page 22-21

- “Specify Complex Input” on page 22-21
- “Specify numerictype of Fixed-Point Input” on page 22-21
- “Specify fimath of Fixed-Point Input” on page 22-22
- “Specify Multiple Properties of Input” on page 22-22

Specify Any Class

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input *U* to a 32-bit signed integer, call:

```
...
assert(isa(U,'int32'));
...
```

If you set the class of an input parameter to *fi*, you must also set its *numerictype*, see “Specify numerictype of Fixed-Point Input” on page 22-21. You can also set its *fimath* properties, see “Specify fimath of Fixed-Point Input” on page 22-22. If you do not set the *fimath* properties, *codegen* uses the MATLAB default *fimath* value.

If you set the class of an input parameter to *struct*, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class *fi* (fixed-point numeric object). For example, to set the class of input *U* to *fi*, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U, 'embedded.fi'));
...
```

If you set the class of an input parameter to `fi`, you must also set its `numericType`, see “Specify `numericType` of Fixed-Point Input” on page 22-21. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 22-22. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Structure Class

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...

or

...
assert(isa(U, 'struct'));
...
```

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Fixed Size

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...  
assert(all(size(U)== [3 2]));  
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )  
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. To set the size of input *U* to scalar, call:

```
...  
assert(isscalar(U));  
...
```

or

```
...  
assert(all(size(U)== [1]));  
...
```

Specify Upper Bounds for Variable-Size Inputs

```
assert ( all(size(param)<=[N0 N1 ...]));  
assert ( all(size(param)<[N0 N1 ...]));
```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input *U* to be less than or equal to a 3-by-2 matrix, call:

```
assert(all(size(U)<=[3 2]));
```

Note You can also specify upper bounds for variable-size inputs using `coder.versizecoder.versizecoder.versize`.

Specify Inputs with Fixed- and Variable-Size Dimensions

```
assert ( all(size(param)>=[M0 M1 ...]));
```

```
assert ( all(size(param)<=[N0 N1 ...]));
```

When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, *k*, the lower-bound *M_k* must be less than or equal to the upper-bound *N_k*.
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be non-negative.

To fix the size of the first dimension of input *U* to 3 and set the second dimension as variable size with upper-bound of 2, call:

```
assert(all(size(U)>=[3 0]));  
assert(all(size(U)<=[3 2]));
```

Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);  
assert (size(param, k)<=Nk);  
assert (size(param, k)<Nk);
```

You can specify individual dimensions as well as specifying all dimensions simultaneously or instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension *k* of input parameter *param*. To set the upper-bound size of the first dimension of input *U* to 3, call:

```
assert(size(U,1)<=3)
```

To fix the size of the second dimension of input *U* to 2, call:

```
assert(size(U,2)==2)
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *param* is real. To specify that input *U* is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *param* is complex. To specify that input *U* is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *fiparam* to the `numerictype` object *T*. For example, to specify the `numerictype` property of fixed-point input *U* as a signed `numerictype` object *T* with 32-bit word length and 30-bit fraction length, use the following code:

```
%#codegen
...
% Define the numerictype object.
```

```
T = numericitytype(1, 32, 30);

% Set the numericitytype property of input U to T.
assert(isequal(numericitytype(U),T));
...
```

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *fiparam* to the `fimath` object *F*. For example, to specify the `fimath` property of fixed-point input *U* so that it saturates on integer overflow, use the following code:

```
%#codegen
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

If you do not specify the `fimath` properties using `assert`, `codegen` uses the MATLAB default `fimath` value.

Specify Multiple Properties of Input

```
assert ( function1 ( params ) &&
         function2 ( params ) &&
         function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input *U* is a double, complex, 3-by-3 matrix, and input *V* is a 16-bit unsigned integer:

```
%#codegen
...
assert(isa(U,'double') &&
       ~isreal(U) &&
```

```

    all(size(U) == [3 3]) &&
    isa(V, 'uint16'));
...

```

Rules for Using assert Function

When using the `assert` function to specify the properties of primary function inputs, follow these rules:

- Call `assert` functions at the beginning of the primary function, before any control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with MATLAB Coder only for specifying properties of primary function inputs before converting your MATLAB code to C/C++ code.
- If you set the class of an input parameter to `fi`, you must also set its `numericType`. See “Specify `numericType` of Fixed-Point Input” on page 22-21. You can also set its `fimath` properties. See “Specify `fimath` of Fixed-Point Input” on page 22-22. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(param) >= [M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
 - You must also specify an upper-bound size for each dimension of the input parameter.
 - For each dimension, k , the lower-bound M_k must be less than or equal to the upper-bound N_k .
 - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
 - Bounds must be non-negative.
- If you specify individual dimensions, the following rules apply:
 - You must specify the size of each dimension at least once.

- The last dimension specification takes precedence over earlier specifications.

Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
pennywhistle	class	int16
	size	220500-by-1 vector
	complexity	real (by default)
win	class	double
	size	1024-by-1 vector
	complexity	real (by default)

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...
```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...
```


Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Toolbox 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
<code>class</code>	<code>fi</code>
<code>numerictype</code>	numerictype object <code>T</code> , as specified in the primary function
<code>fimath</code>	fimath object <code>F</code> , as specified in the primary function
<code>size</code>	scalar
<code>complexity</code>	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);
```

Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of `S` and its fields when passed as an input to your MATLAB function:

```
%#codegen
```

```
function y = fcn(S)

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));
...
```

In most cases, when you don't explicitly specify values for properties, MATLAB Coder uses defaults — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of a structure field

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 2-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

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

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [2 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of all fields.

Accelerate Code for Variable-Size Data (TBD)

Enable and Disable Dynamic Memory Allocation (TBD)

Fix Runtime Stack Overflows (TBD)

Check Code Using the MATLAB Code Analyzer

The code analyzer checks your code for problems and recommends modifications to maximize performance and maintainability. 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.

Fix Errors Detected at Code Generation Time

When the code generation software detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “MATLAB Algorithm Design Basics” “Algorithm Design Basics”. Choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies” on page 8-25.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generation software determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls in MATLAB Generated Code” “Resolution of Function Calls in MATLAB Generated Code” on page 10-2. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

See Also

- “Create and Use Fixed-Point Code Generation Reports” on page 8-52
-
- “When to Generate Code from MATLAB Algorithms” “When to Generate Code from MATLAB Algorithms” on page 15-2
- “Debugging Strategies” on page 8-25
- “Declaring MATLAB Functions as Extrinsic Functions” “Declaring MATLAB Functions as Extrinsic Functions” on page 10-12

System Objects Supported for Code Generation

System Objects Supported for Code Generation

In this section...
“Code Generation for System Objects” on page 24-2
“Computer Vision System Toolbox System Objects” on page 24-2
“Communications System Toolbox System Objects” on page 24-7
“DSP System Toolbox System Objects” on page 24-13

Code Generation for System Objects

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

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

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

Computer Vision System Toolbox System Objects

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

Supported Computer Vision System Toolbox System Objects

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

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.CornerDetector</code>	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
<code>vision.Convolver</code>	Compute 2-D discrete convolution of two input matrices
<code>vision.ImageFilter</code>	Perform 2-D FIR filtering of input matrix
<code>vision.MedianFilter</code>	2D median filtering
Geometric Transformations	
<code>vision.GeometricRotator</code>	Rotate image by specified angle
<code>vision.GeometricScaler</code>	Enlarge or shrink image size
<code>vision.GeometricShearer</code>	Shift rows or columns of image by linearly varying offset
<code>vision.GeometricTransformer</code>	Apply projective or affine transformation to an image
<code>vision.GeometricTransformEstimator</code>	Estimate geometric transformation from matching point pairs
<code>vision.GeometricTranslator</code>	Translate image in two-dimensional plane using displacement vector
Morphological Operations	
<code>vision.ConnectedComponentLabeler</code>	Label and count the connected regions in a binary image
<code>vision.MorphologicalClose</code>	Perform morphological closing on image
<code>vision.MorphologicalDilate</code>	Perform morphological dilation on an image
<code>vision.MorphologicalErode</code>	Perform morphological erosion on an image

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.MorphologicalOpen</code>	Perform morphological opening on an image
Object Detection	
<code>vision.HistogramBasedTracker</code>	Track object in video based on histogram. This object supports tunable properties in code generation.
Sinks	
<code>vision.DeployableVideoPlayer</code>	Send video data to computer screen
<code>vision.VideoFileWriter</code>	Write video frames and audio samples to multimedia file
Sources	
<code>vision.VideoFileReader</code>	Read video frames and audio samples from compressed multimedia file
Statistics	
<code>vision.Autocorrelator</code>	Compute 2-D autocorrelation of input matrix
<code>vision.BlobAnalysis</code>	Compute statistics for connected regions in a binary image
<code>vision.Crosscorrelator</code>	Compute 2-D cross-correlation of two input matrices
<code>vision.Histogram</code>	Generate histogram of each input matrix. This object has no tunable properties.
<code>vision.LocalMaximaFinder</code>	Find local maxima in matrices
<code>vision.Maximum</code>	Find maximum values in input or sequence of inputs

Supported Computer Vision System Toolbox System Objects (Continued)

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

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
Utilities	
<code>vision.ImagePadder</code>	Pad or crop input image along its rows, columns, or both

Communications System Toolbox System Objects

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

Supported Communications System Toolbox System Objects

Object	Description
Source Coding	
<code>comm.DifferentialDecoder</code>	Decode binary signal using differential decoding
<code>comm.DifferentialEncoder</code>	Encode binary signal using differential coding
Channels	
<code>comm.AWGNChannel</code>	Add white Gaussian noise to input signal
<code>comm.LTEMIMOChannel</code>	Filter input signal through LTE MIMO multipath fading channel
<code>comm.MIMOChannel</code>	Filter input signal through MIMO multipath fading channel
<code>comm.BinarySymmetricChannel</code>	Introduce binary errors
Equalizers	
<code>comm.MLSEEqualizer</code>	Equalize using maximum likelihood sequence estimation
Filters	
<code>comm.IntegrateAndDumpFilter</code>	Integrate discrete-time signal with periodic resets

Supported Communications System Toolbox System Objects (Continued)

Object	Description
Measurements	
comm.ACPR	Measure adjacent channel power ratio
comm.CCDF	Measure complementary cumulative distribution function
comm.EVM	Measure error vector magnitude
comm.MER	Measure modulation error ratio
Sources	
comm.BarkerCode	Generate Barker code
comm.HadamardCode	Generate Hadamard code
comm.KasamiSequence	Generate a Kasami sequence
comm.OVSFCode	Generate OVSF code
comm.PNSequence	Generate a pseudo-noise (PN) sequence
comm.WalshCode	Generate Walsh code from orthogonal set of codes
Error Detection and Correction – Block Coding	
comm.BCHDecoder	Decode data using BCH decoder
comm.BCHEncoder	Encode data using BCH encoder
comm.LDPCDecoder	Decode binary low-density parity-check code
comm.LDPCEncoder	Encode binary low-density parity-check code
comm.RSDecoder	Decode data using Reed-Solomon decoder
comm.RSEncoder	Encode data using Reed-Solomon encoder
Error Detection and Correction – Convolutional Coding	
comm.ConvolutionalEncoder	Convolutionally encode binary data
comm.ViterbiDecoder	Decode convolutionally encoded data using Viterbi algorithm
Error Detection and Correction – Cyclic Redundancy Check Coding	

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.CRCDetector</code>	Detect errors in input data using cyclic redundancy code
<code>comm.CRCGenerator</code>	Generate cyclic redundancy code bits and append to input data
<code>comm.HDLCRCGenerator</code>	Generate CRC code bits and append to input data, optimized for HDL code generation
<code>comm.TurboDecoder</code>	Decode input signal using parallel concatenated decoding scheme
<code>comm.TurboEncoder</code>	Encode input signal using parallel concatenated encoding scheme
Interleavers – Block	
<code>comm.AlgebraicDeinterleaver</code>	Deinterleave input symbols using algebraically derived permutation vector
<code>comm.AlgebraicInterleaver</code>	Permute input symbols using an algebraically derived permutation vector
<code>comm.BlockDeinterleaver</code>	Deinterleave input symbols using permutation vector
<code>comm.BlockInterleaver</code>	Permute input symbols using a permutation vector
<code>comm.MatrixDeinterleaver</code>	Deinterleave input symbols using permutation matrix
<code>comm.MatrixInterleaver</code>	Permute input symbols using permutation matrix
<code>comm.MatrixHelicalScanDeinterleaver</code>	Deinterleave input symbols by filling a matrix along diagonals
<code>comm.MatrixHelicalScanInterleaver</code>	Permute input symbols by selecting matrix elements along diagonals
Interleavers – Convolutional	
<code>comm.ConvolutionalDeinterleaver</code>	Restore ordering of symbols using shift registers
<code>comm.ConvolutionalInterleaver</code>	Permute input symbols using shift registers

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.HelicalDeinterleaver</code>	Restore ordering of symbols using a helical array
<code>comm.HelicalInterleaver</code>	Permute input symbols using a helical array
<code>comm.MultiplexedDeinterleaver</code>	Restore ordering of symbols using a set of shift registers with specified delays
<code>comm.MultiplexedInterleaver</code>	Permute input symbols using a set of shift registers with specified delays
MIMO	
<code>comm.OSTBCCombiner</code>	Combine inputs using orthogonal space-time block code
<code>comm.OSTBCEncoder</code>	Encode input message using orthogonal space-time block code
Digital Baseband Modulation – Phase	
<code>comm.BPSKDemodulator</code>	Demodulate using binary PSK method
<code>comm.BPSKModulator</code>	Modulate using binary PSK method
<code>comm.DBPSKModulator</code>	Modulate using differential binary PSK method
<code>comm.DPSKDemodulator</code>	Demodulate using M-ary DPSK method
<code>comm.DPSKModulator</code>	Modulate using M-ary DPSK method
<code>comm.DQPSKDemodulator</code>	Demodulate using differential quadrature PSK method
<code>comm.DQPSKModulator</code>	Modulate using differential quadrature PSK method
<code>comm.DBPSKDemodulator</code>	Demodulate using M-ary DPSK method
<code>comm.QPSKDemodulator</code>	Demodulate using quadrature PSK method
<code>comm.QPSKModulator</code>	Modulate using quadrature PSK method
<code>comm.PSKDemodulator</code>	Demodulate using M-ary PSK method
<code>comm.PSKModulator</code>	Modulate using M-ary PSK method
<code>comm.OQPSKDemodulator</code>	Demodulate offset quadrature PSK modulated data

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.OQPSKModulator</code>	Modulate using offset quadrature PSK method
Digital Baseband Modulation – Amplitude	
<code>comm.GeneralQAMDemodulator</code>	Demodulate using arbitrary QAM constellation. This object has no tunable properties in code generation.
<code>comm.GeneralQAMModulator</code>	Modulate using arbitrary QAM constellation
<code>comm.PAMDemodulator</code>	Demodulate using M-ary PAM method
<code>comm.PAMModulator</code>	Modulate using M-ary PAM method
<code>comm.RectangularQAMDemodulator</code>	Demodulate using rectangular QAM method
<code>comm.RectangularQAMModulator</code>	Modulate using rectangular QAM method
Digital Baseband Modulation – Frequency	
<code>comm.FSKDemodulator</code>	Demodulate using M-ary FSK method
<code>comm.FSKModulator</code>	Modulate using M-ary FSK method
Digital Baseband Modulation – Trellis Coded	
<code>comm.GeneralQAMTCMDemodulator</code>	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation
<code>comm.GeneralQAMTCMModulator</code>	Convolutionally encode binary data and map using arbitrary QAM constellation
<code>comm.PSKTCMDemodulator</code>	Demodulate convolutionally encoded data mapped to M-ary PSK constellation
<code>comm.PSKTCMModulator</code>	Convolutionally encode binary data and map using M-ary PSK constellation
<code>comm.RectangularQAMTCMDemodulator</code>	Demodulate convolutionally encoded data mapped to rectangular QAM constellation
<code>comm.RectangularQAMTCMModulator</code>	Convolutionally encode binary data and map using rectangular QAM constellation
Digital Baseband Modulation – Continuous Phase	

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.CPFSKDemodulator</code>	Demodulate using CPFSK method and Viterbi algorithm
<code>comm.CPFSKModulator</code>	Modulate using CPFSK method
<code>comm.CPMDemodulator</code>	Demodulate using CPM method and Viterbi algorithm
<code>comm.CPMModulator</code>	Modulate using CPM method
<code>comm.GMSKDemodulator</code>	Demodulate using GMSK method and the Viterbi algorithm
<code>comm.GMSKModulator</code>	Modulate using GMSK method
<code>comm.MSKDemodulator</code>	Demodulate using MSK method and the Viterbi algorithm
<code>comm.MSKModulator</code>	Modulate using MSK method
RF Impairments	
<code>comm.MemorylessNonlinearity</code>	Apply memoryless nonlinearity to input signal
<code>comm.PhaseFrequencyOffset</code>	Apply phase and frequency offsets to input signal. The <code>PhaseOffset</code> property of this object is not tunable in code generation.
<code>comm.PhaseNoise</code>	Apply phase noise to complex baseband signal
<code>comm.ThermalNoise</code>	Add receiver thermal noise
Synchronization – Timing Phase	
<code>comm.EarlyLateGateTimingSynchronizer</code>	Recover symbol timing phase using early-late gate method
<code>comm.GardnerTimingSynchronizer</code>	Recover symbol timing phase using Gardner's method
<code>comm.GMSKTimingSynchronizer</code>	Recover symbol timing phase using fourth-order nonlinearity method
<code>comm.MSKTimingSynchronizer</code>	Recover symbol timing phase using fourth-order nonlinearity method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
<code>comm.MuellerMullerTimingSynchronizer</code>	Recover symbol timing phase using Mueller-Muller method
Synchronization Utilities	
<code>comm.CPMCarrierPhaseSynchronizer</code>	Recover carrier phase of baseband CPM signal
<code>comm.DiscreteTimeVCO</code>	Generate variable frequency sinusoid
Converters	
<code>comm.BitToInteger</code>	Convert vector of bits to vector of integers
<code>comm.IntegerToBit</code>	Convert vector of integers to vector of bits
Sequence Operators	
<code>comm.Descrambler</code>	Descramble input signal
<code>comm.GoldSequence</code>	Generate Gold sequence
<code>comm.Scrambler</code>	Scramble input signal

DSP System Toolbox System Objects

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

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
<code>dsp.BurgAREstimator</code>	Compute estimate of autoregressive model parameters using Burg method

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.BurgSpectrumEstimator</code>	Compute parametric spectral estimate using Burg method
<code>dsp.CepstralToLPC</code>	Convert cepstral coefficients to linear prediction coefficients
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
<code>dsp.LPCToAutocorrelation</code>	Convert linear prediction coefficients to autocorrelation coefficients
<code>dsp.LPCToCepstral</code>	Convert linear prediction coefficients to cepstral coefficients
<code>dsp.LPCToLSF</code>	Convert linear prediction coefficients to line spectral frequencies
<code>dsp.LPCToLSP</code>	Convert linear prediction coefficients to line spectral pairs
<code>dsp.LPCToRC</code>	Convert linear prediction coefficients to reflection coefficients
<code>dsp.LSFToLPC</code>	Convert line spectral frequencies to linear prediction coefficients
<code>dsp.LSPToLPC</code>	Convert line spectral pairs to linear prediction coefficients
<code>dsp.RCToAutocorrelation</code>	Convert reflection coefficients to autocorrelation coefficients
<code>dsp.RCToLPC</code>	Convert reflection coefficients to linear prediction coefficients
Filters	
<code>dsp.AllpoleFilter</code>	IIR Filter with no zeros. Only the Denominator property is tunable for code generation.
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.CICDecimator</code>	Decimate input using Cascaded Integrator-Comb filter
<code>dsp.CICInterpolator</code>	Interpolate signal using Cascaded Integrator-Comb filter
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations. The <code>SOSMatrix</code> and <code>ScaleValues</code> properties are not supported for code generation.
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRFilter</code>	Static or time-varying FIR filter. Only the <code>Numerator</code> property is tunable for code generation.
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.IIRFilter</code>	Infinite Impulse Response (IIR) filter. Only the <code>Numerator</code> and <code>Denominator</code> properties are tunable for code generation.
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
Math Operations	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements

Supported DSP System Toolbox System Objects (Continued)

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

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.VariableFractionalDelay</code>	Delay input by time-varying fractional number of sample periods
<code>dsp.VariableIntegerDelay</code>	Delay input by time-varying integer number of sample periods
<code>dsp.Window</code>	Generate or apply window function. This object has no tunable properties for code generation.
<code>dsp.ZeroCrossingDetector</code>	Calculate number of zero crossings of a signal
Sinks	
<code>dsp.AudioPlayer</code>	Write audio data to computer's audio device
<code>dsp.AudioFileWriter</code>	Write audio file
<code>dsp.UDPSender</code>	Send UDP packets to the network
Sources	
<code>dsp.AudioFileReader</code>	Read audio samples from an audio file
<code>dsp.AudioRecorder</code>	Read audio data from computer's audio device
<code>dsp.SignalSource</code>	Import variable from workspace
<code>dsp.SineWave</code>	Generate discrete sine wave. This object has no tunable properties for code generation.
<code>dsp.UDPReceiver</code>	Receive UDP packets from the network
Statistics	
<code>dsp.Autocorrelator</code>	Compute autocorrelation of vector inputs
<code>dsp.Crosscorrelator</code>	Compute cross-correlation of two inputs
<code>dsp.Histogram</code>	Output histogram of an input or sequence of inputs. This object has no tunable properties for code generation.
<code>dsp.Maximum</code>	Compute maximum value in input
<code>dsp.Mean</code>	Compute average or mean value in input

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.Median	Compute median value in input
dsp.Minimum	Compute minimum value in input
dsp.RMS	Compute root-mean-square of vector elements
dsp.StandardDeviation	Compute standard deviation of vector elements
dsp.Variance	Compute variance of input or sequence of inputs
Transforms	
dsp.AnalyticSignal	Compute analytic signals of discrete-time inputs
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input
dsp.IDCT	Compute inverse discrete cosine transform (IDCT) of input
dsp.IFFT	Compute inverse fast Fourier transform (IFFT) of input

System Objects

- “Create System Objects” on page 25-2
- “Set Up System Objects” on page 25-6
- “Process Data Using System Objects” on page 25-11
- “Tuning System object Properties in MATLAB” on page 25-16
- “Find Help and Examples for System Objects” on page 25-19
- “Use System Objects in MATLAB Code Generation” on page 25-21

Create System Objects

In this section...
“Create a System object” on page 25-3
“Define a New System object” on page 25-3
“Change a System object Property” on page 25-4
“Check if a System object Property Has Changed” on page 25-4
“Run a System object” on page 25-4
“Display Available System Objects” on page 25-5

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

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets.

System objects support fixed-point arithmetic. To use 64-bit data types, you must have Fixed-Point Toolbox software. System objects also support C-code generation from MATLAB and Simulink. With System objects, you can optionally generate code to target the desktop or external hardware. You can use System objects in Simulink® models via the MATLAB Function block. You can compile code that contains System objects within MATLAB functions using MATLAB Compiler software. (The compiler product does not support System objects in MATLAB scripts.)

Note System objects predefined in the software do not support sparse matrices. System objects you define support sparse matrices (see “Define a New System object” on page 25-3).

Create a System object

To use System objects, you must first create an object. For example,

```
H = dsp.FFT           % Create default FFT object, H

% Create input data
Fs = 1000;           % Sampling frequency
T = 1/Fs;            % Sample time
L = 1024;            % Length of signal
t = (0:L-1)*T        % Time vector

% Sum of two sinusoids
X = 0.7*sin(2*pi*50*t.) + sin(2*pi*120*t.);

H = vision.FFT       % Create default FFT object, H

% Create input data
Fs = 1000;           % Sampling frequency
T = 1/Fs;            % Sample time
L = 1024;            % Length of signal
t = (0:L-1)*T        % Time vector

% Sum of two sinusoids
X = 0.7*sin(2*pi*50*t.) + sin(2*pi*120*t.);

Hram = hdlram        % Create default hdlram object, H

H = phased.LinearFMWaveform;
```

Define a New System object

You can define a System object to implement your algorithm. For information and examples, see “Define New System Objects” “Define New System Objects” “Define New System Objects” “Define New System Objects”.

Change a System object Property

In general, you should set the object properties before you use the `step` method to run data through the object. To change the value of a property, use this format,

```
H.Normalize = true    % Set the Normalize property
```

The property values of the FFT object, H, are displayed.

```
H.RAMType = 'Dual Port'    % Set the RAMType property
```

The property values of the `hdlram` object, H, are displayed.

```
H.SweepBandwidth = 2e5;    % Set the SweepBandwidth property  
H.SweepDirection = 'Down' % Set the SweepDirection property
```

The property values of the linear FM pulse waveform object, H, are displayed.

Check if a System object Property Has Changed

To check if a tunable property has changed since `step` was last called, use this syntax:

```
flag = isChangedProperty(H, 'Normalize')
```

`flag` is true if the `Normalize` property of object H has changed.

Run a System object

To execute a System object, use the `step` method.

```
Y = step(H,X);           % Process input data, X
```

```
Y = step(H);
```

The output data from the `step` method is stored in Y, which, in this case, is the FFT of X.

The output data from the `step` method is stored in Y, which, in this case, is the FFT of X.

The output data from the `step` method is stored in `Y`, which, in this case, is a vector of samples from the linear FM pulse waveform.

The output data from the `step` method is stored in `Y`, which, in this case, is port input and output data.

Display Available System Objects

To see a list of all the System objects for a particular package, type `help dsphelp commhelp phasedhelp visionhelp hdlverifier`. To display help for specific objects, properties, or methods, see “Find Help and Examples for System Objects” on page 25-19 .

Set Up System Objects

In this section...

“Create a New System object” on page 25-6

“Retrieve System object Property Values” on page 25-6

“Set System object Property Values” on page 25-7

Create a New System object

You must create a `System` object before using it. You can create the object at the MATLAB command line or within a program file. Your command-line code and programs can pass MATLAB variables into and out of `System` objects.

For general information about working with MATLAB objects, see “Object-Oriented Programming” in the MATLAB documentation.

Retrieve System object Property Values

`System` objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *Name-Value pair*. You can display the list of relevant property names and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handle>.<Name>`.

Example

This example retrieves and displays the `TransferFunction` property value for the previously created `DigitalFilter` object:

This example retrieves and displays the `InitialCondition` property value for the previously created `DifferentialDecoder` object:

This example retrieves and displays the `Threshold` property value for the previously created `EdgeDetector` object:

This example retrieves and displays the `PeakPower` property value for the previously created `Transmitter` object:

This example retrieves and displays the `RAMType` property value for the previously created `hdlram` object:

`H.TransferFunction`

`H.InitialCondition`

`H.Threshold`

`H.PeakPower`

`H.RAMType`

Set System object Property Values

You set the property values of a System object to model the desired algorithm.

Note When you use Name-Value pair syntax, the object sets property values in the order you list them. If you specify a dependent property value before its parent property, an error or warning may occur.

Set Properties for a New System object

To set a property when you first create the object, use Name-Value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
H1 = dsp.DigitalFilter('CoefficientsSource','Input port')
```

```
H1 = comm.DifferentialDecoder('InitialCondition',1)
```

```
H1 = vision.EdgeDetector('ThresholdSource','Property')
```

```
H1 = phased.Transmitter('PeakPower',6000)
```

```
H1 = hdlram('RAMType','Single port')
```

where

- H1 is the handle to the object
- dsp is the package name.
comm is the package name.
vision is the package name.
phased is the package name.
- DigitalFilter is the object name.
DifferentialDecoder is the object name.
EdgeDetector is the object name.
Transmitter is the object name.
hdlram is the object name.
- CoefficientsSource is the property name.
InitialCondition is the property name.
ThresholdSource is the property name.
PeakPower is the property name.
RAMType is the property name.
- 'Input port' is the property value.
1 is the property value.
'Property' is the property value.
6000 is the property value.
'Single port' is the property value.

Set Properties for an Existing System object

To set a property after you have created an object, use either of the following syntaxes:

```

H1.CoefficientsSource = 'Property'

H1.InitialCondition = 0

H1.ThresholdSource = 'Input port'

H1.PeakPower = 6500

H1.RAMType = 'Dual port'

or

set(H1,'CoefficientsSource','Property')

set(H1,'InitialCondition',0)

set(H1,'ThresholdSource','Input property')

set(H1,'PeakPower',6500)

set(H1,'RAMType','Dual port')

```

Use Value-Only Inputs

Some object properties have no useful default values or must be specified every time you create an object. For these properties, you can specify only the value without specifying the corresponding property name. If you use value-only inputs, those inputs must be in a specific order, which is the same as the order in which the properties are displayed. Refer to the object reference page for details.

```
H2 = dsp.FIRDecimator(3,[1 .5 1])
```

specifies the DecimationFactor as 3 and the Numerator as [1 .5 1].

```
H2 = vision.VideoFileReader('viptrain.avi')
```

specifies the Filename as viptrain.avi.

```
hURA = phased.URA([2 3],0.25);
```

specifies the `Size` property as `[2 3]` and the `ElementSpacing` property as `0.25`.

Process Data Using System Objects

In this section...

“What are System object Methods?” on page 25-11

“The Step Method” on page 25-11

“Common Methods” on page 25-13

“Advantages of Using Methods” on page 25-15

What are System object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the `step` method and other available methods, see the descriptions in “Common Methods” on page 25-13.

Calculate the Effect of Propagating a Signal in Free Space

This example uses two different `step` methods. The first `step` method is associated with the `phased.LinearFMWaveform` object and the second `step` method is associated with the `phased.Freespace` object.

Construct a linear FM waveform with a pulse duration of 50 microseconds, a sweep bandwidth of 100 kHz, an increasing instantaneous frequency, and a pulse repetition frequency (PRF) of 10 kHz..

```
hFM = phased.LinearFMWaveform('SampleRate',1e6,...
```

```
'PulseWidth',5e-5,'PRF',1e4,...  
'SweepBandwidth',1e5,'SweepDirection','Up',...  
'OutputFormat','Pulses','NumPulses',1);
```

Obtain the waveform using the `step` method. Note that the input to the `step` method is a handle to a `phased.LinearFMWaveform` object.

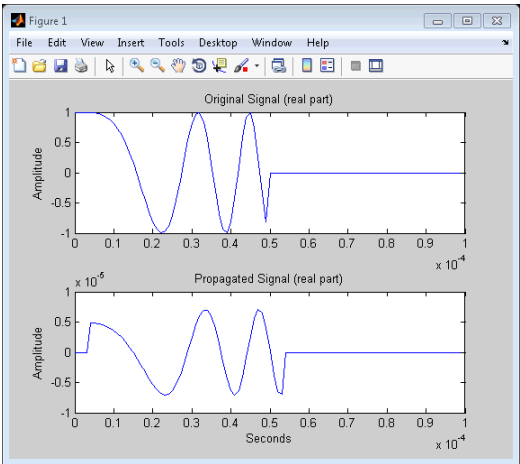
```
Sig = step(hFM);
```

Construct a free space object with a propagation speed equal to the speed of light, an operating frequency of 3 GHz, and a sample rate of 1 MHz. The free space object is constructed to model one way propagation.

```
hFS = phased.FreeSpace(...  
    'PropagationSpeed',physconst('LightSpeed'),...  
    'OperatingFrequency',3e9,'TwoWayPropagation',false,...  
    'SampleRate',1e6);
```

Calculate the effect on the waveform of one-way propagation in free space from coordinates [0;0;0] to [500; 1e3; 20] and plot the results for comparison.

```
PropSig = step(hFS,Sig,[0; 0; 0],[500; 1e3; 20],...  
    [0;0;0],[0;0;0]);  
% compare the original signal to the propagated waveform  
t = unigrid(0,1/hFS.SampleRate,length(Sig)*1/hFS.SampleRate,'[]');  
subplot(211)  
plot(t,real(Sig)); title('Original Signal (real part)');  
ylabel('Amplitude');  
subplot(212)  
plot(t,real(PropSig)); title('Propagated Signal (real part)');  
xlabel('Seconds'); ylabel('Amplitude');
```



Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
<code>step</code>	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the <code>step</code> method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The <code>step</code> method returns regular MATLAB variables. Example: <code>Y = step(H,X)</code>
<code>release</code>	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the <code>release</code> method instead of a destructor. See “Understand System object Modes” on page 25-16.

Method	Description
<code>clone</code>	Creates another object with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object is locked. See “Understand System object Modes” on page 25-16.
<code>reset</code>	Resets the internal states of the object to the initial values for that object
<code>isDone</code>	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns <code>false</code> .
<code>isChangedProperty</code>	Returns <code>true</code> if the specified tunable property value has changed since the last call to <code>step</code> . Example: <code>flag = isChangedProperty(obj, 'propertyName')</code>
<code>info</code>	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.
<code>getNumInputs</code>	Returns the number of inputs (excluding the object itself) expected by the <code>step</code> method. This number varies for an object depending on whether any properties enable additional inputs.
<code>getNumOutputs</code>	Returns the number of outputs expected from the <code>step</code> method. This number varies for an object depending on whether any properties enable additional outputs.
<code>getDiscreteState</code>	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the <code>step</code> method on it or after you have released the object), the states are empty. If the object has no discrete states, <code>getDiscreteState</code> returns an empty structure.

Advantages of Using Methods

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

Tuning System object Properties in MATLAB

In this section...
“Understand System object Modes” on page 25-16
“Change Properties While Running System Objects” on page 25-17
“Change System object Input Complexity or Dimensions” on page 25-18

Understand System object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

The object initializes and locks when it begins processing data. The typical way in which an object becomes locked is when the `step` method is called on that object. To determine if an object is locked, use the `isLocked` method. To unlock an object, use the `release` method. When the object is locked, you cannot change any of the following:

- Number of inputs or outputs
- Data type
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data. Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.
- Value of any nontunable property

Several System objects do not allow changing the complexity of inputs from real to complex. You can, however, change the input complexity from complex to real without unlocking the object.

These restrictions allow the object to maintain states and allocate memory appropriately.

Change Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding reference page or use a command of this form:

```
help dsp.FFT.Normalize
```

```
help comm.DifferentialDecoder.InitialCondition
```

```
help vision.EdgeDetector.Threshold
```

```
help phased.Transmitter.PeakPower
```

```
help hdlram.RAMType
```

where

- `dsp` is the package name.
`comm` is the package name.
`vision` is the package name.
`phased` is the package name.
- `FFT` is the object name.
`DifferentialDecoder` is the object name.
`EdgeDetector` is the object name.
`Transmitter` is the object name.
`hdlram` is the object name.
- `Normalize` is the property name.
`InitialCondition` is the property name.
`Threshold` is the property name.
`PeakPower` is the property name.
`RAMType` is the property name.

Note Unless otherwise specified, System object properties are not tunable.

For information on locked and unlocked modes, see “Understand System object Modes” on page 25-16.

Change System object Input Complexity or Dimensions

During simulations you can change an input’s complexity from complex to real, but not from real to complex. You cannot change any input complexity during code generation.

For objects that do not support variable-size input, if you change the input dimensions while the object is in locked mode, the object produces a warning and unlocks. The object then reinitializes the next time you call the step method. See the object’s reference page for more information. You can change the value of a tunable property and the input size without a warning or error being produced. For all other changes at runtime, an error occurs.

Find Help and Examples for System Objects

Refer to the following resources for more information about System objects.

- Package help – `help dsp`, where `dsp` is a product package name
 Package help – `help comm`, where `comm` is a product package name
 Package help – `help vision`, where `vision` is a product package name
 Package help – `help phased`, where `phased` is a product package name
- Object help – `help dsp.FFT`, where `FFT` is the object name
 Object help – `help comm.DifferentialDecoder`, where `DifferentialDecoder` is the object name
 Object help – `help vision.EdgeDetector`, where `EdgeDetector` is the object name
 Object help – `help phased.Transmitter`, where `Transmitter` is the object name
 Object help – `help hdlverifier.HdlCosimulation`
 Object help – `help hdlram`
- Documentation reference pages for an object – `doc dsp.FFT`
 Documentation reference pages for an object – `doc comm.DifferentialDecoder`
 Documentation reference pages for an object – `doc vision.EdgeDetector`
 Documentation reference pages for an object – `doc phased.Transmitter`
 Documentation pages for object – `doc hdlverifier.HdlCosimulation`
 Documentation pages for object – `doc hdlram`
- Property help — `help dsp.FFT.Normalize`, where `Normalize` is the property name.
 Property help — `help comm.DifferentialDecoder.InitialCondition`, where `InitialCondition` is the property name.
 Property help — `help vision.EdgeDetector.Threshold`, where `Threshold` is the property name.

Property help — `help phased.Transmitter.PeakPower`, where `PeakPower` is the property name.

Property help — `help hdlverifier.HdlCosimulation`

Property help — `help hdlram.RAMType`

- Fixed-point property help — `dsp.FFT.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.

Fixed-point property help — `comm.DifferentialDecoder.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.

Fixed-point property help — `vision.EdgeDetector.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.

Fixed-point property help — `hdlram.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.

- Method help — `help dsp.FFT.step`, where `step` is the method name.

Method help — `help comm.DifferentialDecoder.step`, where `step` is the method name.

Method help — `help vision.EdgeDetector.step`, where `step` is the method name.

Method help — `help phased.Transmitter.step`, where `step` is the method name.

Method help — `help hdlram.step`, where `step` is the method name.

To view examples, go to the Help contents for the associated product. Under Examples, select MATLAB Examples. Under Examples, select Cosimulation with Cadence Incisive or Cosimulation with Mentor Graphics ModelSim.

Use System Objects in MATLAB Code Generation

In this section...

“Considerations for Using System Objects in Generated Code” on page 25-21

“Use System Objects with codegen” on page 25-26

“Use System Objects with the MATLAB Function Block” on page 25-26

“Use System Objects with MATLAB® Compiler™” on page 25-26

Considerations for Using System Objects in Generated Code

You can generate C/C++ code from System objects using MATLAB Coder product. Using this product with System objects, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. System objects also support code generation using the MATLAB Function block in Simulink and the MATLAB Coder codegen function.

For general information on generating code, see

- MATLAB Coder product
- Simulink Coder product
- Embedded Coder® product

The following example, which uses System objects, shows the key factors to consider, such as using persistent variables, passing property values, and extrinsic functions, when you make MATLAB code suitable for code generation.

```
function lmssystemidentification
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.
```

```
persistent hlms hfilt;

% Initialize persistent System objects only once
% Do this with 'if isempty(persistent variable).'
% This condition will be false after the first time.

if isempty(hlms)

    % Create LMS adaptive filter used for system
    % identification. Pass property value arguments
    % as constructor arguments. Property values must
    % be constants during compile time.

    hlms = dsp.LMSFilter(11, 'StepSize', 0.01);

    % Create system (an FIR filter) to be identified.

    hfilt = dsp.DigitalFilter(...
        'TransferFunction', 'FIR (all zeros)', ...
        'Numerator', fir1(10, .25));
end

x = randn(1000,1); % Input signal
d = step(hfilt, x) + 0.01*randn(1000,1); % Desired signal
[~,~,w] = step(hlms, x, d); % Filter weights

% Declare functions called into MATLAB that do not generate
% code as extrinsic.

coder.extrinsic('stem');

stem([get(hfilt, 'Numerator').', w]);
end

% To compile this function use codegen lmssystemidentification.
% This produces a mex file with the same name in the current
% directory.

function ex_system_codegen
```



```
% Find corresponding interest points between a pair of images using local
% neighborhoods.

%#codegen

% Declare System objects as persistent.
persistent cornerDetector colorSpaceConverter

% Initialize persistent System objects only once
% Do this with 'if isempty(persistent variable).'
```

```
% Find corners
points1 = step(cornerDetector, I1);
points2 = step(cornerDetector, I2);

% Extract neighborhood features
[features1, valid_points1] = extractFeatures(I1, points1);
[features2, valid_points2] = extractFeatures(I2, points2);

% Match features
index_pairs = matchFeatures(features1, features2);

% Retrieve locations of corresponding points for each image
matched_points1 = valid_points1(index_pairs(:, 1), :);
matched_points2 = valid_points2(index_pairs(:, 2), :);

% Visualize corresponding points
coder.extrinsic('showMatchedFeatures')
figure; showMatchedFeatures(I1, I2, matched_points1, matched_points2);
```

For a detailed code generation example, see “Generate Code for MATLAB Handle Classes and System Objects” in the MATLAB Coder product documentation.

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

Usage Rules for System Objects in Generated MATLAB Code

- Assign System objects to persistent variables.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a compiler options object. For example,

```
f = coder.MEXConfig;
f.GlobalSyncMethod='NoSync'
```

Then, include `'-config f'` in your `codegen` command.

- Initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty()`.
- Call the constructor exactly once for each System object.
- Set arguments to System object constructors as compile-time constants.
- Use the object constructor to set System object properties because you cannot use dot notation for code generation. You can use the `get` method to display properties.
- Test your code in simulation before generating code.

Limitations on Using System Objects in Generated MATLAB Code

- Ensure that size, type and complexity of inputs do not change.
- Ensure that the value assigned to a nontunable or public property is a constant and that there is at most one assignment to that property (including the assignment in the constructor).
- For most System objects predefined in the software, the only time you can set their properties during code generation is when you construct the objects. System objects that support tunable properties at any time during code generation are listed in the product's code generation support table. For System objects that you define, you can also change their tunable properties at any time during code generation.
- Do not change the size of properties during code generation.
- The only System object methods supported in code generation are
 - `get`
 - `getNumInputs`
 - `getNumOutputs`
 - `isDone` (for sources only)
 - `reset`
 - `step`
- Do not set System objects to become outputs from the 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 (i.e., functions called in interpreted mode) using the `coder.extrinsic` function. Do not return System objects from any extrinsic functions.

Use System Objects with `codegen`

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” and “MATLAB Classes” for more information.

Use System Objects with the MATLAB Function Block

Using the MATLAB Function block, you can include a MATLAB language function in a Simulink model. This model can then generate embeddable code. You can include any System object in the MATLAB Function block. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “What Is a MATLAB Function Block?” in the Simulink documentation.

Use System Objects with MATLAB Compiler

Note MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

A

ANSI C

- compared with `fi` objects 1-22

arguments

- limit on number for code generation from MATLAB 10-19

arithmetic

- fixed-point 4-11
- with [Slope Bias] signals 4-17

arithmetic operations

- fixed-point 1-10

B

binary conversions 1-25

C

C/C++ code generation for supported functions 20-1

casts

- fixed-point 1-19

clone method 25-14

Code generation

- fixed-point 8-1

code generation from MATLAB

- benefits of 15-2

best practices

- generate code generation report 8-48
- preserving your code 8-51
- separating test bench from function code 8-51
- specifying input properties 8-48
- using build scripts 8-49
- using file naming convention 8-51
- using the MATLAB code analyzer 8-50

- best practices for working with variables 14-3

- calling local functions 10-9

- calling MATLAB functions 10-11

- calling MATLAB functions using `feval` 10-16

- characters 12-6

- communications system toolbox System objects 24-7

- compilation directive `%#codegen` 10-8

- compiler options for MEX code generation 8-30

- computer vision system toolbox System objects 24-2

- controlling run-time checks 8-71

- converting `mxArrays` to known types 10-18

- declaring MATLAB functions as extrinsic functions 10-12

- defining persistent variables 14-10

- defining variables 14-2

- defining variables by assignment 14-3

- dsp system toolbox System objects 24-13

- eliminating redundant copies of function inputs 18-4

- eliminating redundant copies of uninitialized variables 14-7

- how it resolves function calls 10-2

- how to disable run-time checks 8-72

- initializing persistent variables 14-10

- inlining functions 18-3

- limit on number of function arguments 10-19
- pragma 10-8

- recommended options for `fiaccel` 8-48

- resolving extrinsic function calls during simulation 10-16

- resolving extrinsic function calls in generated code 10-17

- rules for defining uninitialized variables 14-7

- setting properties of indexed variables 14-6

- supported toolbox functions 10-10

- unrolling for-loops 18-2

- using Code Analyzer 8-26

- using type cast operators in variable definitions 14-6

- variables, complex 12-4

- when not to use 15-2
 - when to disable run-time checks 8-72
 - when to use 15-2
 - which features to use 15-4
 - working with mxArray 10-17
- `coder.extrinsic` 10-12
- `coder.nullcopy`
 - uninitialized variables 14-7
- communications system toolbox System objects
 - supported for code generation from MATLAB 24-7
- compiler options parameters
 - for MEX code generation from MATLAB 8-30
- compilers
 - supported for generating MEX functions with `fiaccel` 8-15
- complex multiplication
 - fixed-point 1-13
- computer vision system toolbox System objects
 - supported for code generation from MATLAB 24-2
- controlling run-time checks
 - code generation from MATLAB 8-71

D

- data type override 5-12
- defining uninitialized variables
 - rules 14-7
- defining variables
 - for C/C++ code generation 14-3
- design considerations
 - when writing MATLAB Code for code generation 15-7
- display preferences
 - setting 5-5
- dsp system toolbox System objects
 - supported for code generation from MATLAB 24-13

E

- eliminating redundant copies of function inputs 18-4
- extrinsic functions 10-12

F

- `fi` objects
 - constructing 2-2
- `fiaccel`
 - recommended options 8-48
 - supported compilers 8-15
- `fimath` objects
 - properties
 - setting in the Model Explorer 4-8
 - setting properties in the Model Explorer 4-8
- `fimath` objects 1-16
 - constructing 4-2
- `fipref` objects
 - constructing 5-2
- fixed-point arithmetic 4-11
- fixed-point data
 - reading from workspace 9-2
 - writing to workspace 9-2
- fixed-point data types
 - addition 1-12
 - arithmetic operations 1-10
 - casts 1-19
 - complex multiplication 1-13
 - modular arithmetic 1-10
 - multiplication 1-13
 - overflow handling 1-5
 - precision 1-5
 - range 1-5
 - rounding 1-6
 - saturation 1-5
 - scaling 1-4
 - subtraction 1-12
 - two's complement 1-11
 - wrapping 1-5

- fixed-point math 4-11
- Fixed-Point MATLAB code 8-1
- fixed-point run-time API 9-6
- fixed-point signal logging 9-6
- functions
 - limit on number of arguments for code generation 10-19
- Functions supported for C/C++ code
 - generation 20-1
 - alphabetical list 20-2
 - arithmetic operator functions 20-76
 - bit-wise operation functions 20-77
 - casting functions 20-77
 - Communications System Toolbox functions 20-78
 - complex number functions 20-78
 - Computer Vision System Toolbox functions 20-79
 - data type functions 20-80
 - derivative and integral functions 20-80
 - discrete math functions 20-81
 - error handling functions 20-81
 - exponential functions 20-81
 - filtering and convolution functions 20-82
 - Fixed-Point Toolbox functions 20-82
 - histogram functions 20-91
 - Image Processing Toolbox functions 20-91
 - input and output functions 20-92
 - interpolation and computational geometry functions 20-92
 - linear algebra functions 20-92
 - logical operator functions 20-93
 - MATLAB Compiler functions 20-93
 - matrix/array functions 20-94
 - nonlinear numerical methods 20-98
 - polynomial functions 20-98
 - relational operator functions 20-98
 - rounding and remainder functions 20-99
 - set functions 20-99
 - signal processing functions 20-100

- Signal Processing Toolbox functions 20-100
- special value functions 20-105
- specialized math functions 20-105
- statistical functions 20-106
- string functions 20-106
- structure functions 20-107
- trigonometric functions 20-107
- Functions supported for MEX and C/C++ code
 - generation
 - categorized list 20-75

G

- getDiscreteState method 25-14
- getNumInputs method 25-14
- getNumOutputs method 25-14

H

- how to disable run-time checks
 - code generation from MATLAB 8-72

I

- indexed variables
 - setting properties for code generation from MATLAB 14-6
- info method 25-14
- initialization
 - persistent variables 14-10
- interoperability
 - fi objects with DSP System Toolbox 9-7
 - fi objects with Filter Design Toolbox 9-11
 - fi objects with Simulink 9-2
- isChangedProperty method 25-14
- isDone method 25-14
- isLocked method 25-14

L

- locked vs. unlocked mode 25-16

- logging
 - overflows and underflows 5-7
- logging modes
 - setting 5-7

M

- math
 - with [Slope Bias] signals 4-17
- MATLAB
 - features not supported for code generation 15-14
- MATLAB Coder
 - best practices
 - using the MATLAB code analyzer 23-5
 - combining property specifications 22-24
 - specifying general properties of primary inputs 22-24
- MATLAB for code generation
 - variable types 14-18
- MATLAB Function block
 - using with Model Explorer and fixed-point models 8-77
- MATLAB functions
 - and generating code for mxArrayArrays 10-17
- Model Explorer
 - setting `embedded.fimath` properties 4-8
 - setting `embedded.numericity` properties 6-9
 - using with fixed-point code generation for MATLAB 8-77
- modular arithmetic 1-10
- multiplication
 - fixed-point 1-13
- mxArrayArrays
 - converting to known types 10-18
 - for code generation from MATLAB 10-17

N

- numericity objects
 - properties
 - setting in the Model Explorer 6-9
 - setting properties in the Model Explorer 6-9
- numericity objects
 - constructing 6-2

O

- one's complement 1-11
- overflow handling 1-5
 - compared with ANSI C 1-28
- overflows
 - logging 5-7

P

- padding 1-19
- persistent variables
 - defining for code generation from MATLAB 14-10
 - initializing for code generation from MATLAB 14-10
- precision
 - fixed-point data types 1-5
- property values 25-7
 - quantizer objects 7-3

Q

- quantizer objects
 - constructing 7-2
 - property values 7-3

R

- range
 - fixed-point data types 1-5
- reading fixed-point data from workspace 9-2
- release method 25-13

- reset method 25-14
- rounding
 - fixed-point data types 1-6
- run-time API
 - fixed-point data 9-6

S

- saturation 1-5
- scaling 1-4
- signal logging
 - fixed-point 9-6
- signal processing functions
 - for C/C++ code generation 20-100
- [Slope Bias] arithmetic 4-17
- step method 25-13
- streaming data
 - using System objects 25-15
- System object
 - clone method 25-14
 - creating 25-6
 - description 25-2
 - getDiscreteState method 25-14
 - getNumInputs method 25-14
 - getNumOutputs method 25-14
 - info method 25-14
 - isChangedProperty method 25-14
 - isDone method 25-14
 - isLocked 25-14
 - locked vs. unlocked mode 25-16
 - methods 25-11
 - properties 25-6
 - property values 25-7
 - release method 25-13
 - reset method 25-14
 - step method 25-13
 - tunable property 25-17
 - using with MATLAB code generation 25-21
 - value-only input 25-9

T

- tunable 25-17
- two's complement 1-11
- type cast operators
 - using in variable definitions 14-6

U

- unary conversions 1-24
- underflows
 - logging 5-7
- uninitialized variables
 - eliminating redundant copies in generated code 14-7

V

- value-only input 25-9
- variable types supported for code generation
 - from MATLAB 14-18
- variables
 - eliminating redundant copies in C/C++ code generated from MATLAB 14-7

Variables

- defining by assignment for code generation
 - from MATLAB 14-3
- defining for code generation from MATLAB 14-2

W

- when to disable run-time checks
 - code generation from MATLAB 8-72
- wrapping
 - fixed-point data types 1-5
- writing fixed-point data to workspace 9-2