

Fixed-Point Designer™

User's Guide



MATLAB®

R2017a

 MathWorks®

How to Contact MathWorks



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



Phone: 508-647-7000



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

Fixed-Point Designer™ User's Guide

© COPYRIGHT 2013–2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (R2013b)
March 2014	Online only	Revised for Version 4.2 (R2014a)
October 2014	Online Only	Revised for Version 4.3 (R2014b)
March 2015	Online Only	Revised for Version 5.0 (R2015a)
September 2015	Online Only	Revised for Version 5.1 (R2015b)
October 2015	Online only	Rereleased for Version 5.0.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 5.2 (R2016a)
September 2016	Online only	Revised for Version 5.3 (R2016b)
March 2017	Online only	Revised for Version 5.4 (R2017a)

Fixed-Point Designer for MATLAB Code

		Fixed-Point Concepts
1		
	Fixed-Point Data Types	1-2
	Scaling	1-4
	Compute Slope and Bias	1-5
	What Is Slope Bias Scaling?	1-5
	Compute Slope and Bias	1-5
	Precision and Range	1-8
	Range	1-8
	Precision	1-9
	Arithmetic Operations	1-13
	Modulo Arithmetic	1-13
	Two's Complement	1-14
	Addition and Subtraction	1-15
	Multiplication	1-16
	Casts	1-22
	fi Objects and C Integer Data Types	1-25
	Integer Data Types	1-25
	Unary Conversions	1-27
	Binary Conversions	1-28
	Overflow Handling	1-30

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	2-18
Data Properties	2-18
fimath Properties	2-18
numerictype Properties	2-20
Setting fi Object Properties	2-21
fi Object Functions	2-24

Fixed-Point Topics

3

Set Up Fixed-Point Objects	3-2
Create Fixed-Point Data	3-2
View Fixed-Point Number Circles	3-16
Perform Binary-Point Scaling	3-29
Develop Fixed-Point Algorithms	3-34
Calculate Fixed-Point Sine and Cosine	3-46
Calculate Fixed-Point Arctangent	3-69
Compute Sine and Cosine Using CORDIC Rotation Kernel	3-94
Perform QR Factorization Using CORDIC	3-99

Compute Square Root Using CORDIC	3-134
Convert Cartesian to Polar Using CORDIC Vectoring Kernel	3-144
Set Data Types Using Min/Max Instrumentation	3-149
Convert Fast Fourier Transform (FFT) to Fixed Point	3-163
Detect Limit Cycles in Fixed-Point State-Space Systems	3-183
Compute Quantization Error	3-195
Normalize Data for Lookup Tables	3-204
Implement Fixed-Point Log2 Using Lookup Table . .	3-210
Implement Fixed-Point Square Root Using Lookup Table	3-215
Set Fixed-Point Math Attributes	3-220

Working with fimath Objects

4

fimath Object Construction	4-2
fimath Object Syntaxes	4-2
Building fimath Object Constructors in a GUI	4-3
fimath Object Properties	4-5
Math, Rounding, and Overflow Properties	4-5
How Properties are Related	4-9
Setting fimath Object Properties	4-11
fimath Properties Usage for Fixed-Point Arithmetic .	4-13
fimath Rules for Fixed-Point Arithmetic	4-13
Binary-Point Arithmetic	4-15
[Slope Bias] Arithmetic	4-18

fimath for Rounding and Overflow Modes	4-21
fimath for Sharing Arithmetic Rules	4-23
Default fimath Usage to Share Arithmetic Rules	4-23
Local fimath Usage to Share Arithmetic Rules	4-23
fimath ProductMode and SumMode	4-26
Example Setup	4-26
FullPrecision	4-27
KeepLSB	4-28
KeepMSB	4-29
SpecifyPrecision	4-30
How Functions Use fimath	4-32
Functions that use then discard attached fimath	4-32
Functions that ignore and discard attached fimath ...	4-32
Functions that do not perform math	4-33

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	6-4
numerictype Object Properties	6-6
Data Type and Scaling Properties	6-6
How Properties are Related	6-9
Set numerictype Object Properties	6-10
numerictype of Fixed-Point Objects	6-12
Valid Values for numerictype Object Properties	6-12
Properties That Affect the Slope	6-14
Stored Integer Value and Real World Value	6-14
numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects	6-15
Example 1	6-15
Example 2	6-16

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

Fixed-Point Conversion Workflows	8-2
Choosing a Conversion Workflow	8-2
Automated Workflow	8-3
Manual Workflow	8-3
Automated Fixed-Point Conversion	8-4
Automated Fixed-Point Conversion Capabilities	8-4
Code Coverage	8-5
Proposing Data Types	8-9
Locking Proposed Data Types	8-11
Viewing Functions	8-11
Viewing Variables	8-18
Log Data for Histogram	8-20
Function Replacements	8-22
Validating Types	8-23
Testing Numerics	8-23
Detecting Overflows	8-24
Best Practices for Debugging Out-of-the-Box Conversion by the Fixed-Point Converter	8-25
Prerequisites	8-25
Convert to Fixed Point Using Default Configuration ..	8-30
Determine Where Numerical Issues Originated	8-33
Adjust <code>fimath</code> Settings	8-35
Convert <code>kalman_filter</code> Function Using New <code>fimath</code> Settings	8-36
Adjust Word Length Settings	8-38
Replace Constant Functions	8-39
MATLAB Language Features Supported for Automated Fixed-Point Conversion	8-42
MATLAB Language Features Supported for Automated Fixed-Point Conversion	8-42
MATLAB Language Features Not Supported for Automated Fixed-Point Conversion	8-44
Generated Fixed-Point Code	8-45
Location of Generated Fixed-Point Files	8-45
Minimizing <code>fi</code> -casts to Improve Code Readability	8-46
Avoiding Overflows in the Generated Fixed-Point Code	8-46

Controlling Bit Growth	8-47
Avoiding Loss of Range or Precision	8-47
Handling Non-Constant mpower Exponents	8-49
Fixed-Point Code for MATLAB Classes	8-51
Automated Conversion Support for MATLAB Classes	8-51
Unsupported Constructs	8-51
Coding Style Best Practices	8-52
Automated Fixed-Point Conversion Best Practices	8-54
Create a Test File	8-54
Prepare Your Algorithm for Code Acceleration or Code Generation	8-55
Check for Fixed-Point Support for Functions Used in Your Algorithm	8-56
Manage Data Types and Control Bit Growth	8-56
Convert to Fixed Point	8-57
Use the Histogram to Fine-Tune Data Type Settings	8-58
Optimize Your Algorithm	8-59
Avoid Explicit Double and Single Casts	8-61
Replacing Functions Using Lookup Table Approximations	8-62
Custom Plot Functions	8-63
Generate Fixed-Point MATLAB Code for Multiple Entry- Point Functions	8-65
Convert Code Containing Global Data to Fixed Point	8-69
Workflow	8-69
Declare Global Variables	8-69
Define Global Data	8-70
Define Constant Global Data	8-71
Limitations of Using Global Data	8-73
Convert Code Containing Global Variables to Fixed- Point	8-74
Convert Code Containing Structures to Fixed Point	8-78

Data Type Issues in Generated Code	8-81
Enable the Highlight Option in the Fixed-Point Converter App	8-81
Enable the Highlight Option at the Command Line . . .	8-81
Stowaway Doubles	8-81
Stowaway Singles	8-82
Expensive Fixed-Point Operations	8-82
System Objects Supported by Fixed-Point Converter App	8-84
Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App	8-85
Create DSP Filter Function and Test Bench	8-85
Convert the Function to Fixed-Point	8-86

Automated Conversion Using Fixed-Point Converter App

9

Specify Type Proposal Options	9-2
Detect Overflows	9-6
Propose Data Types Based on Simulation Ranges	9-15
Propose Data Types Based on Derived Ranges	9-29
View and Modify Variable Information	9-43
View Variable Information	9-43
Modify Variable Information	9-43
Revert Changes	9-45
Promote Sim Min and Sim Max Values	9-46
Replace the exp Function with a Lookup Table	9-47
Convert Fixed-Point Conversion Project to MATLAB Scripts	9-55
Replace a Custom Function with a Lookup Table	9-57

Visualize Differences Between Floating-Point and Fixed-Point Results	9-65
Enable Plotting Using the Simulation Data Inspector	9-76
Add Global Variables by Using the App	9-77
Automatically Define Input Types by Using the App .	9-78
Define Constant Input Parameters Using the App ...	9-79
Define or Edit Input Parameter Type by Using the App	9-80
Define or Edit an Input Parameter Type	9-80
Specify an Enumerated Type Input Parameter by Type	9-81
Specify a Fixed-Point Input Parameter by Type	9-82
Specify a Structure Input Parameter	9-82
Specify a Cell Array Input Parameter	9-83
Define Input Parameter by Example by Using the App	9-88
Define an Input Parameter by Example	9-88
Specify Input Parameters by Example	9-89
Specify a Structure Type Input Parameter by Example	9-90
Specify a Cell Array Type Input Parameter by Example	9-91
Specify an Enumerated Type Input Parameter by Example	9-93
Specify a Fixed-Point Input Parameter by Example ...	9-94
Specify Global Variable Type and Initial Value Using the App	9-96
Why Specify a Type Definition for Global Variables? ..	9-96
Specify a Global Variable Type	9-96
Define a Global Variable by Example	9-96
Define or Edit Global Variable Type	9-97
Define Global Variable Initial Value	9-98
Define Global Variable Constant Value	9-99
Remove Global Variables	9-99
Specify Properties of Entry-Point Function Inputs Using the App	9-100
Why Specify Input Properties?	9-100
Specify an Input Definition Using the App	9-100

Detect Dead and Constant-Folded Code	9-101
What Is Dead Code?	9-101
Detect Dead Code	9-102
Fix Dead Code	9-104

Automated Conversion Using Programmatic Workflow

10

Propose Data Types Based on Simulation Ranges	10-2
Propose Data Types Based on Derived Ranges	10-7
Detect Overflows	10-14
Replace the exp Function with a Lookup Table	10-19
Replace a Custom Function with a Lookup Table ...	10-21
Visualize Differences Between Floating-Point and Fixed-Point Results	10-23
Enable Plotting Using the Simulation Data Inspector	10-29

Single-Precision Conversion

11

Generate Single-Precision MATLAB Code	11-2
Prerequisites	11-2
Create a Folder and Copy Relevant Files	11-2
Set Up the Single-Precision Configuration Object	11-4
Generate Single-Precision MATLAB Code	11-4
View the Type Proposal Report	11-5
View Generated Single-Precision MATLAB Code	11-6
View Potential Data Type Issues	11-6
Compare the Double-Precision and Single-Precision Variables	11-6

MATLAB Language Features Supported for Single-Precision Conversion	11-10
MATLAB Language Features Supported for Single-Precision Conversion	11-10
MATLAB Language Features Not Supported for Single-Precision Conversion	11-11
Single-Precision Conversion Best Practices	11-12
Use Integers for Index Variables	11-12
Limit Use of assert Statements	11-12
Initialize MATLAB Class Properties in Constructor ..	11-12
Provide a Test File That Calls Your MATLAB Function	11-12
Prepare Your Code for Code Generation	11-13
Use the -args Option to Specify Input Properties ...	11-13
Test Numerics and Log I/O Data	11-13

12 Fixed-Point Conversion — Manual Conversion

Manual Fixed-Point Conversion Workflow	12-2
Manual Fixed-Point Conversion Best Practices	12-4
Create a Test File	12-4
Prepare Your Algorithm for Code Acceleration or Code Generation	12-5
Check for Fixed-Point Support for Functions Used in Your Algorithm	12-6
Manage Data Types and Control Bit Growth	12-7
Separate Data Type Definitions from Algorithm	12-8
Convert to Fixed Point	12-10
Optimize Data Types	12-12
Optimize Your Algorithm	12-15
Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros	12-19
Fixed-Point Design Exploration in Parallel	12-44
Overview	12-44
Setup	12-44

Using Parallel for-Loops For Design Exploration	12-44
Description of System Under Test	12-46

Viewing Test Results With Simulation Data Inspector

13

Inspecting Data Using the Simulation Data Inspector	13-2
What Is the Simulation Data Inspector?	13-2
Import Logged Data	13-2
Export Logged Data	13-2
Group Signals	13-3
Run Options	13-3
Create Report	13-3
Comparison Options	13-3
Enabling Plotting Using the Simulation Data Inspector	13-3
Save and Load Simulation Data Inspector Sessions . . .	13-4

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

14

Code Acceleration and Code Generation from MATLAB	14-3
Requirements for Generating Compiled C Code Files	14-4
Functions Supported for Code Acceleration or C Code Generation	14-5
Workflow for Fixed-Point Code Acceleration and Generation	14-11
Set Up C Compiler	14-12

Accelerate Code Using fiaccel	14-13
Speeding Up Fixed-Point Execution with fiaccel	14-13
Running fiaccel	14-13
Generated Files and Locations	14-14
Data Type Override Using fiaccel	14-16
Specifying Default fimath Values for MEX Functions	14-17
File Infrastructure and Paths Setup	14-19
Compile Path Search Order	14-19
Naming Conventions	14-19
Detect and Debug Code Generation Errors	14-23
Debugging Strategies	14-23
Error Detection at Design Time	14-24
Error Detection at Compile Time	14-24
Set Up C Code Compilation Options	14-25
C Code Compiler Configuration Object	14-25
Compilation Options Modification at the Command Line	
Using Dot Notation	14-25
How fiaccel Resolves Conflicting Options	14-26
MEX Configuration Dialog Box Options	14-27
See Also	14-31
Best Practices for Accelerating Fixed-Point Code ...	14-32
Recommended Compilation Options for fiaccel	14-32
Build Scripts	14-33
Check Code Interactively Using MATLAB Code	
Analyzer	14-34
Separating Your Test Bench from Your Function Code	14-34
Preserving Your Code	14-34
File Naming Conventions	14-35
Use Fixed-Point Code Generation Reports	14-36
Code Generation Report Creation	14-36
Code Generation Report Opening	14-37
Viewing Your MATLAB Code	14-37
Viewing Variables in the Variables Tab	14-39
See Also	14-40
Generate C Code from Code Containing Global Data	14-41
Workflow Overview	14-41

Declaring Global Variables	14-41
Defining Global Data	14-42
Synchronizing Global Data with MATLAB	14-43
Limitations of Using Global Data	14-45
Define Input Properties Programmatically in MATLAB	
File	14-46
How to Use <code>assert</code>	14-46
Rules for Using <code>assert</code> Function	14-50
Specifying Properties of Primary Fixed-Point Inputs	14-51
Specifying Properties of Cell Arrays	14-51
Specifying Class and Size of Scalar Structure	14-53
Specifying Class and Size of Structure Array	14-54
Specify Cell Array Inputs at the Command Line	14-55
Specify Cell Array Inputs by Example	14-55
Specify the Type of the Cell Array Input	14-56
Make a Homogeneous Copy of a Type	14-56
Make a Heterogeneous Copy of a Type	14-57
Specify Variable-Size Cell Array Inputs	14-58
Specify Type Name for Heterogeneous Cell Array Inputs	14-60
Specify Constant Cell Array Inputs	14-60
Specify Global Cell Arrays at the Command Line	14-62
Control Run-Time Checks	14-64
Types of Run-Time Checks	14-64
When to Disable Run-Time Checks	14-64
How to Disable Run-Time Checks	14-65
Fix Run-Time Stack Overflows	14-66
Code Generation with MATLAB Coder	14-67
Code Generation with MATLAB Function Block	14-68
Composing a MATLAB Language Function in a Simulink Model	14-68
MATLAB Function Block with Data Type Override	14-68
Fixed-Point Data Types with MATLAB Function Block	14-70

Generate Fixed-Point FIR Code Using MATLAB	
Function Block	14-77
Program the MATLAB Function Block	14-77
Prepare the Inputs	14-78
Create the Model	14-78
Define the fimath Object Using the Model Explorer . .	14-80
Run the Simulation	14-80
Fixed-Point FIR Code Example Parameter Values . .	14-81
Accelerate Code for Variable-Size Data	14-83
Disable Support for Variable-Size Data	14-83
Control Dynamic Memory Allocation	14-83
Accelerate Code for MATLAB Functions with Variable- Size Data	14-84
Accelerate Code for a MATLAB Function That Expands a Vector in a Loop	14-86
Accelerate Fixed-Point Simulation	14-93
Code Generation Readiness Tool	14-96
Summary Tab	14-97
Code Structure Tab	14-99
Check Code Using the Code Generation Readiness Tool	14-102
Run Code Generation Readiness Tool at the Command Line	14-102
Run the Code Generation Readiness Tool From the Current Folder Browser	14-102
See Also	14-102
Check Code Using the MATLAB Code Analyzer	14-103
Fix Errors Detected at Code Generation Time	14-104
See Also	14-104
Avoid Multiword Operations in Generated Code . . .	14-105
Find Potential Data Type Issues in Generated Code	14-108
Data Type Issues Overview	14-108
Enable Highlighting of Potential Data Type Issues .	14-108
Find and Address Cumbersome Operations	14-109

Find and Address Expensive Rounding	14-110
Find and Address Expensive Comparison Operations	14-112
Find and Address Multiword Operations	14-113

Interoperability with Other Products

15

fi Objects with Simulink	15-2
View and Edit fi objects in Model Explorer	15-2
Reading Fixed-Point Data from the Workspace	15-3
Writing Fixed-Point Data to the Workspace	15-4
Setting the Value and Data Type of Block Parameters	15-8
Logging Fixed-Point Signals	15-8
Accessing Fixed-Point Block Data During Simulation	15-8
fi Objects with DSP System Toolbox	15-10
Reading Fixed-Point Signals from the Workspace	15-10
Writing Fixed-Point Signals to the Workspace	15-10
Ways to Generate Code	15-15

Calling Functions for Code Generation

16

Resolution of Function Calls for Code Generation	16-2
Key Points About Resolving Function Calls	16-4
Compile Path Search Order	16-4
When to Use the Code Generation Path	16-5
Resolution of File Types on Code Generation Path	16-6
Compilation Directive %#codegen	16-8
Extrinsic Functions	16-9
Declaring MATLAB Functions as Extrinsic Functions	16-10
Calling MATLAB Functions Using feval	16-14
Resolution of Extrinsic Functions During Simulation	16-14

Working with mxArray	16-15
Restrictions on Extrinsic Functions for Code Generation	16-17
Limit on Function Arguments	16-17
Code Generation for Recursive Functions	16-18
Compile-Time Recursion	16-18
Run-Time Recursion	16-19
Disallow Recursion	16-19
Disable Run-Time Recursion	16-19
Recursive Function Limitations for Code Generation	16-20
Force Code Generator to Use Run-Time Recursion	16-21
Treat the Input to the Recursive Function as a Nonconstant	16-21
Make the Input to the Recursive Function Variable- Size	16-22
Assign Output Variable Before the Recursive Call	16-23

Code Generation for MATLAB Classes

17

MATLAB Classes Definition for Code Generation	17-2
Language Limitations	17-2
Code Generation Features Not Compatible with Classes	17-3
Defining Class Properties for Code Generation	17-4
Calls to Base Class Constructor	17-6
Inheritance from Built-In MATLAB Classes Not Supported	17-7
Classes That Support Code Generation	17-8
Generate Code for MATLAB Value Classes	17-9
Generate Code for MATLAB Handle Classes and System Objects	17-14
MATLAB Classes in Code Generation Reports	17-17
What Reports Tell You About Classes	17-17

How Classes Appear in Code Generation Reports . . .	17-17
Troubleshooting Code Generation Issues with MATLAB	
Classes	17-20
Class class does not have a property with name name	17-20
Handle Object Limitations for Code Generation	17-22
A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop	17-22
A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object	17-22
System Objects in MATLAB Code Generation	17-25
Usage Rules and Limitations for System Objects for Generating Code	17-25
System Objects in codegen	17-28
System Objects in the MATLAB Function Block	17-28
System Objects in the MATLAB System Block	17-28
System Objects and MATLAB Compiler Software	17-28
Specify Objects as Inputs	17-29
Consistency Between coder.ClassType Object and Class Definition File	17-31
Limitations for Using Objects as Entry-Point Function Inputs	17-31

Defining Data for Code Generation

18

Data Definition for Code Generation	18-2
Code Generation for Complex Data	18-4
Restrictions When Defining Complex Variables	18-4
Code Generation for Complex Data with Zero-Valued Imaginary Parts	18-4
Results of Expressions That Have Complex Operands	18-8
Code Generation for Character Arrays	18-9

Array Size Restrictions for Code Generation	18-10
See Also	18-10
Code Generation for Constants in Structures and Arrays	18-11

Defining Functions for Code Generation

19

Code Generation for Variable Length Argument Lists	19-2
Specify Number of Entry-Point Function Input or Output Arguments to Generate	19-3
Control Number of Input Arguments	19-3
Control the Number of Output Arguments	19-4
Code Generation for Anonymous Functions	19-6
Anonymous Function Limitations for Code Generation	19-6
Code Generation for Nested Functions	19-7
Nested Function Limitations for Code Generation	19-7

Defining MATLAB Variables for C/C++ Code Generation

20

Variables Definition for Code Generation	20-2
Best Practices for Defining Variables for C/C++ Code Generation	20-3
Define Variables By Assignment Before Using Them . .	20-3
Use Caution When Reassigning Variables	20-5
Use Type Cast Operators in Variable Definitions	20-5
Define Matrices Before Assigning Indexed Variables . .	20-6

Eliminate Redundant Copies of Variables in Generated Code	20-7
When Redundant Copies Occur	20-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables	20-7
Defining Uninitialized Variables	20-8
Reassignment of Variable Properties	20-9
Reuse the Same Variable with Different Properties .	20-10
When You Can Reuse the Same Variable with Different Properties	20-10
When You Cannot Reuse Variables	20-10
Limitations of Variable Reuse	20-13
Avoid Overflows in for-Loops	20-14
Supported Variable Types	20-16

Design Considerations for C/C++ Code Generation

21

When to Generate Code from MATLAB Algorithms ...	21-2
When Not to Generate Code from MATLAB Algorithms	21-2
Which Code Generation Feature to Use	21-4
Prerequisites for C/C++ Code Generation from MATLAB	21-6
MATLAB Code Design Considerations for Code Generation	21-7
See Also	21-8
Differences Between Generated Code and MATLAB Code	21-9
Character Size	21-9
Order of Evaluation in Expressions	21-9

Termination Behavior	21-11
Size of Variable-Size N-D Arrays	21-11
Size of Empty Arrays	21-11
Size of Empty Array That Results from Deleting Elements of an Array	21-11
Floating-Point Numerical Results	21-12
NaN and Infinity Patterns	21-13
Code Generation Target	21-13
MATLAB Class Initial Values	21-13
Variable-Size Data	21-13
Complex Numbers	21-13
Potential Differences Reporting	21-14
Addressing Potential Differences Messages	21-14
Disabling and Enabling Potential Differences Reporting	21-14
Potential Differences Messages	21-16
Automatic Dimension Incompatibility	21-16
mtimes No Dynamic Scalar Expansion	21-17
Matrix-Matrix Indexing	21-17
Vector-Vector Indexing	21-18
Size Mismatch	21-18
MATLAB Language Features Supported for C/C++ Code Generation	21-21
MATLAB Features That Code Generation Supports .	21-21
MATLAB Language Features That Code Generation Does Not Support	21-22

22

Code Generation for Enumerated Data

Code Generation for Enumerations	22-2
Define Enumerations for Code Generation	22-2
Use Allowed Operations on Enumerations	22-4
Use MATLAB Toolbox Functions That Support Enumerations	22-4

Customize Enumerated Types in Generated Code . . .	22-6
Specify a Default Enumeration Value	22-7
Specify a Header File	22-8
Include Class Name Prefix in Generated Enumerated Type Value Names	22-8

Code Generation for Function Handles

23

Function Handle Limitations for Code Generation . . .	23-2
--	-------------

Generate Efficient and Reusable Code

24

Modularize MATLAB Code	24-2
Eliminate Redundant Copies of Function Inputs	24-3
Inline Code	24-6
Prevent Function Inlining	24-6
Use Inlining in Control Flow Statements	24-6
Control Inlining	24-8
Control Size of Functions Inlined	24-8
Control Size of Functions After Inlining	24-9
Control Stack Size Limit on Inlined Functions	24-9
Fold Function Calls into Constants	24-11
Control Stack Space Usage	24-13
Stack Allocation and Performance	24-14
Allocate Heap Space from Command Line	24-14
Allocate Heap Space Using the MATLAB Coder App	24-14
Dynamic Memory Allocation and Performance	24-15
When Dynamic Memory Allocation Occurs	24-15

Minimize Dynamic Memory Allocation	24-16
Provide Maximum Size for Variable-Size Arrays	24-17
Disable Dynamic Memory Allocation During Code Generation	24-23
Set Dynamic Memory Allocation Threshold	24-24
Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App	24-24
Set Dynamic Memory Allocation Threshold at the Command Line	24-25
Excluding Unused Paths from Generated Code	24-26
Prevent Code Generation for Unused Execution Paths	24-27
Prevent Code Generation When Local Variable Controls Flow	24-27
Prevent Code Generation When Input Variable Controls Flow	24-28
Generate Code with Parallel for-Loops (parfor)	24-29
Minimize Redundant Operations in Loops	24-31
Unroll for-Loops	24-33
Disable Support for Integer Overflow or Non-Finites	24-35
Disable Support for Integer Overflow	24-35
Disable Support for Non-Finite Numbers	24-36
Integrate External/Custom Code	24-37
MATLAB Coder Optimizations in Generated Code . .	24-43
Constant Folding	24-43
Loop Fusion	24-44
Successive Matrix Operations Combined	24-44
Unreachable Code Elimination	24-45
Generate Reusable Code	24-46

Code Generation for MATLAB Structures

25

Structure Definition for Code Generation	25-2
Structure Operations Allowed for Code Generation . .	25-3
Define Scalar Structures for Code Generation	25-4
Restrictions When Defining Scalar Structures by Assignment	25-4
Adding Fields in Consistent Order on Each Control Flow Path	25-4
Restriction on Adding New Fields After First Use	25-5
Define Arrays of Structures for Code Generation	25-6
Ensuring Consistency of Fields	25-6
Using repmat to Define an Array of Structures with Consistent Field Properties	25-6
Defining an Array of Structures by Using struct	25-7
Defining an Array of Structures Using Concatenation .	25-7
Index Substructures and Fields	25-8
Assign Values to Structures and Fields	25-10
Pass Large Structures as Input Parameters	25-12

Functions, Classes, and System Objects Supported for Code Generation

26

Functions and Objects Supported for C/C++ Code Generation — Alphabetical List	26-2
Functions and Objects Supported for C/C++ Code Generation — Category List	26-67
Aerospace Toolbox	26-69
Arithmetic Operations in MATLAB	26-69
Audio System Toolbox	26-70

Automated Driving System Toolbox	26-71
Bit-Wise Operations MATLAB	26-72
Casting in MATLAB	26-73
Character Functions in MATLAB	26-73
Communications System Toolbox	26-74
Complex Numbers in MATLAB	26-80
Computer Vision System Toolbox	26-81
Control Flow in MATLAB	26-85
Control System Toolbox	26-85
Data and File Management in MATLAB	26-85
Data Types in MATLAB	26-86
Desktop Environment in MATLAB	26-87
Discrete Math in MATLAB	26-87
DSP System Toolbox	26-87
Error Handling in MATLAB	26-93
Exponents in MATLAB	26-93
Filtering and Convolution in MATLAB	26-94
Fixed-Point Designer	26-94
Histograms in MATLAB	26-100
Image Acquisition Toolbox	26-100
Image Processing in MATLAB	26-101
Image Processing Toolbox	26-101
Input and Output Arguments in MATLAB	26-105
Interpolation and Computational Geometry in MATLAB	26-105
Linear Algebra in MATLAB	26-106
Logical and Bit-Wise Operations in MATLAB	26-107
MATLAB Compiler	26-107
Matrices and Arrays in MATLAB	26-108
Neural Network Toolbox	26-111
Numerical Integration and Differentiation in MATLAB	26-111
Optimization Functions in MATLAB	26-112
Optimization Toolbox	26-112
Phased Array System Toolbox	26-113
Polynomials in MATLAB	26-119
Programming Utilities in MATLAB	26-120
Relational Operators in MATLAB	26-120
Robotics System Toolbox	26-120
Rounding and Remainder Functions in MATLAB ..	26-122
Set Operations in MATLAB	26-122
Signal Processing in MATLAB	26-123
Signal Processing Toolbox	26-123
Special Values in MATLAB	26-126

Specialized Math in MATLAB	26-127
Statistics in MATLAB	26-127
Statistics and Machine Learning Toolbox	26-128
System Identification Toolbox	26-134
System object Methods	26-135
Trigonometry in MATLAB	26-135
Wavelet Toolbox	26-137
WLAN System Toolbox	26-138

Code Generation for Variable-Size Data

27

Code Generation for Variable-Size Arrays	27-2
Memory Allocation for Variable-Size Arrays	27-3
Enabling and Disabling Support for Variable-Size Arrays	27-3
Variable-Size Arrays in a Code Generation Report ...	27-3
Control Memory Allocation for Variable-Size Arrays .	27-5
Provide Upper Bounds for Variable-Size Arrays	27-5
Disable Dynamic Memory Allocation	27-5
Configure Code Generator to Use Dynamic Memory Allocation for Arrays Bigger Than a Threshold	27-6
Specify Upper Bounds for Variable-Size Arrays	27-7
Specify Upper Bounds for Variable-Size Inputs	27-7
Specify Upper Bounds for Local Variables	27-7
Define Variable-Size Data for Code Generation	27-9
Use a Matrix Constructor with Nonconstant Dimensions	27-9
Assign Multiple Sizes to the Same Variable	27-10
Define Variable-Size Data Explicitly by Using coder.varsize	27-10
C Code Interface for Arrays	27-15
C Code Interface for Statically Allocated Arrays	27-15
C Code Interface for Dynamically Allocated Arrays ..	27-16
Utility Functions for Creating emxArray Data Structures	27-18

Diagnose and Fix Variable-Size Data Errors	27-20
Diagnosing and Fixing Size Mismatch Errors	27-20
Diagnosing and Fixing Errors in Detecting Upper Bounds	27-22
Incompatibilities with MATLAB in Variable-Size Support for Code Generation	27-24
Incompatibility with MATLAB for Scalar Expansion .	27-24
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	27-26
Incompatibility with MATLAB in Determining Size of Empty Arrays	27-27
Incompatibility with MATLAB in Determining Class of Empty Arrays	27-28
Incompatibility with MATLAB in Matrix-Matrix Indexing	27-29
Incompatibility with MATLAB in Vector-Vector Indexing	27-29
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation	27-30
Incompatibility with MATLAB in Concatenating Variable- Size Matrices	27-31
Differences When Curly-Brace Indexing of Variable- Size Cell Array Inside Concatenation Returns No Elements	27-31
Variable-Sizing Restrictions for Code Generation of Toolbox Functions	27-33
Common Restrictions	27-33
Toolbox Functions with Restrictions for Variable-Size Data	27-34

Code Generation for Cell Arrays

Code Generation for Cell Arrays	28-2
Homogeneous vs. Heterogeneous Cell Arrays	28-2
Controlling Whether a Cell Array Is Homogeneous or Heterogeneous	28-3
Cell Arrays in Reports	28-3

Control Whether a Cell Array Is Variable-Size	28-5
Define Cell Array Inputs	28-8
Cell Array Limitations for Code Generation	28-9
Cell Array Element Assignment	28-9
Definition of Variable-Size Cell Array by Using cell . .	28-10
Cell Array Indexing	28-13
Growing a Cell Array by Using {end + 1}	28-14
Variable-Size Cell Arrays	28-15
Cell Array Contents	28-16
Cell Arrays in Structures	28-16
Passing Cell Arrays to External C/C++ Functions . . .	28-16

Primary Functions

29

Specify Properties of Entry-Point Function Inputs . . .	29-2
Why You Must Specify Input Properties	29-2
Properties to Specify	29-2
Rules for Specifying Properties of Primary Inputs	29-4
Methods for Defining Properties of Primary Inputs . . .	29-4
Define Input Properties by Example at the Command Line	29-5
Specify Constant Inputs at the Command Line	29-7
Specify Variable-Size Inputs at the Command Line . . .	29-9
Define Input Properties Programmatically in the MATLAB File	29-12
How to Use assert with MATLAB Coder	29-12
Rules for Using assert Function	29-18
Specifying General Properties of Primary Inputs	29-19
Specifying Properties of Primary Fixed-Point Inputs .	29-20
Specifying Properties of Cell Arrays	29-20
Specifying Class and Size of Scalar Structure	29-22
Specifying Class and Size of Structure Array	29-23

30 System Objects Supported for Code Generation

Code Generation for System Objects	30-2
--	------

Fixed-Point Designer for Simulink Models

31 Getting Started

Installation	31-2
Sharing Fixed-Point Models	31-3
Physical Quantities and Measurement Scales	31-4
Introduction	31-4
Selecting a Measurement Scale	31-5
Select a Measurement Scale for Temperature	31-6
Why Use Fixed-Point Hardware?	31-12
Why Use the Fixed-Point Designer Software?	31-14
Developing and Testing Fixed-Point Systems	31-15
Supported Data Types	31-18
Configure Blocks with Fixed-Point Output	31-19
Specify the Output Data Type and Scaling	31-20
Specify Fixed-Point Data Types with the Data Type Assistant	31-22
Rounding	31-25
Overflow Handling	31-25
Lock the Output Data Type Setting	31-26
Real-World Values Versus Stored Integer Values ...	31-26

Configure Blocks with Fixed-Point Parameters	31-29
Specify Fixed-Point Values Directly	31-29
Specify Fixed-Point Values Via Parameter Objects . .	31-30
Pass Fixed-Point Data Between Simulink Models and MATLAB	31-32
Read Fixed-Point Data from the Workspace	31-32
Write Fixed-Point Data to the Workspace	31-32
Log Fixed-Point Signals	31-35
Access Fixed-Point Block Data During Simulation . . .	31-35
Cast from Doubles to Fixed Point	31-36
About This Example	31-36
Block Descriptions	31-37
Simulations	31-38

Data Types and Scaling

32

Data Types and Scaling in Digital Hardware	32-2
Binary Point Interpretation	32-2
Signed Fixed-Point Numbers	32-4
Floating-Point Data Types	32-4
Scaling	32-6
Binary-Point-Only Scaling	32-7
Slope and Bias Scaling	32-7
Unspecified Scaling	32-7
Quantization	32-9
Fixed-Point Format	32-9
Range and Precision	32-11
Range	32-11
Precision	32-12
Fixed-Point Data Type Parameters	32-12
Range of an 8-Bit Fixed-Point Data Type — Binary-Point-Only Scaling	32-12
Range of an 8-Bit Fixed-Point Data Type — Slope and Bias Scaling	32-13

Fixed-Point Numbers in Simulink	32-15
Constant Scaling for Best Precision	32-15
Fixed-Point Data Type and Scaling Notation	32-17
Display Port Data Types	32-20
Scaled Doubles	32-22
What Are Scaled Doubles?	32-22
When to Use Scaled Doubles	32-23
Use Scaled Doubles to Avoid Precision Loss	32-24
About the Model	32-24
Running the Example	32-25
Floating-Point Numbers	32-26
Floating-Point Numbers	32-26
Scientific Notation	32-26
The IEEE Format	32-27
Range and Precision	32-29
Exceptional Arithmetic	32-31

Arithmetic Operations

33

Fixed-Point Arithmetic Operations	33-3
Limitations on Precision	33-4
Rounding	33-5
Choose a Rounding Mode	33-5
Rounding Modes for Fixed-Point Simulink Blocks ...	33-6
Rounding Mode: Ceiling	33-8
Rounding Mode: Convergent	33-9
Rounding Mode: Floor	33-11
Rounding Mode: Nearest	33-12

Rounding Mode: Round	33-14
Rounding Mode: Simplest	33-16
Optimize Rounding for Casts	33-16
Optimize Rounding for High-Level Arithmetic Operations	33-16
Optimize Rounding for Intermediate Arithmetic Operations	33-18
Rounding Mode: Zero	33-20
Rounding to Zero Versus Truncation	33-22
Pad with Trailing Zeros	33-23
Limitations on Precision and Errors	33-24
Maximize Precision	33-25
Net Slope and Net Bias Precision	33-26
What are Net Slope and Net Bias?	33-26
Detecting Net Slope and Net Bias Precision Issues ..	33-27
Fixed-Point Constant Underflow	33-27
Fixed-Point Constant Overflow	33-27
Fixed-Point Constant Precision Loss	33-28
Detect Net Slope and Bias Precision Issues	33-29
Detect Fixed-Point Constant Precision Loss	33-30
Limitations on Range	33-31
Saturation and Wrapping	33-33
What Are Saturation and Wrapping?	33-33
Saturation and Wrapping	33-33
Guard Bits	33-36
Determine the Range of Fixed-Point Numbers	33-37
Handle Overflows in Simulink Models	33-38
Recommendations for Arithmetic and Scaling	33-40
Arithmetic Operations and Fixed-Point Scaling	33-40

Addition	33-41
Accumulation	33-44
Multiplication	33-44
Gain	33-47
Division	33-49
Summary	33-51
Parameter and Signal Conversions	33-52
Introduction	33-52
Parameter Conversions	33-53
Signal Conversions	33-53
Rules for Arithmetic Operations	33-56
Computational Units	33-56
Addition and Subtraction	33-56
Multiplication	33-58
Division	33-63
Shifts	33-64
The Summation Process	33-66
The Multiplication Process	33-69
The Division Process	33-71
Shifts	33-72
Shifting Bits to the Right	33-72
Conversions and Arithmetic Operations	33-74

Realization Structures

34

Realizing Fixed-Point Digital Filters	34-2
Introduction	34-2
Realizations and Data Types	34-2
Targeting an Embedded Processor	34-4
Introduction	34-4
Size Assumptions	34-4

Operation Assumptions	34-4
Design Rules	34-5
Canonical Forms	34-7
Direct Form II	34-9
Series Cascade Form	34-13
Parallel Form	34-16

Fixed-Point Advisor

35

Preparation for Fixed-Point Conversion	35-2
Introduction	35-2
Best Practices	35-2
Run the Fixed-Point Advisor	35-4
Fix a Task Failure	35-5
Manually Fixing Failures	35-5
Automatically Fixing Failures	35-6
Batch Fixing Failures	35-6
Restore Points	35-7
Save a Restore Point	35-7
Load a Restore Point	35-8

Fixed-Point Tool

36

Fixed-Point Tool	36-2
Introduction to the Fixed-Point Tool	36-2
Using the Fixed-Point Tool	36-2
Run Management	36-4
Run Management	36-4
Run Management with the Shortcut Editor	36-4
Manual Run Management	36-5

Use Shortcuts to Manage Runs	36-6
Why Use Shortcuts to Manage Runs	36-6
When to Use Shortcuts to Manage Runs	36-6
Add Shortcuts	36-7
Edit Shortcuts	36-7
Delete Shortcuts	36-8
Capture Model Settings Using the Shortcut Editor ...	36-9
Debug a Fixed-Point Model	36-10
Simulating the Model to See the Initial Behavior ...	36-10
Debugging the Model	36-12
Simulating the Model Using a Different Input	
Stimulus	36-14
Debugging the Model with the New Input	36-14
Proposing Fraction Lengths for Math2 Based on	
Simulation Results	36-15
Verifying the New Settings	36-15
Logging Simulation Ranges for Referenced Models .	36-17
Viewing Simulation Ranges for Referenced Models . .	36-17
Fixed-Point Instrumentation and Data Type Override	
Settings	36-19
Log Simulation Ranges for Referenced Models	36-21
Propose Data Types for a Referenced Model	36-27
View Signal Names in Fixed-Point Tool	36-30
Model Multiple Data Type Behaviors Using a Data	
Dictionary	36-32
Change Data Types of Model Parameters	36-36

Convert Floating-Point Model to Fixed Point

Learning Objectives	37-2
Model Description	37-4
Model Overview	37-4

Model Set Up	37-5
Before You Begin	37-7
Convert Floating-Point Model to Fixed Point	37-8
Open the Model	37-8
Prepare Floating-Point Model for Conversion to Fixed Point	37-8
Propose Data Types	37-13
Apply Fixed-Point Data Types	37-15
Verify Fixed-Point Settings	37-15
Test Fixed-Point Settings With New Input Data	37-16
Gather a Floating-Point Benchmark	37-17
Propose Data Types for the New Input	37-18
Apply the New Fixed-Point Data Types	37-19
Verify New Fixed-Point Settings	37-19
Clean Up Model Settings	37-20
Key Points to Remember	37-20
Where to Learn More	37-21

Producing Lookup Table Data

38

Producing Lookup Table Data	38-2
Worst-Case Error for a Lookup Table	38-3
Approximate the Square Root Function	38-3
Create Lookup Tables for a Sine Function	38-5
Introduction	38-5
Parameters for <code>fixpt_look1_func_approx</code>	38-5
Setting Function Parameters for the Lookup Table ...	38-7
Using <code>errmax</code> with Unrestricted Spacing	38-7
Using <code>nptsmx</code> with Unrestricted Spacing	38-10
Using <code>errmax</code> with Even Spacing	38-12
Using <code>nptsmx</code> with Even Spacing	38-13
Using <code>errmax</code> with Power of Two Spacing	38-14
Using <code>nptsmx</code> with Power of Two Spacing	38-16
Specifying Both <code>errmax</code> and <code>nptsmx</code>	38-17
Comparison of Example Results	38-18

Use Lookup Table Approximation Functions	38-20
Effects of Spacing on Speed, Error, and Memory	
Usage	38-21
Criteria for Comparing Types of Breakpoint Spacing .	38-21
Model That Illustrates Effects of Breakpoint Spacing	38-21
Data ROM Required for Each Lookup Table	38-22
Determination of Out-of-Range Inputs	38-22
How the Lookup Tables Determine Input Location . .	38-23
Interpolation for Each Lookup Table	38-25
Summary of the Effects of Breakpoint Spacing	38-27

Automatic Data Typing

39

Choosing a Range Collection Method	39-2
Before Proposing Data Types for Your Model	39-5
Best Practices for Fixed-Point Workflow	39-7
Use a Known Working Simulink Model	39-7
Back Up Your Simulink Model	39-7
Capture the Current Data Type Override Settings . . .	39-7
Convert Individual Subsystems	39-7
Isolate the System Under Design	39-7
Do Not Use “Save as” on Referenced Models and MATLAB Function blocks	39-8
Use Lock Output Data Type Setting	39-8
Save Simulink Signal Objects	39-8
Test Update Diagram Failure	39-8
Disable Fast Restart	39-9
Models That Might Cause Data Type Propagation Errors	39-10
Autoscaling Using the Fixed-Point Tool	39-12
Workflow for Automatic Data Typing	39-12
Set Up the Model	39-14

Select System Under Design	39-16
Collect Ranges	39-17
Collect Ranges Using Simulation Data	39-17
Collect Ranges Using Range Analysis	39-18
Propose Data Types	39-19
Examine Results to Resolve Conflicts	39-22
Proposed Data Type Summary	39-24
Needs Attention	39-24
Range Information	39-25
Examine the Results and Resolve Conflicts	39-26
Apply Proposed Data Types	39-27
Verify New Settings	39-29
Clean Up Model Settings	39-30
Automatic Data Typing of Simulink Signal Objects .	39-31
Get Proposals for Results with Inherited Types	39-32
How to Get Proposals for Objects That Use an Inherited Output Data Type	39-32
When the Fixed-Point Tool Will Not Propose for Inherited Data Types	39-33
Propose Fraction Lengths	39-34
How to Propose Fraction Lengths	39-34
About the Feedback Controller Example Model	39-34
Propose Fraction Lengths Using Simulation Range Data	39-41
How the Fixed-Point Tool Proposes Word Lengths ..	39-50
How the Fixed-Point Tool Uses Range Information ..	39-50
How the Fixed-Point Tool Uses Target Hardware Information	39-51
Propose Word Lengths Based on Simulation Data ..	39-52
How to Propose Word Lengths	39-52
Propose Word Lengths Based on Simulation Data ...	39-52

Propose Data Types Using Multiple Simulations	39-58
About This Example	39-58
Running the Simulation	39-60
View Simulation Results	39-63
Compare Runs	39-63
Compare Signals	39-64
Inspect Signals	39-65
Histogram Plot of Signal	39-66
Viewing Results With the Simulation Data Inspector	39-70
Why Use the Simulation Data Inspector	39-70
When to Use the Simulation Data Inspector	39-70
What You Can Inspect in the Simulation Data Inspector	39-70
Fixed-Point Instrumentation and Data Type Override	39-72
Control Instrumentation Settings	39-72
Control Data Type Override	39-73

Range Analysis

40

How Range Analysis Works	40-2
Analyzing a Model with Range Analysis	40-2
Automatic Stubbing	40-5
Model Compatibility with Range Analysis	40-6
Derive Ranges	40-7
Derive Ranges at the Subsystem Level	40-9
Deriving Ranges at the Subsystem Level	40-9
Derive Ranges at the Subsystem Level	40-10
View Derived Ranges in the Fixed-Point Tool	40-11
Derive Ranges Using Design Ranges	40-13
Derive Ranges Using Block Initial Conditions	40-16

Derive Ranges for Simulink.Parameter Objects	40-19
Insufficient Design Range Information	40-22
Troubleshoot Range Analysis of System Objects	40-25
Providing More Design Range Information	40-28
Fixing Design Range Conflicts	40-31
Derive Ranges for a Referenced Model	40-34
Propose Data Types for a Referenced Model	40-40
Deriving Ranges for a Referenced Model	40-41
Viewing Derived Minimum and Maximum Values for Referenced Models	40-41
Data Type Override Settings	40-42
Intermediate Range Results	40-43
Unsupported Simulink Software Features	40-46
Simulink Blocks Supported for Range Analysis	40-48
Overview of Simulink Block Support	40-48
Limitations of Support for Model Blocks	40-58

Working with the MATLAB Function Block

41

Convert MATLAB Function Block to Fixed Point	41-2
Open the Model	41-2
Collect Range Information	41-3
Propose Data Types	41-4
Inspect Code Using the Code View	41-5
Apply Proposed Data Types	41-7
Verify Results	41-8
 Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool	 41-11

Replace Functions in a MATLAB Function Block with a Lookup Table	41-12
Derive Ranges of MATLAB Function Block Variables	41-16
Explore the Model	41-16
Derive Ranges	41-17
Propose Data Types	41-17
Apply Proposed Data Types	41-18
Convert a System Object to Fixed Point Using the Fixed-Point Tool	41-19
Using the Fixed-Point Tool with System Objects	41-19
Convert a System Object to Fixed-Point	41-20

Working with Data Objects in the Fixed-Point Workflow

42

Refine Data Types of a Model with Buses Using Simulation Data	42-2
Open and Simulate the Model	42-2
Use Data Type Override to Resolve Overflows	42-3
Propose New Fraction Lengths	42-5
Examine and Apply Proposed Types	42-6
Verify New Fixed-Point Settings	42-8
Save the Model and New Bus Data Types	42-9
Convert a Model with Buses to Fixed-Point Using Range Analysis	42-11
Open the Model	42-11
Set Design Minimums and Maximums	42-12
Open the Fixed-Point Tool and Select System for Conversion	42-14
Derive Minimum and Maximum Values	42-14
Propose Fraction Lengths	42-14
Examine and Apply Proposed Types	42-14
Save the Model and the New Bus Data Types	42-17

Bus Objects in the Fixed-Point Workflow	42-19
How Data Type Proposals Are Determined for Bus Objects	42-19
Bus Naming Conventions with Data Type Override ..	42-21
Limitations of Bus Objects in the Fixed-Point Workflow	42-22
Convert Model with Bus Object with Structure Initial Conditions to Fixed-Point	42-23
Autoscale Simulink.AliasType Objects	42-27
Autoscale Simulink.NumericType Objects	42-30
Autoscale Simulink.Parameter Objects	42-33
Autoscale a Model Using Parameter Objects	42-34

Command Line Interface for the Fixed-Point Tool

43

The Command-Line Interface for the Fixed-Point Tool	43-2
Convert a Model to Fixed Point Using the Command Line	43-4

Code Generation

44

Fixed-Point Code Generation Support	44-2
Introduction	44-2
Languages	44-2
Data Types	44-2
Rounding Modes	44-2
Overflow Handling	44-2
Blocks	44-3

Scaling	44-3
Accelerating Fixed-Point Models	44-4
Using External Mode or Rapid Simulation Target ...	44-6
Introduction	44-6
External Mode	44-6
Rapid Simulation Target	44-7
Net Slope Computation	44-8
Handle Net Slope Computation	44-8
Use Division to Handle Net Slope Computation	44-9
Improve Numerical Accuracy of Simulation Results with Rational Approximations to Handle Net Slope	44-10
Improve Efficiency of Generated Code with Rational Approximations to Handle Net Slope	44-15
Use Integer Division to Handle Net Slope Computation	44-21
Control the Generation of Fixed-Point Utility Functions	44-22
Optimize Generated Code Using Specified Minimum and Maximum Values	44-22
Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values	44-24
Optimize Generated Code with the Model Advisor ...	44-28
Identify Blocks that Generate Expensive Fixed-Point and Saturation Code	44-28
Identify Questionable Fixed-Point Operations	44-31
Identify Blocks that Generate Expensive Rounding Code	44-34
Lookup table optimization	44-36
Selecting Data Types for Basic Operations	44-38
Restrict Data Type Word Lengths	44-38
Avoid Fixed-Point Scalings with Bias	44-38
Wrap and Round to Floor or Simplest	44-39
Limit the Use of Custom Storage Classes	44-40
Use of Shifts by C Code Generation Products	44-41
Introduction to Shifts by Code Generation Products ..	44-41

Modeling Sources of Shifts	44-43
Controlling Shifts in Generated Code	44-43

Fixed-Point Advisor Reference

45

Fixed-Point Advisor	45-2
Fixed-Point Advisor Overview	45-3
Preparing Model for Conversion	45-6
Prepare Model for Conversion Overview	45-7
Verify model simulation settings	45-8
Verify update diagram status	45-11
Address unsupported blocks	45-12
Set up signal logging	45-14
Create simulation reference data	45-15
Verify Fixed-Point Conversion Guidelines Overview .	45-17
Check model configuration data validity diagnostic parameters settings	45-18
Implement logic signals as Boolean data	45-19
Simulation range checking	45-20
Check for implicit signal resolution	45-21
Preparing for Data Typing and Scaling	45-23
Prepare for Data Typing and Scaling Overview	45-24
Review locked data type settings	45-25
Verify Stateflow charts have strong data typing with Simulink	45-26
Specify block minimum and maximum values	45-27
Return to the Fixed-Point Tool to Perform Data Typing and Scaling	45-28
See Also	45-28

What Is the Difference Between Fixed-Point and Built-In Integer Types?	46-3
Negative Fraction Length	46-4
Fraction Length Greater Than Word Length	46-6
fi Constructor Does Not Follow globalfimath Rules	46-8
Decide Which Workflow Is Right For Your Application	46-9
Tips for Making Generated Code More Efficient	46-11
fimath Settings for Efficient Code	46-11
Replace Functions With More Efficient Fixed-Point Implementations	46-11
Know When a Function is Supported for Instrumentation and Acceleration	46-13
What to Do If a Function Is Not Supported for Fixed-Point Conversion	46-14
Isolate the Unsupported Functions	46-14
Create a Replacement Function	46-14
fi*non-fi Errors	46-16
Data Type Mismatch Errors	46-17
Mismatched fimath Errors	46-18
Why Does the Fixed-Point Converter App Not Propose Data Types for System Objects?	46-19
Slow Operations in the Fixed-Point Converter App	46-20
Using Blocks that Don't Support Fixed-Point Data Types	46-21
Find Blocks That Support Fixed-Point Data Types	46-21

What to Do When a Block Is Not Supported For Fixed-Point Conversion	46-21
Prevent The Fixed-Point Tool From Overriding Integer Data Types	46-23
Why Did The Fixed-Point Tool Not Propose Data Types?	46-24
Inadequate Range Information	46-24
Inherited Output Data Types	46-24
Frequently Asked Questions About Fixed-Point Numbers	46-25
Fraction Length Greater Than Word Length	46-25
Negative Fraction Length	46-25
Why am I missing data type proposals for MATLAB Function block variables?	46-26
Data Type Propagation Errors After Applying Proposed Data Types	46-27
Shared Data Type Groups	46-27
Block Constraints	46-28
Internal Block Rules	46-29
Resolve Range Analysis Issues	46-30
Data Type Mismatch and Structure Initial Conditions	46-31
Specify Bus Signal Initial Conditions Using Simulink.Parameter Objects	46-31
Data Type Mismatch and Masked Atomic Subsystems	46-32
Compile-Time Recursion Limit Reached	46-33
Issue	46-33
Cause	46-33
Solutions	46-33
Force Run-Time Recursion	46-33
Increase the Compile-Time Recursion Limit	46-36
Output Variable Must Be Assigned Before Run-Time Recursive Call	46-38
Issue	46-38

Cause	46-38
Solution	46-38

Unable to Determine That Every Element of Cell Array

Is Assigned	46-41
Issue	46-41
Cause	46-41
Solution	46-42

Nonconstant Index into varargin or varargin in a for-

Loop	46-45
Issue	46-45
Cause	46-45
Solution	46-46

Single-Precision Conversion in Simulink

47

Getting Started with Single Precision Converter	47-2
Select System Under Design	47-2
Check Compatibility	47-2
Convert	47-3
Verify	47-4
Convert a System to Single Precision	47-5

Writing Fixed-Point S-Functions

A

Data Type Support	A-2
Supported Data Types	A-2
The Treatment of Integers	A-3
Data Type Override	A-3
Structure of the S-Function	A-5

Storage Containers	A-7
Introduction	A-7
Storage Containers in Simulation	A-7
Storage Containers in Code Generation	A-10
Data Type IDs	A-12
The Assignment of Data Type IDs	A-12
Registering Data Types	A-13
Setting and Getting Data Types	A-15
Getting Information About Data Types	A-15
Converting Data Types	A-17
Overflow Handling and Rounding Methods	A-18
Tokens for Overflow Handling and Rounding Methods	A-18
Overflow Logging Structure	A-19
Create MEX-Files	A-20
Fixed-Point S-Function Examples	A-21
Get the Input Port Data Type	A-23
Set the Output Port Data Type	A-25
Interpret an Input Value	A-26
Write an Output Value	A-28
Determine Output Type Using the Input Type	A-30
API Function Reference	A-31

Fixed-Point Designer for MATLAB Code

Fixed-Point Concepts

- “Fixed-Point Data Types” on page 1-2
- “Scaling” on page 1-4
- “Compute Slope and Bias” on page 1-5
- “Precision and Range” on page 1-8
- “Arithmetic Operations” on page 1-13
- “fi Objects and C Integer Data Types” on page 1-25

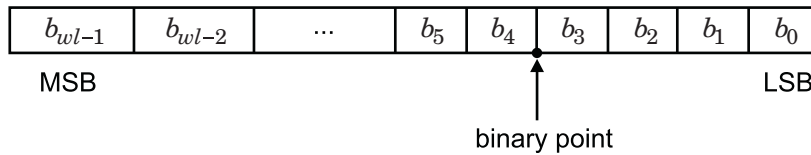
Fixed-Point Data Types

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 Designer™ documentation.

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 Designer 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

$$\text{real-world value} = 2^{-\text{fraction length}} \times \text{integer}$$

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

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

For an example of how to compute slope and bias in MATLAB[®], see “Compute Slope and Bias” on page 1-5

Compute Slope and Bias

In this section...

“What Is Slope Bias Scaling?” on page 1-5

“Compute Slope and Bias” on page 1-5

What Is Slope Bias Scaling?

With slope bias scaling, you must specify the slope and bias of a number. The real-world value of a slope bias scaled number can be represented by:

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

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

Compute Slope and Bias

Start with the endpoints that you want, signedness, and word length.

```
lower_bound = 999;
upper_bound = 1000;
is_signed = true;
word_length = 16;
```

To find the range of a `fi` object with a specified word length and signedness, use the `range` function.

```
[Q_min, Q_max] = range(fi([], is_signed, word_length, 0));
```

To find the slope and bias, solve the system of equations:

$$\text{lower_bound} = \text{slope} * \text{Q_min} + \text{bias}$$

$$\text{upper_bound} = \text{slope} * \text{Q_max} + \text{bias}$$

Rewrite these equations in matrix form.

$$\begin{bmatrix} \text{lower_bound} \\ \text{upper_bound} \end{bmatrix} = \begin{bmatrix} \text{Q_min} & 1 \\ \text{Q_max} & 1 \end{bmatrix} \begin{bmatrix} \text{slope} \\ \text{bias} \end{bmatrix}$$

Solve for the slope and bias.

```
A = double ([Q_min, 1; Q_max, 1]);  
b = double ([lower_bound; upper_bound]);  
x = A\b;  
format long g
```

To find the slope, or precision, call the first element of the slope-bias vector, *x*.

```
slope = x(1)  
  
slope =  
  
1.52590218966964e-05
```

To find the bias, call the second element of vector *x*.

```
bias = x(2)  
  
bias =  
  
999.500007629511
```

Create a `numeric_type` object with slope bias scaling.

```
T = numeric_type(is_signed, word_length, slope, bias)  
  
T =
```

```
DataTypeMode: Fixed-point: slope and bias scaling  
Signedness: Signed  
WordLength: 16  
Slope: 1.5259021896696368e-5  
Bias: 999.500007629511
```

Create a `fi` object with `numeric_type` *T*.

```
a = fi(999.255, T)  
  
a =  
  
999.254993514916
```

```
DataTypeMode: Fixed-point: slope and bias scaling  
Signedness: Signed
```



```
WordLength: 16
Slope: 1.5259021896696368e-5
Bias: 999.500007629511
```

Verify that the `fi` object that you created has the correct specifications by finding the range of `a`.

```
range(a)
```

```
ans =
```

```
999          1000
```

```
DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 1.5259021896696368e-5
Bias: 999.500007629511
```

Precision and Range

In this section...

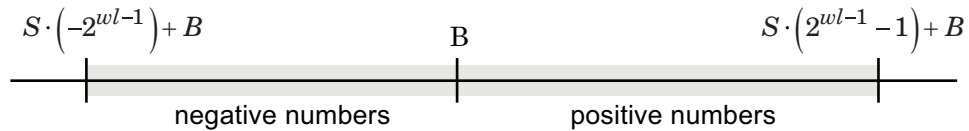
“Range” on page 1-8

“Precision” on page 1-9

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

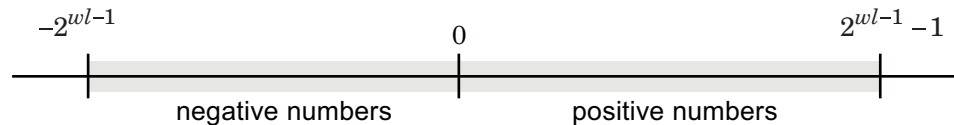
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl , scaling S and bias B is illustrated below:



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 Designer 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.

When you create a `fi` object, any overflows are saturated. The `OverflowAction` property of the default `fi` object is `saturate`. You can log overflows and underflows by setting the `LoggingMode` property of the `fi` 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 Designer software currently supports the following rounding methods:

- **Ceiling** 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.
- **Zero** 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 the Fixed-Point Designer product.

Fixed-Point Designer Rounding Mode	Cost	Bias	Possibility of Overflow
Ceiling	Low	Large positive	Yes
Convergent	High	Unbiased	Yes
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	Low	Large negative	No
Nearest	Moderate	Small positive	Yes
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
Simplest (Simulink® only)	Low	Depends on the operation	No

Arithmetic Operations

In this section...

“Modulo Arithmetic” on page 1-13

“Two's Complement” on page 1-14

“Addition and Subtraction” on page 1-15

“Multiplication” on page 1-16

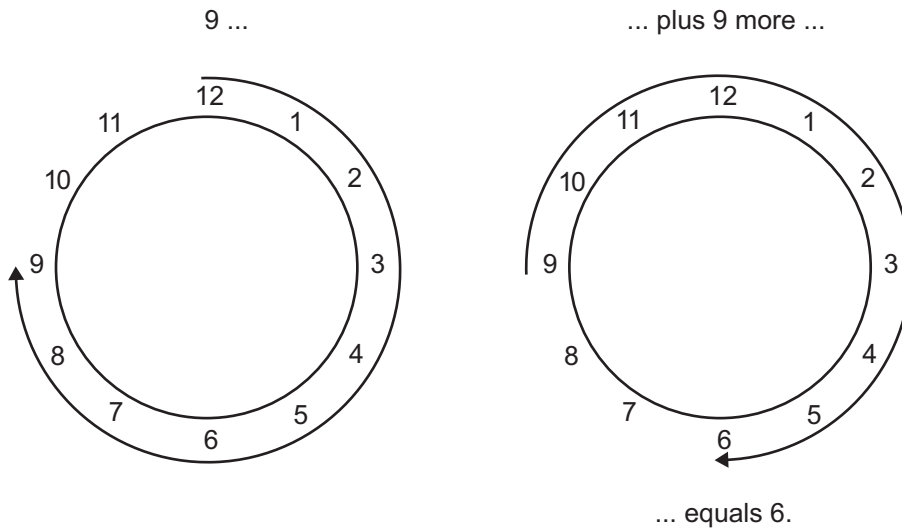
“Casts” on page 1-22

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):

$$\begin{array}{r} 010010.100 \quad (18.5) \\ -0110.110 \quad (6.75) \\ \hline \end{array}$$

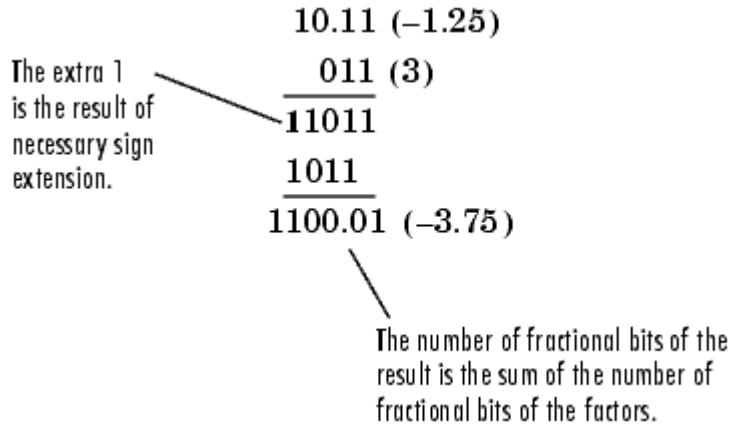
The default `fimath` has a value of 1 (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

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):

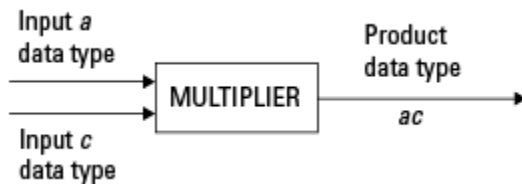


Multiplication Data Types

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

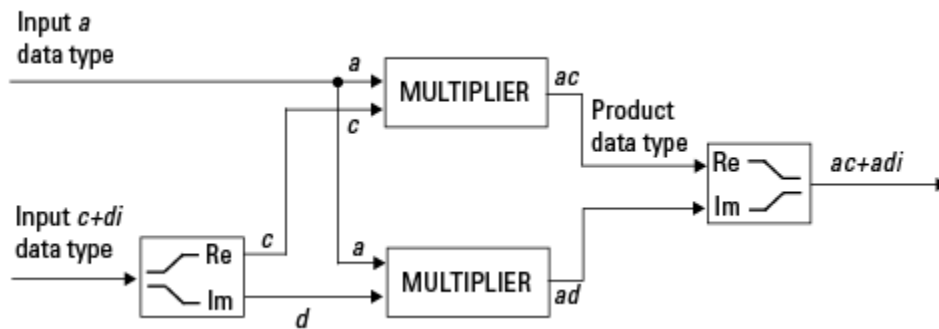
Real-Real Multiplication

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.



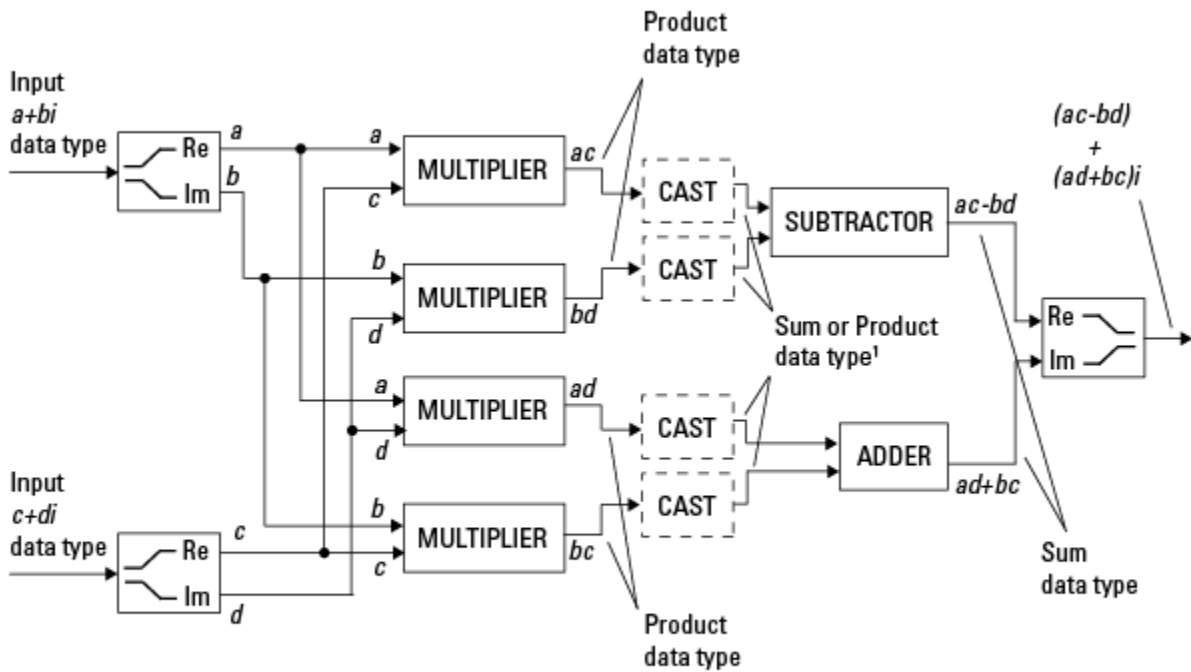
Real-Complex Multiplication

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. 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*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. As a result, the word length of the full-precision result has one more bit than the sum of the word lengths of the multiplicands:

```
x = fi(5+6i,T1,F)
```

```
x =
```

```
5.0000 + 6.0000i
```

```

        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*y
```

```
z =
```

```
38.0000 +70.0000i
```

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 41

```

FractionLength: 30

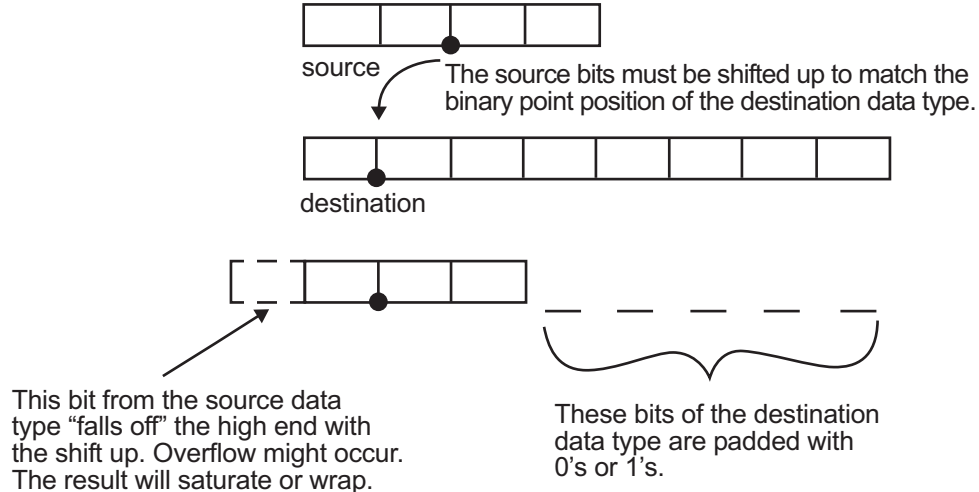
Casts

The `fi` 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:



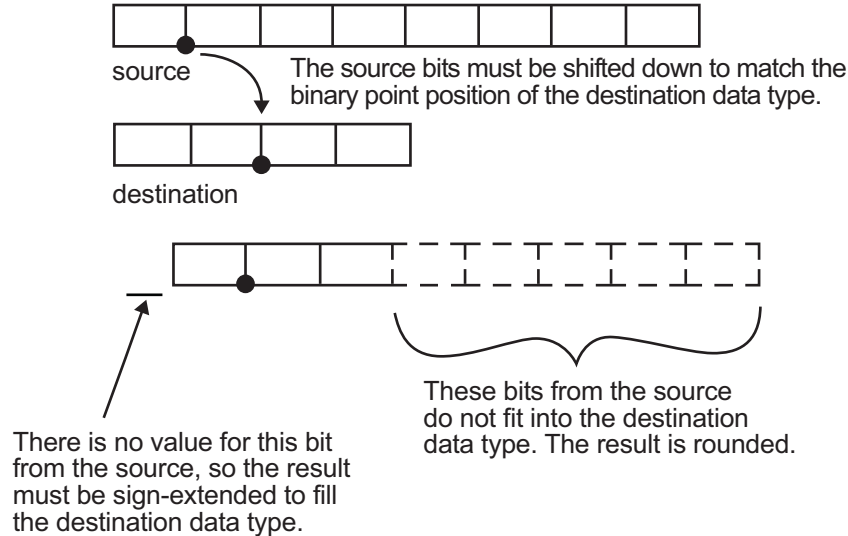
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 1-25

“Unary Conversions” on page 1-27

“Binary Conversions” on page 1-28

“Overflow Handling” on page 1-30

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 1-14 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 the ranges 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 Designer 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}	$2^{n-1} - 1$	Not applicable
<code>fi(x,0,n,0)</code>	No	n (2 to 65,535)	0	0	$2^n - 1$	Not applicable
<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
<code>fi(x,1,32,0)</code>	Yes	32	0	-2,147,483,648	2,147,483,647	long int
<code>fi(x,0,32,0)</code>	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
<code>char</code> or <code>short</code>	<code>int</code>
<code>unsigned char</code> or <code>unsigned short</code>	<code>int</code> or <code>unsigned int</code> ¹
<code>float</code>	<code>float</code>
Array of <code>T</code>	Pointer to <code>T</code>
Function returning <code>T</code>	Pointer to function returning <code>T</code>

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	$\sim x = \text{not}(x)$	Result is logical .
$\sim x$	<code>bitcmp(x)</code>	Result is same numeric type as operand.
*x	No equivalent	Not applicable
$x \ll n$	<code>bitshift(x, n)</code> positive n	Result is same numeric type as operand. Round mode is always floor . Overflow mode is obeyed. 0-valued bits are shifted in on the right.
$x \gg n$	<code>bitshift(x, -n)</code>	Result is same numeric type as operand. Round mode is always floor . 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

Type of One Operand	Type of Other Operand	ANSI C Conversion
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
<code>fi</code>	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
<code>fi</code>	<code>int8</code>	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 8 • FractionLength = 0
<code>fi</code>	<code>uint8</code>	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 8 • FractionLength = 0
<code>fi</code>	<code>int16</code>	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 16 • FractionLength = 0
<code>fi</code>	<code>uint16</code>	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 16 • FractionLength = 0
<code>fi</code>	<code>int32</code>	<ul style="list-style-type: none"> • Signed = 1

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
		<ul style="list-style-type: none"> • 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 Designer 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
OverflowAction	'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.
ProductMode	'FullPrecision'	<p>Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than <code>MaxProductWordLength</code>.</p> <p>The rules for computing the resulting product word and fraction lengths are given in “fimath Object Properties” on page 4-5 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 <code>ProductWordLength</code> property determines the resulting word length. If <code>ProductWordLength</code> is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If <code>ProductWordLength</code> 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 “fimath Object Properties” on page 4-5 in the Property Reference.</p>
	'KeepMSB'	The most significant bits of the product are kept. Overflow is prevented, but precision may be lost.

fimath Object Properties Related to Overflow Handling	Property Value	Description
		<p>The ProductWordLength property determines the resulting word length. 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 “fimath Object Properties” on page 4-5 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'.
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 “fimath Object Properties” on page 4-5 in the Property Reference.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'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 <code>SumWordLength</code> property determines the resulting word length. If <code>SumWordLength</code> is greater than is necessary for the full-precision sum, then the result is stored in the least significant bits. If <code>SumWordLength</code> 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 “fimath Object Properties” on page 4-5 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 <code>SumWordLength</code> property determines the resulting word length. If <code>SumWordLength</code> is greater than is necessary for the full-precision sum, then the result is stored in the most significant bits. If <code>SumWordLength</code> is less than is necessary for the full-precision sum, then rounding occurs.</p> <p>The rule for computing the resulting sum fraction length is given in “fimath Object Properties” on page 4-5 in the Property Reference.</p>
	'SpecifyPrecision'	<p>You can specify both the word length and the fraction length of the resulting sum.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
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 2-18
- “fi Object Functions” on page 2-24

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 Designer software in any of the following ways:

- You can use the `fi` constructor function to create a `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, to create a `fi` object with the default data type and a value of 0:

```
a = fi(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.

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 has a local `fimath` object. The `fi` object uses default values for the remaining unspecified `fimath` object properties.

- If you do not specify any `fi` object properties in the `fi` object constructor, the `fi` object uses default `fi` 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 `fi` object with a `fi` 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 **numericType** 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

```

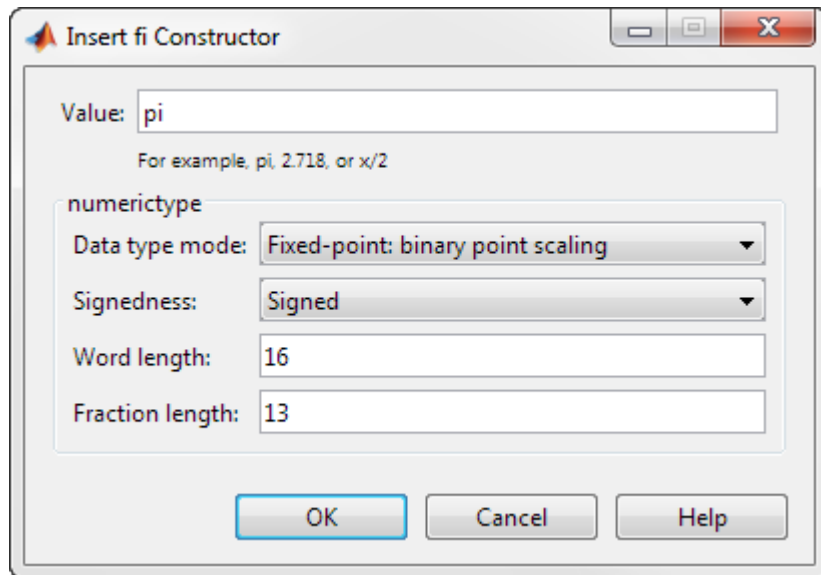
Building `fi` Object Constructors in a GUI

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 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:

- 1 On the **Home** tab, in the **File** section, click **New > Script** to open the MATLAB Editor.
- 2 On the **Editor** tab, in the **Edit** section, click  in the **Insert** button group. Click **Insert `fi`...** to open the **Insert `fi` Constructor** dialog box.
- 3 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



- 4 To insert the `fi` object constructor 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 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:

```
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
```

Creating fi Objects For Use in a Types Table

You can write a reusable MATLAB algorithm by keeping the data types of the algorithmic variables in a separate types table. For example,

```
function T = mytypes(dt)
    switch dt
        case 'double'
            T.b = double([]);
            T.x = double([]);
            T.y = double([]);

        case 'fixed16'
            T.b = fi([],1,16,15);
            T.x = fi([],1,16,15);
            T.y = fi([],1,16,14);
    end
end
```

Cast the variables in the algorithm to the data types in the types table as described in “Manual Fixed-Point Conversion Best Practices” on page 12-4.

```
function [y,z]=myfilter(b,x,z,T)
    y = zeros(size(x),'like',T.y);
    for n=1:length(x)
        z(:) = [x(n); z(1:end-1)];
        y(n) = b * z;
```

```
end  
end
```

In a separate test file, set up input data to feed into your algorithm, and specify the data types of the inputs.

```
% Test inputs  
b = fir1(11,0.25);  
t = linspace(0,10*pi,256)';  
x = sin((pi/16)*t.^2);  
% Linear chirp  
  
% Cast inputs  
T=mytypes('fixed16');  
b=cast(b,'like',T.b);  
x=cast(x,'like',T.x);  
z=zeros(size(b),'like',T.x);  
  
% Run  
[y,z] = myfilter(b,x,z,T);
```

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 Designer `reinterpretcast` Function
- Casting with the `cast` 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, 1, 8, 0);
B = fi(32767, 1, 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, 1, 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 Designer `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], 0, 8, 5)
T = numerictype(1, 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
```

Casting with the cast Function

Using the `cast` function, you can convert the value of a variable to the same numeric type, complexity, and `fimath` as another variable.

In the following example, `a` is cast to the data type of `b`. The output, `c`, has the same numeric type and `fimath` properties as `b`, and the value of `a`.

```
a = pi;  
b = fi([],1,16,13,'RoundingMethod',Floor);  
c = cast(a,'like',b)  
  
c =  
  
    3.1415  
  
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 16
```

```
FractionLength: 13  
  
RoundingMethod: Floor  
OverflowAction: Saturate  
  ProductMode: FullPrecision  
    SumMode: FullPrecision
```

Using this syntax allows you to specify data types separately from your algorithmic code as described in “Manual Fixed-Point Conversion Best Practices” on page 12-4.

`fi` Object Properties

In this section...
“Data Properties” on page 2-18
“ <code>fimath</code> Properties” on page 2-18
“ <code>numerictype</code> Properties” on page 2-20
“Setting <code>fi</code> Object Properties” on page 2-21

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
- `int` — Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type
- `oct` — Stored integer value of a `fi` object in octal

To learn more about these properties, see “`fi` Object Properties” in the Fixed-Point Designer 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 `fi` properties associated with `fi` objects determine how fixed-point arithmetic is performed. These `fi` properties can come from a local `fi` object or from default `fi` property values. To learn more about `fi` objects in fixed-point arithmetic, see “`fi` Rules for Fixed-Point Arithmetic” on page 4-13.

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

For more information, see “fimath Object Properties” on page 4-5.

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 more information, see “`numerictype` Object Properties” on page 6-6.

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

- “Setting Fixed-Point Properties at Object Creation” on page 2-21
- “Using Direct Property Referencing with fi” on page 2-22

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
```

To set the stored integer value of a `fi` object, use the parameter/value pair for the `'int'` property when you create the object. For example, create a signed `fi` object with a stored integer value of 4, 16-bit word length, and 15-bit fraction length.

```
x = fi(0,1,16,15,'int',4);
```

Verify that the `fi` object has the expected integer setting.

```
x.int
```

```
ans =
```

```
4
```

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 `OverflowAction` 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 3-16
- “Perform Binary-Point Scaling” on page 3-29
- “Develop Fixed-Point Algorithms” on page 3-34
- “Calculate Fixed-Point Sine and Cosine” on page 3-46
- “Calculate Fixed-Point Arctangent” on page 3-69
- “Compute Sine and Cosine Using CORDIC Rotation Kernel” on page 3-94
- “Perform QR Factorization Using CORDIC” on page 3-99
- “Compute Square Root Using CORDIC” on page 3-134
- “Convert Cartesian to Polar Using CORDIC Vectoring Kernel” on page 3-144
- “Set Data Types Using Min/Max Instrumentation” on page 3-149
- “Convert Fast Fourier Transform (FFT) to Fixed Point” on page 3-163
- “Detect Limit Cycles in Fixed-Point State-Space Systems” on page 3-183
- “Compute Quantization Error” on page 3-195
- “Normalize Data for Lookup Tables” on page 3-204
- “Implement Fixed-Point Log2 Using Lookup Table” on page 3-210
- “Implement Fixed-Point Square Root Using Lookup Table” on page 3-215
- “Set Fixed-Point Math Attributes” on page 3-220

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.

```
originalFormat = get(0, 'format');
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);
```

```
bin(a)
```



```
ans =
```

```
'10100011110101110000101000111101011100001111010111000010100011110101110000101001'
```

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 =
    int16
    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(π)`

`a.bin = '0110010010001000'`

`a =`

`3.1416015625`

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

`fi(ϕ)`

```
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 =
```

```
int16
```

```
10
```

```
storedInteger(b)
```

```
ans =  
  
    int16  
  
    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 = numerictype('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
```

```
bin(a)
```

```
ans =
```

```
'01111111'
```

while a 100-bit fractional number cannot (1 is displayed, when the exact value is $1 - 2^{-99}$):

```
b = sfi(1,100,99)
```

```
b =
```

```
1
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 100  
    FractionLength: 99
```

Note, however, that the full precision is preserved in the internal representation of `fi`

```
bin(b)
```


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

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

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

```
originalFormat = get(0, 'format'); format
```

```
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 =

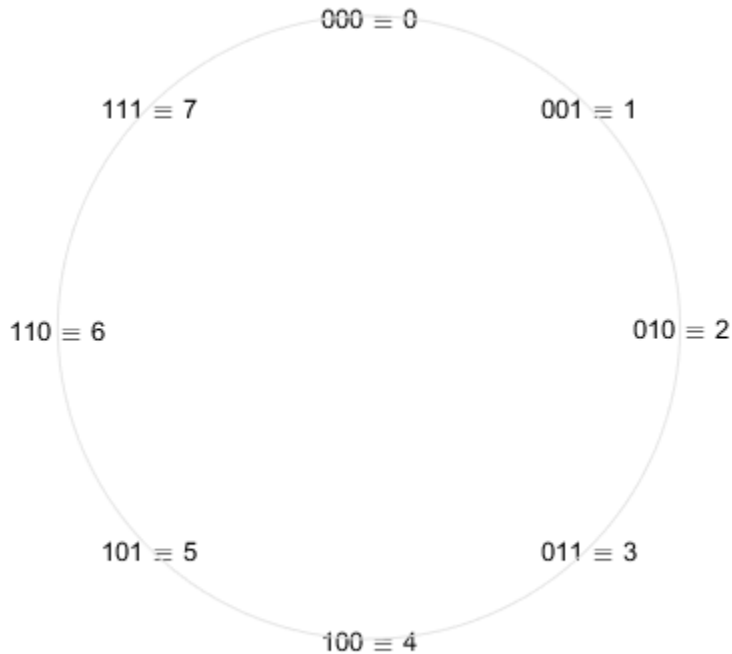
8×3 char array

```
'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)
```

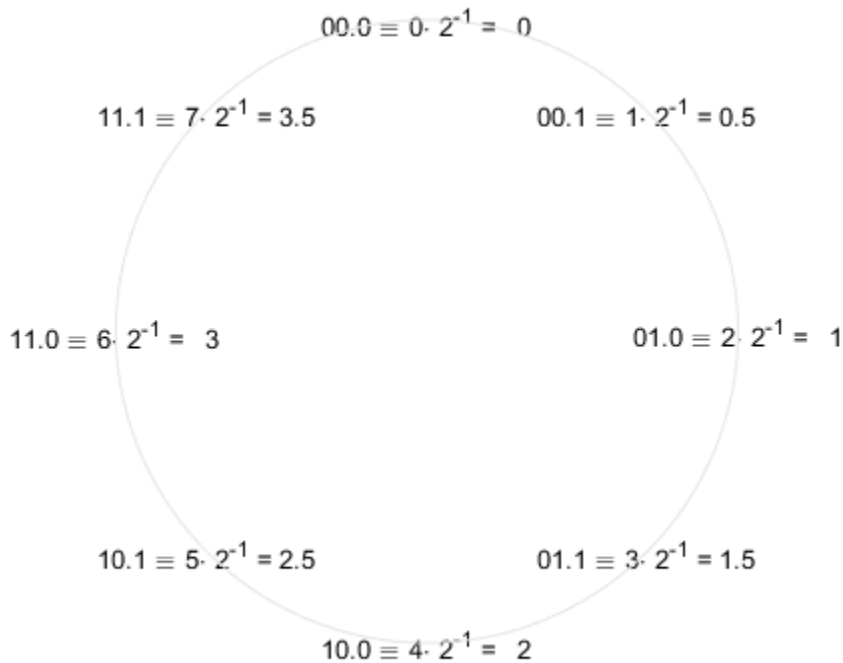
```
uf =  
  
    0  
0.5000  
1.0000  
1.5000  
2.0000  
2.5000  
3.0000  
3.5000
```

```
b =  
  
8×3 char array  
  
'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 fraction length f is equal to the wordlength n , which produces a scaling such that the range of numbers is between 0 and $1 \cdot 2^{-f}$, 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.

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

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
0.1250
0.2500
0.3750
0.5000
0.6250
0.7500
0.8750
```

```
b =
```

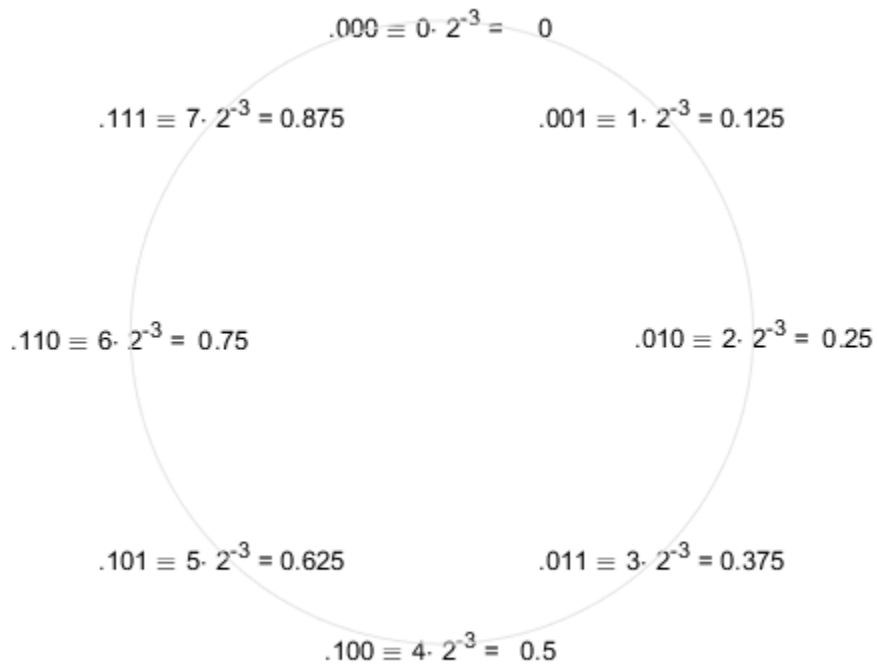
```
8×3 char array
```

```
'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 =
```

```
8×3 char array
```

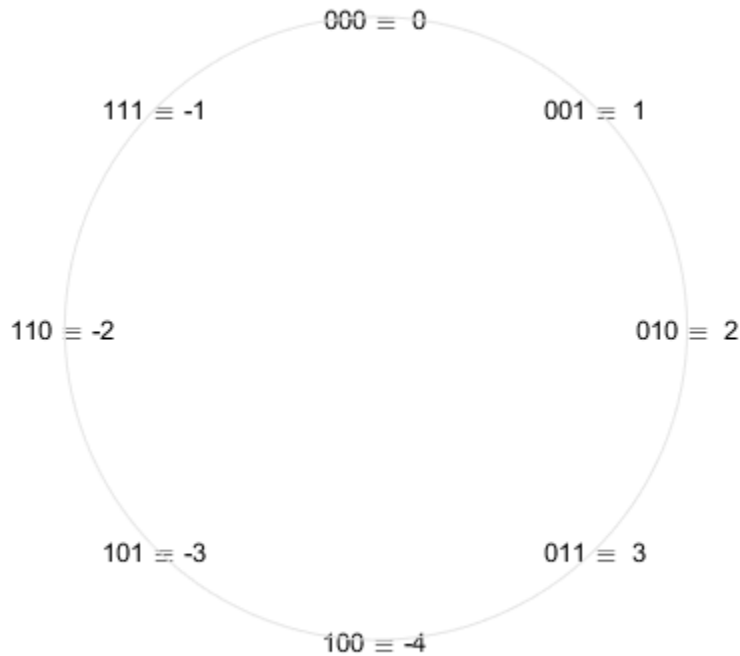
```
'100'
'101'
'110'
'111'
'000'
'001'
'010'
'011'
```

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*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 =
```

```
-2.0000
-1.5000
-1.0000
-0.5000
  0
  0.5000
  1.0000
  1.5000
```

```
b =
```

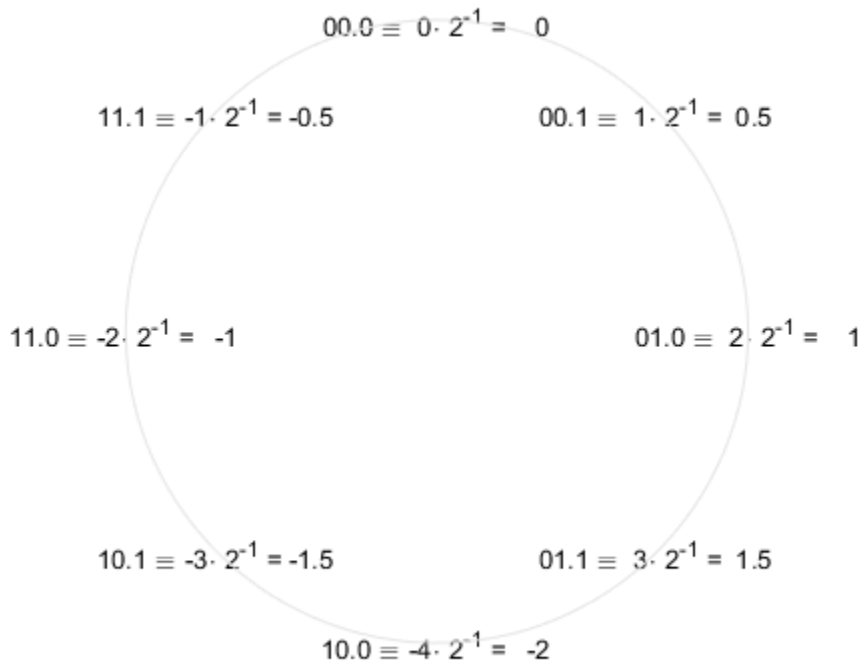
```
8×3 char array
```

```
'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 fraction length f is one less than the wordlength n , which produces a scaling such that the range of numbers is between -1 and $1-2^{-f}$, 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*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 =
```

```
-1.0000
-0.7500
-0.5000
-0.2500
  0
  0.2500
  0.5000
  0.7500
```

```
b =
```

```
8×3 char array
```

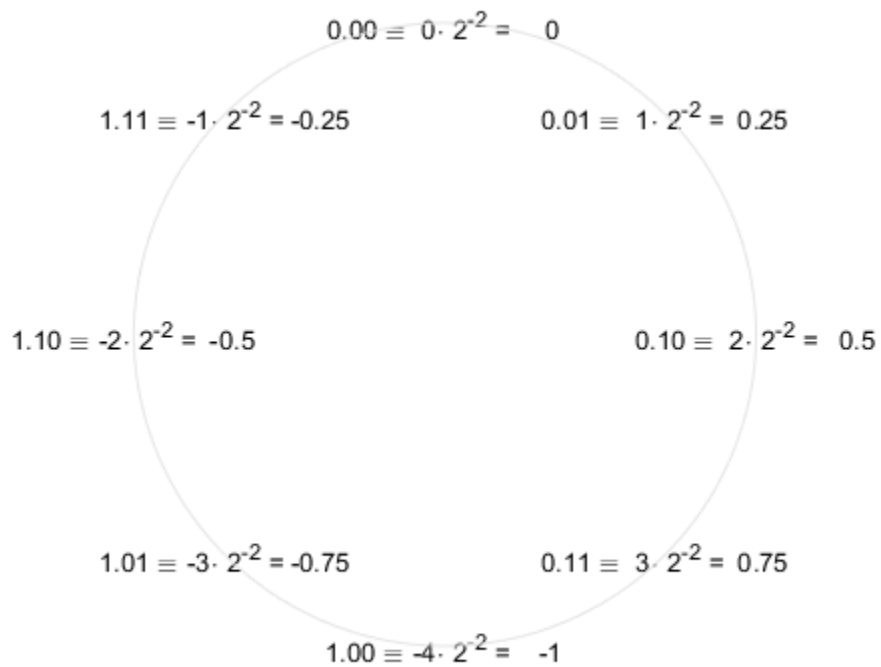
```
'100'
'101'
'110'
'111'
'000'
'001'
'010'
'011'
```

Signed Fractional Fixed-Point Number Circle.

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

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

```
set(0, 'format', originalFormat);
```



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 =
```

```
0.5000
```

```

    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 =
```

```
-0.5000
```

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

```
ans =
```

```
'111'
```

```
ans =
```

```
int8
```

```
-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).

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

`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 =
```

```
int8
```

```
-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.

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

`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 =
```

```
int8
```

```
-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:

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

`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 =  
    int8  
    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: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.

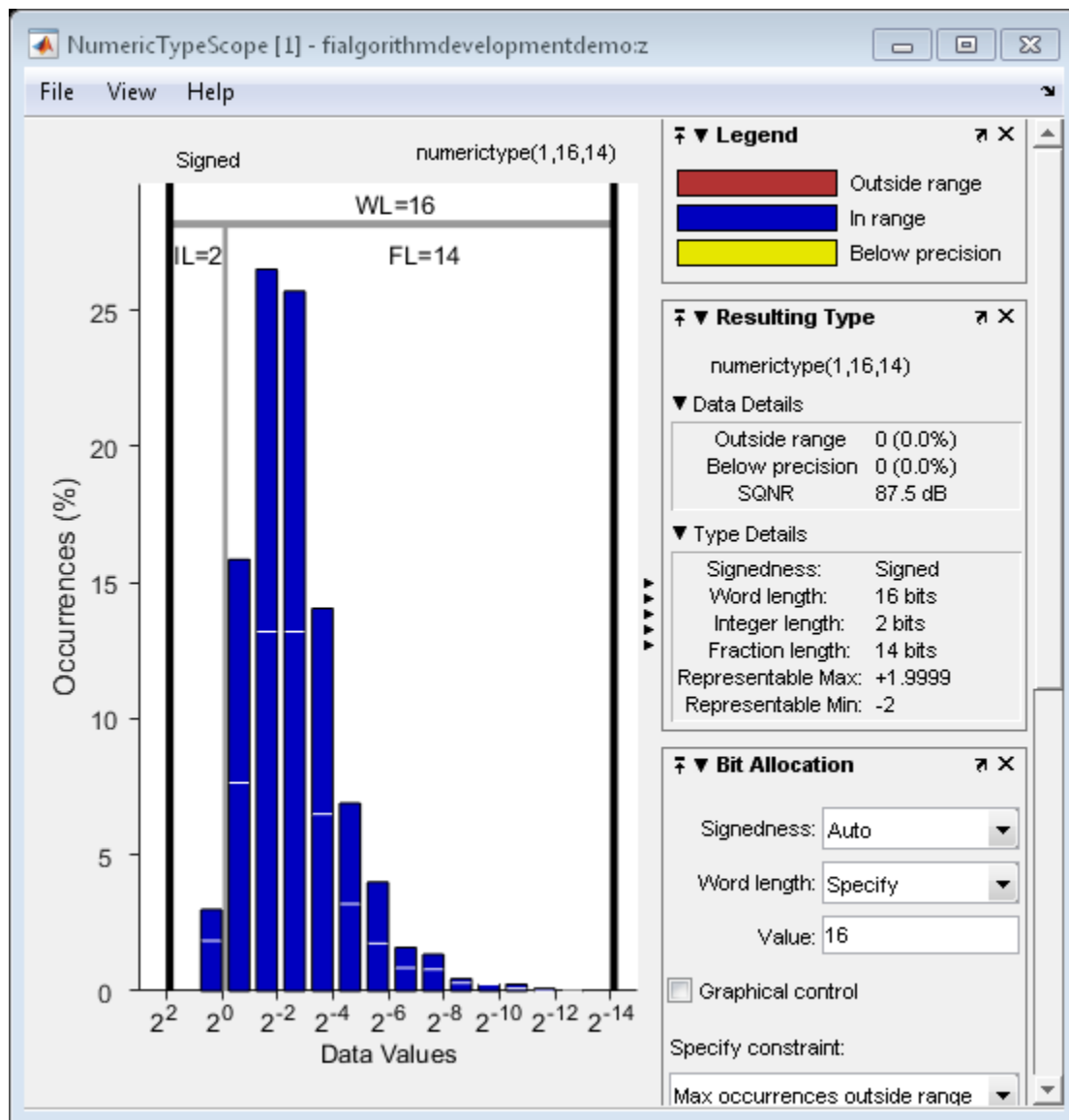
Instrument Floating-Point Code

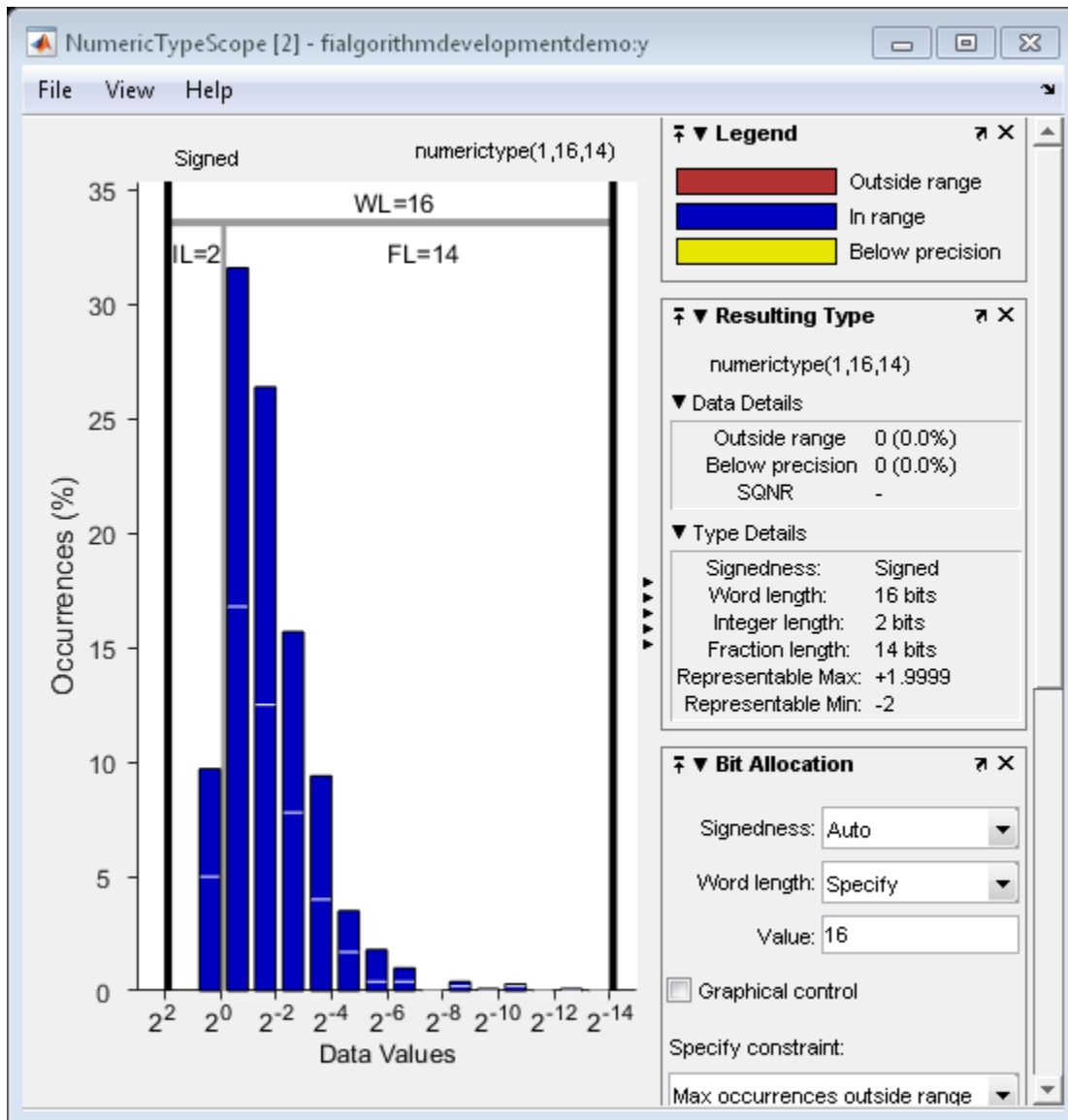
```

% Reset states
z = [0;0];

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 $(2^1 2^{-12}]$.

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 2 integer bit to avoid overflows. 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 1.51 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 $(2^1 2^{-13}]$.

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 2 integer bits to avoid overflows. With this suggested type you see no overflows or underflows.

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.
FIPREF_STATE = get(fipref);
reset(fipref)
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

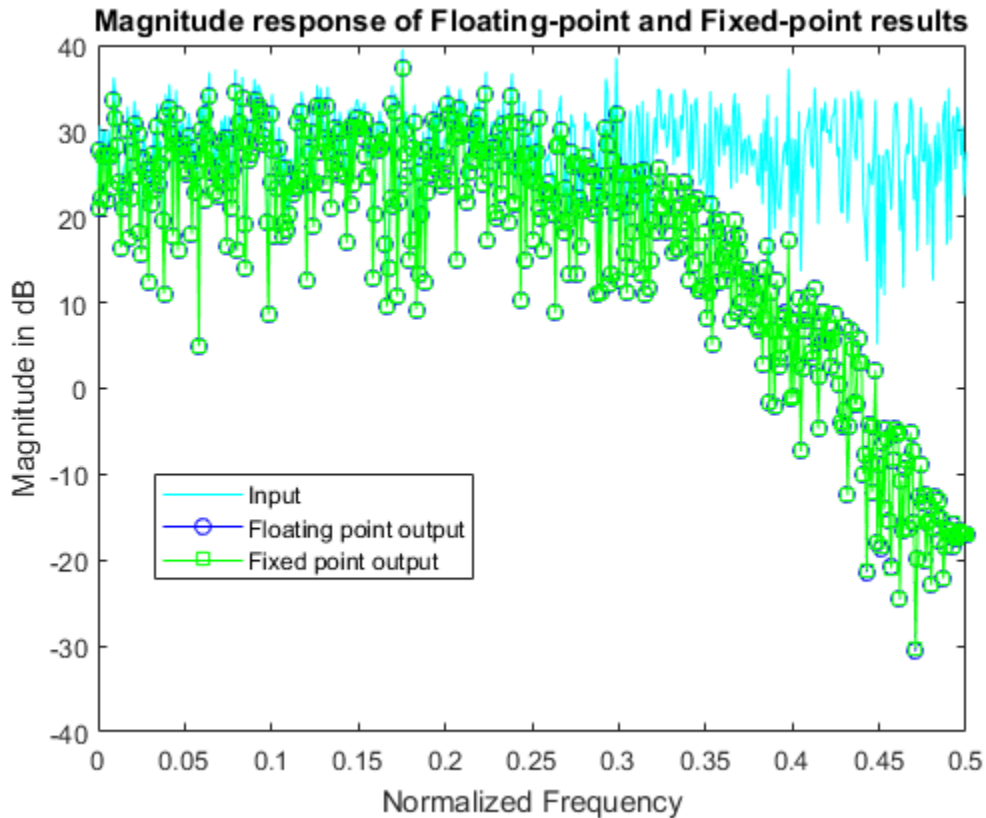
```
for k=1:length(x)
    y(k) = b(1)*x(k) + z(1);
    z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
    z(2) = b(3)*x(k)          - a(3)*y(k);
end
% Reset the logging mode.
fp.LoggingMode = default_loggingmode;
```

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.

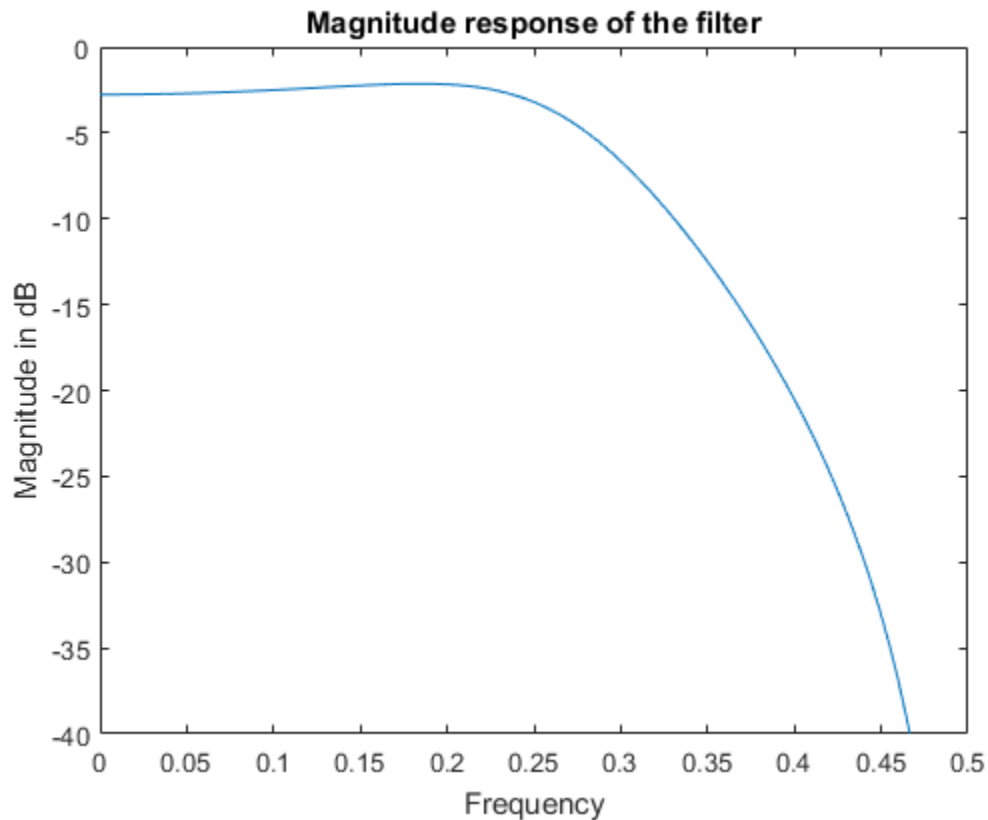
```
n = length(x);
f = linspace(0,0.5,n/2);
x_response = 20*log10(abs(fft(double(x))));
ydouble_response = 20*log10(abs(fft(ydouble)));
y_response = 20*log10(abs(fft(double(y))));
plot(f,x_response(1:n/2),'c-',...
     f,ydouble_response(1:n/2),'bo-',...
     f,y_response(1:n/2),'gs-');
ylabel('Magnitude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



```

h = fft(double(b),n)./fft(double(a),n);
h = h(1:end/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');

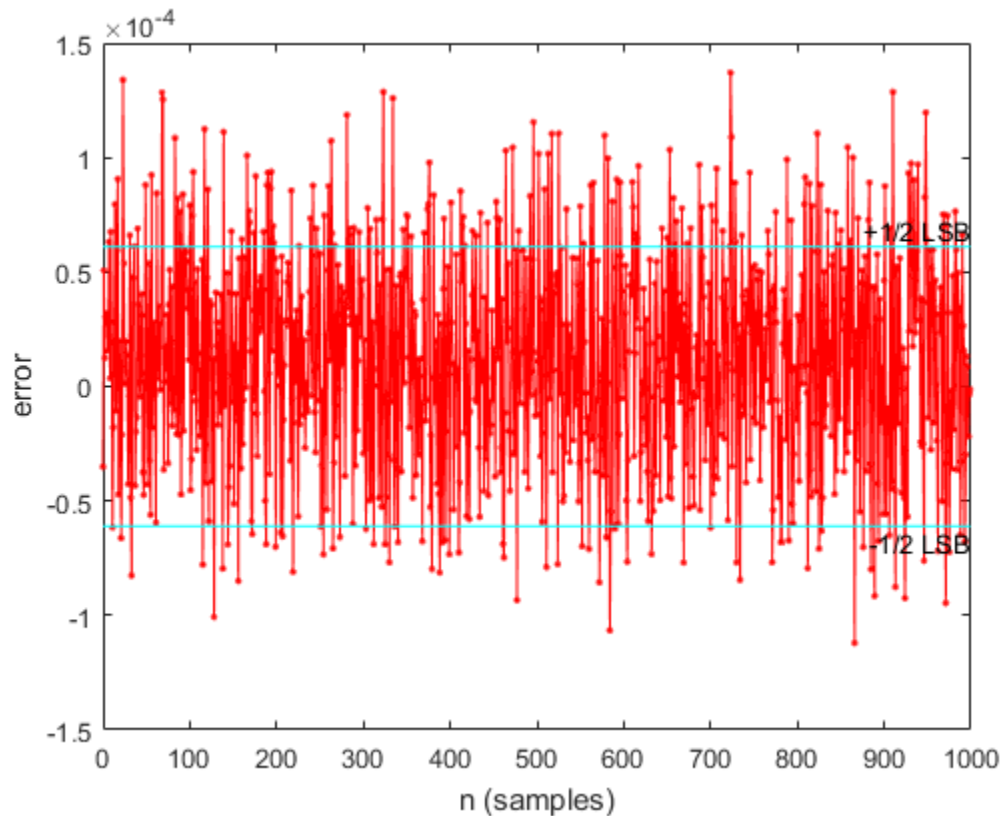
```



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 Fixed-Point Designer™, 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.hasSimulinkLicense
    % 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

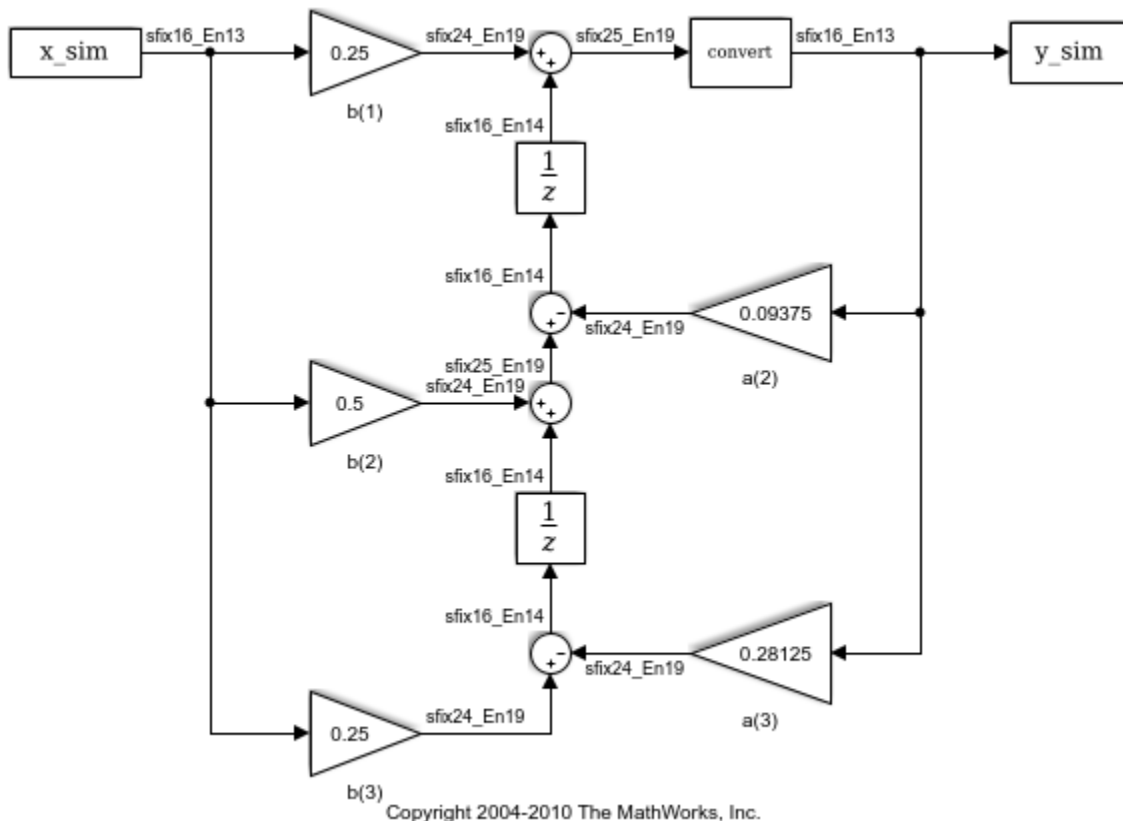
% Verify that the Simulink results are the same as the MATLAB file
isequal(y, out_sim.get('y_sim'))

end

ans =

logical

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);  
fipref(FIPREF_STATE);
```

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 Designer™ 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 `cordiccoss` 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:

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

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

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

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 `cordiccoss` 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 `atan(2.^-(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(:) = z + inpLUT(idx);
        x(:) = x + ytmp;
        y(:) = y - xtmp;
    else
        z(:) = z - inpLUT(idx);
        x(:) = x - ytmp;
        y(:) = 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.

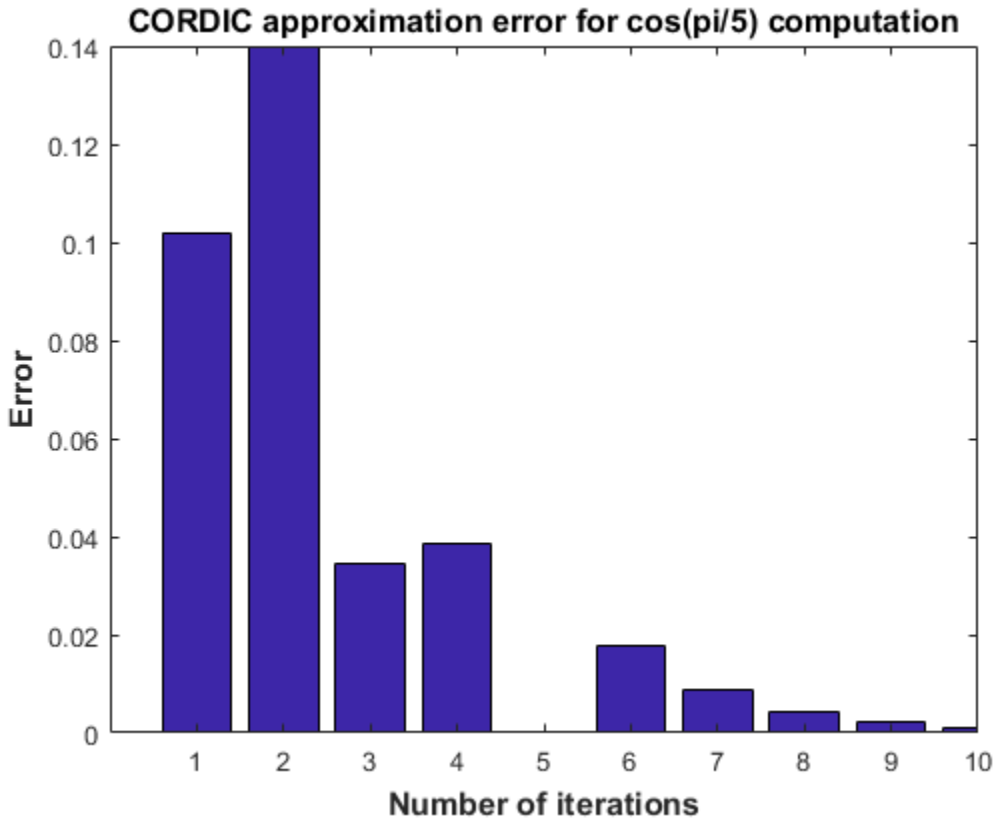
```
theta = pi/5; % input angle in radians
nitters = 10; % number of iterations
sinTh = sin(theta); % reference result
cosTh = cos(theta); % reference result
y_sin = zeros(nitters, 1);
sin_err = zeros(nitters, 1);
x_cos = zeros(nitters, 1);
cos_err = zeros(nitters, 1);
fprintf('\n\nNITERS \tERROR\n');
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]);
```

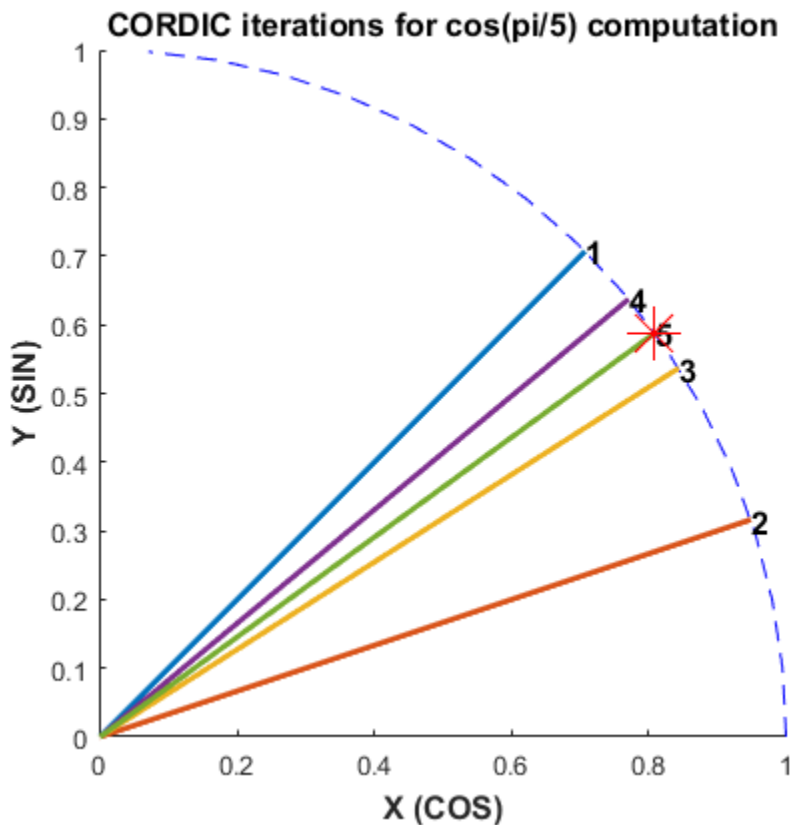


Plot the X-Y results for 5 iterations

```

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 result
    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`

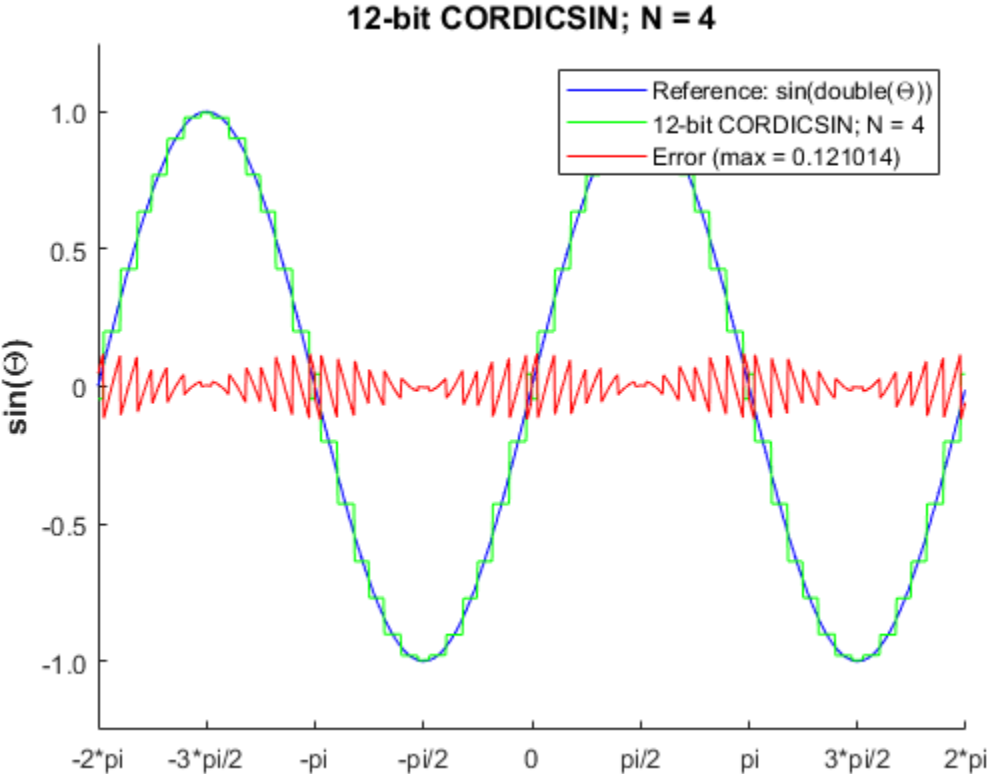
Create 1024 points between $[-2\pi, 2\pi]$

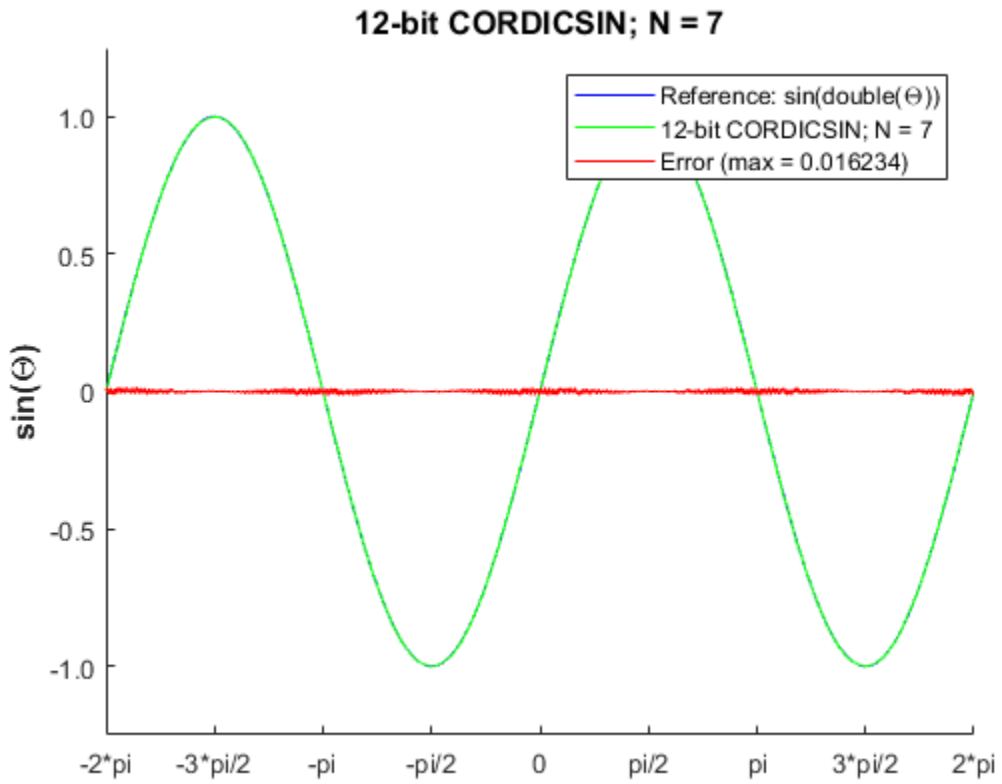
```
stepSize = pi/256;
thRadDb1 = (-2*pi):stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDb1, 12); % signed, 12-bit fixed-point values
sinThRef = sin(double(thRadFxp)); % reference results
```

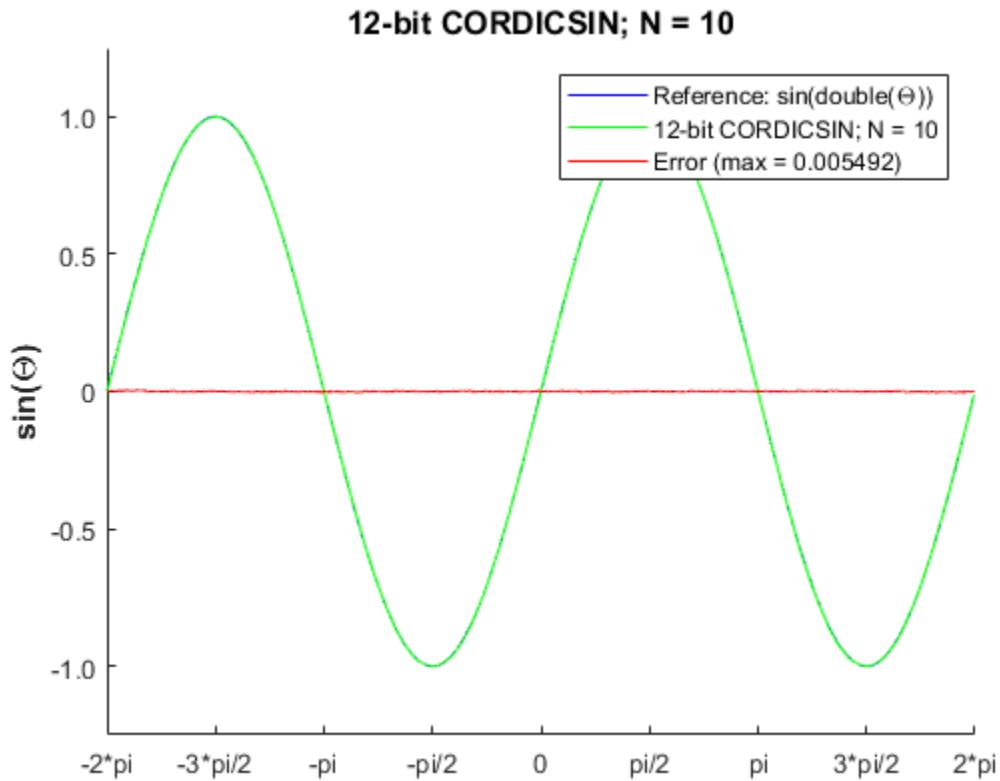
Compare fixed-point CORDIC vs. double-precision trig function results

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
```

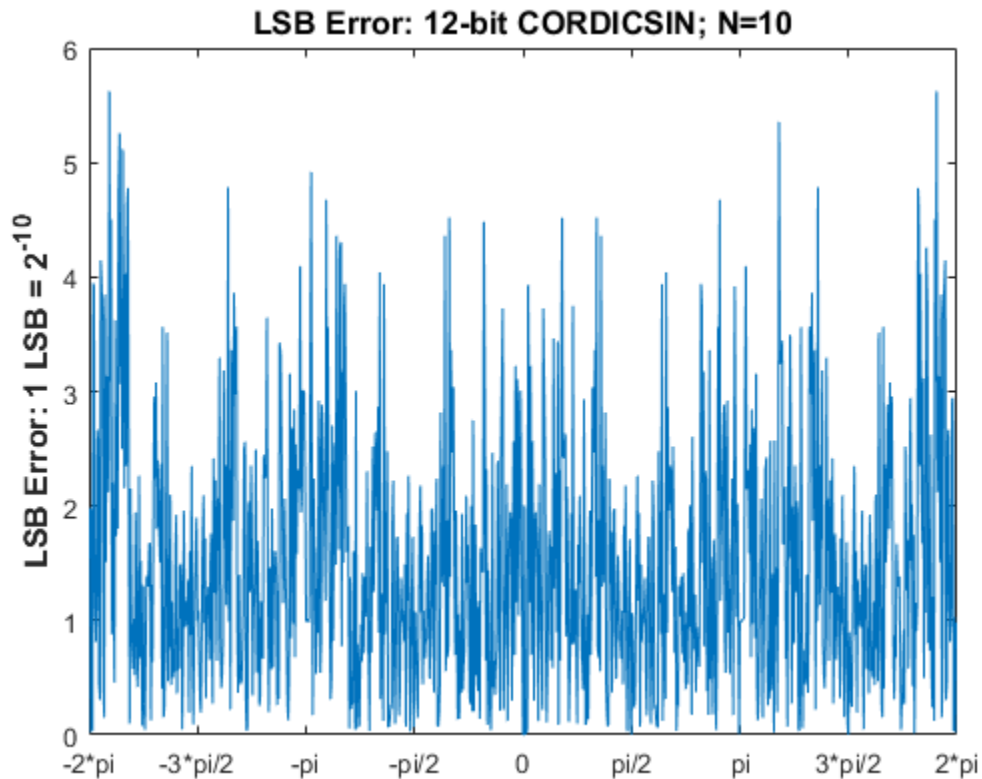







Compute the LSB Error for N = 10

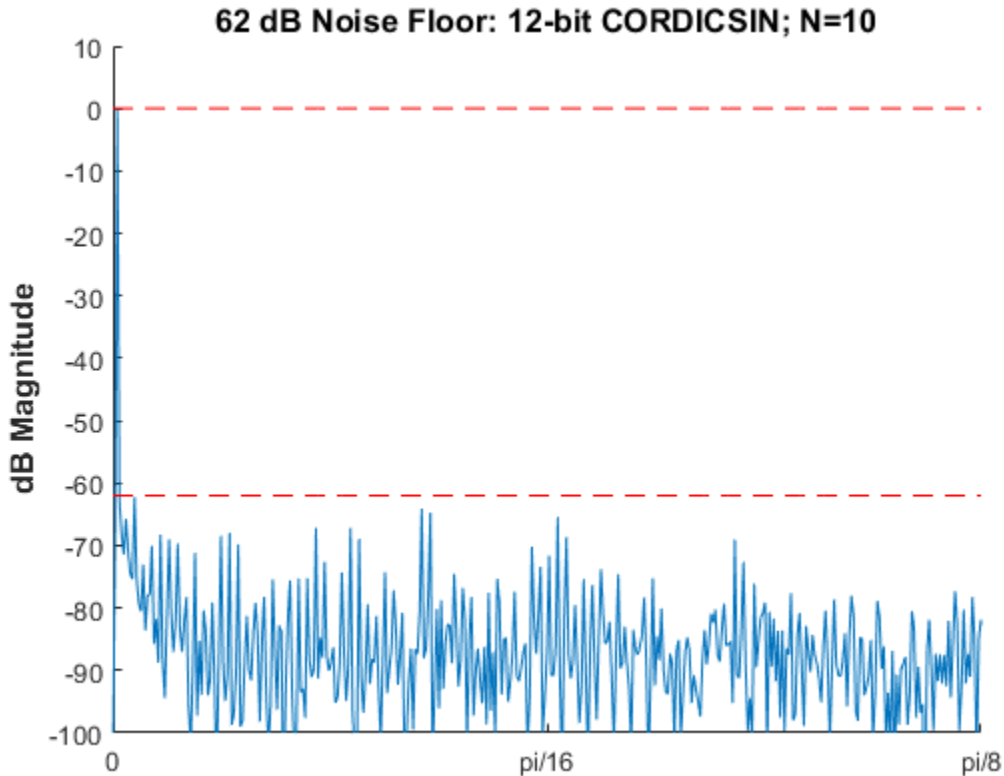
```
figure;
fracLen = cdcSinTh.FractionLength;
plot(thRadFxp, abs(errCdcRef) * pow2(fracLen));
set(gca, 'XTick', -2*pi:pi/2:2*pi);
set(gca, 'XTickLabel', ...
    {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
    '0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
ylabel(sprintf('LSB Error: 1 LSB = 2^{-%d}', fracLen), 'fontsize', 12, 'fontweight', 'b');
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® `fiaccl` 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 `fiaccl`.

The `fiaccl` 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 function  
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 Designer 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 Designer `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., `tblValsNT = numerictype(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);
quarterSinDb1FltPtVals = (sin(2*pi*((0:63) ./ 256)))';
endpointQuantized_Plus1 = 1.0 - double(eps(fi(0,tblValsNT)));

halfSinWaveDb1FltPtVals = ...
    [quarterSinDb1FltPtVals; ...
    endpointQuantized_Plus1; ...
    flipud(quarterSinDb1FltPtVals(2:end))];
```

```
fullSinWaveDb1FltPtVals = ...  
    [halfSinWaveDb1FltPtVals; -halfSinWaveDb1FltPtVals];  
  
FI_SIN_LUT = fi(fullSinWaveDb1FltPtVals, tblValsNT);
```

Overview of Algorithm Implementation

The implementation of the Fixed-Point Designer `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 $[0, 2\pi]$. For this purpose, a modulo- 2π operation is performed to obtain the fixed-point input value `inpValInRange` in the range $[0, 2\pi]$ 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:

```
idxUFI16 = 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 * idxUFI16;  
idxUFI16(:) = fullScaleIdx;
```

The top 8 most significant bits (MSBs) of this full-scale unsigned 16-bit index `idxUFI16` 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 = storedInteger(bitsliceget(idxUFI16, 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 `idxUFI16` 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] radians
inpValInRange = fi(0.425, inpInRangeNT); % arbitrary fixed-point input angle

% Normalize its stored integer to get full-scale unsigned 16-bit integer index
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 = 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:

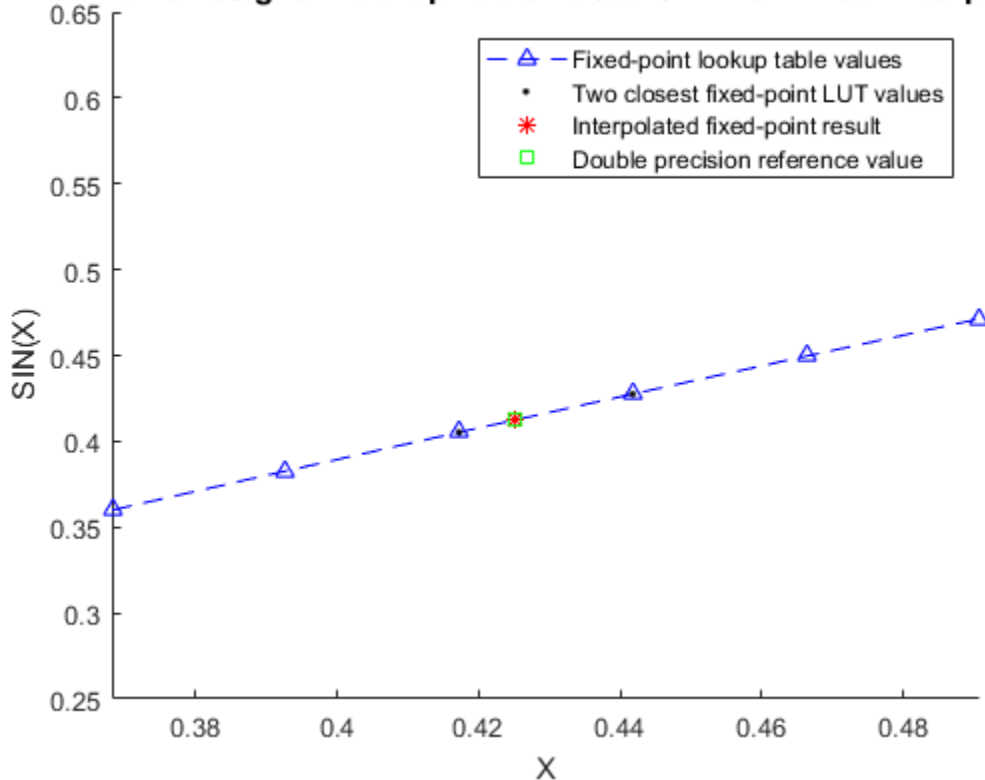
```

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 Designer Lookup Table Based SIN with Linear Interpolation', ...
      'fontsize',12,'fontweight','b');
```

Fixed-Point Designer 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-point
thRadFxp = sfi(thRadDb1, 12); % signed, 12-bit fixed-point inputs
```

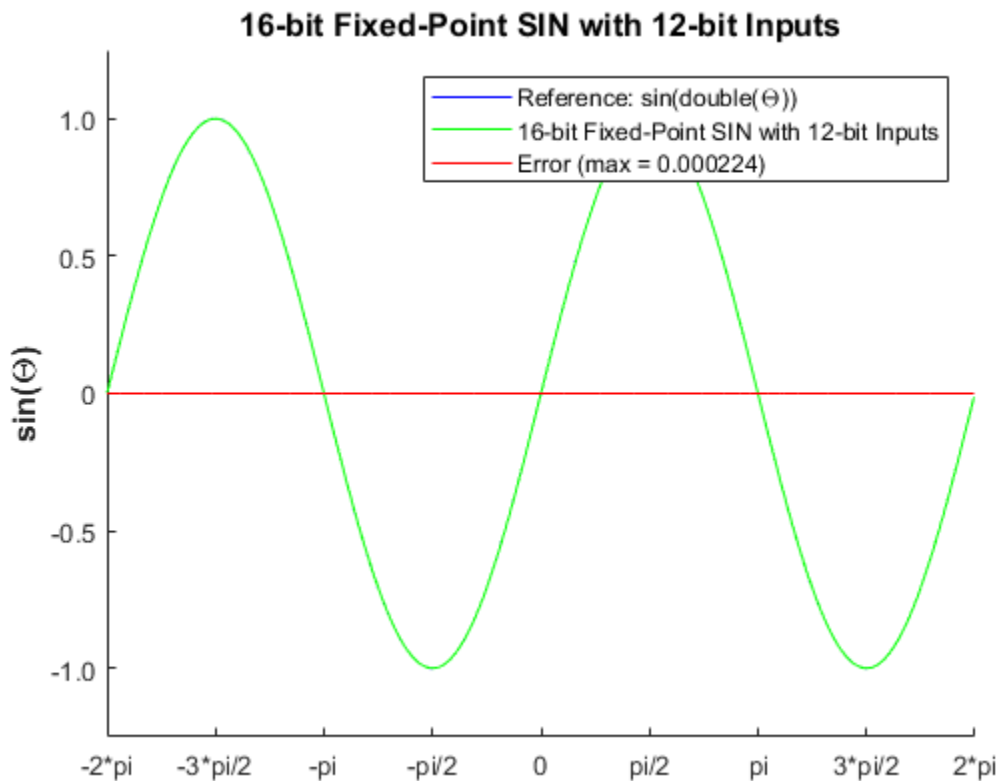
Compare fixed-point SIN vs. double-precision SIN results

```
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');

```

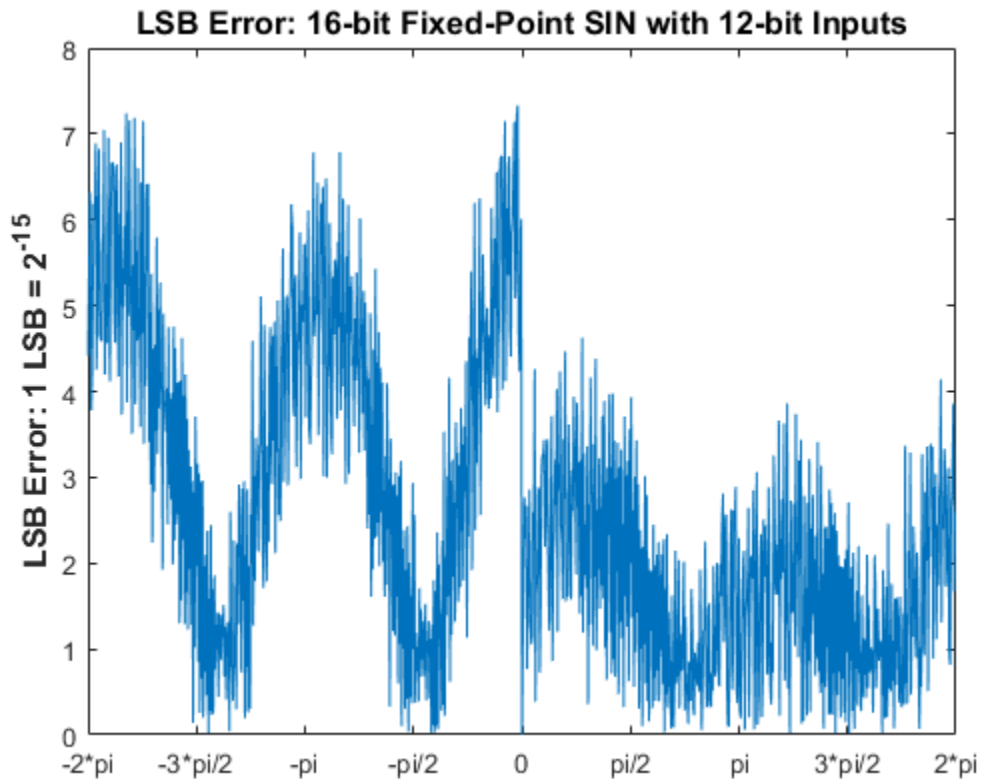


Compute the LSB Error

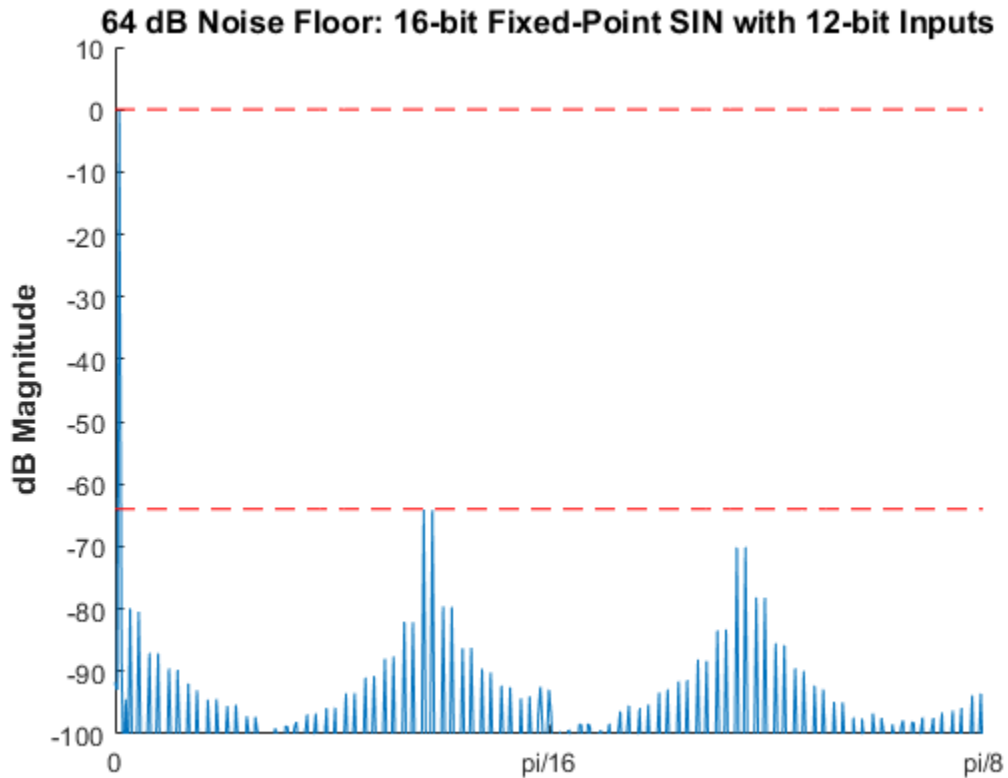
```

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, 'fontweight', 'b');
title('LSB Error: 16-bit Fixed-Point SIN with 12-bit Inputs', 'fontsize', 12, 'fontweight');
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(:) = x - ytmp;
        y(:) = y + xtmp;
        z(:) = z - inpLUT(idx);
    else
        x(:) = x + ytmp;
        y(:) = y - xtmp;
        z(:) = 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_private(inX, inY,
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 bits: %10g\n',
    (1:Niter); angleAccumulator(:)'; angleError(:)'; log2(abs(zf(:) - theta)));
```

```
Iteration:  1, Calculated angle:  45.000, Error in degrees:           2, Error in bits:
Iteration:  2, Calculated angle:  18.435, Error in degrees:  -24.5651, Error in bits:
Iteration:  3, Calculated angle:  32.471, Error in degrees:  -10.5288, Error in bits:
Iteration:  4, Calculated angle:  39.596, Error in degrees:   -3.40379, Error in bits:
Iteration:  5, Calculated angle:  43.173, Error in degrees:   0.172543, Error in bits:
Iteration:  6, Calculated angle:  41.383, Error in degrees:  -1.61737, Error in bits:
Iteration:  7, Calculated angle:  42.278, Error in degrees:  -0.722194, Error in bits:
Iteration:  8, Calculated angle:  42.725, Error in degrees:  -0.27458, Error in bits:
Iteration:  9, Calculated angle:  42.949, Error in degrees: -0.0507692, Error in bits:
Iteration: 10, Calculated angle:  43.061, Error in degrees:  0.0611365, Error in bits:
```

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 iterations
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 $\sqrt{x_0^2 + y_0^2} = 1$.

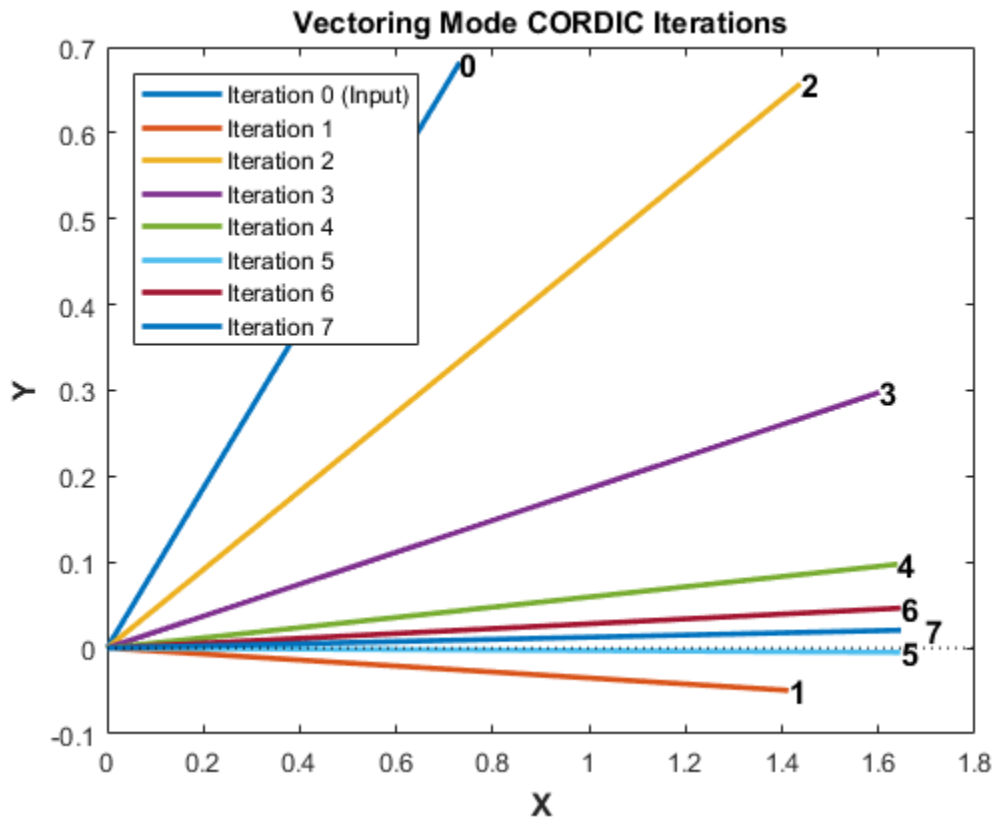
```
y_n = yf(end)
```

```
y_n =
    -0.0018
```

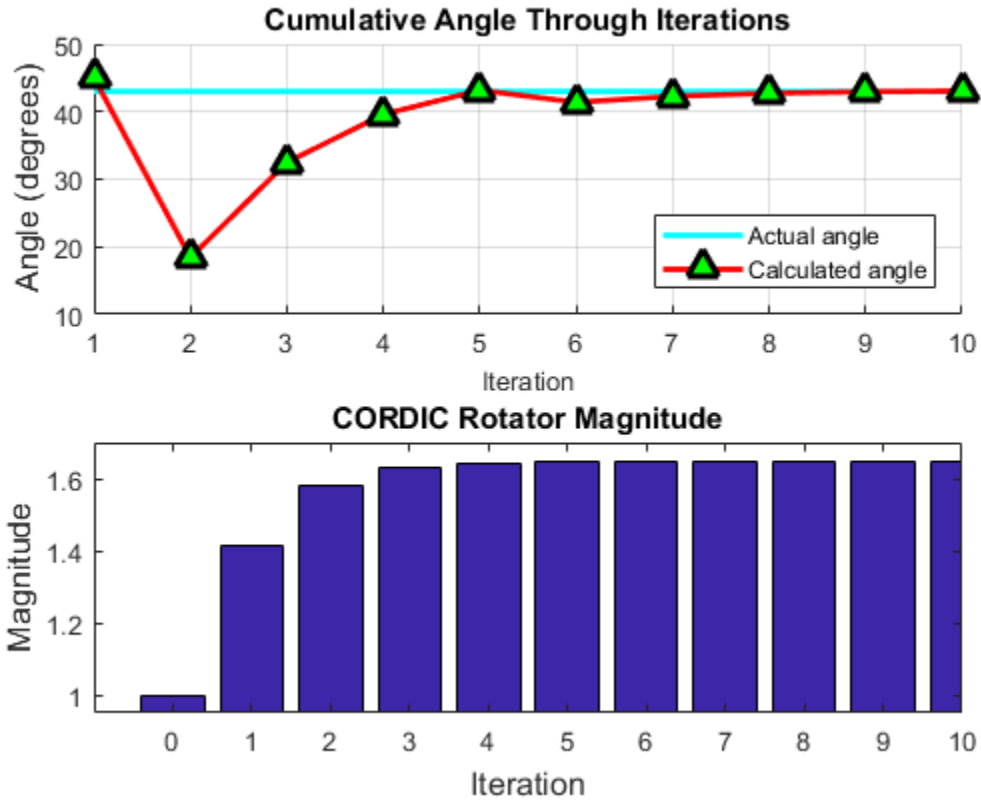
```
x_n = xf(end)
```

```
x_n =
    1.6468
```

```
figno = 1;
fidemo.fixpt_atan2_demo_plot(figno, xf, yf) %Vectoring Mode CORDIC Iterations
```



```
figno = figno + 1; %Cumulative Angle and Rotator Magnitude Through Iterations
fidemo.fixpt_atan2_demo_plot(figno,Niter, theta, angleAccumulator, rotatorMagnitude)
```



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 `bitsra(y,i)` and `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.

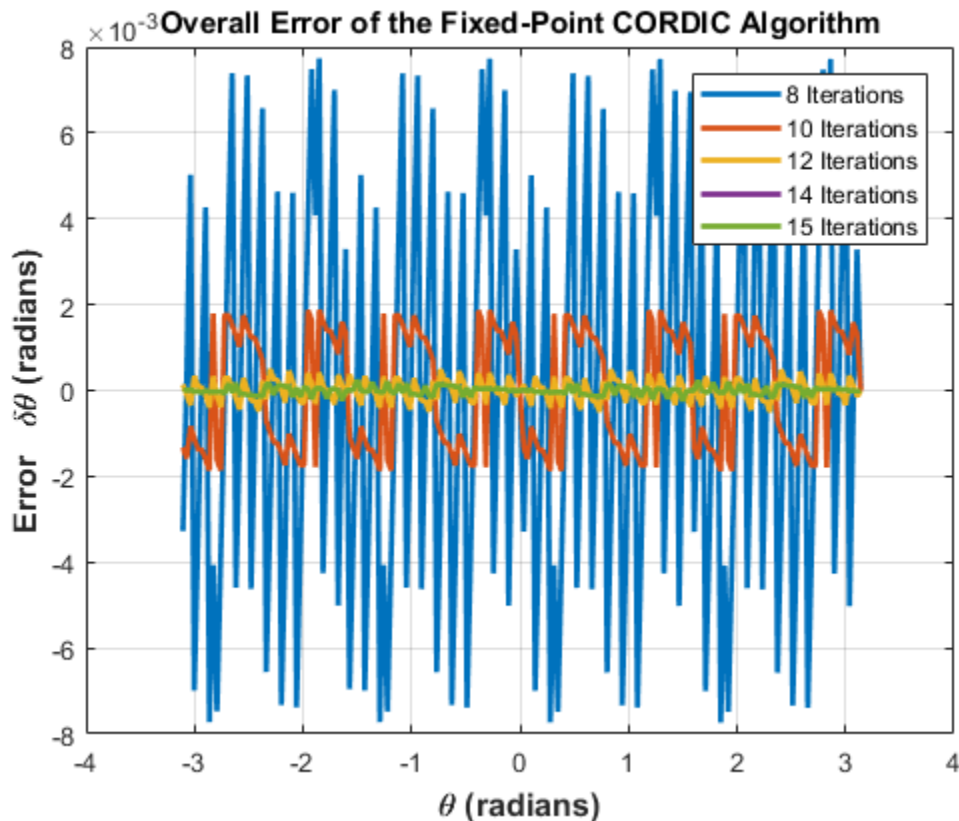
```
iterations = [8, 10, 12, 14, 15];
max_cordicErr_real_world_value = max(abs(cordic_err'));
fprintf('Iterations: %2d, Max error in real-world-value: %g\n',...
        [iterations; max_cordicErr_real_world_value]);
```

```
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 FIACCCEL

You can generate a MEX function from MATLAB code using the MATLAB® `fiaccel` 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 `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 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 the function
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

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

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 $\operatorname{atan}(x)$ 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(:)']');
```

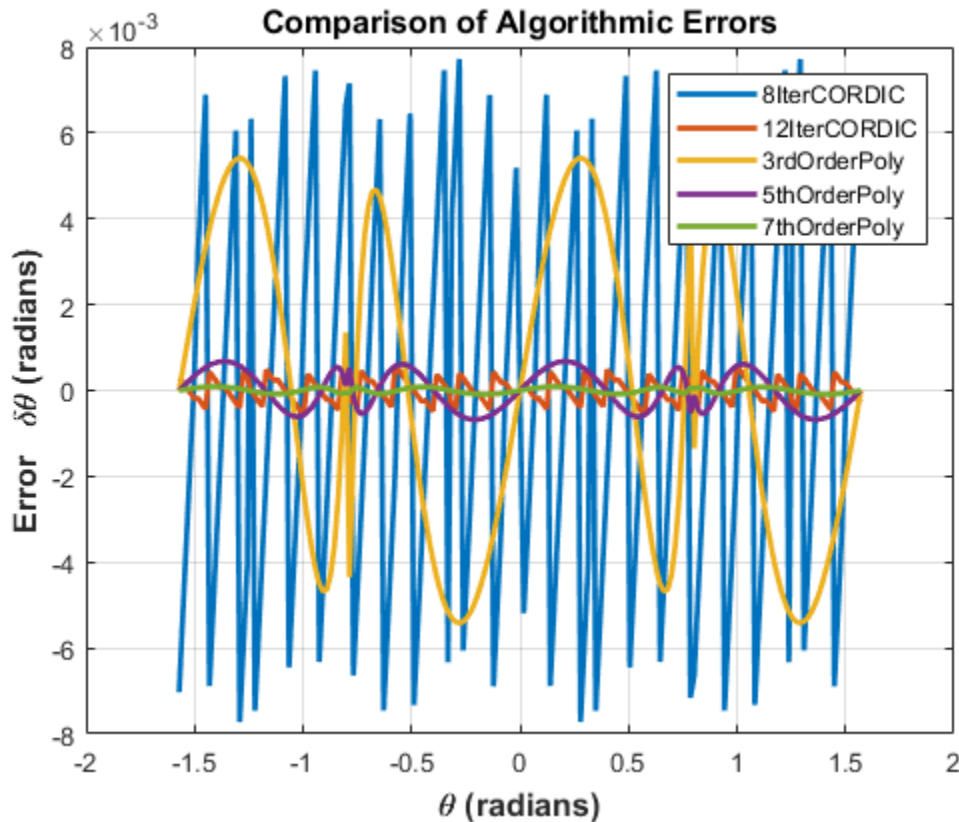
```
Order: 3, Polynomial approximation algorithmic error in real-world-value: 0.00541647
Order: 5, Polynomial approximation algorithmic error in real-world-value: 0.000679384
Order: 7, Polynomial approximation algorithmic error in real-world-value: 9.16204e-05
```

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\n',...
        [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
end
```



```

    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', RoundingMethodStr);
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, RoundingMethodStr);
    else
        % (pi/4, pi/2)
        z(idx) = pi/2 - feval(fhandle, abs(x(idx)), abs(y(idx)), N, constA, Tz, RoundingMethodStr);
    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
    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'); % 3rd order
zfix5p = fidemo.poly_atan2(inYfix, inXfix, ord(2), constAfix5, Tz, 'Floor'); % 5th order
zfix7p = fidemo.poly_atan2(inYfix, inXfix, ord(3), constAfix7, Tz, 'Floor'); % 7th order
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]);
```

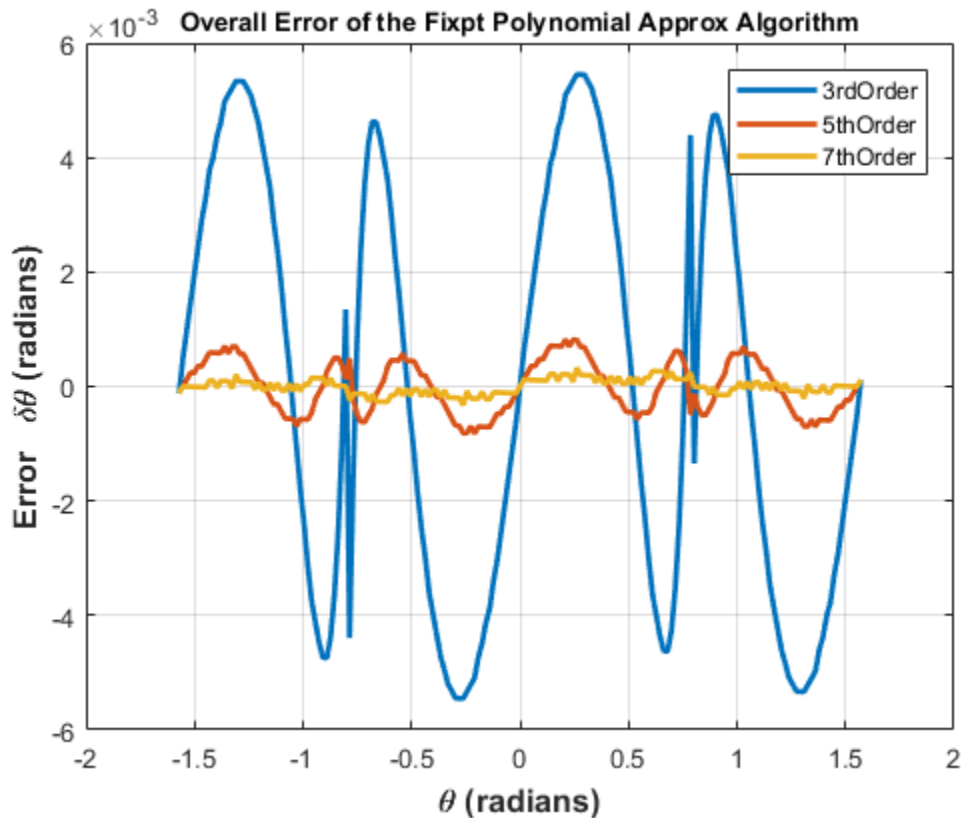
```
PolyOrder: 3, Overall error in bits: -7.51907
```

```
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 Designer™ 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 Designer `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 Designer 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 = bitsliceget(idxUFIX16, 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 `idxUFI16` 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(idxUFI16,8,1), 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

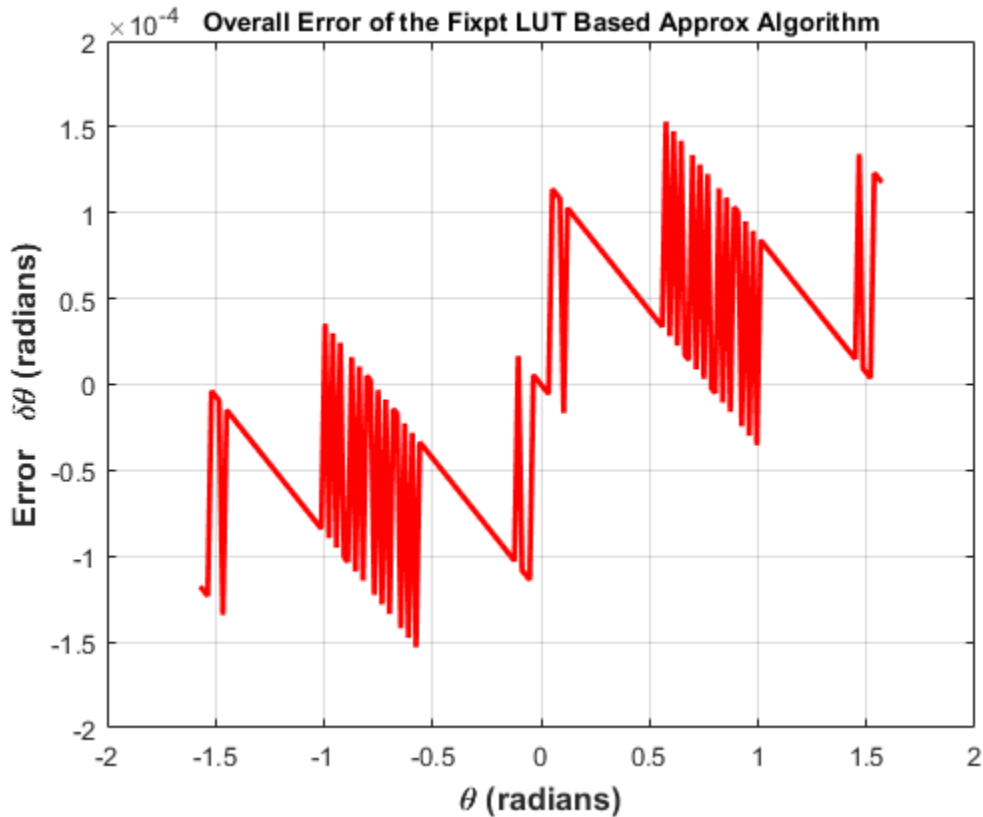
You can compute the overall error by comparing the fixed-point lookup table based approximation to the builtin `atan2` function. The ideal reference output is `zfltRef`.

```
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;
```

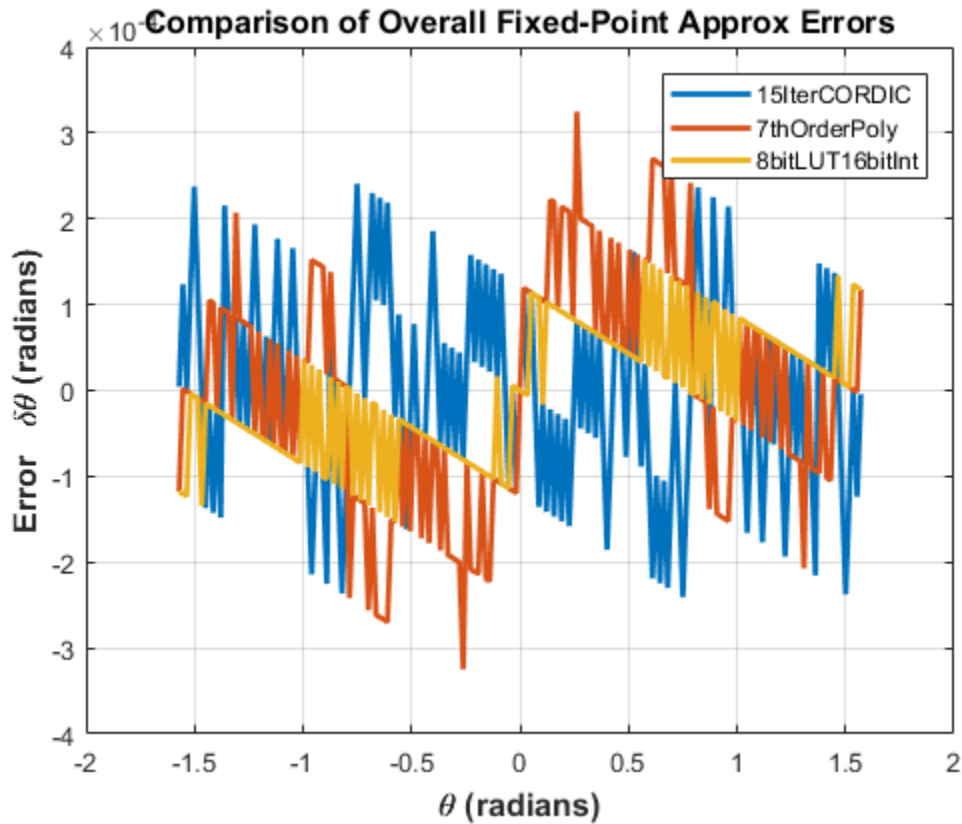
```
fidemo.fixpt_atan2_demo_plot(figno, theta, lut_err)
```



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_err)
```



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 $\text{atan}(2.^{- (0:N-1)})$.
- 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

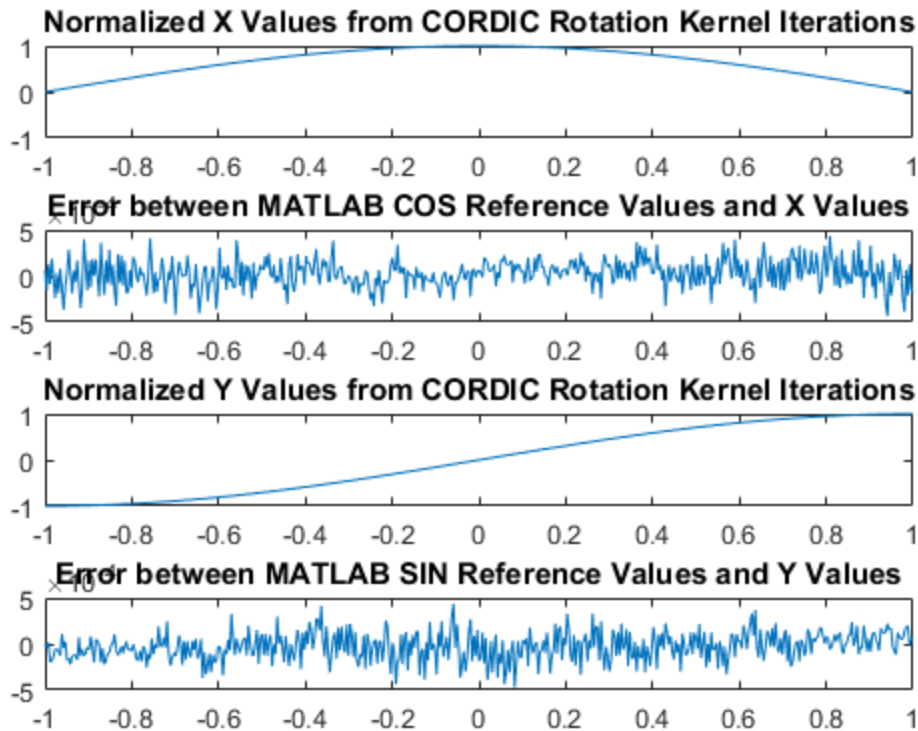
for idx = 1:length(theta)
    % CORDIC rotation kernel iterations
    [x_out(idx), y_out(idx), z_out(idx)] = ...
        fidemo.cordic_rotation_kernel(...
            fi(inv_An, xyNT), fi(0, xyNT), theta(idx), inpLUT, nitters);
end

% 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 = get(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, $Q = \text{eye}(m)$ 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	0
1	1.414213562373095	0.414213562373095	0.707106781186547	-0.292893218813453
2	1.581138830084190	0.166925267711095	0.632455532033676	-0.074651249152872
3	1.629800601300662	0.048661771216473	0.613571991077896	-0.018883540955780
4	1.642484065752237	0.012683464451575	0.608833912517752	-0.004738078560144
5	1.645688915757255	0.003204850005018	0.607648256256168	-0.001185656261584
6	1.646492278712479	0.000803362955224	0.607351770141296	-0.000296486114872
7	1.646693254273644	0.000200975561165	0.607277644093526	-0.000074126047770
8	1.646743506596901	0.000050252323257	0.607259112298893	-0.000018531794633
9	1.646756070204878	0.000012563607978	0.607254479332562	-0.000004632966330
10	1.646759211139822	0.000003140934944	0.607253321089875	-0.000001158242687
11	1.646759996375617	0.000000785235795	0.607253031529134	-0.000000289560741
12	1.646760192684695	0.000000196309077	0.607252959138945	-0.000000072390190
13	1.646760241761972	0.000000049077277	0.607252941041397	-0.000000018097548
14	1.646760254031292	0.000000012269320	0.607252936517010	-0.000000004524387
15	1.646760257098622	0.000000003067330	0.607252935385914	-0.000000001131097
16	1.646760257865455	0.000000000766833	0.607252935103139	-0.000000000282774
17	1.646760258057163	0.000000000191708	0.607252935032446	-0.000000000070694
18	1.646760258105090	0.000000000047927	0.607252935014772	-0.000000000017673
19	1.646760258117072	0.000000000011982	0.607252935010354	-0.000000000004418
20	1.646760258120067	0.000000000002995	0.607252935009249	-0.000000000001105
21	1.646760258120816	0.000000000000749	0.607252935008973	-0.000000000000276
22	1.646760258121003	0.000000000000187	0.607252935008904	-0.000000000000069
23	1.646760258121050	0.000000000000047	0.607252935008887	-0.000000000000017
24	1.646760258121062	0.000000000000012	0.607252935008883	-0.000000000000004
25	1.646760258121065	0.000000000000003	0.607252935008882	-0.000000000000001
26	1.646760258121065	0.000000000000001	0.607252935008881	-0.000000000000000
27	1.646760258121065	0	0.607252935008881	0
28	1.646760258121065	0	0.607252935008881	0
29	1.646760258121065	0	0.607252935008881	0
30	1.646760258121065	0	0.607252935008881	0
31	1.646760258121065	0	0.607252935008881	0
32	1.646760258121065	0	0.607252935008881	0

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 $Q = \text{eye}(3)$.

$$R = A = \begin{bmatrix} -0.8201 & 0.3573 & -0.0100 \\ -0.7766 & -0.0096 & -0.7048 \\ -0.7274 & -0.6206 & -0.8901 \end{bmatrix}$$

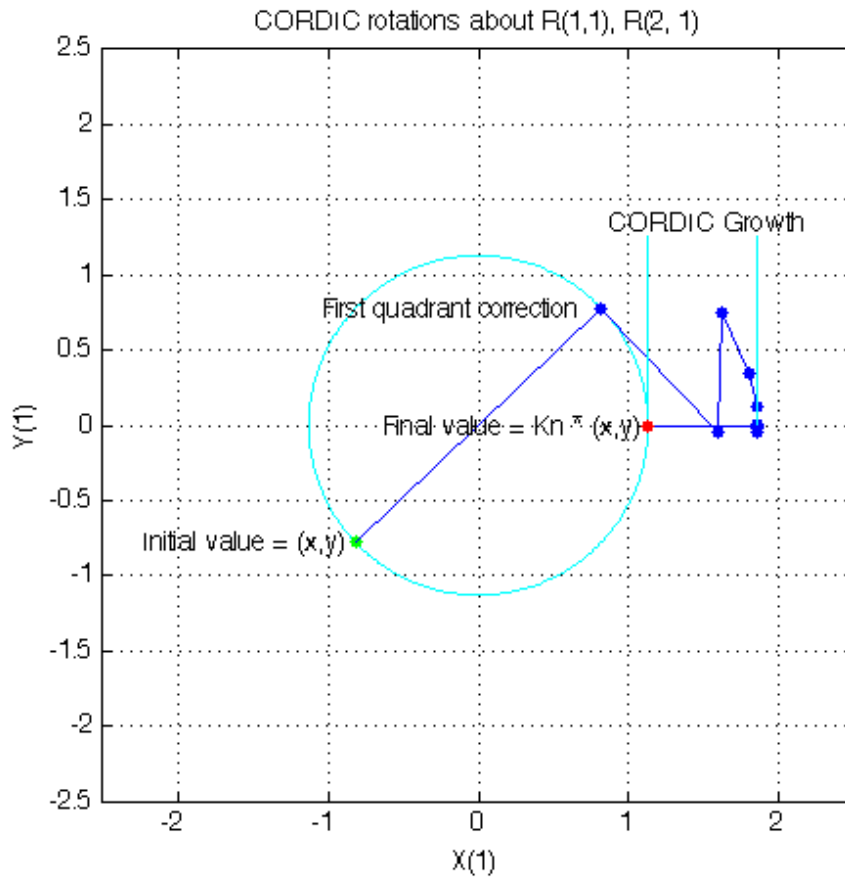
$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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.

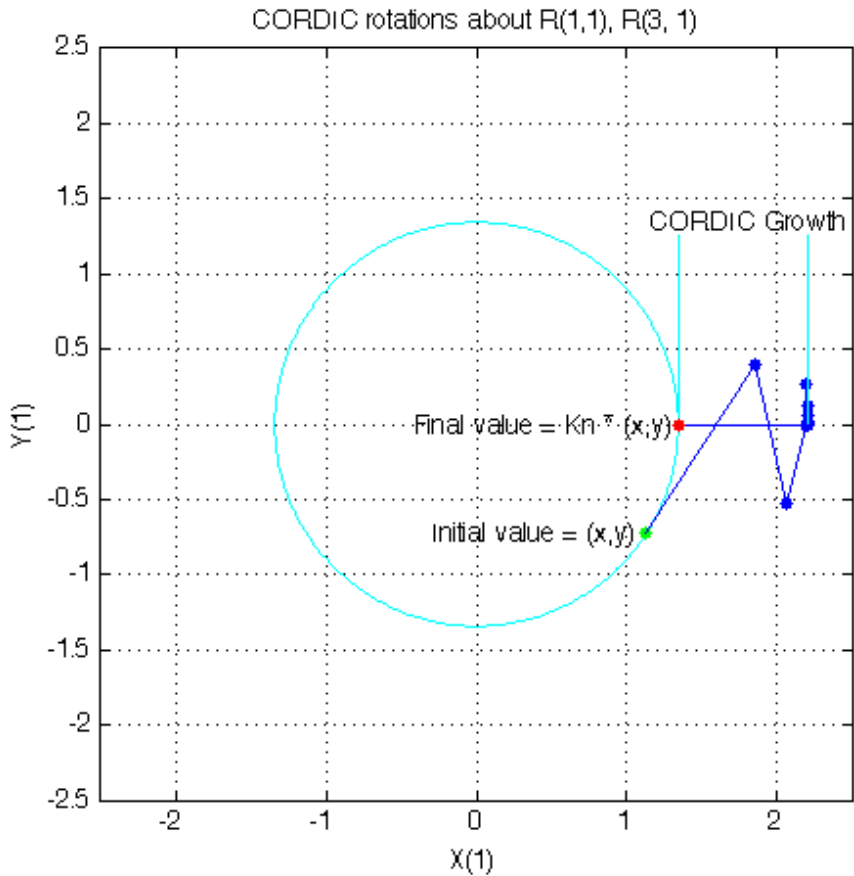


The second rotation is about the first and third row of R and the first and third column of Q. Element $R(1, 1)$ is the pivot and $R(3, 1)$ rotates to 0.

	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]	

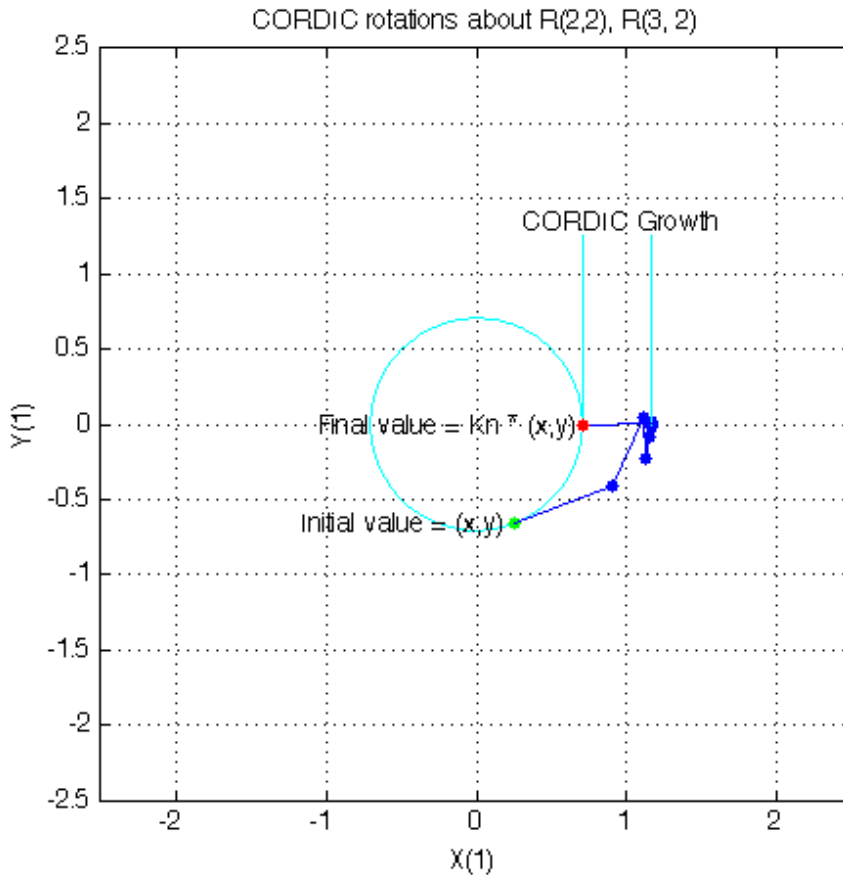
$$\begin{bmatrix} -0.6876 & -0.7261 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} -0.5781 & -0.7261 & -0.3723 \\ -0.5415 & 0 & 0.8407 \end{bmatrix}$$



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.

<p>R before the third rotation</p> $\begin{bmatrix} 1.3434 & 0.1235 & 0.8954 \\ 0 & 0.2527 & 0.5049 \\ 0 & -0.6586 & -0.4820 \end{bmatrix}$ <p>Q before the third rotation</p>	<p>-></p>	<p>R after the third rotation</p> $\begin{bmatrix} 1.3434 & 0.1235 & 0.8954 \\ 0 & 0.7054 & 0.6308 \\ 0 & 0 & 0.2987 \end{bmatrix}$ <p>Q after the third rotation</p>
--	--------------	---

$$\begin{array}{r}
 -0.6105 \quad [\quad u \quad] \quad [\quad v \quad] \\
 -0.5781 \quad [0.6876] \quad [-0.3932] \\
 -0.5415 \quad [-0.7261] \quad [-0.3723] \quad \rightarrow \\
 \quad \quad \quad [\quad \quad 0] \quad [0.8407]
 \end{array}
 \begin{array}{r}
 -0.6105 \quad [\quad u \quad] \quad [\quad v \quad] \\
 -0.5781 \quad [0.6134] \quad [0.5011] \\
 -0.5415 \quad [0.0875] \quad [-0.8113] \\
 \quad \quad \quad [-0.7849] \quad [0.3011]
 \end{array}$$



This completes the QR factorization. R is upper triangular, and Q is orthogonal.

$$R = \begin{array}{ccc}
 1.3434 & 0.1235 & 0.8954 \\
 0 & 0.7054 & 0.6308 \\
 0 & 0 & 0.2987
 \end{array}$$

```

Q =
-0.6105    0.6134    0.5011
-0.5781    0.0875   -0.8113
-0.5415   -0.7849    0.3011

```

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' =  1.0000    0.0000    0.0000
        0.0000    1.0000         0
        0.0000         0    1.0000

Q'*Q =  1.0000    0.0000   -0.0000
        0.0000    1.0000   -0.0000
       -0.0000   -0.0000    1.0000

```

You can see the error difference by subtracting the identity matrix.

```

Q*Q' - eye(size(Q)) =
                        0  2.7756e-16  3.0531e-16
                    2.7756e-16  4.4409e-16         0
                    3.0531e-16         0  6.6613e-16

```

You can verify that Q^*R is close to A by subtracting to see the error difference.

```

Q*R - A =  -3.7802e-11  -7.2325e-13  -2.7756e-17
           -3.0512e-10   1.1708e-12  -4.4409e-16
           3.6836e-10  -4.3487e-13  -7.7716e-16

```

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) || isscaleddouble(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.

```
niter = 52;
[Q,R] = cordicqr(A,niter)
```

$$Q = \begin{bmatrix} 0.5000 & -0.5000 & 0.5000 & 0.5000 \\ 0.5000 & 0.5000 & -0.5000 & 0.5000 \\ 0.5000 & -0.5000 & -0.5000 & -0.5000 \end{bmatrix}$$

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
    
```

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.');
```

```
end
```

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 `Q*R - 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
```


$$\begin{bmatrix} 0.0000 & 1.0000 & 0 \\ 0 & 0 & 1.0000 \end{bmatrix}$$

$Q*Q'$

ans =

$$\begin{bmatrix} 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.0000 & -0.0000 \\ 0.0000 & -0.0000 & 1.0000 \end{bmatrix}$$

And they both satisfy the requirement that $Q*R - A$ is small:

$Q*Q'R_0 - A$

ans =

$$1.0e-15 * \begin{bmatrix} -0.1110 & -0.1110 & -0.1110 \\ -0.1110 & -0.1110 & -0.1110 \\ -0.1110 & -0.1110 & -0.1110 \end{bmatrix}$$

$Q*R - A$

ans =

$$1.0e-15 * \begin{bmatrix} -0.2220 & 0.2220 & 0.2220 \\ 0.4441 & 0 & 0 \\ 0.2220 & 0.2220 & 0.2220 \end{bmatrix}$$

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 Designer™

- `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

This example shows how to compute square root using a CORDIC 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 ($i = 4, 13, 40, 121, \dots, k, 3k+1, \dots$) are repeated to achieve result convergence.

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.

CORDIC EQUATIONS IN HYPERBOLIC VECTORING MODE

The hyperbolic vectoring mode is used for computing **square root**.

For the vectoring mode, the CORDIC equations are as follows:

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

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

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

where

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

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

- $x_N \approx A_N \sqrt{x_0^2 - y_0^2}$
- $y_N \approx 0$
- $z_N \approx z_0 + \operatorname{atanh}(y_0/x_0)$

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.

Note also that for **square root** we will use only the x_N result.

MATLAB Implementation of a CORDIC Hyperbolic Vectoring Algorithm

A MATLAB code implementation example of the CORDIC Hyperbolic 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 data types.

CORDIC Hyperbolic Vectoring Kernel

```
k = 4; % Used for the repeated (3*k + 1) iteration steps
for idx = 1:n
    xtmp = bitsra(x, idx); % multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % multiply by 2^(-idx)
    if y < 0
        x(:) = x + ytmp;
        y(:) = y + xtmp;
```

```

        z(:) = z - atanhLookupTable(idx);
else
    x(:) = x - ytmp;
    y(:) = y - xtmp;
    z(:) = z + atanhLookupTable(idx);
end

if idx==k
    xtmp = bitsra(x, idx); % multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % multiply by 2^(-idx)
    if y < 0
        x(:) = x + ytmp;
        y(:) = y + xtmp;
        z(:) = z - atanhLookupTable(idx);
    else
        x(:) = x - ytmp;
        y(:) = y - xtmp;
        z(:) = z + atanhLookupTable(idx);
    end
    k = 3*k + 1;
end
end % idx loop

```

CORDIC-Based Square Root Computation

Square Root Computation Using the CORDIC Hyperbolic Vectoring 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:

- x_0 is set to $v + 0.25$.
- y_0 is set to $v - 0.25$.

After N iterations, these initial values lead to the following output as N approaches $+\infty$:

$$x_N \approx A_N \sqrt{(v + 0.25)^2 - (v - 0.25)^2}$$

This may be further simplified as follows:

$$x_N \approx A_N \sqrt{v}$$

where A_N is the CORDIC gain as defined above.

Note: for square root, z and `atanhLookupTable` have no impact on the result. Hence, z and `atanhLookupTable` are not used.

MATLAB Implementation of a CORDIC Square Root Kernel

A MATLAB code implementation example of the CORDIC Square Root Kernel algorithm follows (for the case of scalar x and y). This same code can be used for both fixed-point and floating-point data types.

CORDIC Square Root Kernel

```

k = 4; % Used for the repeated (3*k + 1) iteration steps

for idx = 1:n
    xtmp = bitsra(x, idx); % multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % multiply by 2^(-idx)
    if y < 0
        x(:) = x + ytmp;
        y(:) = y + xtmp;
    else
        x(:) = x - ytmp;
        y(:) = y - xtmp;
    end

    if idx==k
        xtmp = bitsra(x, idx); % multiply by 2^(-idx)
        ytmp = bitsra(y, idx); % multiply by 2^(-idx)
        if y < 0
            x(:) = x + ytmp;
            y(:) = y + xtmp;
        else
            x(:) = x - ytmp;
            y(:) = y - xtmp;
        end
        k = 3*k + 1;
    end
end % idx loop

```

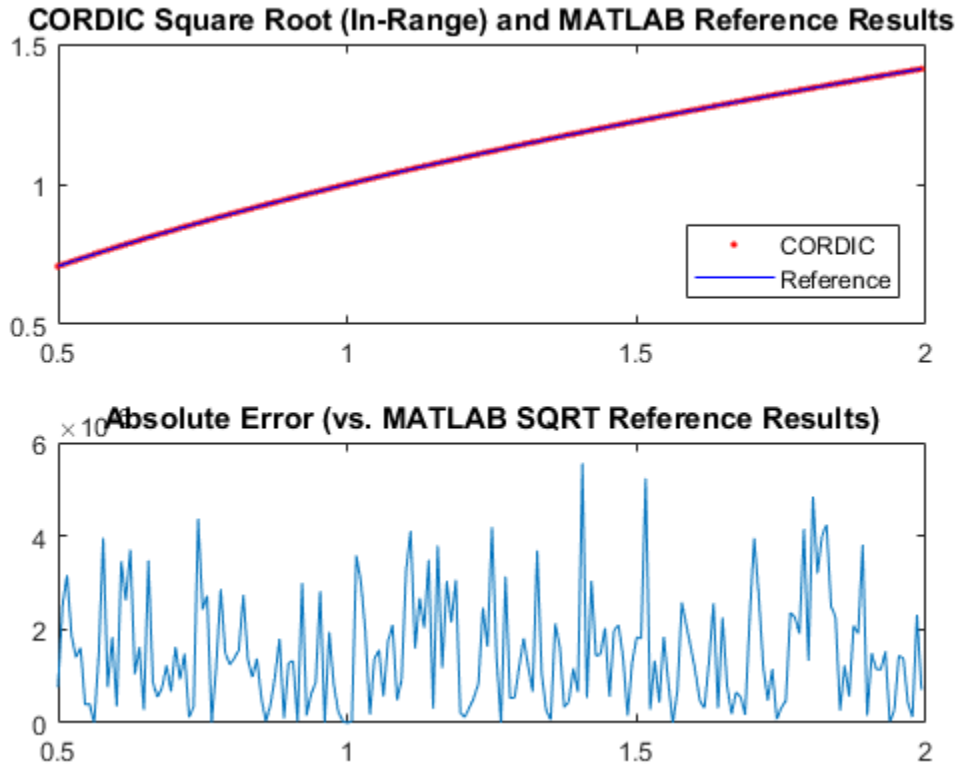
This code is identical to the **CORDIC Hyperbolic Vectoring Kernel** implementation above, except that z and `atanhLookupTable` are not used. This is a cost savings of 1 table lookup and 1 addition per iteration.

Example

Use the CORDICSQRT function to compute the approximate square root of `v_fix` using ten CORDIC kernel iterations:

```
step = 2^-7;
v_fix = fi(0.5:step:(2-step), 1, 20); % fixed-point inputs in range [.5, 2)
niter = 10; % number of CORDIC iterations
x_sqr = cordicsqrt(v_fix, niter);

% Get the Real World Value (RWV) of the CORDIC outputs for comparison
% and plot the error between the MATLAB reference and CORDIC sqrt values
x_cdc = double(x_sqr); % CORDIC results (scaled by An_hp)
v_ref = double(v_fix); % Reference floating-point input values
x_ref = sqrt(v_ref); % MATLAB reference floating-point results
figure;
subplot(211);
plot(v_ref, x_cdc, 'r.', v_ref, x_ref, 'b-');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root (In-Range) and MATLAB Reference Results');
subplot(212);
absErr = abs(x_ref - x_cdc);
plot(v_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');
```



Overcoming Algorithm Input Range Limitations

Many square root algorithms normalize the input value, v , to within the range of $[0.5, 2)$ range. This pre-processing is typically done using a fixed word length normalization, and can be used to support small as well as large input value ranges.

The CORDIC-based square root algorithm implementation is particularly sensitive to inputs outside of this range. The function `CORDICSQRT` overcomes this algorithm range limitation through a normalization approach based on the following mathematical relationships:

$$v = u * 2^n, \text{ for some } 0.5 \leq u < 2 \text{ and some even integer } n.$$

Thus:

$$\sqrt{v} = \sqrt{u} * 2^{n/2}$$

In the CORDICSQRT function, the values for u and n , described above, are found during normalization of the input v . n is the number of leading zero most significant bits (MSBs) in the binary representation of the input v . These values are found through a series of bitwise logic and shifts. Note: because n must be even, if the number of leading zero MSBs is odd, one additional bit shift is made to make n even. The resulting value after these shifts is the value $0.5 \leq u < 2$.

u becomes the input to the CORDIC-based square root kernel, where an approximation to \sqrt{u} is calculated. The result is then scaled by $2^{n/2}$ so that it is back in the correct output range. This is achieved through a simple bit shift by $n/2$ bits. The (left or right) shift direction depends on the sign of n .

Example

Compute the square root of 10-bit fixed-point input data with a small non-negative range using CORDIC. Compare the CORDIC-based algorithm results to the floating-point MATLAB reference results over the same input range.

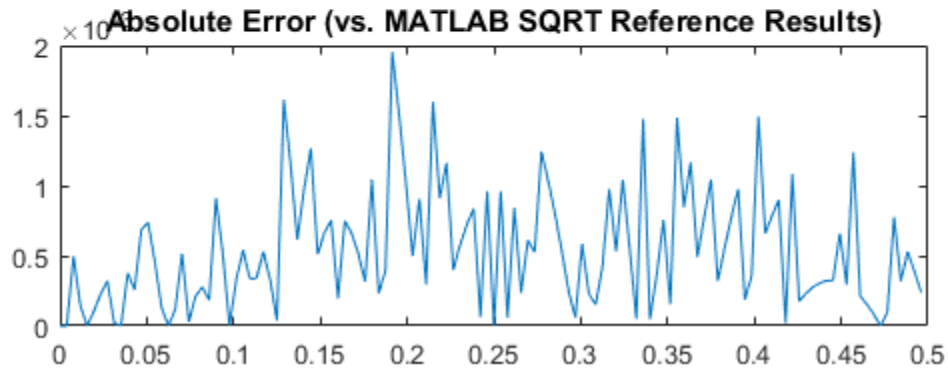
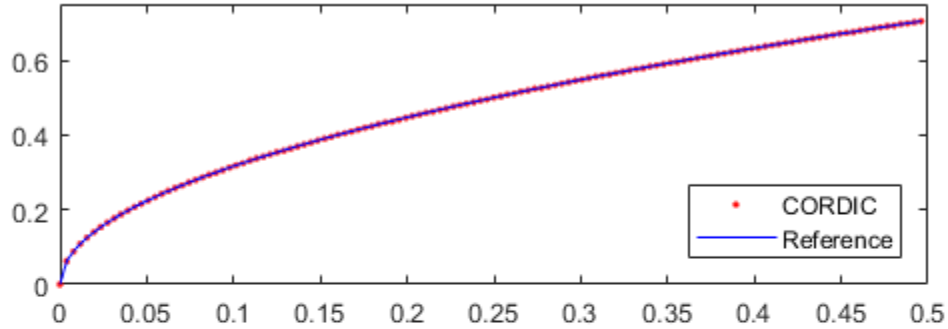
```

step      = 2^-8;
u_ref     = 0:step:(0.5-step); % Input array (small range of values)
u_in_arb  = fi(u_ref,0,10); % 10-bit unsigned fixed-point input data values
u_len     = numel(u_ref);
sqrt_ref  = sqrt(double(u_in_arb)); % MATLAB sqrt reference results
niter     = 10;
results   = zeros(u_len, 2);
results(:,2) = sqrt_ref(:);

% Compute the equivalent Real World Value result for plotting.
% Plot the Real World Value (RWV) of CORDIC and MATLAB reference results.
x_out = cordicsqrt(u_in_arb, niter);
results(:,1) = double(x_out);
figure;
subplot(211);
plot(u_ref, results(:,1), 'r.', u_ref, results(:,2), 'b-');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root (Small Input Range) and MATLAB Reference Results');
axis([0 0.5 0 0.75]);
subplot(212);
absErr = abs(results(:,2) - results(:,1));
plot(u_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');

```


CORDIC Square Root (Small Input Range) and MATLAB Reference Results



Example

Compute the square root of 16-bit fixed-point input data with a large positive range using CORDIC. Compare the CORDIC-based algorithm results to the floating-point MATLAB reference results over the same input range.

```

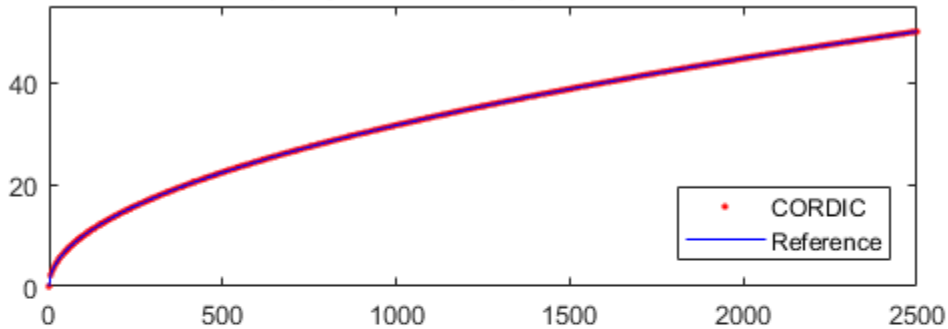
u_ref    = 0:5:2500;          % Input array (larger range of values)
u_in_arb = fi(u_ref,0,16);   % 16-bit unsigned fixed-point input data values
u_len    = numel(u_ref);
sqrt_ref = sqrt(double(u_in_arb)); % MATLAB sqrt reference results
niter    = 16;
results  = zeros(u_len, 2);
results(:,2) = sqrt_ref(:);
    
```

```

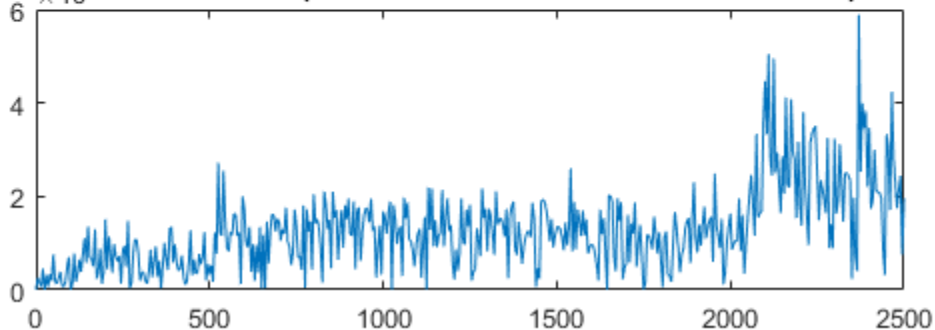
% Compute the equivalent Real World Value result for plotting.
% Plot the Real World Value (RWV) of CORDIC and MATLAB reference results.
x_out = cordicsqrt(u_in_arb, niter);
results(:,1) = double(x_out);
figure;
subplot(211);
plot(u_ref, results(:,1), 'r.', u_ref, results(:,2), 'b-');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root (Large Input Range) and MATLAB Reference Results');
axis([0 2500 0 55]);
subplot(212);
absErr = abs(results(:,2) - results(:,1));
plot(u_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');

```

CORDIC Square Root (Large Input Range) and MATLAB Reference Results



Absolute 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:

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

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

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

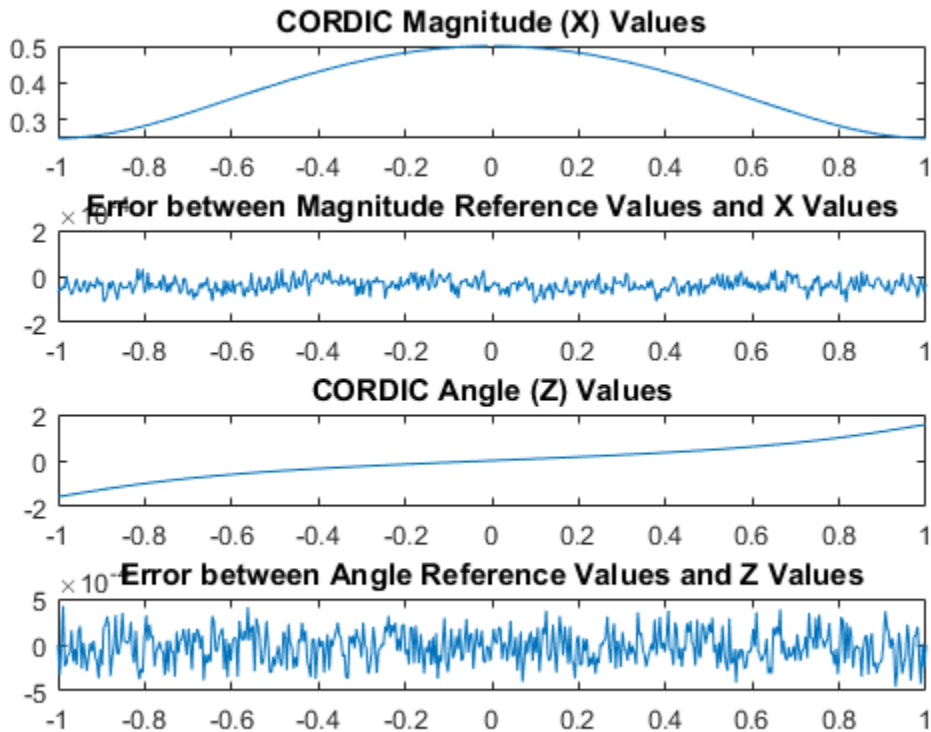
% Get the Real World Value (RWV) of the CORDIC outputs for comparison
% and plot the error between the (magnitude, angle) values
AnGain = prod(sqrt(1+2.^(-2*(0:(nitters-1))))); % CORDIC gain
x_c2p_RWV = (1/AnGain) .* double(x_c2p); % Magnitude (scaled by CORDIC gain)
z_c2p_RWV = (pi/2) .* double(z_c2p); % 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 = get(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 `fi` 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 `fi`, 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

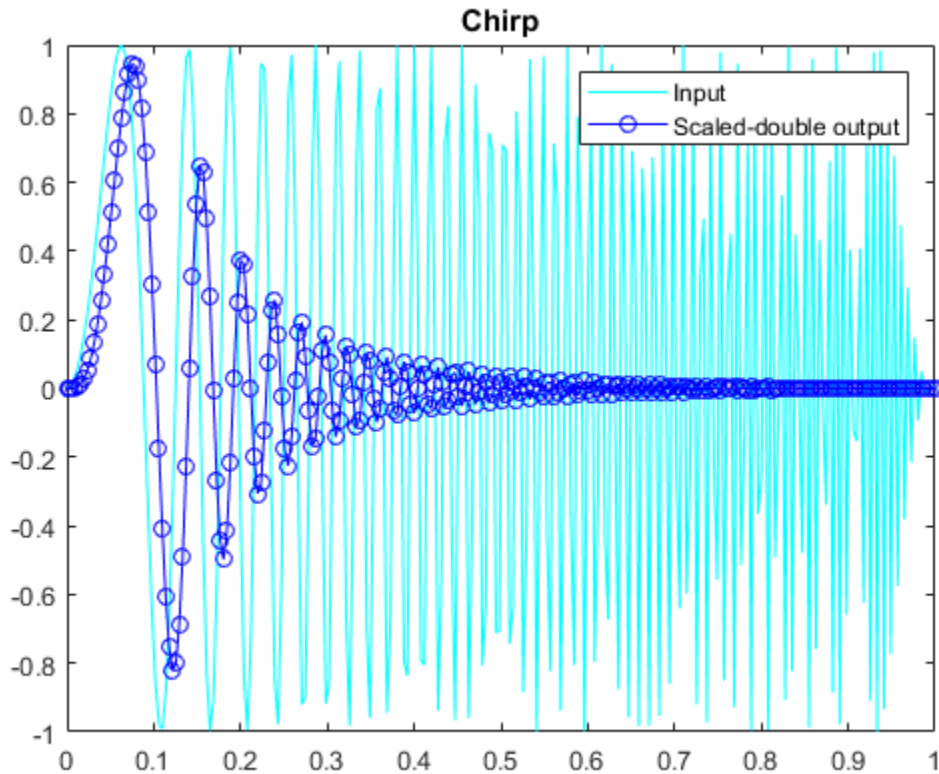
The instrumented MEX function must be run to record minimum and maximum values for that simulation run. Subsequent runs accumulate the instrumentation results until they are cleared with `clearInstrumentationResults`.

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:

- `-defaultDT T` Default data type to propose for doubles, where `T` is a numeric type object, or one of the strings `{remainFloat, double, single, int8, int16, int32, int64, uint8, uint16, uint32, uint64}`. The default is `remainFloat`.

- `-nocode` Do not show MATLAB code in the printable report. Display only the logged variables tables. This option only has effect in combination with the `-printable` option.
- `-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 and open in the system browser.
- `-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.

The screenshot shows the MATLAB Code Generation Report for the function `fi_2nd_order_df2t_filter`. The code is as follows:

```

1 function [y,z] = fi_2nd_order_df2t_filter(b,a,k,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(k)
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

```

Information for the selected expression:

Size	1 x 1
Class	embedded.fi
Complex	No
DT Mode	ScaledDouble
Signedness	Signed
WL	40
FL	29
Percent of Current Range	< 1
Always Whole Number	No
SimMin	-0.47000708911889938
SimMax	0.542233211730554

Summary of instrumentation results:

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	15	95	No	-0.8158220312628653	0.9446434665609258
2	z	IO	2 x 1	embedded.fi	No	ScaledDouble	Signed	16	15	15	93	No	-0.8098257866372123	0.9227265381763932
3	b	Input	1 x 3	embedded.fi	No	-	Signed	16	19	-	96	No	0.029954910278320312	0.059909820556640625
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	-0.999999482421875	0.99993896484375
6	j	Local	1 x 1	double	No	-	-	-	-	-	-	Yes	1	256

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.

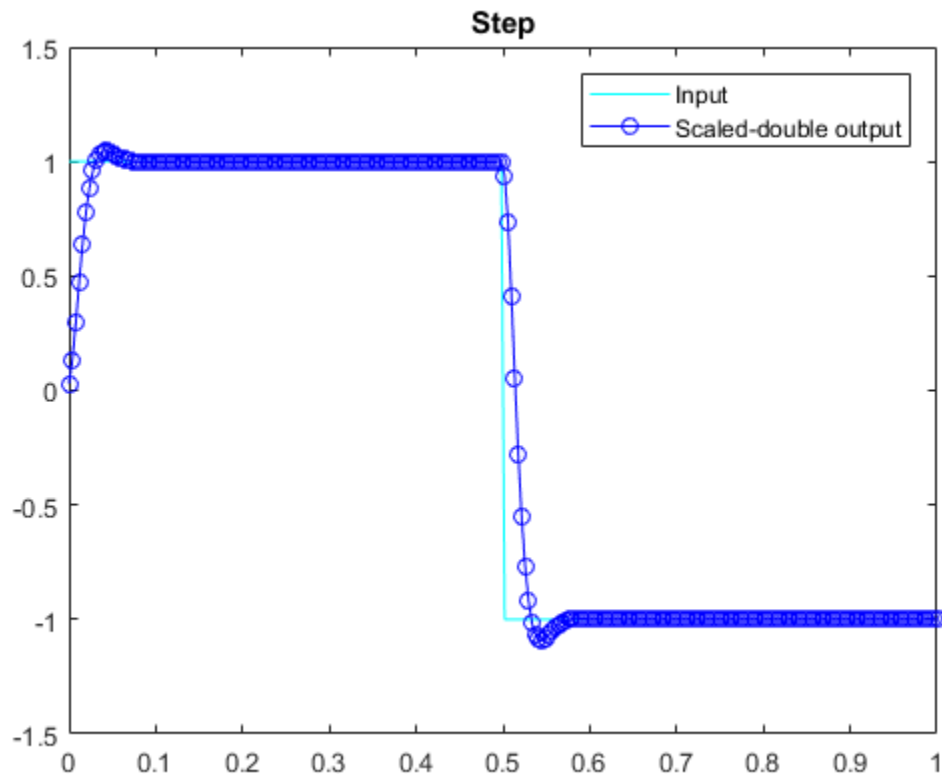
```
xstep = [ones(N/2,1); -ones(N/2,1)];
x(:) = xstep;
```

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
```


Code Generation Report

MATLAB code Call stack

Function: `fi_2nd_order_df2t_filter`

```

1 function [y,z] = fi_2nd_order_df2t_filter(b,a,k,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,z] = 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	296 x 1	embedded fi	No	ScaledDouble	Signed	16	15	14	110	No	-1.0811287150891327	1.045564982263348
2	z	IO	2 x 1	embedded fi	No	ScaledDouble	Signed	16	15	14	137	No	-1.0611738048108124	1.0156109861363434
3	b	Input	1 x 3	embedded fi	No	-	Signed	16	19	-	95	No	0.029954910278320312	0.059909820556640625
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	0.999969482421875
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.

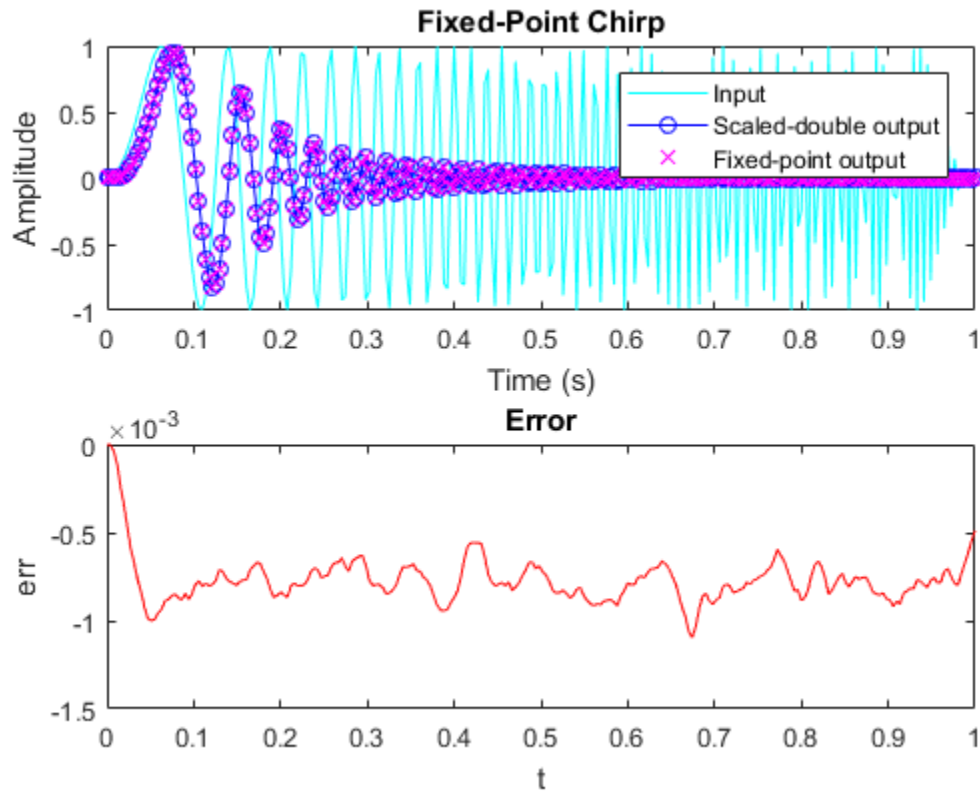
Validate with Chirp Input

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 a window titled "Code Generation Report" with a MATLAB code editor and a summary table.

MATLAB code:

```

Function: fi_2nd_order_df2t_filter
1 function [y,z] = fi_2nd_order_df2t_filter(b,a,k,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     z[j] = b(1)*x[j] + z(1);
7     z(1) = (b(2)*x[j] + z(2)) - a(2) * z[j];
8     z(2) = b(3)*x[j] - a(3) * z[j];
9 end
10 end
11

```

Summary Table:

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.f	No	Signed	16	14	48	No	-0.82061767578125	0.84378662106375
2	z	IO	2 x 1	embedded.f	No	Signed	16	14	47	No	-0.81000791015625	0.82183603515625
3	b	Input	1 x 3	embedded.f	No	Signed	16	19	96	No	0.029954910278320312	0.05890982055640625
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	-0.999999482421875	0.99993896484375
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);

```

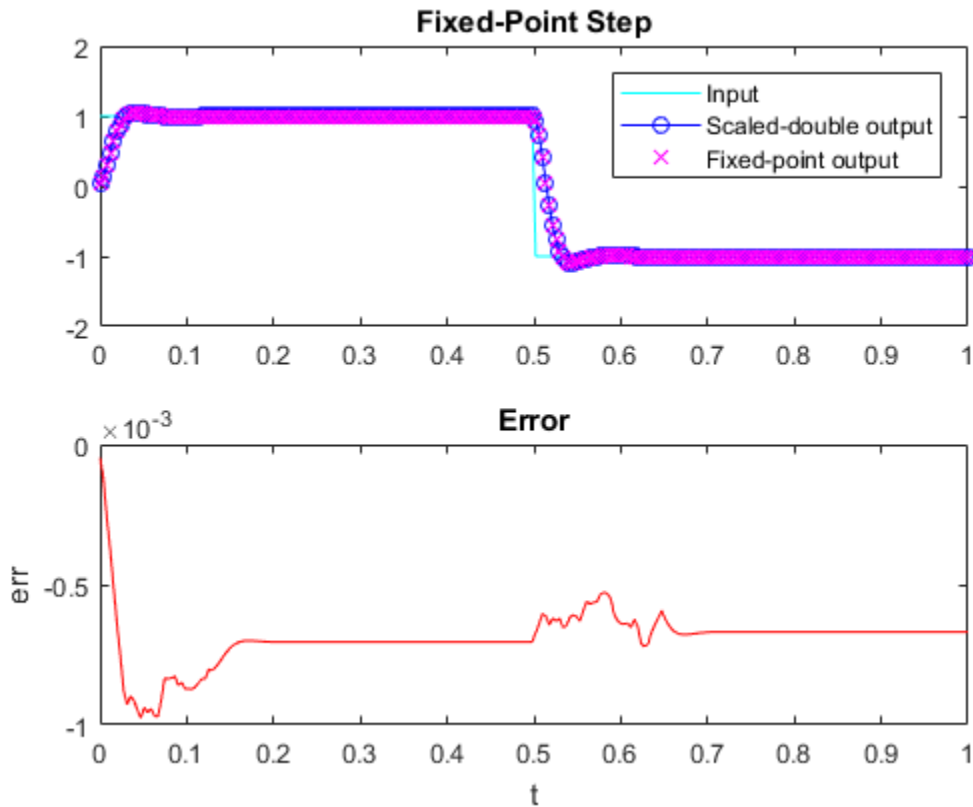
Plot the fixed-point outputs against the scaled-double outputs to verify that they meet your design criteria.

```
clf
```

```

subplot(211);plot(t,x,'c',t,ysd,'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,k,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	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	IO	256 x 1	embedded.f	No	Signed	16	14	55	No	-1.09173583684375	1.04461666921875
2	z	IO	2 x 1	embedded.f	No	Signed	16	14	54	No	-1.061767578125	1.01470947265625
3	b	Input	1 x 3	embedded.f	No	Signed	16	19	96	No	0.029954910278320312	0.059909820556640625
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	-1	0.999999482421875
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 Designer™ examples directory into a temporary directory so this example doesn't interfere with your own work.

```
tempdirObj = fidemo.fiTempdir('fi_radix2fft_demo');

% Copying important functions to the temporary directory
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
    'fi_m_radix2fft_algorithm1_6_2.m'),'.','f');
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
    'fi_m_radix2fft_algorithm1_6_2_typed.m'),'.','f');
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
    'fi_m_radix2fft_withscaling_typed.m'),'.','f');
```

Run the following code to capture current states, and reset the global states.

```
FIPREF_STATE = get(fipref);
reset(fipref)
```

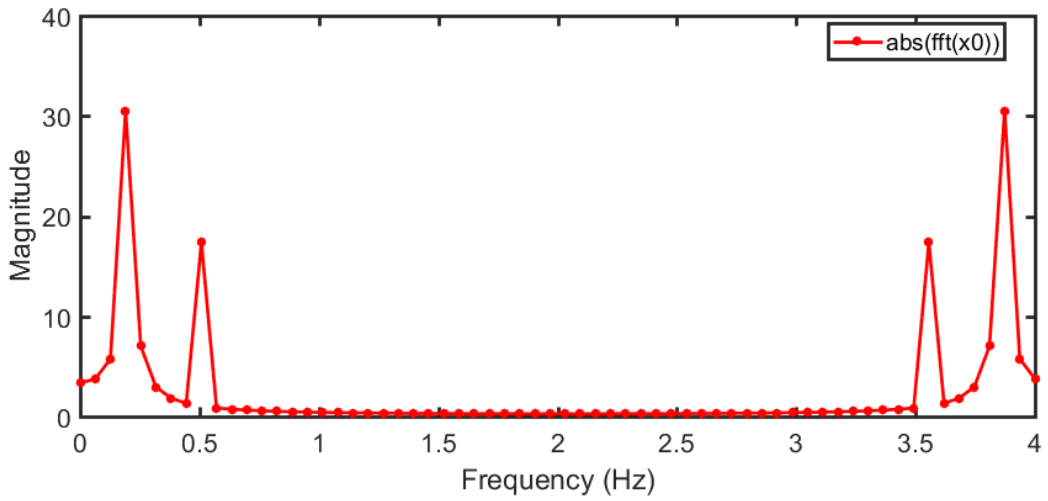
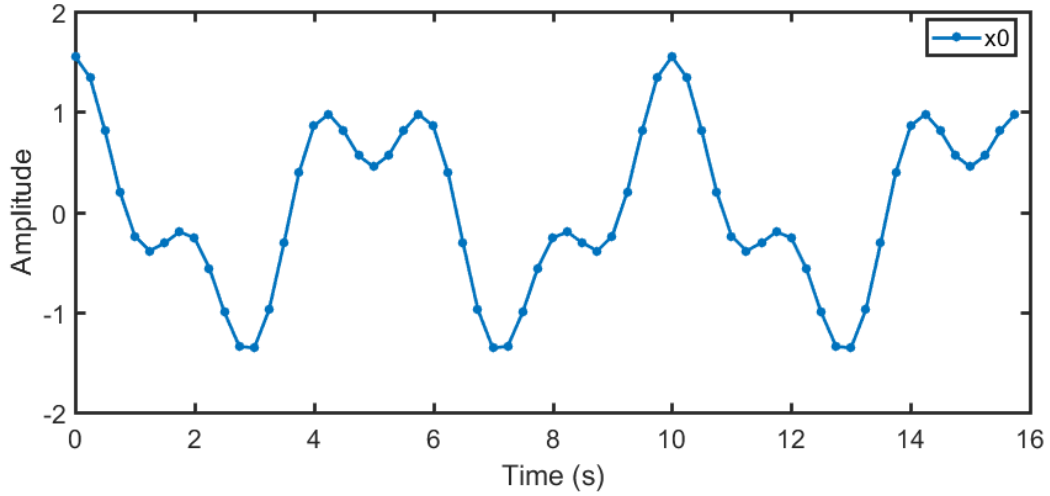
Textbook FFT Algorithm

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
y0 = fft(x0); % Frequency-domain transformation
% fft() is a MATLAB built-in
% function

fidemo.fi_fft_demo_ini_plot(t,x0,f,y0); % Plotting the results from fft
```

% and time-domain signal



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 provides you with an example of how you can 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.

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  $q = 1 : t$ 
   $L = 2^q$ ;  $r = n/L$ ;  $L_* = L/2$ ;
  for  $k = 0 : r - 1$ 
    for  $j = 0 : L_* - 1$ 
       $\tau = w^{(L_* - 1 + j)} \cdot x(kL + j + L_*)$ 
       $x(kL + j + L_*) = x(kL + j) - \tau$ 
       $x(kL + j) = x(kL + j) + \tau$ 
    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)
%FI_RADIX2TWIDDLES Twiddle factors for radix-2 FFT example.
% W = FI_RADIX2TWIDDLES(N) computes the length N-1 vector W of
% twiddle factors to be used in the FI_M_RADIX2FFT example code.
%
% See also FI_RADIX2FFT_DEMO.

% Reference:
%
% Twiddle factors for Algorithm 1.6.2, p. 45, Charles Van Loan,
% Computational Frameworks for the Fast Fourier Transform, SIAM,
% Philadelphia, 1992.
%
% Copyright 2003-2011 The MathWorks, Inc.
%

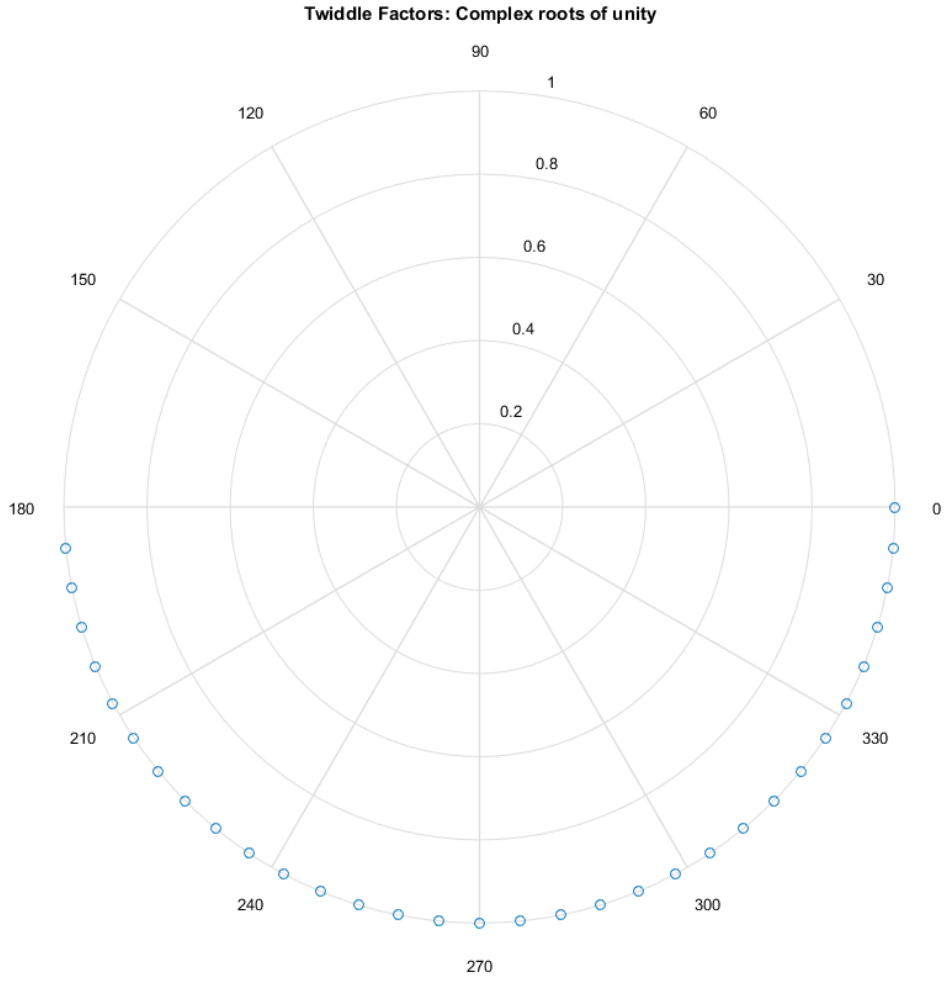
t = log2(n);
if floor(t) ~= t
    error('N must be an exact power of two.');
```

```

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

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. 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)
%FI_M_RADIX2FFT_ALGORITHM1_6_2 Radix-2 FFT example.
% Y = FI_M_RADIX2FFT_ALGORITHM1_6_2(X, W) computes the radix-2 FFT of
% input vector X with twiddle-factors W. Input X is assumed to be
% complex.
%
% The length of vector X must be an exact power of two.
% Twiddle-factors W are computed via
% W = fidemo.fi_radix2twiddles(N)
% where N = length(X).
%
% This version of the algorithm has no scaling before the stages.
%
% See also FI_RADIX2FFT_DEMO, FI_M_RADIX2FFT_WITHSCALING.
%
% Reference:
% Charles Van Loan, Computational Frameworks for the Fast Fourier
% Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2, p. 45.
%
% Copyright 2004-2015 The MathWorks, Inc.

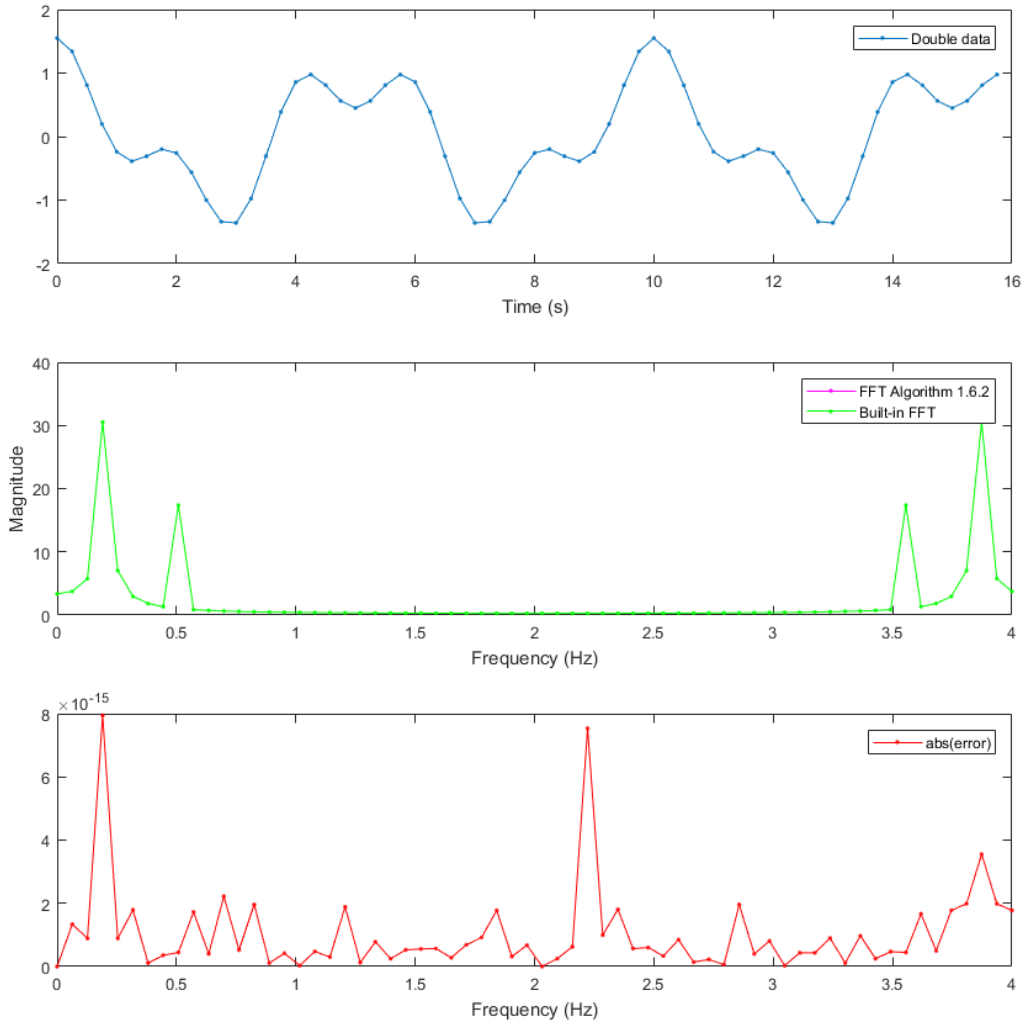
n = length(x); t = log2(n);
x = fidemo.fi_bitreverse(x,n);
for q=1:t
    L = 2^q; r = n/L; L2 = L/2;
    for k=0:(r-1)
        for j=0:(L2-1)
            temp = w(L2-1+j+1) * x(k*L+j+L2+1);
            x(k*L+j+L2+1) = x(k*L+j+1) - temp;
            x(k*L+j+1) = x(k*L+j+1) + temp;
        end
    end
end
end
```

Visualization

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.

As seen in the plot below, the error is within tolerance of the MATLAB built-in FFT function, verifying that you have correctly implemented the algorithm.

```
y = fi_m_radix2fft_algorithm1_6_2(x0, w0);  
fidemo.fi_fft_demo_plot(real(x0),y,y0,Fs,'Double data', ...  
    {'FFT Algorithm 1.6.2','Built-in FFT'});
```



Convert Functions to use Types Tables

To separate data types from the algorithm:

- 1 Create a table of data type definitions.
- 2 Modify the algorithm code to use data types from that table.

This example shows the iterative steps by creating different files. In practice, you can make the iterative changes to the same file.

Original types table

Create a types table using a structure with prototypes for the variables set to their original types. Use the baseline types to validate that you made the initial conversion correctly, and to programmatically toggle your function between floating point and fixed point types. The index variables are automatically converted to integers by MATLAB Coder™, so you don't need to specify their types in the table.

Specify the prototype values as empty ([]) since the data types are used, but not the values.

```
function T = fi_m_radix2fft_original_types()
%FI_M_RADIX2FFT_ORIGINAL_TYPES Types Table Example
%
% Copyright 2015 The MathWorks, Inc.

    T.x = double([]);
    T.w = double([]);
    T.n = double([]);

end
```

Type-aware algorithm function

Add types table T as an input to the function and use it to cast variables to a particular type, while keeping the body of the algorithm unchanged.

```
function x = fi_m_radix2fft_algorithm1_6_2_typed(x, w, T)
%FI_M_RADIX2FFT_ORIGINAL_TYPED Radix-2 FFT example.
% Y = FI_M_RADIX2FFT_ALGORITHM1_6_2_TYPED(X, W, T) computes the radix-2
% FFT of input vector X with twiddle-factors W. Input X is assumed to be
% complex.
%
% The length of vector X must be an exact power of two.
% Twiddle-factors W are computed via
```

```

%      W = fidemo.fi_radix2twiddles(N)
%      where N = length(X).
%
%      T is a types table to cast variables to a particular type, while keeping
%      the body of the algorithm unchanged.
%
%      This version of the algorithm has no scaling before the stages.
%
%      See also FI_RADIX2FFT_DEMO, FI_M_RADIX2FFT_WITHSCALING.
%
%      Reference:
%      Charles Van Loan, Computational Frameworks for the Fast Fourier
%      Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2, p. 45.
%
%      Copyright 2015 The MathWorks, Inc.
%
%#codegen

n = length(x);
t = log2(n);
x = fidemo.fi_bitreverse_typed(x,n,T);
LL = cast(2.^(1:t), 'like', T.n);
rr = cast(n./LL, 'like', T.n);
LL2 = cast(LL./2, 'like', T.n);
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) = x(k*L+j+1) - temp;
            x(k*L+j+1) = x(k*L+j+1) + temp;
        end
    end
end
end
end
end

```

Type-aware bitreversal function

Add types table T as an input to the function and use it to cast variables to a particular type, while keeping the body of the algorithm unchanged.


```

function x = fi_bitreverse_typed(x,n0,T)
%FI_BITREVERSE_TYPED Bit-reverse the input.
% X = FI_BITREVERSE_TYPED(x,n,T) bit-reverse the input sequence X, where
% N=length(X).
%
% T is a types table to cast variables to a particular type, while keeping
% the body of the algorithm unchanged.
%
% See also FI_RADIX2FFT_DEMO.

% Copyright 2004-2015 The MathWorks, Inc.
%
%#codegen
n = cast(n0,'like',T.n);
nv2 = bitsra(n,1);
j = cast(1,'like',T.n);
for i=1:(n-1)
    if i<j
        temp = x(j);
        x(j) = x(i);
        x(i) = temp;
    end
    k = nv2;
    while k<j
        j(:) = j-k;
        k = bitsra(k,1);
    end
    j(:) = j+k;
end

```

Validate modified function

Every time you modify your function, validate that the results still match your baseline. Since you used the original types in the types table, the outputs should be identical. This validates that you made the conversion to separate the types from the algorithm correctly.

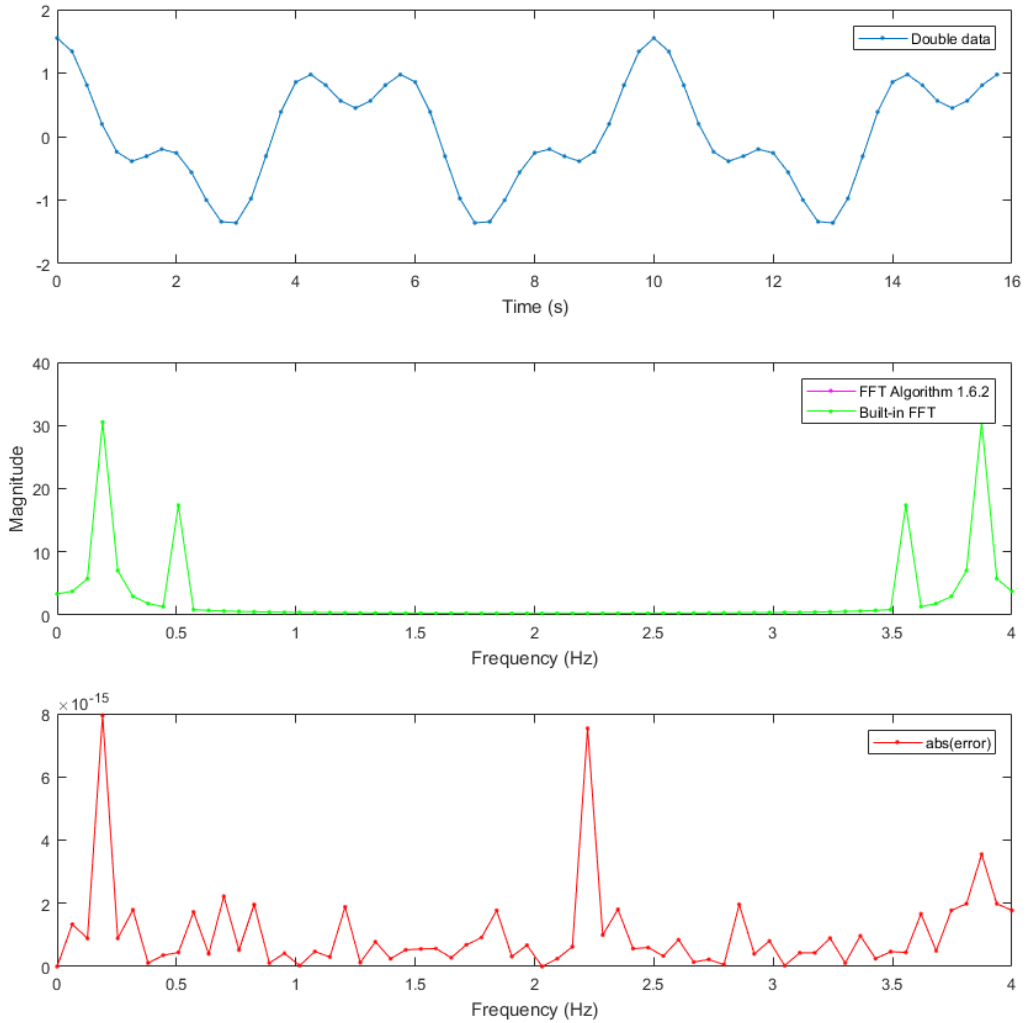
```

T1 = fidemo.fi_m_radix2fft_original_types(); % Getting original data types declared in
x = cast(x0,'like',T1.x);
w = cast(w0,'like',T1.w);

y = fi_m_radix2fft_algorithm1_6_2_typed(x, w, T1);

```

```
fidemo.fi_fft_demo_plot(real(x),y,y0,Fs,'Double data', ...
    {'FFT Algorithm 1.6.2','Built-in FFT'});
```



Create a fixed-point types table

Create a fixed-point types table using a structure with prototypes for the variables. Specify the prototype values as empty ([]) since the data types are used, but not the values.

```
function T = fi_m_radix2fft_fixed_types()
%FI_M_RADIX2FFT_FIXED_TYPES Example function
%
% Copyright 2015 The MathWorks, Inc.

    T.x = fi([],1,16,14); % Picked the following types to ensure that the
    T.w = fi([],1,16,14); % inputs have maximum precision and will not
                        % overflow
    T.n = int32([]);     % Picked int32 as n is an index

end
```

Identify Fixed-Point Issues

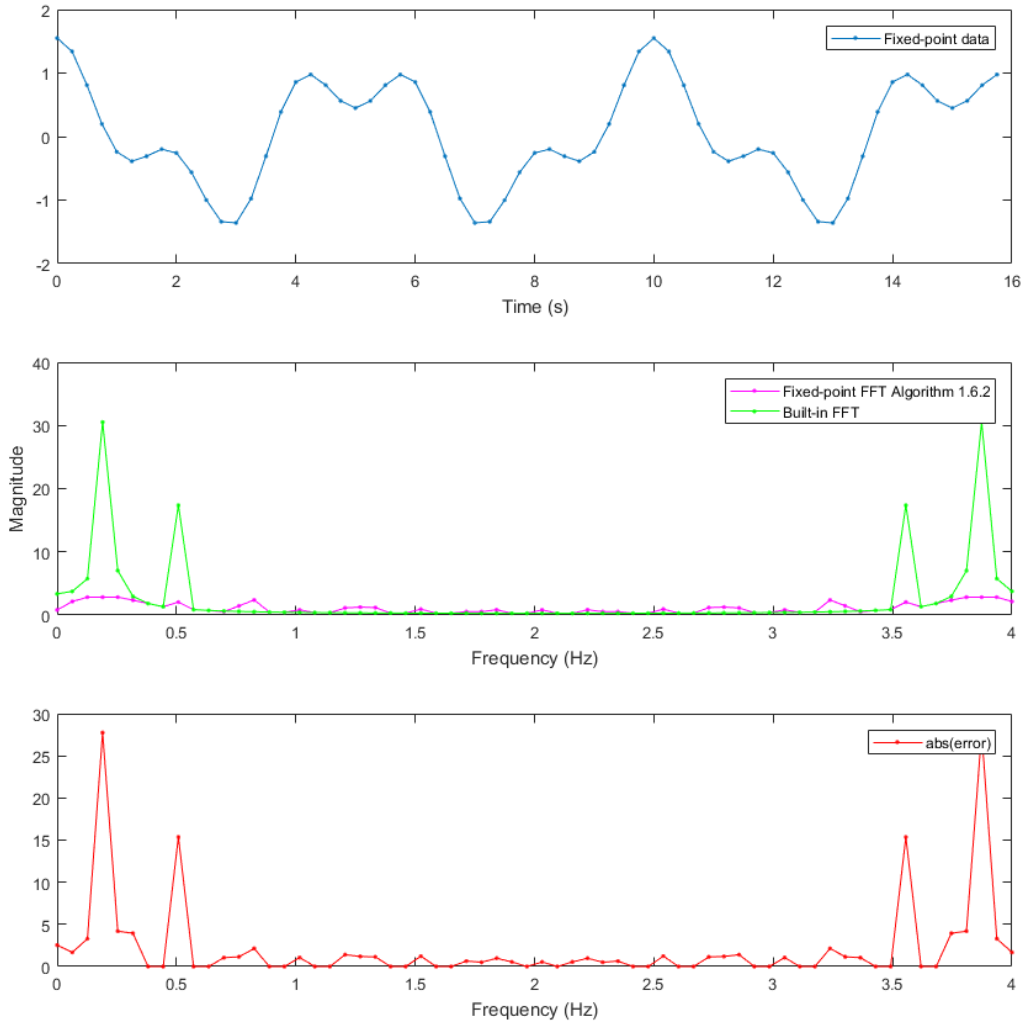
Now, try converting the input data to fixed-point and see if the algorithm still looks good. In this first pass, you use all the defaults for signed fixed-point data by using the `fi` constructor.

```
T2 = fidemo.fi_m_radix2fft_fixed_types(); % Getting fixed point data types declared in

x = cast(x0,'like',T2.x);
w = cast(w0,'like',T2.w);
```

Re-run the same algorithm with the fixed-point inputs

```
y = fi_m_radix2fft_algorithm1_6_2_typed(x,w,T2);
fidemo.fi_fft_demo_plot(real(x),y,y0,Fs,'Fixed-point data', ...
    {'Fixed-point FFT Algorithm 1.6.2','Built-in FFT'});
```



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, 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.

```
function T = fi_m_radix2fft_scaled_fixed_types()
%FI_M_RADIX2FFT_SCALED_FIXED_TYPES Example function
%
% Copyright 2015 The MathWorks, Inc.

    DT = 'ScaledDouble';           % Data type to be used for fi
                                   % constructor
    T.x = fi([],1,16,14,'DataType',DT); % Picked the following types to
    T.w = fi([],1,16,14,'DataType',DT); % ensure that the inputs have
                                   % maximum precision and will not
                                   % overflow
    T.n = int32([]);               % Picked int32 as n is an index

end

T3 = fidemo.fi_m_radix2fft_scaled_fixed_types(); % Getting fixed point data types decla

x_scaled_double = cast(x0,'like',T3.x);
w_scaled_double = cast(w0,'like',T3.w);
```

```
buildInstrumentedMex fi_m_radix2fft_algorithm1_6_2_typed ...
    -o fft_instrumented -args {x_scaled_double w_scaled_double T3}
```

Run the instrumented MEX function to record min/max values.

```
y_scaled_double = fft_instrumented(x_scaled_double,w_scaled_double,T3);
```

Show the instrumentation results.

```
showInstrumentationResults fft_instrumented
```

You can see from the instrumentation results that there were overflows when assigning into the variable x.

The screenshot shows the MATLAB code editor with the function `fi_m_radix2fft_algorithm1_6_2_typed` and its instrumentation results table.

Code Snippet:

```

12 % Add types table T as an input to the function and use it to cast
13 % variables to a particular type, while keeping the body of the algorithm
14 % unchanged.
15 %
16 % This version of the algorithm has no scaling before the stages.
17 %
18 % See also FI_RADIX2FFT_DEMO, FI_M_RADIX2FFT_WITHSCALING.
19 %
20 % Reference:
21 % Charles Van Loan, Computational Frameworks for the Fast Fourier
22 % Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2, p. 45.
23 %
24 % Copyright 2015 The MathWorks, Inc.
25 %
26 %codegen
27
28 n = length(x);
29 L = log2(n);
30 % fidemo_fi_radix2fft_algorithm1_6_2_typed(n,T);
31 LL = cast(2.^[1:L], 'like', T,n);
32 rr = cast(n./LL, 'like', T,n);
33 LL2 = cast(LL./2, 'like', T,n);
34 for q=1:L
35     L = LL(q);
36     x = rr(q);
37     L2 = LL2(q);
38     for k=0:(L-1)
39         temp = w(L2-L+1) + x(k*L+L2+1);
40         x(k*L+L2+1) = x(k*L+1) - temp;
41         x(k*L+1) = x(k*L+1) + temp;
42     end
43 end
44 end
45 end
46 end

```

Instrumentation Results Table:

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	x	IO	1 x 64	embedded:s	Yes	ScaledDouble	Signed	16	14	124%	No	-17.04520175369448	25.320136436369612
2	w	Input	63 x 1	embedded:s	Yes	ScaledDouble	Signed	16	14	51%	No	-1	1
3	T	Input	1 x 1	struct	-	-	-	-	-	-	-	-	-
4	n	Local	1 x 1	double	No	-	-	-	-	-	Yes	64	64
5	L	Local	1 x 1	double	No	-	-	-	-	-	Yes	6	6
6	LL	Local	1 x 6	int32	No	-	-	-	-	-	Yes	2	64
7	rr	Local	1 x 6	int32	No	-	-	-	-	-	Yes	1	32
8	LL2	Local	1 x 6	int32	No	-	-	-	-	-	Yes	1	32
9	q	Local	1 x 1	double	No	-	-	-	-	-	Yes	1	6
10	L	Local	1 x 1	int32	No	-	-	-	-	-	Yes	2	64
11	r	Local	1 x 1	int32	No	-	-	-	-	-	Yes	1	32
12	L2	Local	1 x 1	int32	No	-	-	-	-	-	Yes	1	32
13	k	Local	1 x 1	int32	No	-	-	-	-	-	Yes	0	31
14	j	Local	1 x 1	int32	No	-	-	-	-	-	Yes	0	31
15	temp	Local	1 x 1	embedded:s	Yes	ScaledDouble	Signed	33	28	79%	No	-12.52563147499761	12.525631474997604

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_typed(x, w, T)
%FI_M_RADIX2FFT_WITHSCALING Radix-2 FFT example with scaling at each stage.
% Y = FI_M_RADIX2FFT_WITHSCALING_TYPED(X, W, T) computes the radix-2 FFT of
% input vector X with twiddle-factors W with scaling by 1/2 at each stage.
% Input X is assumed to be complex.
%
% The length of vector X must be an exact power of two.
% Twiddle-factors W are computed via
% W = fidemo.fi_radix2twiddles(N)
% where N = length(X).
%
% T is a types table to cast variables to a particular type, while keeping
% the body of the algorithm unchanged.
%
% This version of the algorithm has no scaling before the stages.
%
% See also FI_RADIX2FFT_DEMO.
%
% Reference:
% Charles Van Loan, Computational Frameworks for the Fast Fourier
% Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2, p. 45.
%
% Copyright 2004-2015 The MathWorks, Inc.
%
%#codegen

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 = cast(2.^(1:t), 'like', T.n);
rr = cast(n./LL, 'like', T.n);
```

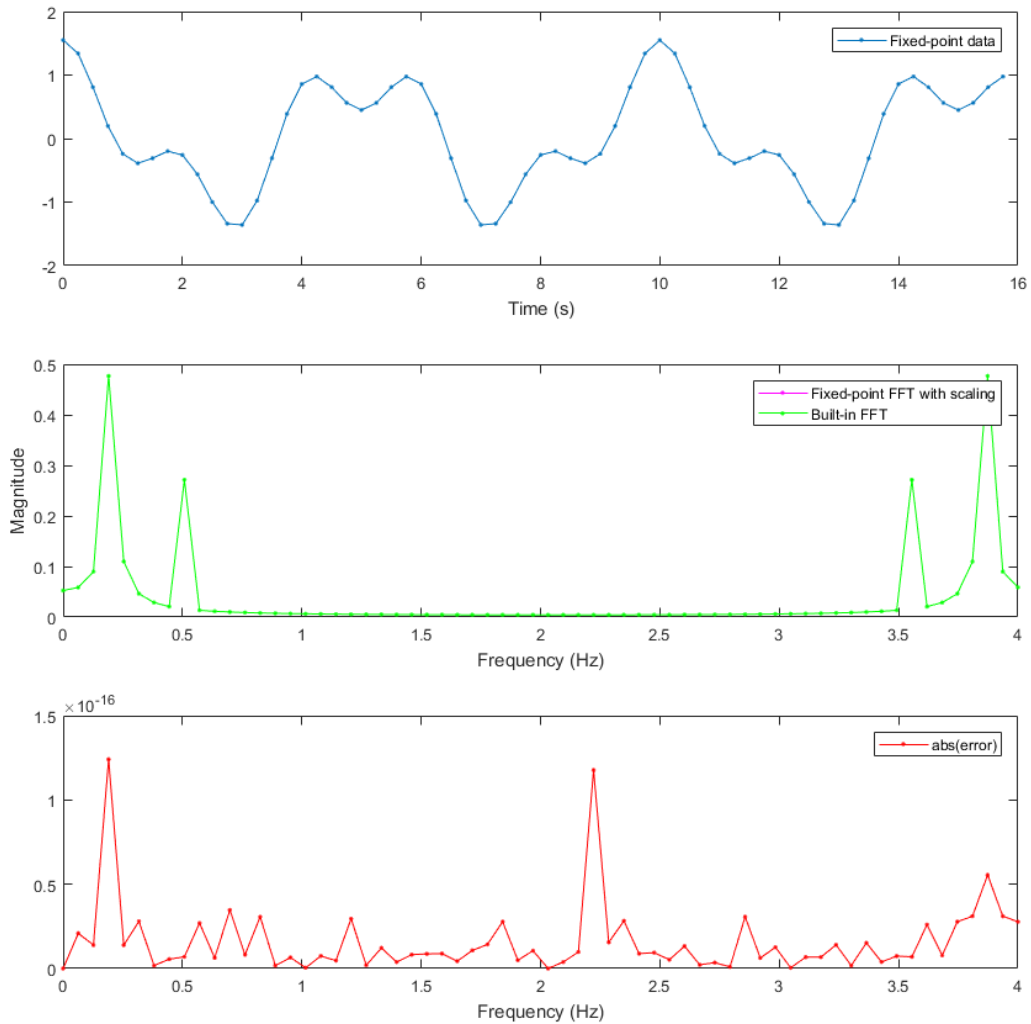
```
LL2 = cast(LL./2, 'like', T.n);
for q=1:t
    L = LL(q); r = rr(q); L2 = LL2(q);
    for k=0:(r-1)
        for j=0:(L2-1)
            temp = w(L2-1+j+1) * x(k*L+j+L2+1);
            x(k*L+j+L2+1) = bitsra(x(k*L+j+1) - temp, 1);
            x(k*L+j+1) = bitsra(x(k*L+j+1) + temp, 1);
        end
    end
end
end
```

Run the scaled algorithm with fixed-point data.

```
x = cast(x0, 'like', T3.x);
w = cast(w0, 'like', T3.w);

y = fi_m_radix2fft_withscaling_typed(x,w,T3);

fidemo.fi_fft_demo_plot(real(x), y, y0/n, Fs, 'Fixed-point data', ...
    {'Fixed-point FFT with scaling', 'Built-in FFT'});
```

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.

Cleve Moler, *Numerical Computing with MATLAB*, SIAM, 2004, Chapter 8 Fourier Analysis.

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.

```
cleanup(tempdirObj);
```

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.

```
originalFormat = get(0, 'format');
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', 'fisisostatespacef
```

```
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
```

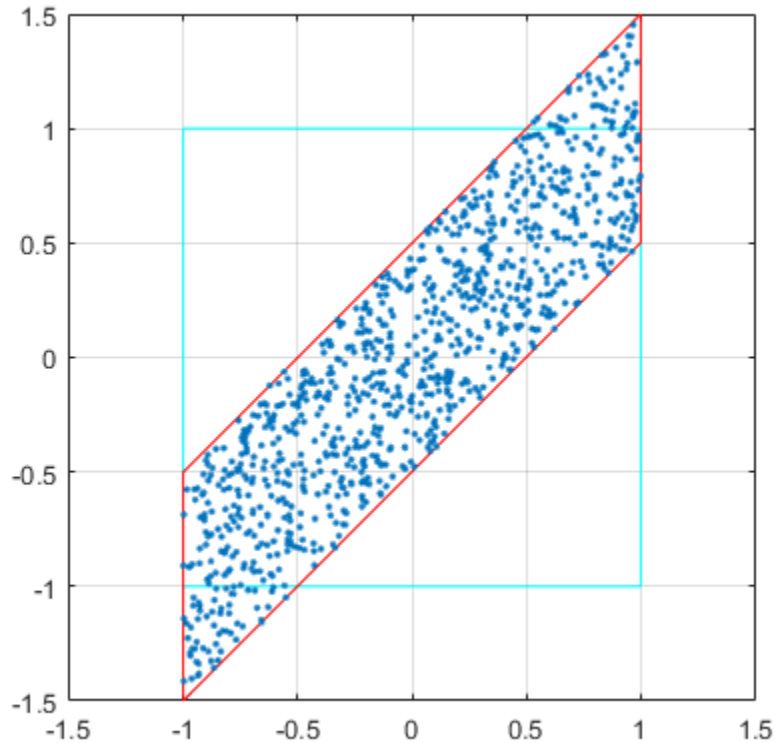
```
% A, B, C, D. Output Y and final conditions Zf are returned.  
  
% Copyright 2004-2011 The MathWorks, Inc.  
  
y = x;  
z(:,2:length(x)+1) = 0;  
for k=1:length(x)  
    y(k) = C*z(:,k) + D*x(k);  
    z(:,k+1) = A*z(:,k) + B*x(k);  
end
```

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:,:),'.')
```



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.

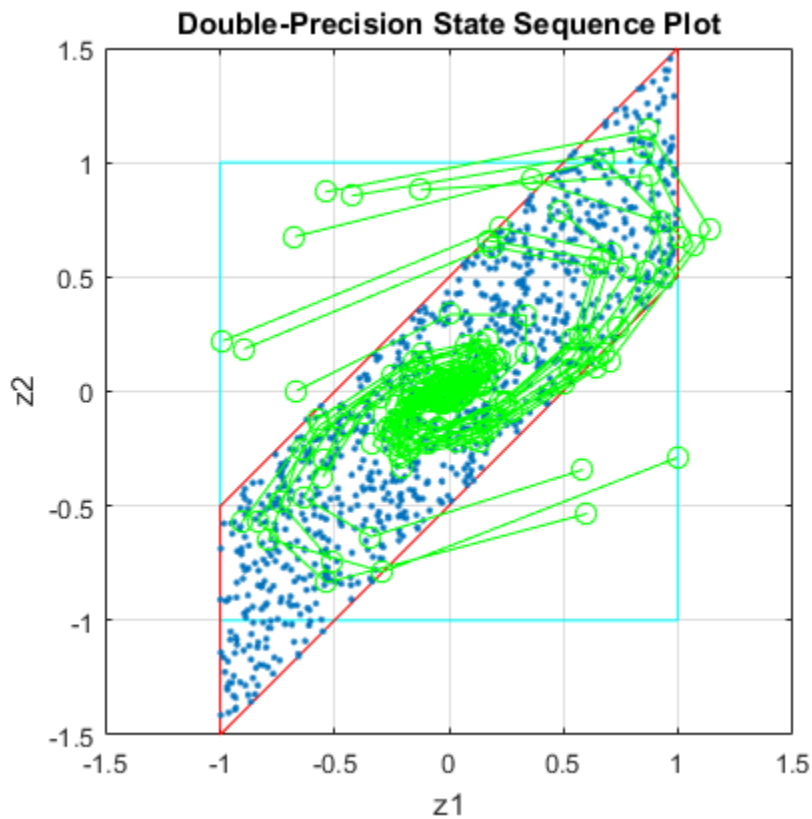
Note that some of the states wander outside the unit square, and that they eventually wind down to the zero state at the origin, $z=[0;0]$.

```
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

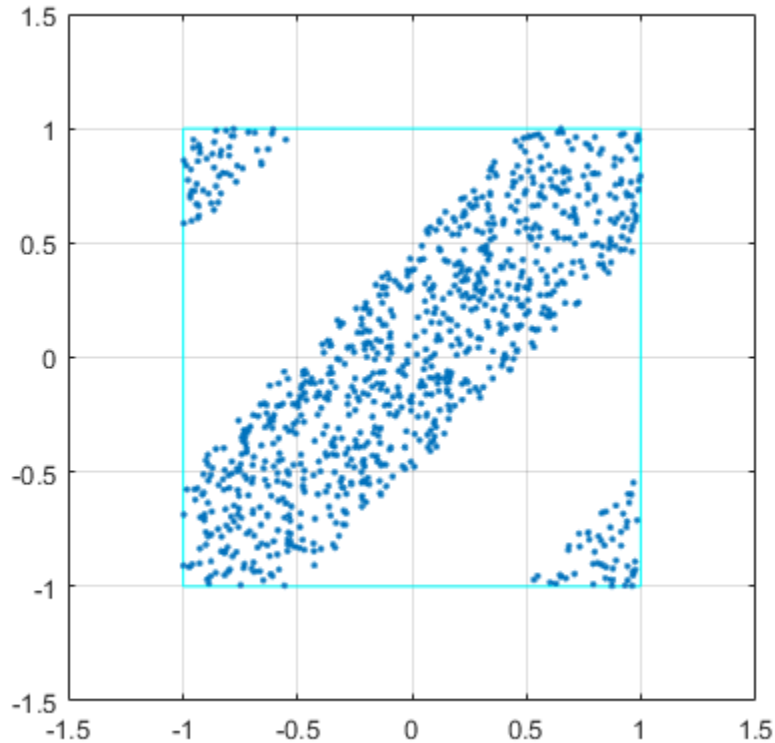
Again, we choose random states within the unit square and observe where they are projected after one step of being multiplied by the state-transition matrix A. The difference is that this time matrix A is fixed-point.

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.

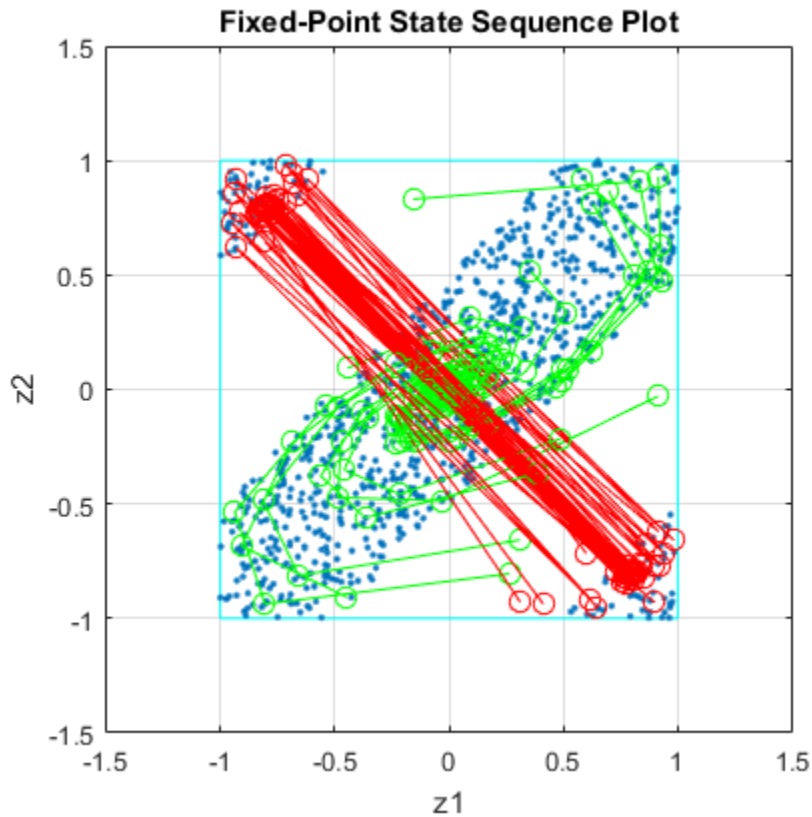
The only difference between this and the previous code is that we are driving it with fixed-point data types.

```
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 A_2 .

```
T = [-2 0; -1 1];  
Tinv = [-.5 0; -.5 1];  
A2 = Tinv*A*T; B2 = Tinv*B; C2 = C*T; D2 = D;
```

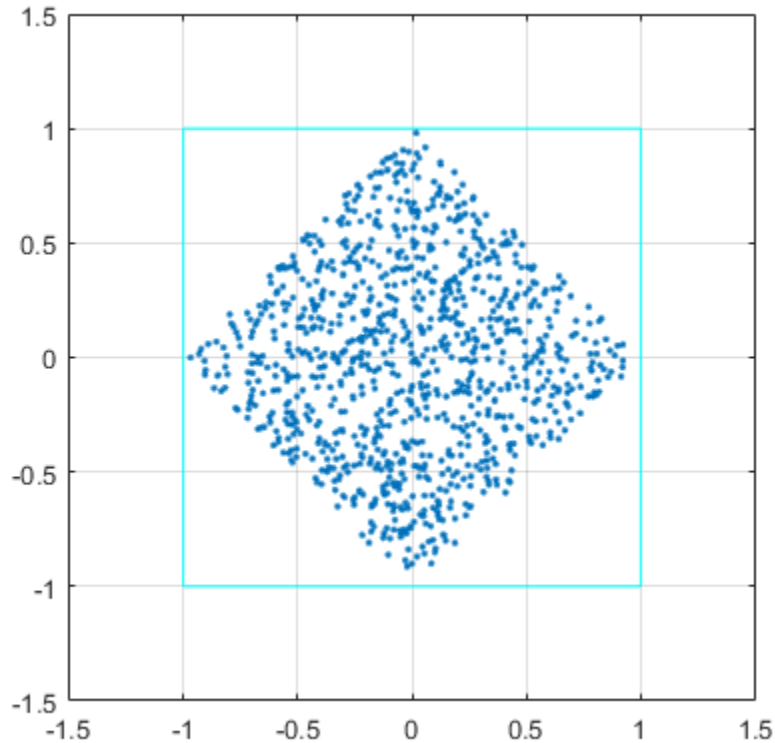
Similarity transformations preserve eigenvalues, as a result of which the system transfer function of the transformed system remains same as before. However, the transformed state transformation matrix A_2 is normal.

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 A_2 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,:), 'r')
```



Plot the State Sequence

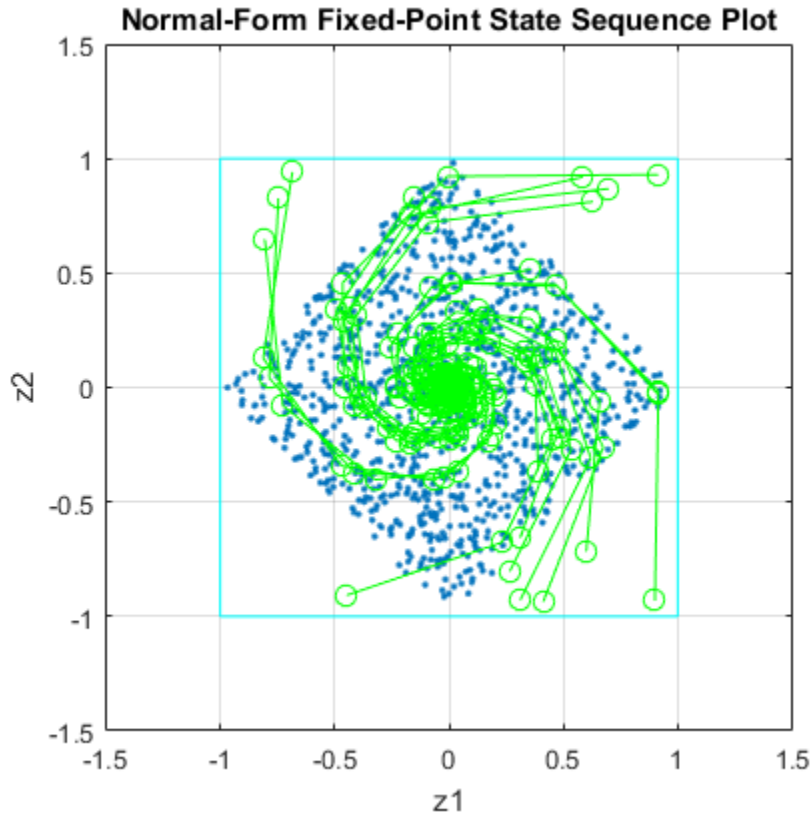
Plotting the state sequences again for the same initial states as before we see that the outputs now spiral towards the origin.

```
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.

```
set(0, 'format', originalFormat);
```

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.

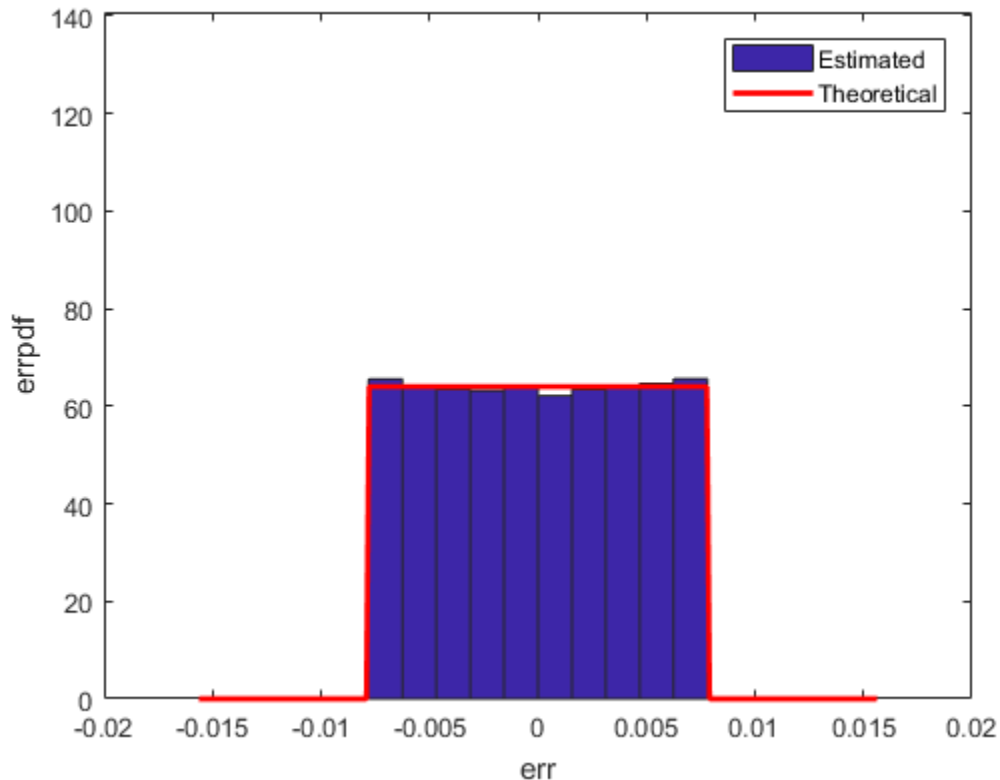
```
q = quantizer([8 7]);
r = realmax(q);
u = r*(2*rand(50000,1) - 1);           % Uniformly distributed (-1,1)
xi=linspace(-2*eps(q),2*eps(q),256);
```

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);
```



```

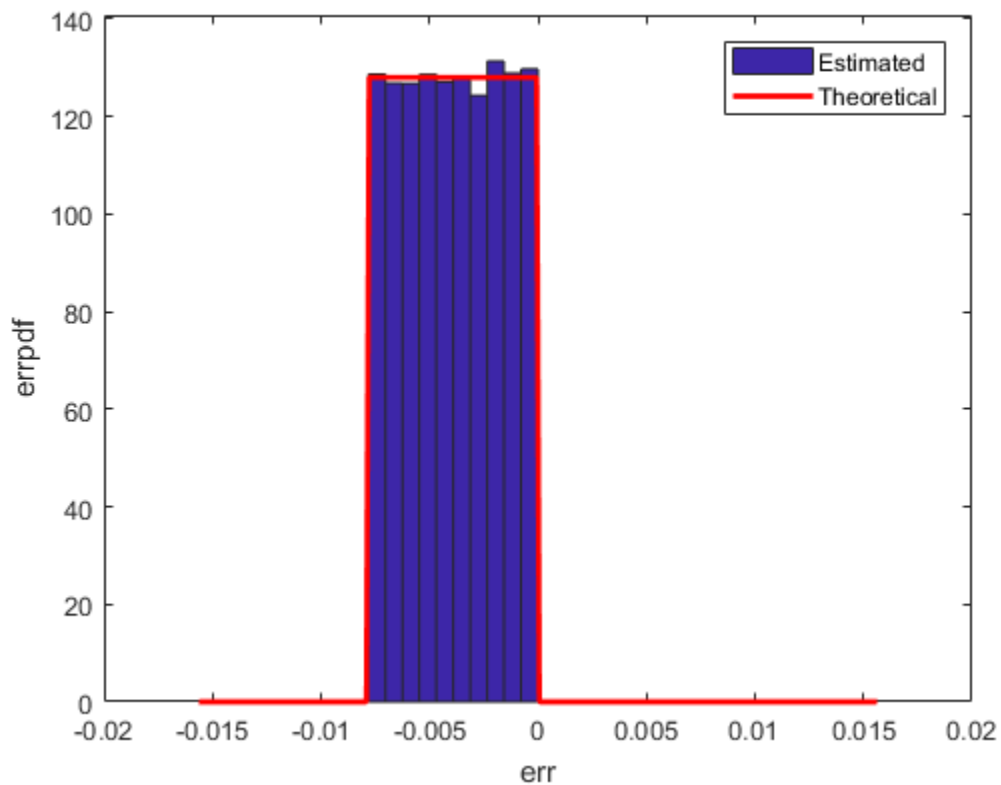
% Theoretical variance = eps(q)^2 / 12
% Theoretical mean     = -eps(q)/2
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)

```

```

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.

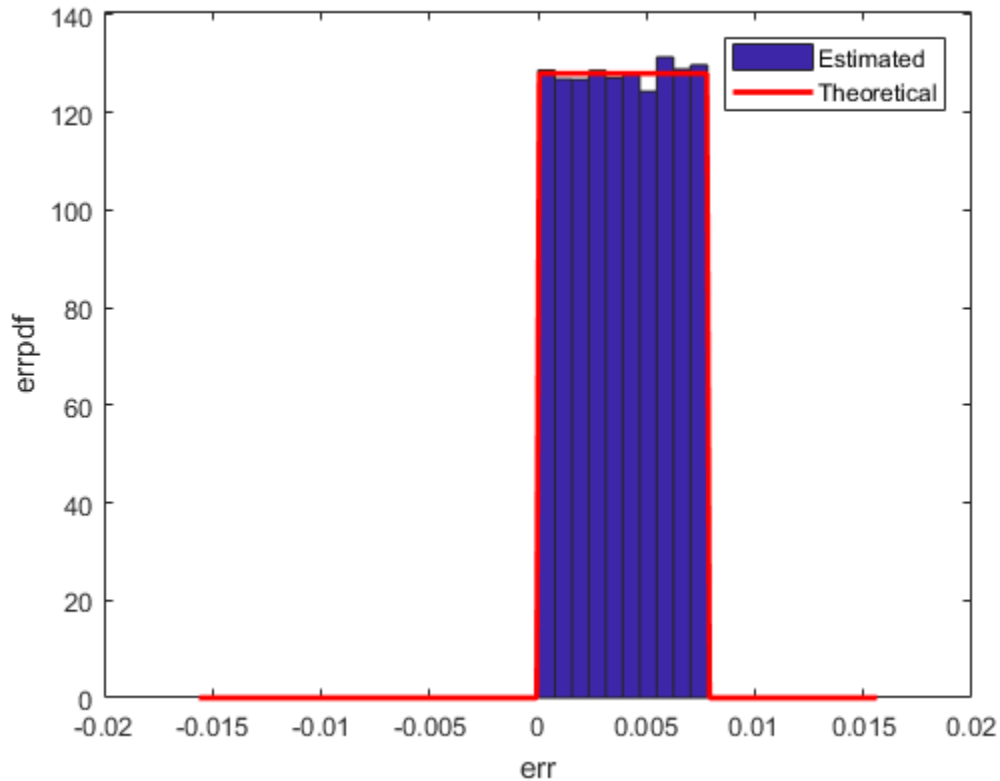
```

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
```

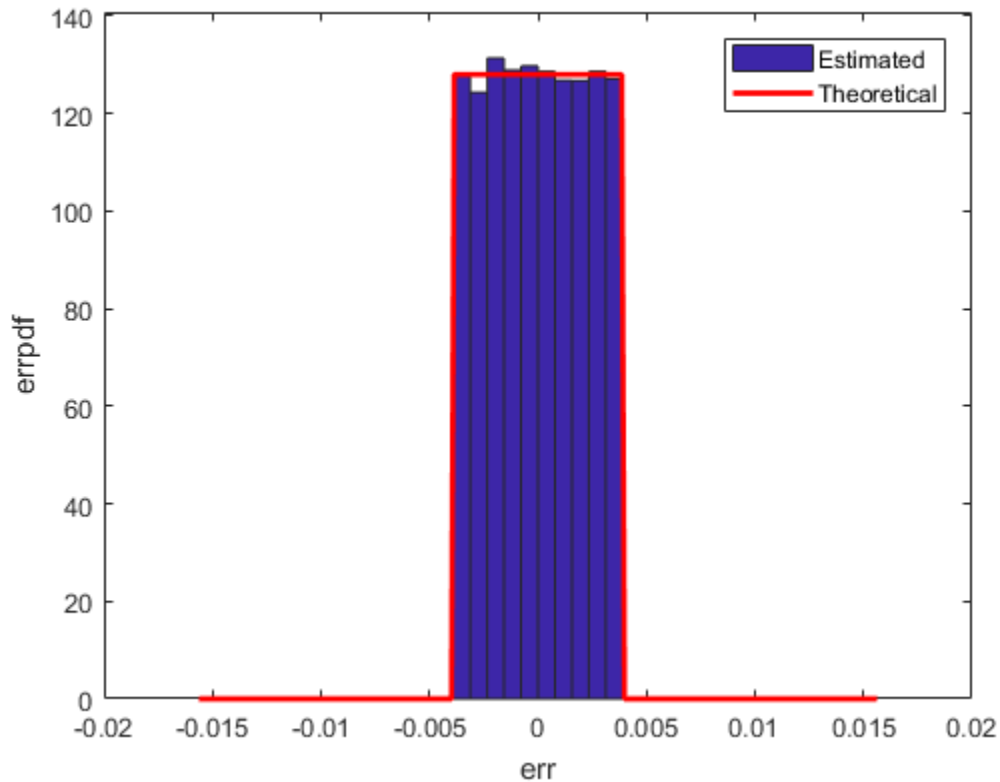


Round: Round to Nearest. In a Tie, Round to Largest Magnitude.

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 = eps(q)^2 / 12  
% Theoretical mean = 0  
fidemo.qerrordemoplots(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
```

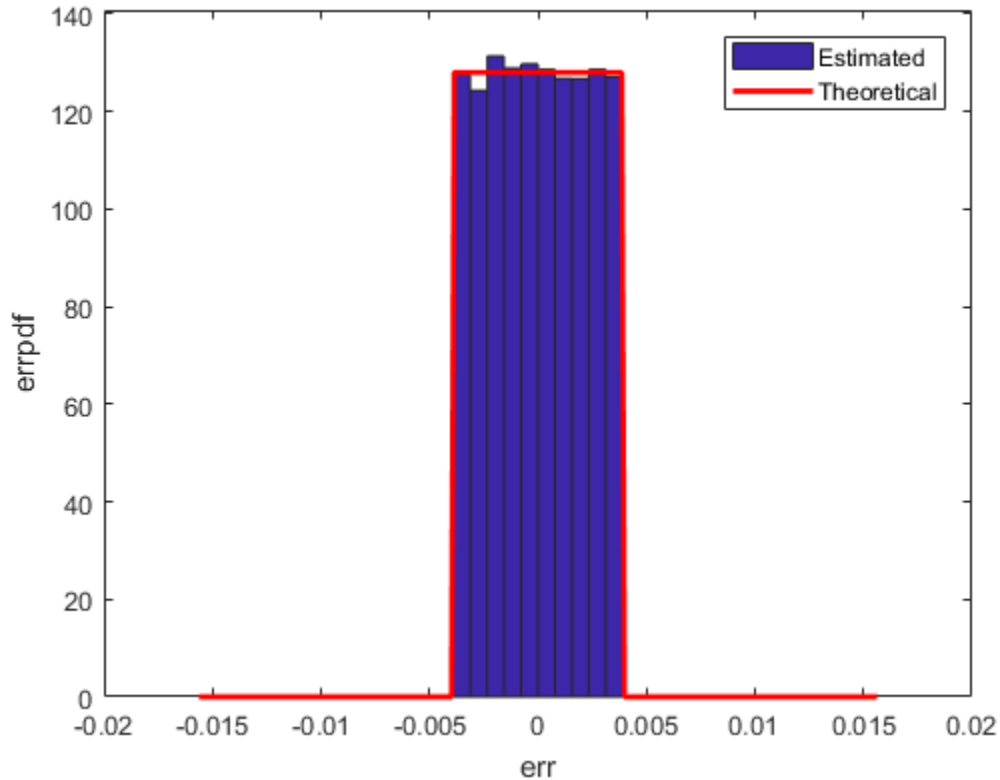


Convergent: Round to Nearest. In a Tie, Round to Even.

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

$$f(\text{err}) = 1/\text{eps}(q), \text{ for } -\text{eps}(q)/2 \leq \text{err} \leq \text{eps}(q)/2, \text{ and } 0 \text{ otherwise}$$

while the error p.d.f. of round is

$$f(\text{err}) = 1/\text{eps}(q), \text{ for } -\text{eps}(q)/2 < \text{err} \leq \text{eps}(q)/2, \text{ and } 0 \text{ otherwise}$$

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','qerrordemoplot.m')
```

```
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  
% used by the example function QERRORDEMO, where Q is the quantizer  
% whose attributes are being analyzed; F_T is the theoretical  
% quantization error probability density function for quantizer Q  
% computed by ERRPDF; XI is the domain of 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 1999-2014 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 = get(fipref); reset(fipref)
```

Function to Normalize Unsigned Data

This algorithm normalizes unsigned data with 8-bit bytes. 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_byte` 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 byte is read from the number-of-leading-zeros lookup table, `NLZLUT`.

```
function [x,n] = fi_normalize_unsigned_8_bit_byte(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 data type');
    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 the
    % 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_byte` 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 ...
                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 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 ...])
```

```

0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0] );

```

end

Example

For example, let

```
u = fi(0.3, 1, 16, 8);
```

In binary, $u = 00000000.01001101_2 = 0.30078125$ (the fixed-point value is not exactly 0.3 because of roundoff to 8 bits). The goal is to normalize such that

$$u = 1.0011010000000000_2 * 2^{(-2)} = x * 2^n.$$

Start with u represented as an unsigned integer.

```

High byte  Low byte
00000000  01001101  Start: u as unsigned integer.

```

The high byte is $0 = 00000000_2$. Add 1 to make an index out of it: $index = 0 + 1 = 1$. The number-of-leading-zeros lookup table at index 1 indicates that there are 8 leading zeros: $NLZLUT(1) = 8$. 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: $index = 77 + 1 = 78$. The number-of-leading-zeros lookup table at index 78 indicates that there is 1 leading zero: $NLZLUT(78) = 1$. 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 = 1.0011010000000000\_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] = fi_normalize_unsigned_8_bit_byte(u)
```

```
x =
```

```
1.203125
```

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

```
n =
```

```
int8
```

```
-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 \log_2 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 = get(fipref); reset(fipref)
```

Log2 Implementation

The \log_2 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_byte()` 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 $i = x_B - 2^{(B-1)} + 1$ 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 `LOG2LUT(i)` and the next value in the table `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 `reinterpretedcast()`.
- 6 Finally, compute the output using the lookup table and linear interpolation:

$$\begin{aligned} \log_2(u) &= \log_2(x * 2^n) \\ &= n + \log_2(x) \\ &= n + \text{LOG2LUT}(i) + r * (\text{LOG2LUT}(i+1) - \text{LOG2LUT}(i)) \end{aligned}$$

```
function y = fi_log2lookup_8_bit_byte(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 = coder.nullcopy(fi(zeros(size(u)), numerictype(LOG2LUT), fimath(LOG2LUT)));
B = 8; % Number of bits in a byte
w = u.WordLength;
for k = 1:numel(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_byte(u(k));
    % Extract the high byte of x
    high_byte = 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    0.033423...
                  0.044394119358453    0.055282435501190    0.066089190457773    0.076815...
                  0.087462841250339    0.098032082960527    0.108524456778169    0.118941...
                  0.129283016944966    0.139551352398794    0.149747119504682    0.159871...
                  0.169925001442312    0.179909090014934    0.189824558880017    0.199672...
                  0.209453365628950    0.219168520462162    0.228818690495881    0.238404...
                  0.247927513443586    0.257387842692652    0.266786540694901    0.276124...
                  0.285402218862248    0.294620748891627    0.303780748177103    0.312882...

```

```

0.321928094887362    0.330916878114617    0.339850002884625    0.348728...
0.357552004618084    0.366322214245816    0.375039431346925    0.383704...
0.392317422778760    0.400879436282184    0.409390936137702    0.417852...
0.426264754702098    0.434628227636725    0.442943495848728    0.451211...
0.459431618637297    0.467605550082997    0.475733430966398    0.483815...
0.491853096329675    0.499845887083205    0.507794640198696    0.515699...
0.523561956057013    0.531381460516312    0.539158811108031    0.546894...
0.554588851677637    0.562242424221073    0.569855608330948    0.577428...
0.584962500721156    0.592457037268080    0.599912842187128    0.607330...
0.614709844115208    0.622051819456376    0.629356620079610    0.636624...
0.643856189774725    0.651051691178929    0.658211482751795    0.665335...
0.672425341971496    0.679480099505446    0.686500527183218    0.693486...
0.700439718141092    0.707359132080883    0.714245517666123    0.721099...
0.727920454563199    0.734709620225838    0.741466986401147    0.748192...
0.754887502163469    0.761551232444479    0.768184324776926    0.774787...
0.781359713524660    0.787902559391432    0.794415866350106    0.800899...
0.807354922057604    0.813781191217037    0.820178962415188    0.826548...
0.832890014164742    0.839203788096944    0.845490050944375    0.851749...
0.857980995127572    0.864186144654280    0.870364719583405    0.876516...
0.882643049361841    0.888743248898259    0.894817763307943    0.900866...
0.906890595608518    0.912889336229962    0.918863237274595    0.924812...
0.930737337562886    0.936637939002571    0.942514505339240    0.948367...
0.954196310386875    0.960001932068081    0.965784284662087    0.971543...
0.977279923499916    0.982993574694310    0.988684686772166    0.994353...
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

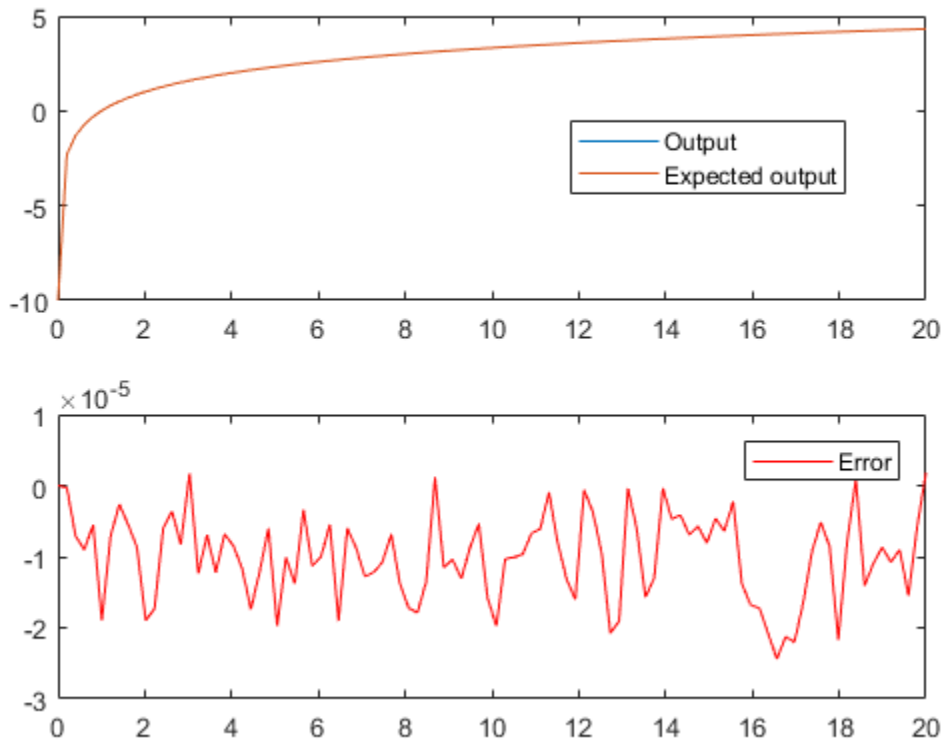
```

Example

```
u = fi(linspace(0.001,20,100));
```



```
y = fi_log2lookup_8_bit_byte(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 = get(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_byte()` 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 $i = x_B - 2^{(B-2)} + 1$ is used as an index to `SQRTLUT` so that $\sqrt{x_B}$ can be evaluated by looking up the index $\sqrt{x_B} = \text{SQRTLUT}(i)$.
- 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 `reinterpretpcast()`.
- 6 Finally, compute the output using the lookup table and linear interpolation:

$$\begin{aligned} \text{sqrt}(u) &= \text{sqrt}(x * 2^n) \\ &= \text{sqrt}(x) * 2^{(n/2)} \\ &= (\text{SQRTLUT}(i) + r * (\text{SQRTLUT}(i+1) - \text{SQRTLUT}(i))) * 2^{(n/2)} \end{aligned}$$

```
function y = fi_sqrtlookup_8_bit_byte(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:numel(u)
    assert(u(k)>=0, 'Input must be non-negative. ');
    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_byte(u(k));
        isodd = storedInteger(bitand(fi(1,1,8,0),fi(n)));
        x = bitsra(x,isodd);
        n = n + isodd;
        % Extract the high byte of x
        high_byte = 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    0.723489
                  0.728868986855663    0.734208757779421    0.739509972887452    0.744773
```

0.7500000000000000	0.755190373349661	0.760345316287277	0.7654655
0.770551750371122	0.775604602874429	0.780624749799800	0.7856128
0.790569415042095	0.795495128834866	0.800390529679106	0.8052561
0.810092587300983	0.814900300650331	0.819679815537750	0.8244316
0.829156197588850	0.833854004007896	0.838525491562421	0.8431710
0.847791247890659	0.852386356061616	0.856956825050130	0.8615033
0.866025403784439	0.870524267324007	0.875000000000000	0.8794529
0.883883476483184	0.888291900221993	0.892678553567856	0.8970433
0.901387818865997	0.905711046636840	0.910013736160065	0.9142961
0.918558653543692	0.922801441264588	0.927024810886958	0.9312290
0.935414346693485	0.939581023648307	0.943729304408844	0.9478599
0.951971638232989	0.956066158798647	0.960143218483576	0.9642033
0.968245836551854	0.972271824131503	0.976281209488332	0.9802741
0.984250984251476	0.988211768802619	0.992156741649222	0.9960860
1.000000000000000	1.003898650263063	1.007782218537319	1.0116508
1.015504800579495	1.019344151893756	1.023169096484056	1.0269797
1.030776406404415	1.034559084827928	1.038327982864759	1.0420833
1.045825033167594	1.049553476484167	1.053268721647045	1.0569709
1.060660171779821	1.064336647870400	1.068000468164691	1.0716517
1.075290658380328	1.078917281352004	1.082531754730548	1.0861341
1.089724735885168	1.093303480283494	1.096870548424015	1.1004266
1.103970108290981	1.107502821666834	1.111024302164449	1.1145346
1.118033988749895	1.121522402807898	1.125000000000000	1.1284668
1.131923142267177	1.135368882786559	1.138804197393037	1.1422299
1.145643923738960	1.149048519428140	1.152443057161611	1.1558270
1.159202311936963	1.162567202358642	1.165922381636102	1.1692679
1.172603939955857	1.175930482639174	1.179247641507075	1.1825551
1.185854122563142	1.189143599402528	1.192424001771182	1.1956954
1.198957880828180	1.202211503854459	1.205456345124119	1.2086924
1.211919964354082	1.215138880951474	1.218349293101120	1.2215512
1.224744871391589	1.227930169024281	1.231107225224513	1.2342761
1.237436867076458	1.240589577579950	1.243734296383275	1.2468710
1.250000000000000	1.253121103485214	1.256234452640111	1.2593401
1.262438117295260	1.265528545707287	1.268611445636527	1.2716866
1.274754878398196	1.277815518766305	1.280868845744950	1.2839144
1.286953767623375	1.289985465034393	1.293010054098575	1.2960279
1.299038105676658	1.302041665999979	1.305038313613819	1.3080280
1.311011060212689	1.313987252601790	1.316956719106592	1.3199199
1.322875655532295	1.325825214724777	1.328768226591831	1.3317047
1.334634781503914	1.337558409939543	1.340475661845451	1.3433869
1.346291201783626	1.349189571557681	1.352081728298996	1.3549677
1.357847561400027	1.360721316067327	1.363589014329464	1.3664500
1.369306393762915	1.372156150006259	1.375000000000000	1.3778379
1.380670127148408	1.383496476323666	1.386317063301177	1.3891319

```

        1.391941090707505    1.394744600276337    1.397542485937369    1.400334
        1.403121520040228    1.405902734900249    1.408678458698081    1.411448
        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);

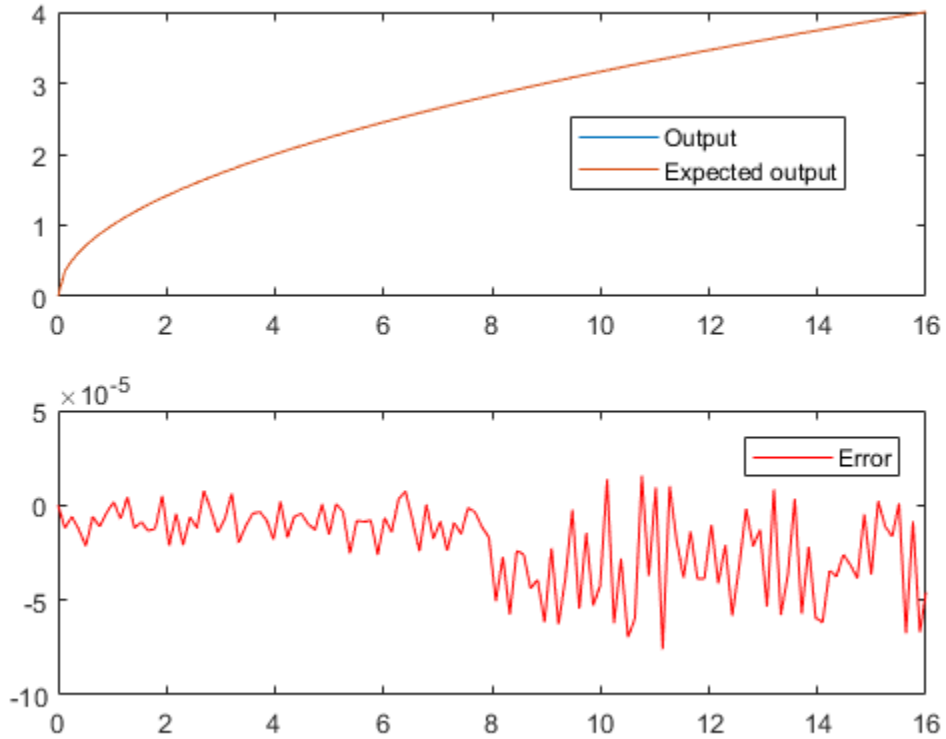
y = fi_sqrtlookup_8_bit_byte(u);

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 insulate your fixed-point operations from global and local `fimath` settings by using the `setfimath` and `removefimath` functions. 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','RoundingMethod','Floor')
>> 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} -launchrepe
```

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)(-23826
            * 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 4-5
- “fimath Properties Usage for Fixed-Point Arithmetic” on page 4-13
- “fimath for Rounding and Overflow Modes” on page 4-21
- “fimath for Sharing Arithmetic Rules” on page 4-23
- “fimath ProductMode and SumMode” on page 4-26
- “How Functions Use fimath” on page 4-32

fimath Object Construction

In this section...

“fimath Object Syntaxes” on page 4-2

“Building fimath Object Constructors in a GUI” on page 4-3

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 Designer 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...)
```

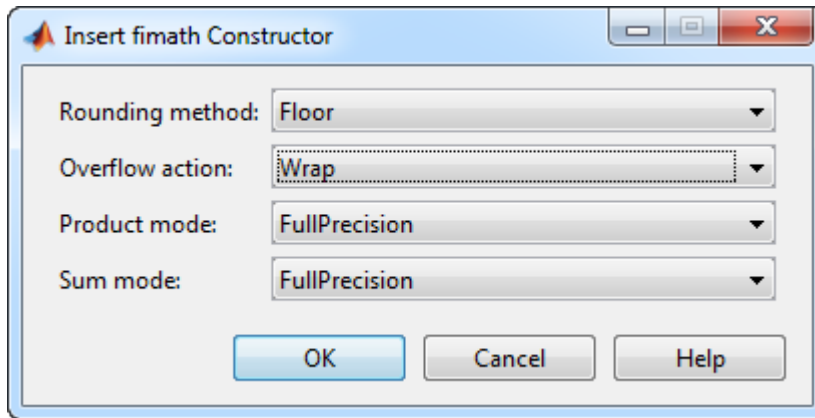
allows you to set properties for a `fimath` object at object creation with property name/property value pairs. Refer to “Setting `fimath` Properties at Object Creation” on page 4-11.

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 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 On the **Home** tab, in the **File** section, click **New > Script** to open the MATLAB Editor
- 2 On the **Editor** tab, in the **Edit** section, click  in the **Insert** button group. Click the **Insert `fimath`...** to open the **Insert `fimath` Constructor** dialog box.
- 3 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



- 4 To insert the `fimath` object constructor 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 in your file:

```
1     fimath('RoundingMethod', 'Floor', ...
2         'OverflowAction', 'Wrap', ...
3         'ProductMode', 'FullPrecision', ...
4         'SumMode', 'FullPrecision')
```

fimath Object Properties

In this section...

“Math, Rounding, and Overflow Properties” on page 4-5

“How Properties are Related” on page 4-9

“Setting fimath Object Properties” on page 4-11

Math, Rounding, and Overflow Properties

You can always write to the following properties of `fimath` objects:

Property	Description	Valid Values
<code>CastBeforeSum</code>	Whether both operands are cast to the sum data type before addition	<ul style="list-style-type: none"> 0 (default) — do not cast before sum 1 — cast before sum <p>Note: This property is hidden when the <code>SumMode</code> is set to <code>FullPrecision</code>.</p>
<code>MaxProductWordLength</code>	Maximum allowable word length for the product data type	<ul style="list-style-type: none"> 65535 (default) Any positive integer
<code>MaxSumWordLength</code>	Maximum allowable word length for the sum data type	<ul style="list-style-type: none"> 65535 (default) Any positive integer
<code>OverflowAction</code>	Action to take on overflow	<ul style="list-style-type: none"> <code>Saturate</code> (default) — Saturate to maximum or minimum value of the fixed-point range on overflow. <code>Wrap</code> — Wrap on overflow. This mode is also known as two's complement overflow.
<code>ProductBias</code>	Bias of the product data type	<ul style="list-style-type: none"> 0 (default) Any floating-point number
<code>ProductFixedExponent</code>	Fixed exponent of the product data type	<ul style="list-style-type: none"> -30 (default) Any positive or negative integer

Property	Description	Valid Values
		<p>Note: The <code>ProductFractionLength</code> is the negative of the <code>ProductFixedExponent</code>. Changing one property changes the other.</p>
<code>ProductFractionLength</code>	Fraction length, in bits, of the product data type	<ul style="list-style-type: none"> • 30 (default) • Any positive or negative integer <p>Note: The <code>ProductFractionLength</code> is the negative of the <code>ProductFixedExponent</code>. Changing one property changes the other.</p>
<code>ProductMode</code>	Defines how the product data type is determined	<ul style="list-style-type: none"> • <code>FullPrecision</code> (default) — The full precision of the result is kept. • <code>KeepLSB</code>— Keep least significant bits. Specify the product word length, while the fraction length is set to maintain the least significant bits of the product. • <code>KeepMSB</code> — Keep most significant bits. Specify the product word length, while the fraction length is set to maintain the most significant bits of the product. • <code>SpecifyPrecision</code>— specify the word and fraction lengths or slope and bias of the product.
<code>ProductSlope</code>	Slope of the product data type	<ul style="list-style-type: none"> • 9.3132e-010 (default) • Any floating-point number <p>Note:</p> $ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$ <p>Changing one of these properties affects the others.</p>

Property	Description	Valid Values
ProductSlopeAdjustmentFactor	Slope adjustment factor of the product data type	<ul style="list-style-type: none"> • 1 (default) • Any floating-point number greater than or equal to 1 and less than 2 <hr/> <p>Note:</p> $ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$ <p>Changing one of these properties affects the others.</p>
ProductWordLength	Word length, in bits, of the product data type	<ul style="list-style-type: none"> • 32 (default) • Any positive integer
RoundingMethod	Rounding method	<ul style="list-style-type: none"> • Nearest (default) — Round toward nearest. Ties round toward positive infinity. • Ceiling — Round toward positive infinity. • Convergent — Round toward nearest. Ties round to the nearest even stored integer (least biased). • Zero — Round toward zero. • Floor — Round toward negative infinity. • Round — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
SumBias	Bias of the sum data type	<ul style="list-style-type: none"> • 0 (default) • Any floating-point number
SumFixedExponent	Fixed exponent of the sum data type	<ul style="list-style-type: none"> • -30 (default) • Any positive or negative integer <hr/> <p>Note: The SumFractionLength is the negative of the SumFixedExponent. Changing one property changes the other.</p>

Property	Description	Valid Values
SumFractionLength	Fraction length, in bits, of the sum data type	<ul style="list-style-type: none"> • 30 (default) • Any positive or negative integer <p>Note: The SumFractionLength is the negative of the SumFixedExponent. Changing one property changes the other.</p>
SumMode	Defines how the sum data type is determined	<ul style="list-style-type: none"> • FullPrecision (default) — The full precision of the result is kept. • KeepLSB — Keep least significant bits. Specify the sum data type word length, while the fraction length is set to maintain the least significant bits of the sum. • KeepMSB — Keep most significant bits. Specify the sum data type word length, while the fraction length is set to maintain the most significant bits of the sum and no more fractional bits than necessary • SpecifyPrecision — Specify the word and fraction lengths or the slope and bias of the sum data type.
SumSlope	Slope of the sum data type	<ul style="list-style-type: none"> • 9.3132e-010 (default) • Any floating-point number <p>Note:</p> $SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$ <p>Changing one of these properties affects the others.</p>

Property	Description	Valid Values
SumSlopeAdjustmentFactor	Slope adjustment factor of the sum data type	<ul style="list-style-type: none"> • 1 (default) • Any floating-point number greater than or equal to 1 and less than 2 <p>Note:</p> $SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$ <p>Changing one of these properties affects the others.</p>
SumWordLength	Word length, in bits, of the sum data type	<ul style="list-style-type: none"> • 32 (default) • Any positive integer

For details about these properties, refer to the “fi Object Properties” on page 2-18. To learn how to specify properties for `fi` objects in Fixed-Point Designer software, refer to “Setting fimath Object Properties” on page 4-11.

How Properties are Related

Sum data type properties

The slope of the sum of two `fi` objects is related to the `SumSlopeAdjustmentFactor` and `SumFixedExponent` properties by

$$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$$

If any of these properties are updated, the others are modified accordingly.

In a `FullPrecision` sum, the resulting word length is represented by

$$W_s = \text{integer length} + F_s$$

where

$$\text{integer length} = \max(W_a - F_a, W_b - F_b) + \text{ceil}(\log_2(\text{NumberOfSummands}))$$

and

$$F_s = \max(F_a, F_b)$$

When the `SumMode` is set to `KeepLSB`, the resulting word length and fraction length is determined by

$$W_s = \text{specified in the } \text{SumWordLength} \text{ property}$$

$$F_s = \max(F_a, F_b)$$

When the `SumMode` is set to `KeepMSB`, the resulting word length and fraction length is determined by

$$W_s = \text{specified in the } \text{SumWordLength} \text{ property}$$

$$F_s = W_s - \text{integer length}$$

where

$$\text{integer length} = \max(W_a - F_a, W_b - F_b) + \text{ceil}(\log_2(\text{NumberOfSummands}))$$

When the `SumMode` is set to `SpecifyPrecision`, you specify both the word and fraction length or slope and bias of the sum data type with the `SumWordLength` and `SumFractionLength`, or `SumSlope` and `SumBias` properties respectively.

Product data type properties

The slope of the product of two `fi` objects is related to the `ProductSlopeAdjustmentFactor` and `ProductFixedExponent` properties by

$$\text{ProductSlope} = \text{ProductSlopeAdjustmentFactor} \times 2^{\text{ProductFixedExponent}}$$

If any of these properties are updated, the others are modified accordingly.

In a `FullPrecision` multiply, the resulting word length and fraction length are represented by

$$W_p = W_a + W_b$$

$$F_p = F_a + F_b$$

When the `ProductMode` is `KeepLSB` the word length and fraction length are determined by

W_p = specified in the `ProductWordLength` property

$$F_p = F_a + F_b$$

When the `ProductMode` is `KeepMSB` the word length and fraction length are

W_p = specified in the `ProductWordLength` property

$$F_p = W_p - \text{integer length}$$

where

$$\text{integer length} = (W_a + W_b) - (F_a - F_b)$$

When the `ProductMode` is set to `SpecifyPrecision`, you specify both the word and fraction length or slope and bias of the product data type with the `ProductWordLength` and `ProductFractionLength`, or `ProductSlope` and `ProductBias` properties respectively.

For more information about how certain functions use the `fimath` properties, see

Setting fimath Object Properties

- “Setting fimath Properties at Object Creation” on page 4-11
- “Using Direct Property Referencing with fimath” on page 4-12

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
```

In addition to creating a `fimath` object at the command line, you can also set `fimath` properties using the **Insert fimath Constructor** dialog box. For an example of this approach, see “Building fimath Object Constructors in a GUI” on page 4-3.

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
```

fimath Properties Usage for Fixed-Point Arithmetic

In this section...

“fimath Rules for Fixed-Point Arithmetic” on page 4-13

“Binary-Point Arithmetic” on page 4-15

“[Slope Bias] Arithmetic” on page 4-18

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 leftmost `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 leftmost `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 14-72.

Binary-Point Arithmetic

The `fimath` object encapsulates the math properties of Fixed-Point Designer 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 4-13 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 add two `fi` objects with data type `double`, but you can not add 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 Designer 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 Designer 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 = numerictype('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.

For more information about the fimath object and its properties, see “fimath Object Properties” on page 4-5

fimath for Sharing Arithmetic Rules

There are two ways of sharing `fimath` properties in Fixed-Point Designer software:

- “Default `fimath` Usage to Share Arithmetic Rules” on page 4-23
- “Local `fimath` Usage to Share Arithmetic Rules” on page 4-23

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 `numericType` object and `fimath` object, and then uses those objects to create two `fi` objects with the same `numericType` 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 4-26

“FullPrecision” on page 4-27

“KeepLSB” on page 4-28

“KeepMSB” on page 4-29

“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 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.

For more information about the `fimath` object and its properties, see “`fimath` Object Properties” on page 4-5

How Functions Use fimath

In this section...
“Functions that use then discard attached fimath” on page 4-32
“Functions that ignore and discard attached fimath” on page 4-32
“Functions that do not perform math” on page 4-33

Functions that use then discard attached fimath

Functions	Note
conv, filter	Error if attached fimaths differ.
mean, median	—

Functions that ignore and discard attached fimath

Functions	Note
accumneg, accumpos	<ul style="list-style-type: none"> By default, use Floor rounding method and Wrap overflow
add, sub, mpy	<ul style="list-style-type: none"> Override and discard any fimath objects attached to the input fi objects Uses the fimath from input, F, as in add(F, a, b)
CORDIC functions — cordicabs, cordicangle, cordicatan2, cordiccart2pol, cordiccexp, cordicccos, cordicpol2cart, cordicrotate, cordicsin, cordicsincos, cordicsqrt	CORDIC functions use their own internal fimath: <ul style="list-style-type: none"> Rounding Mode – Floor Overflow Action – Wrap
mod	—
qr	—
quantize	Uses the math settings on the quantizer object, ignores and discards any fimath settings on the input

Functions	Note
Trigonometric functions — <code>atan2</code> , <code>cos</code> , <code>sin</code>	—

Functions that do not perform math

Functions	Note
Built-in Types— <code>int32</code> , <code>int64</code> , <code>int8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code> , <code>uint8</code>	Ignore any <code>fimath</code> settings on the input. Always use the rounding method <code>Round</code> when casting to the new data type. The output is not a <code>fi</code> object so it has no attached <code>fimath</code> .
<code>bitshift</code> , <code>bitsra</code> , <code>bitsrl</code>	<code>OverflowAction</code> and <code>RoundingMethod</code> are ignored — bits drop off the end.
<code>bitshift</code>	<code>RoundingMethod</code> is ignored, but <code>OverflowAction</code> property is obeyed.

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 2-18. To learn how to specify properties for `fipref` objects in Fixed-Point Designer 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 `numericType` 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 Designer 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 Designer 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 6-6
- “numerictype of Fixed-Point Objects” on page 6-12
- “numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects” on page 6-15

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 6-4

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 Designer 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 Designer 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
```

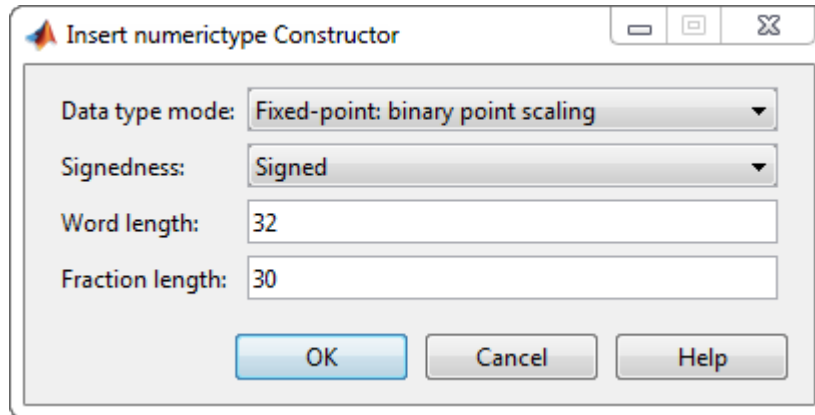
Example: Build numerictype Object Constructors in a GUI

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 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 On the **Home** tab, in the **File** section, click **New > Script** to open the MATLAB Editor
- 2 On the **Editor** tab, in the **Edit** section, click  in the **Insert** button group. Click the **Insert numerictype...** to open the **Insert numerictype Constructor** dialog box.
- 3 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



- 4 To insert the `numerictype` object constructor 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 in your file:

```
5 T numerictype(1, 32, 30)
```

numerictype Object Properties

In this section...

“Data Type and Scaling Properties” on page 6-6

“How Properties are Related” on page 6-9

“Set numerictype Object Properties” on page 6-10

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:

Property	Description	Valid Values
Bias	Bias associated with the object. Along with the slope, the bias forms the scaling of a fixed-point number.	<ul style="list-style-type: none"> Any floating-point number
DataType	Data type category	<ul style="list-style-type: none"> Fixed (default) — Fixed-point or integer data type boolean — Built-in MATLAB boolean data type double — Built-in MATLAB double data type ScaledDouble — Scaled double data type single — Built-in MATLAB single data type
DataTypeMod	Data type and scaling associated with the object	<ul style="list-style-type: none"> Fixed-point: binary point scaling (default) — Fixed-point data type and scaling defined by the word length and fraction length Boolean — Built-in boolean Double — Built-in double

Property	Description	Valid Values
		<ul style="list-style-type: none"> • Fixed-point: slope and bias scaling — Fixed-point data type and scaling defined by the slope and bias • Fixed-point: unspecified scaling — Fixed-point data type with unspecified scaling • Scaled double: binary point scaling — Double data type with fixed-point word length and fraction length information retained • Scaled double: slope and bias scaling — Double data type with fixed-point slope and bias information retained • Scaled double: unspecified scaling — Double data type with unspecified fixed-point scaling • Single — Built-in single
FixedExponent	Fixed-point exponent associated with the object	<ul style="list-style-type: none"> • Any integer <hr/> <p>Note: The FixedExponent property is the negative of the FractionLength. Changing one property changes the other.</p>
FractionLength	Fraction length of the stored integer value, in bits	<ul style="list-style-type: none"> • Best precision fraction length based on value of the object and the word length (default) • Any integer <hr/> <p>Note: The FractionLength property is the negative of the FixedExponent. Changing one property changes the other.</p>

Property	Description	Valid Values
Scaling	Scaling mode of the object	<ul style="list-style-type: none"> • BinaryPoint (default) — Scaling for the <code>fi</code> object is defined by the fraction length. • SlopeBias — Scaling for the <code>fi</code> object is defined by the slope and bias. • Unspecified — A temporary setting that is only allowed at <code>fi</code> object creation, to allow for the automatic assignment of a binary point best-precision scaling.
Signed	<p>Whether the object is signed</p> <hr/> <p>Note: Although the Signed property is still supported, the Signedness property always appears in the numerictype object display. If you choose to change or set the signedness of your numerictype objects using the Signed property, MATLAB updates the corresponding value of the Signedness property.</p>	<ul style="list-style-type: none"> • true (default) — signed • false — unsigned • 1 — signed • 0 — unsigned • [] — auto
Signedness	Whether the object is signed, unsigned, or has an unspecified sign	<ul style="list-style-type: none"> • Signed (default) • Unsigned • Auto — unspecified sign
Slope	<p>Slope associated with the object</p> <p>Along with the bias, the slope forms the scaling of a fixed-point number.</p>	<ul style="list-style-type: none"> • Any floating-point number <hr/> <p>Note:</p> $slope = slope\ adjustment\ factor \times 2^{fixed\ exponent}$ <p>Changing one of these properties changes the other.</p>

Property	Description	Valid Values
Slope AdjustmentFactor	Slope adjustment associated with the object	<ul style="list-style-type: none"> Any number greater than or equal to 1 and less than 2
	The slope adjustment is equivalent to the fractional slope of a fixed-point number.	<p>Note:</p> $\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$ <p>Changing one of these properties changes the other.</p>
WordLength	Word length of the stored integer value, in bits	<ul style="list-style-type: none"> 16 (default) Any positive integer if Signedness is Unsigned or unspecified Any integer greater than one if Signedness is set to Signed

These properties are described in detail in the “fi Object Properties” on page 2-18. To learn how to specify properties for numerictype objects in Fixed-Point Designer software, refer to “Set numerictype Object Properties” on page 6-10.

How Properties are Related

Properties that affect the slope

The **Slope** field of the numerictype object 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

In binary point scaling the numerictype StoredIntegerValue and RealWorldValue properties are related according to

$$\text{real-world value} = \text{stored integer value} \times 2^{\text{fraction length}}$$

In [Slope Bias] scaling the RealWorldValue can be represented by

$$\text{real-world value} = \text{stored integer value} \times (\text{slope adjustment factor} \times 2^{\text{fixed exponent}}) + \text{bias}$$

which is equivalent to

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

If any of these properties are updated, the others are modified accordingly.

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
```

In addition to creating a numerictype object at the command line, you can also set numerictype properties using the **Insert numerictype Constructor** dialog box. For

an example of this approach, see “Example: Build numerictype Object Constructors in a GUI” on page 6-4.

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

```

numerictype of Fixed-Point Objects

In this section...

“Valid Values for numerictype Object Properties” on page 6-12

“Properties That Affect the Slope” on page 6-14

“Stored Integer Value and Real World Value” on page 6-14

Valid Values for numerictype Object Properties

The `numerictype` object contains all the data type and scaling attributes of a fixed-point object. The `numerictype` object behaves like any MATLAB object, except that it only lets you set valid values for defined fields. The following table shows the possible settings of each field of the object.

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.

<code>DataTypeMode</code>	<code>DataType</code>	<code>Scaling</code>	<code>Signedness</code>	<code>Word-Length</code>	<code>Fraction-Length</code>	<code>Slope</code>	<code>Bias</code>
<i>Fixed-point data types</i>							
Fixed-point: binary point scaling	Fixed	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,535	Positive or negative integer	$2^{(-}$ fraction length)	0
Fixed-point: slope and bias scaling	Fixed	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,535	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 65,535	N/A	N/A	N/A
<i>Scaled double data types</i>							
Scaled double: binary point scaling	ScaledDoub1	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,535	Positive or negative integer	2 ^(- fraction length)	0
Scaled double: slope and bias scaling	ScaledDoub1	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,535	N/A	Any floating- point number	Any floating- point number
Scaled double: unspecified scaling	ScaledDoub1	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,535	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 object 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

In binary point scaling the numerictype StoredIntegerValue and RealWorldValue properties are related according to

$$\text{real-world value} = \text{stored integer value} \times 2^{\text{fraction length}}$$

In [Slope Bias] scaling the RealWorldValue can be represented by

$$\text{real-world value} = \text{stored integer value} \times (\text{slope adjustment factor} \times 2^{\text{fixed exponent}}) + \text{bias}$$

which is equivalent to

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

If any of these properties are updated, the others are modified accordingly.

For more detail on these properties see “numerictype Object Properties” on page 6-6.

For more information on scaling see “Data Types and Scaling”

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
```

For more detail on the properties of `numerictype` objects see “`numerictype` Object Properties” on page 6-6.

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 Designer 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 2-18 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.

Automated Fixed-Point Conversion

- “Fixed-Point Conversion Workflows” on page 8-2
- “Automated Fixed-Point Conversion” on page 8-4
- “Best Practices for Debugging Out-of-the-Box Conversion by the Fixed-Point Converter” on page 8-25
- “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 8-42
- “Generated Fixed-Point Code” on page 8-45
- “Fixed-Point Code for MATLAB Classes” on page 8-51
- “Automated Fixed-Point Conversion Best Practices” on page 8-54
- “Replacing Functions Using Lookup Table Approximations” on page 8-62
- “Custom Plot Functions” on page 8-63
- “Generate Fixed-Point MATLAB Code for Multiple Entry-Point Functions” on page 8-65
- “Convert Code Containing Global Data to Fixed Point” on page 8-69
- “Convert Code Containing Global Variables to Fixed-Point” on page 8-74
- “Convert Code Containing Structures to Fixed Point” on page 8-78
- “Data Type Issues in Generated Code” on page 8-81
- “System Objects Supported by Fixed-Point Converter App” on page 8-84
- “Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App” on page 8-85

Fixed-Point Conversion Workflows

In this section...
“Choosing a Conversion Workflow” on page 8-2
“Automated Workflow” on page 8-3
“Manual Workflow” on page 8-3

Choosing a Conversion Workflow

MathWorks® provides a number of solutions for fixed-point conversion. Which conversion method you use depends on your end goal and your level of fixed-point expertise.

Goal	Conversion Method	See Also
Use generated fixed-point MATLAB code for simulation purposes.	If you are new to fixed-point modeling, use the Fixed-Point Converter app.	“Automated Workflow” on page 8-3
	If you are familiar with fixed-point modeling, and want to quickly explore design tradeoffs, convert your code manually.	“Manual Workflow” on page 8-3
Generate fixed-point C code (requires MATLAB Coder™)	MATLAB Coder Fixed-Point Conversion tool	“Convert MATLAB Code to Fixed-Point C Code” (MATLAB Coder)
Generated HDL code (requires HDL Coder™)	HDL Coder Workflow Advisor	“Floating-Point to Fixed-Point Conversion” (HDL Coder)
Integrate fixed-point MATLAB code in larger applications for system-level simulation.	Generate a MEX function from the fixed-point algorithm and call the MEX function instead of the original MATLAB function.	“Propose Data Types Based on Simulation Ranges” on page 9-15 and “Propose Data Types Based on Derived Ranges” on page 9-29

Automated Workflow

If you are new to fixed-point modeling and you are looking for a direct path from floating-point MATLAB to fixed-point MATLAB code, use the automated workflow. Using this automated workflow, you can obtain data type proposals based on simulation ranges, static ranges, or both. For more information, see “Automated Fixed-Point Conversion” on page 8-4, “Propose Data Types Based on Simulation Ranges” on page 9-15, and “Propose Data Types Based on Derived Ranges” on page 9-29.

Manual Workflow

If you have a baseline understanding of fixed-point implementation details and an interest in exploring design tradeoffs to achieve optimized results, use the separate algorithm/data type workflow. Separating algorithmic code from data type specifications allows you to quickly explore design tradeoffs. This approach provides readable, portable fixed-point code that you can easily integrate into other projects. For more information, see “Manual Fixed-Point Conversion Workflow” on page 12-2 and “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros” on page 12-19.

Automated Fixed-Point Conversion

In this section...

“Automated Fixed-Point Conversion Capabilities” on page 8-4

“Code Coverage” on page 8-5

“Proposing Data Types” on page 8-9

“Locking Proposed Data Types” on page 8-11

“Viewing Functions” on page 8-11

“Viewing Variables” on page 8-18

“Log Data for Histogram” on page 8-20

“Function Replacements” on page 8-22

“Validating Types” on page 8-23

“Testing Numerics” on page 8-23

“Detecting Overflows” on page 8-24

Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Converter app or at the command line using the `fiaccel` function -`float2fixed` option. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually entered ranges take precedence over simulation ranges and the app uses them when proposing data types. In addition, you can modify and lock the proposed type so that the app cannot change it. For more information, see “Locking Proposed Data Types” on page 8-11.

For a list of supported MATLAB features and functions, see “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 8-42.

During fixed-point conversion, you can:

- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.

- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test file with the fixed-point types applied.
- View a histogram of bits that each variable uses.
- Detect overflows.

Code Coverage

By default, the app shows code coverage results. Your test files must exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want.

Reviewing code coverage results helps you to verify that your test files are exercising the algorithm adequately. If the code coverage is inadequate, modify the test files or add more test files to increase coverage. If you simulate multiple test files in one run, the app displays cumulative coverage. However, if you specify multiple test files, but run them one at a time, the app displays the coverage of the file that ran last.

The app displays a color-coded coverage bar to the left of the code.

```

11 persistent current_state
12 if isempty( current_state )
13     current_state = S1;
14 end
15
16 % switch to new state based on the value state register
17 switch uint8( current_state )
18     case S1
19         % value of output 'Z' depends both on state and inputs
20         if (A)
21             Z = true;
22             current_state( 1 ) = S1;
23         else
24             Z = false;
25             current_state( 1 ) = S2;
26         end
27     case S2
28         if (A)
29             Z = false;
30             current_state( 1 ) = S1;
31         else
32             Z = true;
33             current_state( 1 ) = S2;
34         end
35     case S3
36         if (A)
37             Z = false;
38             current_state( 1 ) = S2;
39         else
40             Z = true;
41             current_state( 1 ) = S3;
42     end

```

This table describes the color coding.

Coverage Bar Color	Indicates
Green	One of the following situations:

Coverage Bar Color	Indicates
	<ul style="list-style-type: none">• The entry-point function executes multiple times and the code executes more than one time.• The entry-point function executes one time and the code executes one time. <p>Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.</p>
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you place your cursor over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that the section executes.

```

11 persistent current_state
12 if isempty( current_state )
13     current_state = S1;           1 calls
14 end                               51 calls
15
16 % switch to new state based on the value state register
17 switch uint8( current_state )
18     case S1
19         % value of output 'Z' depends both on state and inputs
20         if (A)
21             Z = true;           37 calls
22             current_state( 1 ) = S1;
23         else                       7 calls
24             Z = false;
25             current_state( 1 ) = S2;
26         end
27     case S2                               51 calls
28         if (A)
29             Z = false;           7 calls
30             current_state( 1 ) = S1;
31         else                               0 calls
32             Z = true;
33             current_state( 1 ) = S2;
34         end
35     case S3                               51 calls
36         if (A)
37             Z = false;           0 calls
38             current_state( 1 ) = S2;
39         else
40             Z = true;
41             current_state( 1 ) = S3;
42         end


```

To verify that your test files are testing your algorithm over the intended operating range, review the code coverage results.

Coverage Bar Color	Action
Green	If you expect sections of code to execute more frequently than the coverage shows, either modify the MATLAB code or the test files.
Orange	This behavior is expected for initialization code, for example, the initialization of persistent variables. If you expect the code to execute more than one time, either modify the MATLAB code or the test files.
Red	If the code that does not execute is an error condition, this behavior is acceptable. If you expect the code to execute, either modify the

Coverage Bar Color	Action
	MATLAB code or the test files. If the code is written conservatively and has upper and lower boundary limits, and you cannot modify the test files to reach this code, add static minimum and maximum values. See “Computing Derived Ranges” on page 8-10.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage can speed up simulation. To turn off code coverage, on the **Convert to Fixed Point** page:

- 1 Click the **Analyze** arrow .
- 2 Clear the **Show code coverage** check box.

Proposing Data Types

The app proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data (also known as static ranges), or both. If you run a simulation and compute derived ranges, the app merges the simulation and derived ranges.

Note: You cannot propose data types based on derived ranges for MATLAB classes.

You can manually enter static ranges. These manually entered ranges take precedence over simulation ranges and the app uses them when proposing data types. You can modify and lock the proposed type so that the tool cannot change it. For more information, see “Locking Proposed Data Types” on page 8-11.

Running a Simulation

During fixed-point conversion, the app generates an instrumented MEX function for your entry-point MATLAB file. If the build completes without errors, the app displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the app provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the app

displays them on the **Function Replacements** tab. See “Function Replacements” on page 8-22.

Before running a simulation, specify the test file or files that you want to run. When you run a simulation, the app runs the test file, calling the instrumented MEX function. If you modify the MATLAB design code, the app automatically generates an updated MEX function before running a test file.

If the test file runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the app cannot modify them.

If the test file fails, the errors are displayed on the **Output** tab.

Test files must exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test file covers the operating range of the algorithm with the accuracy that you want. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app merges the simulation results.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Log Data for Histogram” on page 8-20.

Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually entered data types are locked so that the app cannot modify them. The app uses these data types to calculate the input minimum and maximum

values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 8-11.

When you select **Compute Derived Ranges**, the app runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces + / - Inf derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the app performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. When the timeout is reached, the app aborts the analysis.

Locking Proposed Data Types

You can lock proposed data types against changes by the app using one of the following methods:

- Manually setting a proposed data type in the app.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

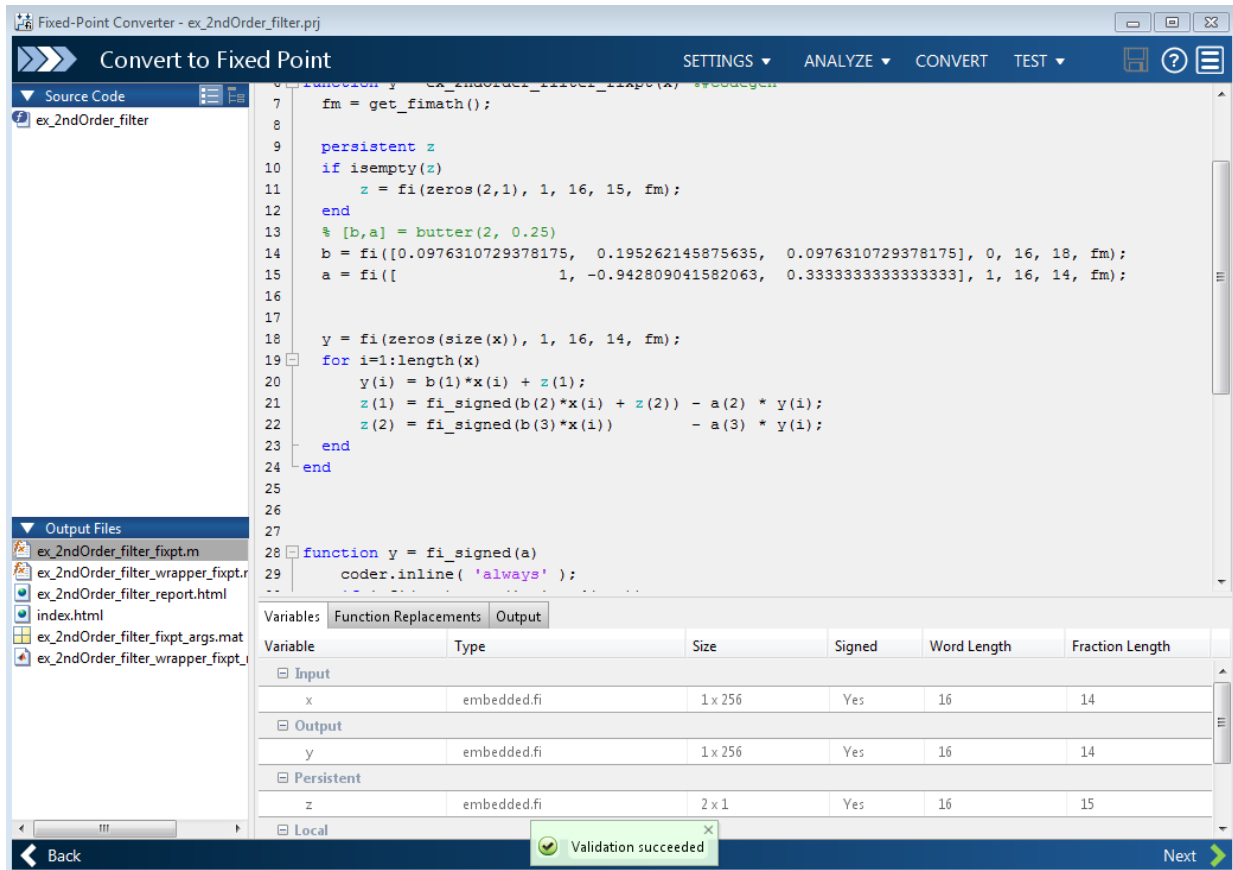
The app displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting **Undo changes**. This action unlocks only the selected type.
- Right-clicking and selecting **Undo changes for all variables**. This action unlocks all locked proposed types.

Viewing Functions

During the **Convert to Fixed Point** step of the fixed-point conversion process, you can view a list of functions in your project in the left pane. This list also includes function specializations and class methods. When you select a function from the list, the MATLAB code for that function or class method is displayed in the code window and the variables that they use are displayed on the **Variables** tab.

After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the app retains the original function name in the fixed-point file name and appends the fixed-point suffix. For example, here the fixed-point version of `ex_2ndOrder_filter.m` is `ex_2ndOrder_filter_fixpt.m`.



Classes

The app displays information for the class and each of its methods. For example, consider a class, `Counter`, that has a static method, `MAX_VALUE`, and a method, `next`.

If you select the class, the app displays the class and its properties on the **Variables** tab.

The screenshot displays the MATLAB IDE interface. On the left, the 'Source Code' pane shows a project structure with files like 'Counter', 'next', 'MAX_VALUE', and 'use_counter'. Below it, the 'Output Files' pane lists generated files such as 'Counter_fixpt.m', 'use_counter_fixpt.m', and 'use_counter_report.html'. The main editor shows the source code for a class named 'Counter'.

```

1 classdef Counter < handle
2     properties
3         Value
4     end
5
6     methods (Static)
7         function t = MAX_VALUE()
8             t = 128;
9         end
10    end
11
12    methods
13        function this = Counter()
14            this.Value = 0;
15        end
16
17        function v = next(this)
18            v = this.Value;
19            if this.Value == this.MAX_VALUE
20                this.Value = 0;
21            else
22                this.Value = this.Value + 1;
23            end
24        end
25    end
26 end
27

```

At the bottom, the 'Type Validation Output' table is displayed, showing the types and ranges for variables used in the selected method.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Output							
this							
Value	double	0	128			Yes	numeric(0, 8, 0)

Navigation buttons 'Back' and 'Next' are visible at the bottom of the window.

If you select a method, the app displays only the variables that the method uses.

The screenshot displays the MATLAB IDE interface. On the left, the 'Source Code' pane shows a project structure with files like 'Counter', 'next', 'MAX_VALUE', and 'use_counter'. The 'Output Files' pane lists generated files such as 'Counter_fixpt.m' and 'use_counter_report.html'. The main editor shows the following MATLAB code:

```

1 classdef Counter < handle
2     properties
3         Value
4     end
5
6     methods (Static)
7         function t = MAX_VALUE()
8             t = 128;
9         end
10    end
11
12    methods
13        function this = Counter()
14            this.Value = 0;
15        end
16
17        function v = next(this)
18            v = this.Value;
19            if this.Value == this.MAX_VALUE
20                this.Value = 0;
21            else
22                this.Value = this.Value + 1;
23            end
24        end
25    end
26 end
27

```

Below the code editor, the 'Type Validation Output' table is visible, showing the following data:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
▲ Output							
t	double	128	128			Yes	numeric(0, 8, 0)

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

Specializations

If a function is specialized, the app lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

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

```
tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));
```

```
ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));
```

```
y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
```



```
end
```

```
function y = foo(u)
    y = u * 2;
end
```

```
function y = bar(u)
    y = u * 4;
end
```

If you select the top-level function, the app displays all the variables on the **Variables** tab.

The screenshot shows a software interface with a source code editor and a variables table. The source code editor displays the following code:

```

1 function y = dut(u, v)
2
3     tt1 = foo(u);
4     tt2 = foo([u v]);
5     tt3 = foo(complex(u,v));
6
7     ss1 = bar(u);
8     ss2 = bar([u v]);
9     ss3 = bar(complex(u,v));
10
11    y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13    -end
14
15 function y = foo(u)
16     y = u * 2;
17 -end
18
19 function y = bar(u)
20     y = u * 4;
21 -end

```

The variables table below the code editor shows the following data:

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	double	10	10	Yes	numerictype(0, 4, 0)
v	double	20	20	Yes	numerictype(0, 5, 0)
Output					
y	double	300	300	Yes	numerictype(0, 9, 0)
Local					
tt1	double	20	20	Yes	numerictype(0, 5, 0)

If you select the tree view, the app also displays the line numbers for the call to each specialization.

The screenshot shows a MATLAB IDE interface. On the left, a 'Source Code' pane displays a tree view of functions: 'dut', 'foo > 3', 'foo > 1', 'foo > 2', 'bar > 1', 'bar > 2', and 'bar > 3'. The main editor shows the source code for these functions. Below the code, a table displays the 'Variables' section, showing the types and ranges for variables 'u' and 'y'.

```

1 function y = dut(u, v)
2
3 tt1 = foo(u);
4 tt2 = foo([u v]);
5 tt3 = foo(complex(u,v));
6
7 ss1 = bar(u);
8 ss2 = bar([u v]);
9 ss3 = bar(complex(u,v));
10
11 y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13 end
14
15 function y = foo(u)
16     y = u * 2;
17 end
18
19 function y = bar(u)
20     y = u * 4;
21 end
    
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	complex double	10	20	Yes	numerictype(0, 5, 0)
Output					
y	complex double	20	40	Yes	numerictype(0, 6, 0)

If you select a specialization, the app displays only the variables that the specialization uses.

The screenshot displays a software interface for automated fixed-point conversion. On the left, a 'Source Code' panel lists several code snippets, with 'dut > foo > 3' selected. The main area shows the corresponding MATLAB code for these snippets. Below the code is a table with tabs for 'Variables', 'Function Replacements', and 'Output'. The 'Output' tab is active, showing a table with columns for Variable, Type, Sim Min, Sim Max, Whole Number, and Proposed Type.

```

1 function y = dut(u, v)
2
3     tt1 = foo(u);
4     tt2 = foo([u v]);
5     tt3 = foo(complex(u,v));
6
7     ss1 = bar(u);
8     ss2 = bar([u v]);
9     ss3 = bar(complex(u,v));
10
11    y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13    ~end
14
15 function y = foo(u)
16     y = u * 2;
17    ~end
18
19 function y = bar(u)
20     y = u * 4;
21    ~end

```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	complex double	10	20	Yes	numerictype(0, 5, 0)
Output					
y	complex double	20	40	Yes	numerictype(0, 6, 0)

In the generated fixed-point code, the number of each fixed-point specialization matches the number in the **Source Code** list, which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for `foo > 1` is named `foo_s1`.

The screenshot displays the MATLAB IDE interface. On the left, the 'Source Code' pane shows a tree view with 'dut' expanded, listing files like 'dut_fixpt.m', 'dut_wrapper_fixpt.m', etc. The main editor shows the source code for 'dut_fixpt.m' with line numbers 7 through 30. Below the code, the 'Variables' tab is active, showing a table with columns: Variable, Type, Sim Min, Sim Max, Whole Number, and Proposed Type.

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	complex double	10	20	Yes	numeric(0, 5, 0)
Output					
y	complex double	40	80	Yes	numeric(0, 7, 0)

At the bottom of the IDE, a green notification box says 'Validation succeeded'. Navigation buttons for 'Back' and 'Next' are visible.

Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the app.

- **Static Min** and **Static Max** — The static minimum and maximum values.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

When you compute derived ranges, the app runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the app.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The app determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the app uses the edited values in subsequent analyses. You can revert to the types proposed by the app.

- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

Because the app does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

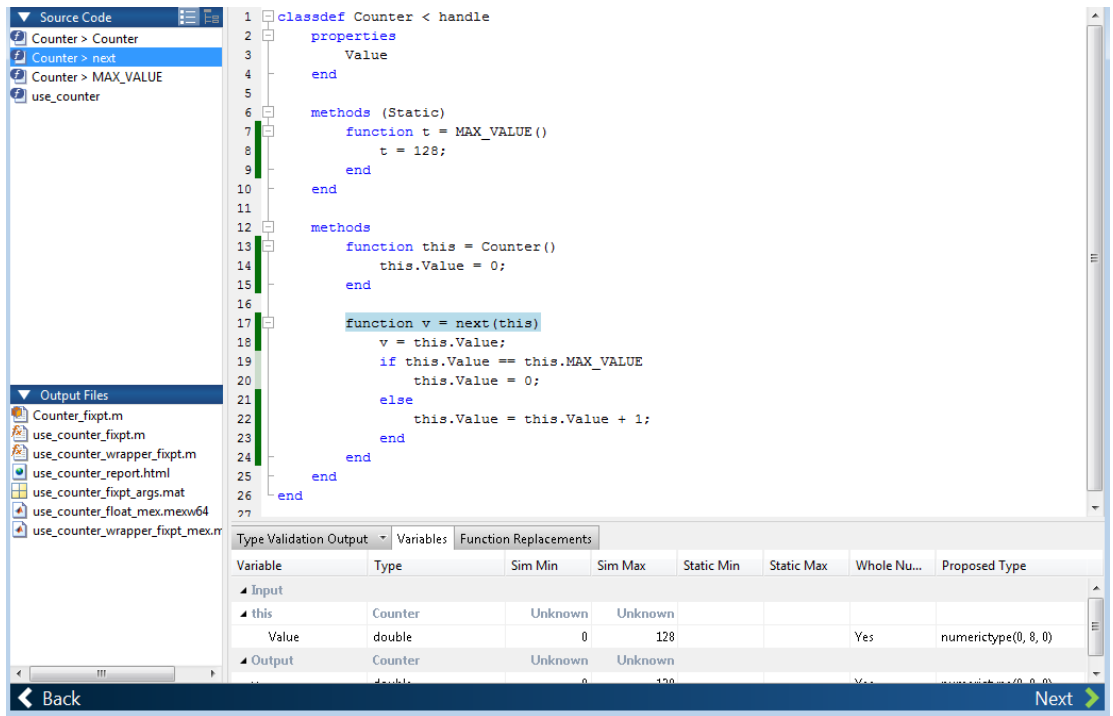
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The app highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Viewing Information for MATLAB Classes

The app displays:

- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Source Code** list on the **Convert to Fixed Point** page to select which class or class method to view. If you select a class method, the app highlights the method in the code window.




- Information about MATLAB classes on the **Variables** tab.

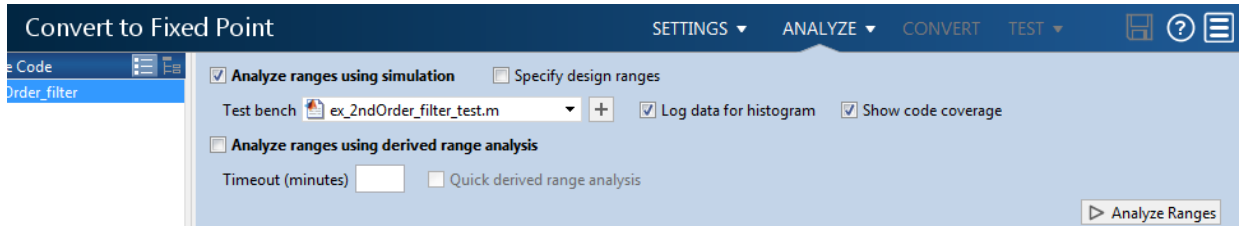
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output							
v	double	0	128			Yes	numerictype(0, 8, 0)

Log Data for Histogram

To log data for histograms:

- On the **Convert to Fixed Point** page, click the **Analyze** arrow .

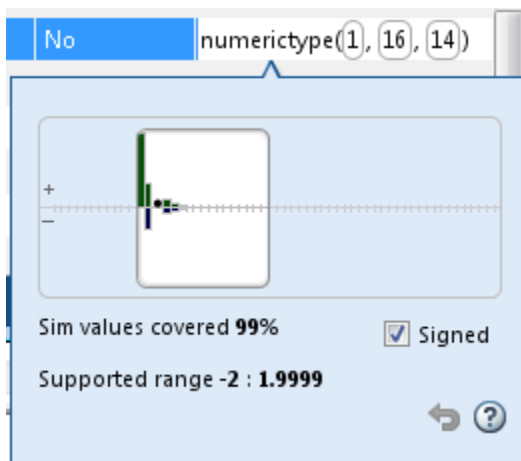
- Select **Log data for histogram**.



- Click **Analyze Ranges**.

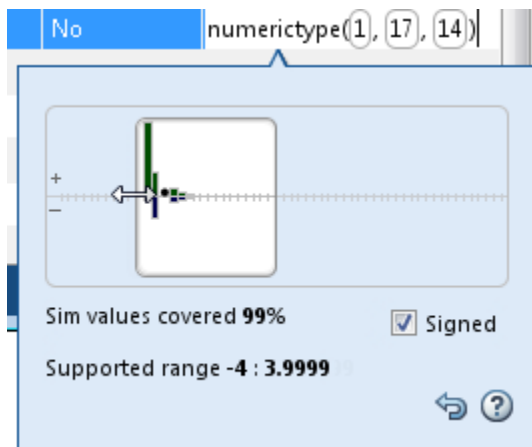
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1, 16, 14)`.




You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the app lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.

Simulation Output		Variables		Function Replacements			
Enter a function to replace						Custom Function ▾	+
Function or Operator	Replacement						
▲ Custom Function	<i>Function Name</i>						
foo	foo_fixedpoint						
▲ Lookup Table	<i>Interpolation Method</i>	<i>Design Min</i>	<i>Design Max</i>	<i>Number of Points</i>			
exp	None	Auto	Auto	1000			

You can add and remove function replacements from this list. If you enter a function replacement for a function, the replacement function is used when you build the

project. If you do not enter a replacement, the app uses the type specified in the original MATLAB code for the function.

Note: Using this table, you can replace the names of the functions but you cannot replace argument patterns.

If code generation readiness screening is disabled, the list of unsupported functions on the **Function Replacements** tab can be incomplete or incorrect. In this case, add the functions manually. See .

Validating Types

Converting the code to fixed point validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

Testing Numerics

After converting code to fixed point and validating the proposed fixed-point data types, click **Test** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test file to define inputs or run a simulation, the app uses this test file to test numerics. Optionally, you can add test files and select to run more than one test file. The app compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the app generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For nonscalar outputs, only the error information is shown.

After fixed-point simulation, if the numerical results do not meet the accuracy that you want, modify fixed-point data type settings and repeat the type validation and numerical










testing steps. You might have to iterate through these steps multiple times to achieve the results that you want.

Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion app runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-precision floating-point, they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type. For more information, see “Scaled Doubles” on page 32-22.

If the app detects overflows, on its **Overflow** tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window

Variables	Function Replacements	Overflows	
	Function	Line	Description
	overflow_fixpt	7	Overflow error in expression 'x'.
	overflow_fixpt	7	Overflow error in expression 'y'.
	overflow_fixpt	10	Overflow error in expression 'z'.
	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
	overflow_fixpt	10	Overflow error in expression 'x'.
	overflow_fixpt	10	Overflow error in expression 'x*y'.
	overflow_fixpt	10	Overflow error in expression 'y'.
	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the app also provides overflow information for these expressions.

See Also

“Detect Overflows” on page 9-6

Best Practices for Debugging Out-of-the-Box Conversion by the Fixed-Point Converter

This example shows some best practices for debugging your fixed-point code when you need more than out of the box conversion.

In this section...

“Prerequisites” on page 8-25

“Convert to Fixed Point Using Default Configuration” on page 8-30

“Determine Where Numerical Issues Originated” on page 8-33

“Adjust `fimath` Settings” on page 8-35

“Convert `kalman_filter` Function Using New `fimath` Settings” on page 8-36

“Adjust Word Length Settings” on page 8-38

“Replace Constant Functions” on page 8-39

Prerequisites

- 1 Create a local working folder, for example, `c:\kalman_filter`.
- 2 In your local working folder, create the following files.

- **kalman_filter function**

This is the main entry-point function for your project.

```
function [y] = kalman_filter(z,N0)
    %#codegen
    A = kalman_stm();

    % Measurement Matrix
    H = [1 0];

    % Process noise variance
    Q = 0;
    % Measurement noise variance
    R = NO ;

    persistent x_est p_est
    if isempty(x_est)
        % Estimated state
```

```
        x_est = [0; 1];
        % Estimated error covariance
        p_est = NO * eye(2, 2);
    end

    % Kalman algorithm
    % Predicted state and covariance
    x_prd = A * x_est;
    p_prd = A * p_est * A' + Q;

    % Estimation
    S = H * p_prd' * H' + R;
    B = H * p_prd';
    klm_gain = matrix_solve(S,B)';

    % Estimated state and covariance
    x_est = x_prd + klm_gain * (z - H * x_prd);
    p_est = p_prd - klm_gain * H * p_prd;

    % Compute the estimated measurements
    y = H * x_est;
end
```

- **kalman_stm function**

This function is called by the `kalman_filter` function and computes the state transition matrix.

```
function A = kalman_stm()
    f0 = 200;
    dt = 1/1e4;
    % Kalman filter initialization
    % State transition Matrix
    A = [cos(2*pi*f0*dt), -sin(2*pi*f0*dt);
        sin(2*pi*f0*dt), cos(2*pi*f0*dt)];
end
```

- **matrix_solve function**

This function is a more efficient implementation of a matrix left divide.

```
function x = matrix_solve(a,b)
    %fixed-point conversion friendly matrix solve: a * x = b

    % initialize x
```

```

x = zeros(size(a,1),size(b,2));
% compute lu decomposition of a
[l, u] = lu_replacement(a);
% solve x = a\b for every column of b
  through forward and backward substitution
for col = 1:size(b,2)
    bcol = b(:,col);
    y = forward_substitute(l,bcol);
    x(:,col) = back_substitute(u,y);
end

```

end

- **lu_replacement function**

This function is called by the `matrix_solve` function.

```

function [l,A]=lu_replacement(A)
N=size(A,1);
l = eye(N);
for n=1:N-1
    piv = A(n,n);
    for k=n+1:N
        mult = divide_no_zero(A(k,n),piv);
        A(k,:) = -mult*A(n,:) + A(k,:);
        l(k,n) = mult;
    end
end
end

```

end

- **forward_substitute function**

This function is called by the `matrix_solve` function.

```

function y = forward_substitute(l,b)
% forward substitution
N = size(b,1);
y = zeros(N,1);
% forward substitution
y(1) = divide_no_zero(b(1),l(1,1));
for n = 2:N
    acc = 0;
    for k = 1:n-1
        acc(:) = acc + y(k)*l(n,k);
    end
end

```

```
        y(n) = divide_no_zero((b(n)-acc),l(n,n));  
    end
```

```
end
```

- **back_substitute function**

This function is called by the `matrix_solve` function.

```
function x = back_substitute(u,y)  
    % backwards substitution  
    N = size(u,1);  
    x = zeros(N,1);  
  
    % backward substitution  
    x(N) = divide_no_zero(y(N),u(N,N));  
  
    for n = (N-1):(-1):(1)  
        acc = 0;  
        for k = n:(N)  
            acc(:) = acc + x(k)*u(n,k);  
        end  
        x(n) = divide_no_zero((y(n) - acc),u(n,n));  
    end  
end
```

- **divide_no_zero function**

This function is called by the `lu_replacement`, `forward_substitute` and `back_substitute` functions.

```
function y = divide_no_zero(num, den)  
    % Divide and avoid division by zero  
    if den == 0  
        y = 0;  
    else  
        y = num/den;  
    end  
end
```

- **kalman_filter_tb test file**

This script generates a noisy sine wave, and calls the `kalman_filter` function to filter the noisy signal. It then plots the signals for comparison.

```
% KALMAN FILTER EXAMPLE TEST BENCH
```

```
clear all
step = ((400*pi)/1000)/10;
TIME_STEPS = 400;
X = 0:step:TIME_STEPS;
rng default;
rng(1);
Orig_Signal = sin(X);
Noisy_Signal = Orig_Signal + randn(size(X));
Clean_Signal = zeros(size(X));
for i = 1:length(X)
Clean_Signal(i) = kalman_filter(Noisy_Signal(i), 1);
end
figure
subplot(5,1,1)
plot(X,rand(size(X)))
axis([1 TIME_STEPS 0 1.25]);
title('Noise')

% Plot Noisy Signal
subplot(5,1,2)
plot(X,Noisy_Signal)
axis([1 TIME_STEPS -4 4]);
title('Noisy Signal')

% Plot Filtered Clean Signal
subplot(5,1,3)
plot(X,Clean_Signal)
axis([1 TIME_STEPS -2 2]);
title('Filtered Signal')

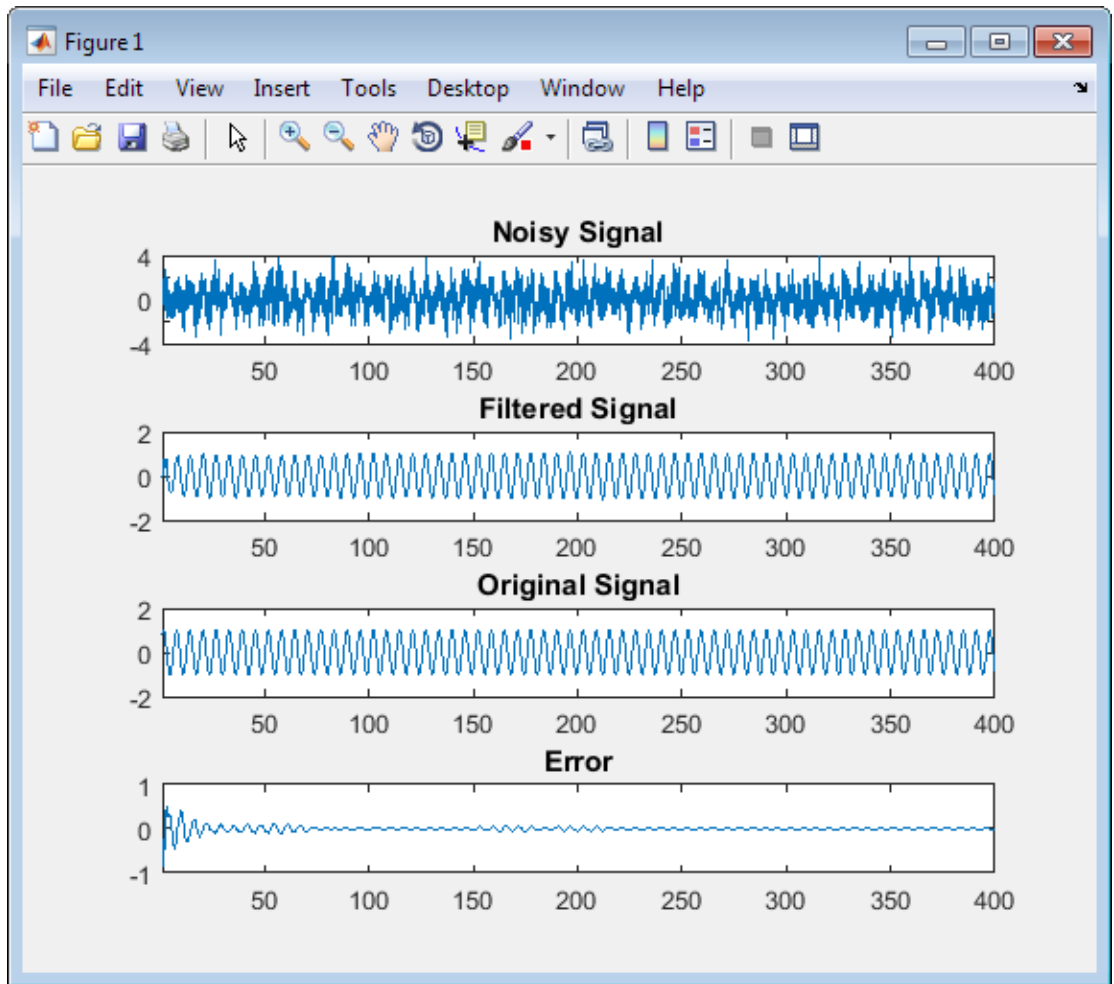
% Plot Original Signal
subplot(5,1,4)
plot(X,Orig_Signal)
axis([1 TIME_STEPS -2 2]);
title('Original Signal')

% Plot Error
subplot(5,1,5)
plot(X, (Clean_Signal - Orig_Signal))
axis([1 TIME_STEPS -1 1]);
title('Error')
figure(gcf)
```

Convert to Fixed Point Using Default Configuration

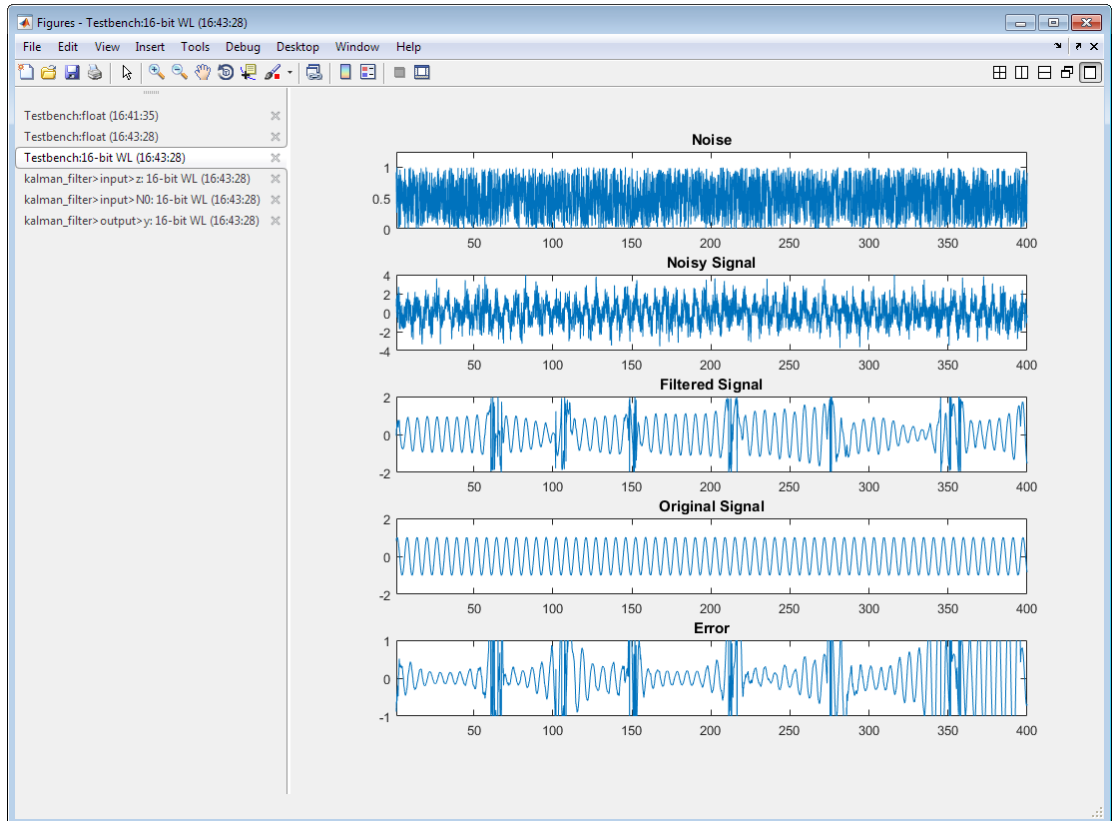
- 1 From the apps gallery, open the Fixed-Point Converter app.
- 2 On the **Select Source Files** page, browse to the `kalman_filter.m` file and click **Open**.
- 3 Click **Next**. On the **Define Input Types** page, browse to the `kalman_filter_tb` file. Click **Autodefine Input Types**.

The test file runs and plots the input noisy signal, the filtered signal, the ideal filtered signal, and the difference between the filtered and the ideal filtered signal.

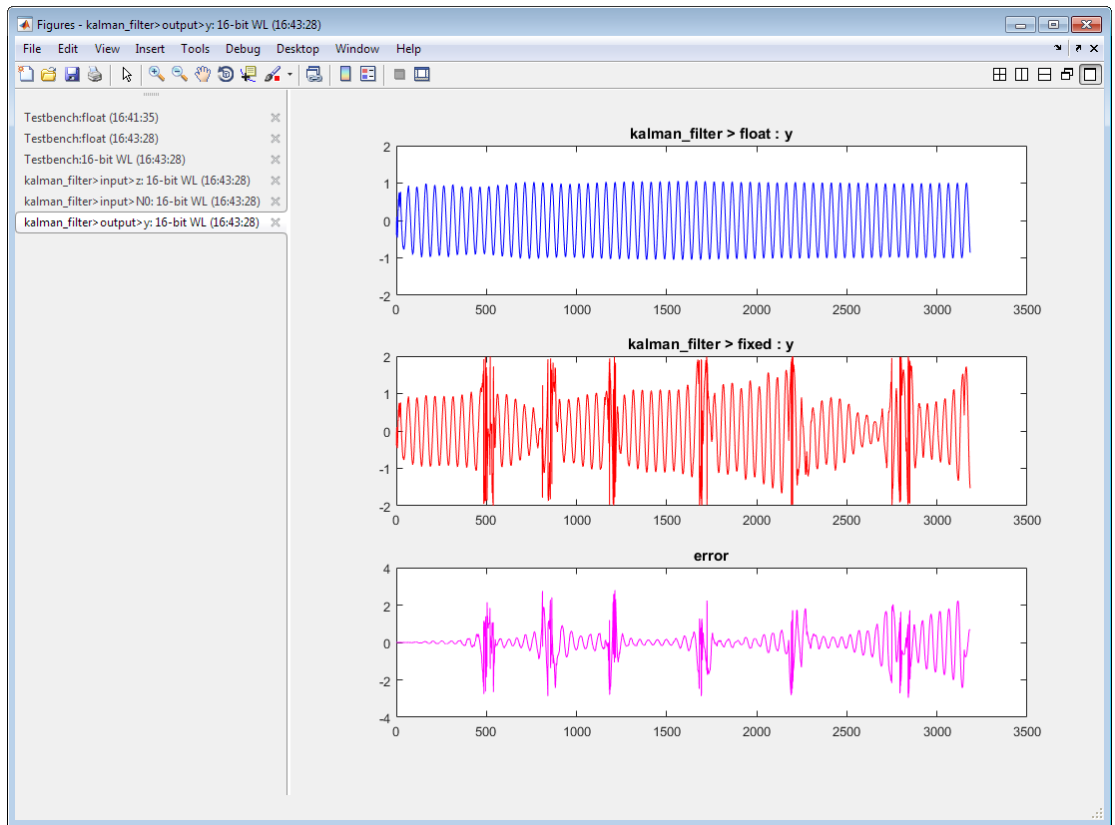


- 4 Click **Next**. On the **Convert to Fixed Point** page, click **Simulate** to gather range information and data type proposals using the default settings.
- 5 Click **Convert** to apply the proposed data types.
- 6 Click the **Test** arrow and select the **Log inputs and outputs for comparison plots** check box. Click **Test**. The Fixed-Point Converter runs the test file `kalman_filter_tb.m` to test the generated fixed-point code. Floating-point and fixed-point simulations are run, with errors calculated for the input and output variables.

The generated plots show that the current fixed-point implementation does not produce good results.



The error for the output variable y is extremely high, at over 282 percent.



Determine Where Numerical Issues Originated

Add any variables that you suspect are the cause of numerical issues to the output arguments of your selected function. Adding these variables to the output arguments enables you to log the signals for comparison plots.

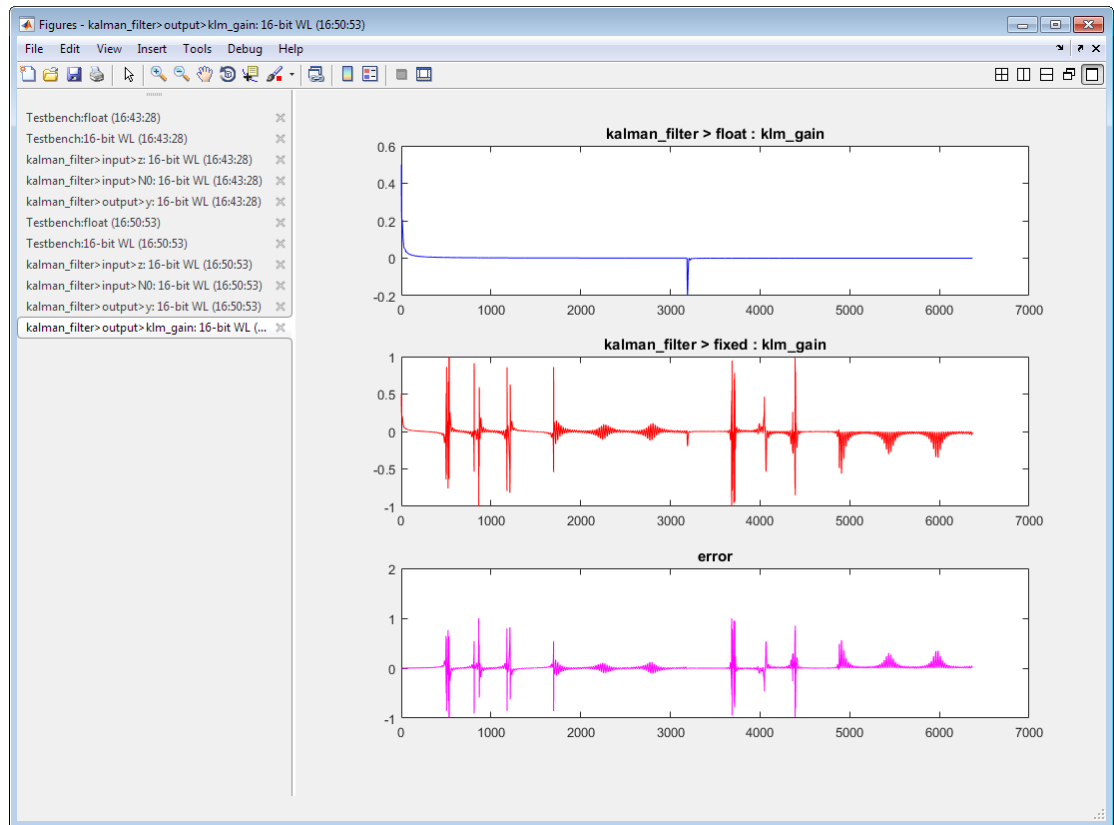
- 1 Click `kalman_filter` in the **Source Code** pane to return to the floating-point code.

The `kalman_filter` function calls the `matrix_solve` function, which contains calls to several other functions. To investigate whether numerical issues are originating in the `matrix_solve` function, within the editor, add `klm_gain`,

the output variable of the `matrix_solve` function, to the output arguments of `kalman_filter`.

- 2 Click **Simulate** to refresh the results, and then click **Convert**.
- 3 Click **Test** and log all inputs and outputs. Because `klm_gain` is now an output argument, the Fixed-Point Converter logs the signal and plots the error of the `matrix_solve` function.

The generated plot shows a large error for the `klm_gain` variable. Investigate the `matrix_solve` function further.



- 4 Close the Fixed-Point Converter and plots and start a new project. On the **Select Source Files** page, browse to the `matrix_solve` file and click **Open**.

- 5 On the **Define Input Types** page, enter the `kalman_filter_tb` test file and click **Autodefine Input Types**.
- 6 On the **Convert to Fixed Point** page, add `l`, `u`, and `y`, the outputs of the other functions called by `matrix_solve`, to the output arguments of the `matrix_solve` function to enable plotting during the **Test** step.

```
function [x, l, u, y] = matrix_solve(a,b)
```

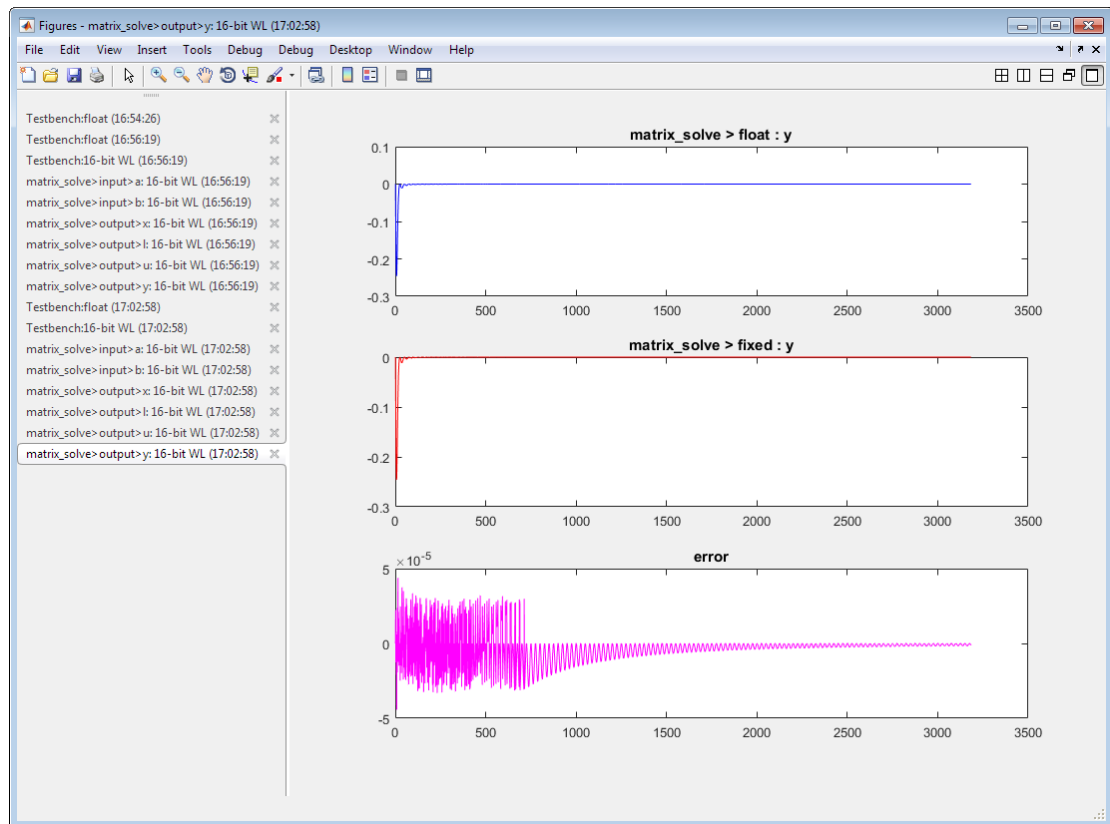
- 7 Click **Simulate** to gather data type proposals, and then click **Convert**.
- 8 Click the arrow next to **Test** and ensure that **Log inputs and outputs for comparison plots** is selected. Click **Test**.

Examine the output plots. Several outputs of the `matrix_solve` function have a very high error. Adjust some of the project settings to address this error.

Adjust `fimath` Settings

Adjust the rounding method and the overflow action used in the generated fixed-point code by changing the `fimath` settings.

- 1 In the same project, click **Settings**.
- 2 Under **fimath**, set the **Rounding method** to **Nearest**. Set the **Overflow action** to **Saturate**. Click **Convert** to apply the new `fimath` settings.
- 3 Click **Test**. Using the new `fimath` settings, the error is now on the order of 10^{-5} .



After you determine the `fimath` settings that address the issues in the `matrix_solve` function, apply these settings when converting the `kalman_filter` function.

- 4 Close the Fixed Point Converter and the plot windows.

Convert `kalman_filter` Function Using New `fimath` Settings

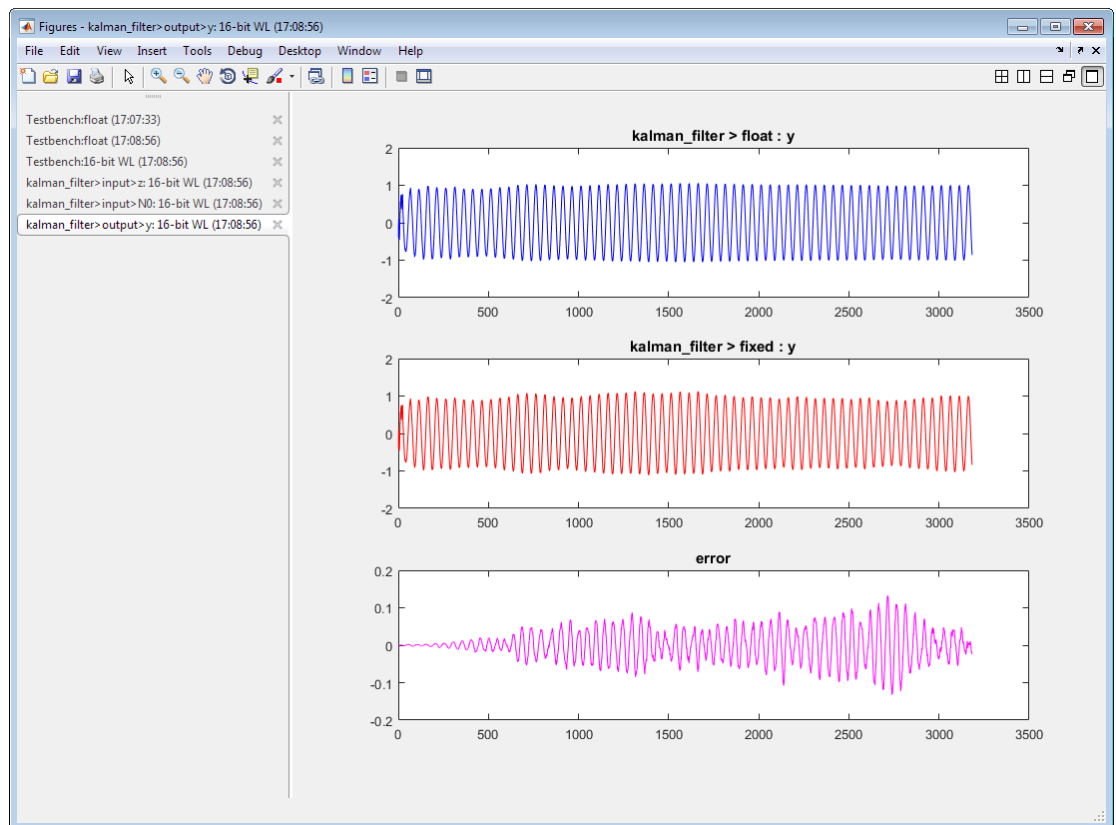
- 1 Remove the additional variables that you added to the output arguments of the `kalman_filter` and `matrix_solve` functions. Save the files.
- 2 Open the Fixed-Point Converter.
- 3 On the **Select Source Files** page, specify `kalman_filter` as the entry-point file.

- 4 On the **Define Input Types** page, add `kalman_filter_tb` as the test file, and click **Autodefine Input Types**.
- 5 On the **Convert to Fixed Point** page, click **Settings**.

Use the same settings that you used when converting the `matrix_solve` function. Under **fimath**, set the **Rounding method** to **Nearest**. Set the **Overflow action** to **Saturate**.

- 6 Click **Simulate** to gather data type proposals, and then click **Convert** to apply the proposals.
- 7 Click the arrow next to **Test** and ensure that **Log inputs and outputs for comparison plots** is selected. Click **Test**.

Examine the plot for output variable, `y`.

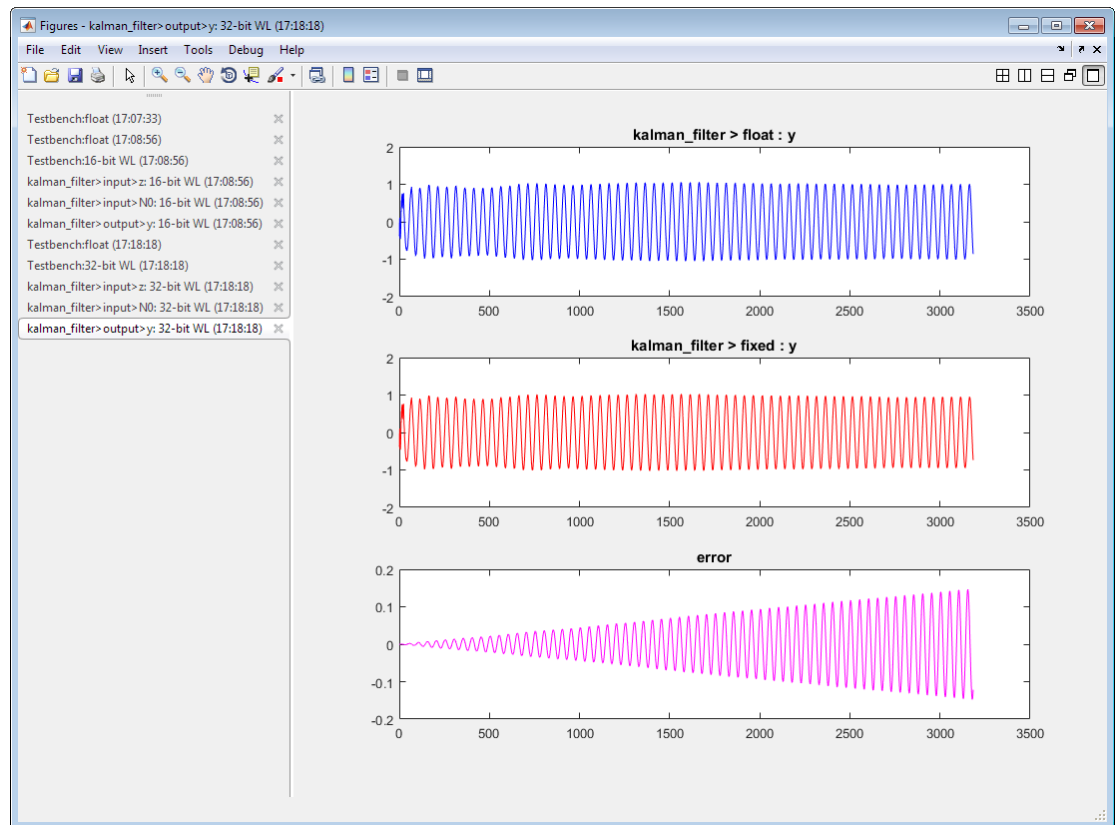


The new `fimath` settings improve the output, but some error still remains.

Adjust Word Length Settings

Adjusting the default word length improves the accuracy of the generated fixed-point design.

- 1 Click **Settings** and change the default word length to 32. Click **Convert** to apply the new settings.
- 2 Click **Test**. The error for the output variable `y` is accumulating.



- 3 Close the Fixed-Point Converter and plot window.

Replace Constant Functions

The `kalman_stm` function computes the state transition matrix, which is a constant. You do not need to convert this function to fixed point. To avoid unnecessary quantization through computation, replace this function with a double-precision constant. By replacing the function with a constant, the state transition matrix undergoes quantization only once.

- 1 Edit the `kalman_filter` function. Replace the call to the `kalman_stm` function with the equivalent double constant.

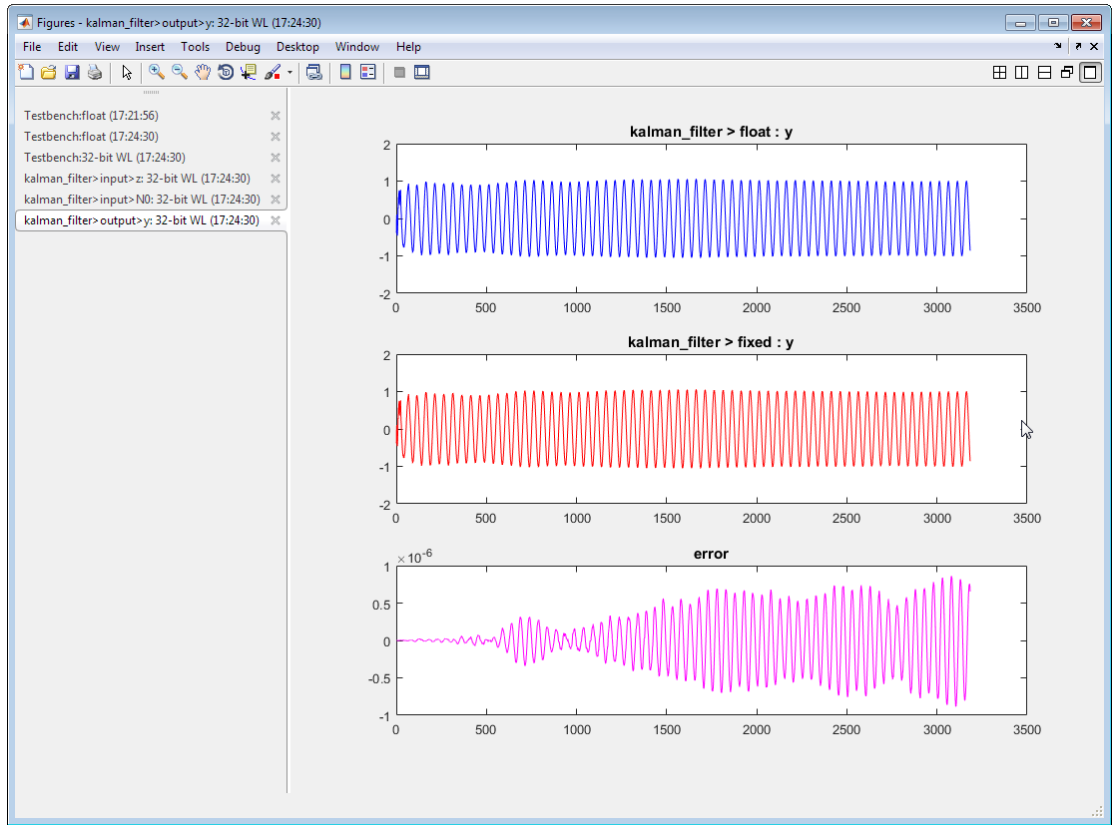
```
A = [0.992114701314478, -0.125333233564304; ...  
     0.125333233564304, 0.992114701314478];
```

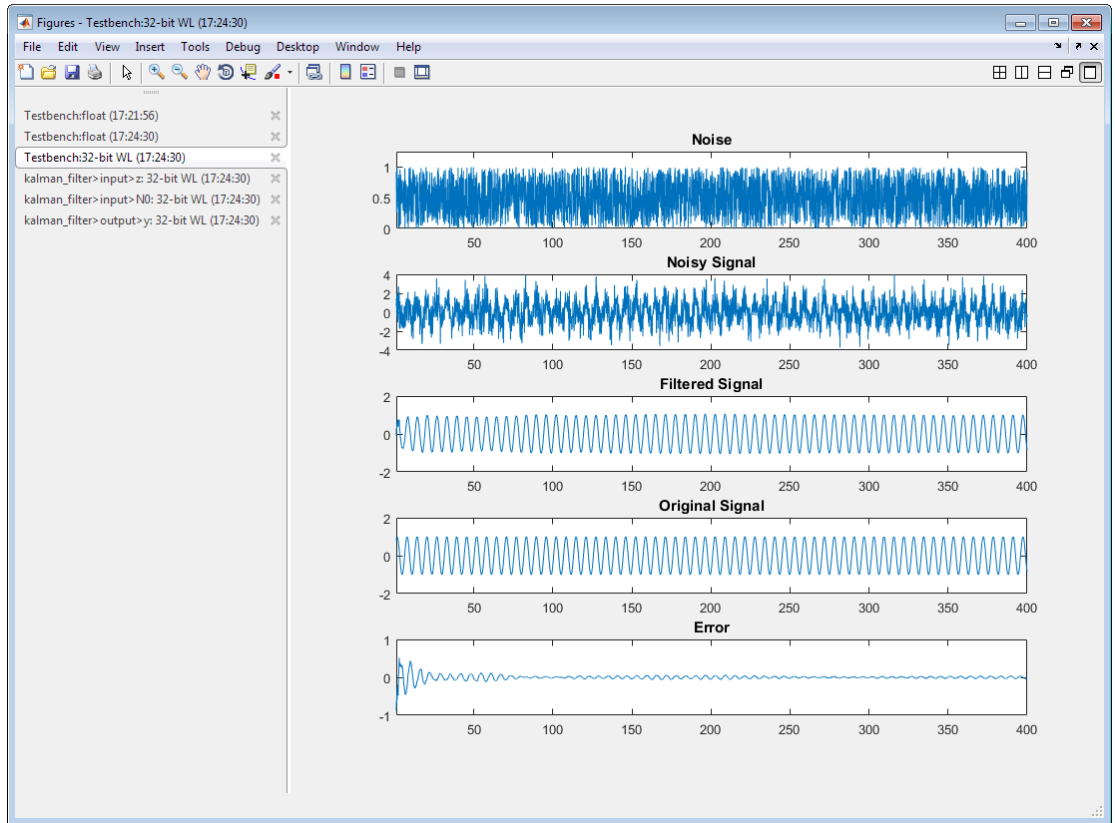
- 2 Open the Fixed-Point Converter app. On the **Select Source Files** page, select `kalman_filter`.
- 3 On the **Define Input Types** page, select `kalman_filter_tb` as the test file.
- 4 On the **Convert to Fixed Point** page, click **Settings**.

Use the same settings as before. Under **fimath**, set the **Rounding method** to **Nearest**. Set the **Overflow action** to **Saturate**. Set the default word length to **32**.

- 5 Click **Simulate**, and then click **Convert**.
- 6 Click the **Test** arrow and select the **Log inputs and outputs for comparison plots** check box. Click **Test**.

The error on the plot for the functions output `y` is now on the order of 10^{-6} , which is acceptable for this design.





MATLAB Language Features Supported for Automated Fixed-Point Conversion

In this section...
“MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 8-42
“MATLAB Language Features Not Supported for Automated Fixed-Point Conversion” on page 8-44

MATLAB Language Features Supported for Automated Fixed-Point Conversion

Fixed-Point Designer supports the following MATLAB language features in automated fixed-point conversion:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Generate Code for Variable-Size Data” (MATLAB Coder)). Range computation for variable-sized data is supported via simulation mode only. Variable-sized data is not supported for comparison plotting.
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 27-30)
- Complex numbers (see “Code Generation for Complex Data” on page 18-4)
- Numeric classes (see “Supported Variable Types” on page 20-16)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB” on page 14-3)
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Global variables
- Persistent variables
- Structures, including arrays of structures. Range computation for structures is supported via simulation mode only.

- Characters

The complete set of Unicode® characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to convert your MATLAB algorithm to fixed point.

- MATLAB classes. Range computation for MATLAB classes is supported via simulation mode only.

Automated conversion supports:

- Class properties
- Constructors
- Methods
- Specializations

It does not support class inheritance or packages. For more information, see “Fixed-Point Code for MATLAB Classes” on page 8-51.

- Ability to call functions (see “Resolution of Function Calls for Code Generation” on page 16-2)
- Subset of MATLAB toolbox functions (see “Functions Supported for Code Acceleration or C Code Generation” on page 14-5).
- Subset of DSP System Toolbox™ System objects.

The DSP System Toolbox System objects supported for automated conversion are:

- `dsp.ArrayVectorAdder`
- `dsp.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.LUFactor`
- `dsp.UpperTriangularSolver`

- `dsp.VariableFractionalDelay`
- `dsp.Window`

MATLAB Language Features Not Supported for Automated Fixed-Point Conversion

Fixed-Point Designer does not support the following features in automated fixed-point conversion:

- Anonymous functions
- Cell arrays
- Function handles
- Java[®]
- Nested functions
- Recursion
- Sparse matrices
- `try/catch` statements
- `varargin` and `varargout`

Generated Fixed-Point Code

In this section...

“Location of Generated Fixed-Point Files” on page 8-45

“Minimizing `fi`-casts to Improve Code Readability” on page 8-46

“Avoiding Overflows in the Generated Fixed-Point Code” on page 8-46

“Controlling Bit Growth” on page 8-47

“Avoiding Loss of Range or Precision” on page 8-47

“Handling Non-Constant `mpower` Exponents” on page 8-49

Location of Generated Fixed-Point Files

By default, the fixed-point conversion process generates files in a folder named `codegen/fcn_name/fixpt` in your local working folder. `fcn_name` is the name of the MATLAB function that you are converting to fixed point.

File name	Description
<code>fcn_name_fixpt.m</code>	Generated fixed-point MATLAB code. To integrate this fixed-point code into a larger application, consider generating a MEX-function for the function and calling this MEX-function in place of the original MATLAB code.
<code>fcn_name_fixpt_exVal.mat</code>	MAT-file containing: <ul style="list-style-type: none"> • A structure for the input arguments. • The name of the fixed-point file.
<code>fcn_name_fixpt_report.html</code>	Link to the type proposal report that displays the generated fixed-point code and the proposed type information.
<code>fcn_name_report.html</code>	Link to the type proposal report that displays the original MATLAB code and the proposed type information.

File name	Description
<code>fcn_name_wrapper_fixpt.m</code>	File that converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during the conversion step. These fixed-point values are fed into the converted fixed-point function, <code>fcn_name_fixpt</code> .

Minimizing `fi`-casts to Improve Code Readability

The conversion process tries to reduce the number of `fi`-casts by analyzing the floating-point code. If an arithmetic operation is comprised of only compile-time constants, the conversion process does not cast the operands to fixed point individually. Instead, it casts the entire expression to fixed point.

For example, here is the fixed-point code generated for the constant expression $x = 1/\sqrt{2}$ when the selected word length is 14.

Original MATLAB Code	Generated Fixed-Point Code
<code>x = 1/sqrt(2);</code>	<code>x = fi(1/sqrt(2), 0, 14, 14, fm);</code> <code>fm</code> is the local <code>fi</code> math.

Avoiding Overflows in the Generated Fixed-Point Code

The conversion process avoids overflows by:

- Using full-precision arithmetic unless you specify otherwise.
- Avoiding arithmetic operations that involve double and `fi` data types. Otherwise, if the word length of the `fi` data type is not able to represent the value in the double constant expression, overflows occur.
- Avoiding overflows when adding and subtracting non fixed-point variables and fixed-point variables.

The fixed-point conversion process casts non-`fi` expressions to the corresponding `fi` type.

For example, consider the following MATLAB algorithm.


```

% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi(A, B)
    % '1024' is non-fi, cast it
    y = A + 1024;
    % 'size(B, 1)*length(A)' is a non-fi, cast it
    y = A + size(B, 1)*length(A);
end

```

The generated fixed-point code is:

```

%#codegen
% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi_fixpt(A, B)
    % '1024' is non-fi, cast it
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
                'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
                'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(A + fi(1024, 0, 11, 0, fm), 0, 11, 0, fm);
    % 'size(B, 1)*length(A)' is a non-fi, cast it
    y(:) = A + fi(size(B, 1)*length(A), 0, 9, 0, fm);
end

```

Controlling Bit Growth

The conversion process controls bit growth by using subscripted assignments, that is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. Using subscripted assignment keeps fixed-point variables fixed point rather than inadvertently turning them into doubles. Maintaining the fixed-point type reduces the number of type declarations in the generated code. Subscripted assignment also prevents bit growth which is useful when you want to maintain a particular data type for the output.

Avoiding Loss of Range or Precision

Avoiding Loss of Range or Precision in Unsigned Subtraction Operations

When the result of the subtraction is negative, the conversion process promotes the left operand to a signed type.

For example, consider the following MATLAB algorithm.

```
% A = 1;  
% B = 5  
function [y,z] = unsigned_subtraction(A,B)  
    y = A - B;  
  
    C = -20;  
    z = C - B;  
end
```

In the original code, both *A* and *B* are unsigned and the result of *A-B* can be negative. In the generated fixed-point code, *A* is promoted to signed. In the original code, *C* is signed, so does not require promotion in the generated code.

```
 %#codegen  
% A = 1;  
% B = 5  
function [y,z] = unsigned_subtraction_fixpt(A,B)  
  
fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...  
           'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...  
           'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
y = fi(fi_signed(A) - B, 1, 3, 0, fm);  
C = fi(-20, 1, 6, 0, fm);  
z = fi(C - B, 1, 6, 0, fm);  
end  
  
function y = fi_signed(a)  
coder.inline( 'always' );  
if isfi( a ) && ~(issigned( a ))  
    nt = numerictype( a );  
    new_nt = numerictype( 1, nt.WordLength + 1, nt.FractionLength );  
    y = fi( a, new_nt, fimath( a ) );  
else  
    y = a;  
end  
end
```

Avoiding Loss of Range When Concatenating Arrays of Fixed-Point Numbers

If you concatenate matrices using `vertcat` and `horzcat`, the conversion process uses the largest `numerictype` among the expressions of a row and casts the leftmost element to that type. This type is then used for the concatenated matrix to avoid loss of range.

For example, consider the following MATLAB algorithm.

```
% A = 1, B = 100, C = 1000
function [y, z] = lb_node(A, B, C)
    %% single rows
    y = [A B C];
    %% multiple rows
    z = [A 5; A B; A C];
end
```

In the generated fixed-point code:

- For the expression $y = [A \ B \ C]$, the leftmost element, A , is cast to the type of C because C has the largest type in the row.
- For the expression $[A \ 5; \ A \ B; \ A \ C]$:
 - In the first row, A is cast to the type of C because C has the largest type of the whole expression.
 - In the second row, A is cast to the type of B because B has the larger type in the row.
 - In the third row, A is cast to the type of C because C has the larger type in the row.

```
##codegen
% A = 1, B = 100, C = 1000
function [y, z] = lb_node_fixpt(A, B, C)
    %% single rows
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', ...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi([fi(A, 0, 10, 0, fm) B C], 0, 10, 0, fm);

    %% multiple rows
    z = fi([fi(A, 0, 10, 0, fm) 5; fi(A, 0, 7, 0, fm) B; ...
           fi(A, 0, 10, 0, fm) C], 0, 10, 0, fm);
end
```

Handling Non-Constant mpower Exponents

If the function that you are converting has a scalar input, and the `mpower` exponent input is not constant, the conversion process sets the `fimath ProductMode` to

SpecifyPrecision in the generated code. With this setting, the output data type can be determined at compile time.

For example, consider the following MATLAB algorithm.

```
% a = 1
% b = 3
function y = exp_operator(a, b)
    % exponent is a constant so no need to specify precision
    y = a^3;
    % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
    y = b^a;
end
```

In the generated fixed-point code, for the expression $y = a^3$, the exponent is a constant, so there is no need to specify precision. For the expression $y = b^a$, the exponent is not constant, so the ProductMode is set to SpecifyPrecision.

```
##codegen
% a = 1
% b = 3
function y = exp_operator_fixpt(a, b)
    % exponent is a constant so no need to specify precision
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(a^3, 0, 2, 0, fm);
    % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
    y(:) = fi(b, 'ProductMode', 'SpecifyPrecision',...
              'ProductWordLength', 2, 'ProductFractionLength', 0)^a;
end
```

Fixed-Point Code for MATLAB Classes

In this section...

“Automated Conversion Support for MATLAB Classes” on page 8-51

“Unsupported Constructs” on page 8-51

“Coding Style Best Practices” on page 8-52

Automated Conversion Support for MATLAB Classes

The automated fixed-point conversion process:

- Proposes fixed-point data types based on simulation ranges for MATLAB classes. It does not propose data types based on derived ranges for MATLAB classes.

After simulation, the Fixed-Point Converter app:

- Function list contains class constructors, methods, and specializations.
- Code window displays the objects used in each function.
- Provides code coverage for methods.

For more information, see “Viewing Information for MATLAB Classes” on page 8-19.

- Supports class methods, properties, and specializations. For each specialization of a class, `class_name`, the conversion generates a separate `class_name_fixpt.m` file. For every instantiation of a class, the generated fixed-point code contains a call to the constructor of the appropriate specialization.
- Supports classes that have `get` and `set` methods such as `get.PropertyName`, `set.PropertyName`. These methods are called when properties are read or assigned. The `set` methods can be specialized. Sometimes, in the generated fixed-point code, assignment statements are transformed to function calls.

Unsupported Constructs

The automated conversion process does not support:

- Class inheritance.
- Packages.
- Constructors that use `nargin` and `varargin`.

Coding Style Best Practices

When you write MATLAB code that uses MATLAB classes:

- Initialize properties in the class constructor.
- Replace constant properties with static methods.

For example, consider the `counter` class.

```
classdef Counter < handle
    properties
        Value = 0;
    end

    properties(Constant)
        MAX_VALUE = 128
    end

    methods
        function out = next(this)
            out = this.Count;
            if this.Value == this.MAX_VALUE
                this.Value = 0;
            else
                this.Value = this.Value + 1;
            end
        end
    end
end
```

To use the automated fixed-point conversion process, rewrite the class to have a static class that initializes the constant property `MAX_VALUE` and a constructor that initializes the property `Value`.

```
classdef Counter < handle
    properties
        Value;
    end

    methods(Static)
        function t = MAX_VALUE()
            t = 128;
        end
    end
end
```

```
methods
function this = Counter()
    this.Value = 0;
end
function out = next(this)
    out = this.Value;
    if this.Value == this.MAX_VALUE
        this.Value = 0;
    else
        this.Value = this.Value + 1;
    end
end
end
end
```

Automated Fixed-Point Conversion Best Practices

In this section...

“Create a Test File” on page 8-54

“Prepare Your Algorithm for Code Acceleration or Code Generation” on page 8-55

“Check for Fixed-Point Support for Functions Used in Your Algorithm” on page 8-56

“Manage Data Types and Control Bit Growth” on page 8-56

“Convert to Fixed Point” on page 8-57

“Use the Histogram to Fine-Tune Data Type Settings” on page 8-58

“Optimize Your Algorithm” on page 8-59

“Avoid Explicit Double and Single Casts” on page 8-61

Create a Test File

A best practice for structuring your code is to separate your core algorithm from other code that you use to test and verify the results. Create a test file to call your original MATLAB algorithm and fixed-point versions of the algorithm. For example, as shown in the following table, you might set up some input data to feed into your algorithm, and then, after you process that data, create some plots to verify the results. Since you need to convert only the algorithmic portion to fixed point, it is more efficient to structure your code so that you have a test file, in which you create your inputs, call your algorithm, and plot the results, and one (or more) algorithmic files, in which you do the core processing.

Original code	Best Practice	Modified code
<pre>% TEST INPUT x = randn(100,1); % ALGORITHM y = zeros(size(x)); y(1) = x(1); for n=2:length(x) y(n)=y(n-1) + x(n); end % VERIFY RESULTS yExpected=cumsum(x); plot(y-yExpected)</pre>	<p>Issue</p> <p>Generation of test input and verification of results are intermingled with the algorithm code.</p> <p>Fix</p> <p>Create a test file that is separate from your algorithm.</p>	<p>Test file</p> <pre>% TEST INPUT x = randn(100,1); % ALGORITHM y = cumulative_sum(x); % VERIFY RESULTS yExpected = cumsum(x); plot(y-yExpected) title('Error')</pre>

Original code	Best Practice	Modified code
<code>title('Error')</code>	Put the algorithm in its own function.	Algorithm in its own function <pre>function y = cumulative_sum(x) y = zeros(size(x)); y(1) = x(1); for n=2:length(x) y(n) = y(n-1) + x(n); end end</pre>

You can use the test file to:

- Verify that your floating-point algorithm behaves as you expect before you convert it to fixed point. The floating-point algorithm behavior is the baseline against which you compare the behavior of the fixed-point versions of your algorithm.
- Propose fixed-point data types.
- Compare the behavior of the fixed-point versions of your algorithm to the floating-point baseline.
- Help you determine initial values for static ranges.

By default, the Fixed-Point Converter app shows code coverage results. Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. For example, for a filter, realistic inputs are impulses, sums of sinusoids, and chirp signals. With these inputs, using linear theory, you can verify that the outputs are correct. Signals that produce maximum output are useful for verifying that your system does not overflow. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results help you verify that your test file is exercising the algorithm adequately. Review code flagged with a red code coverage bar because this code is not executed. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. See “Code Coverage” on page 8-5.

Prepare Your Algorithm for Code Acceleration or Code Generation

The automated conversion process instruments your code and provides data type proposals to help you convert your algorithm to fixed point.

MATLAB algorithms that you want to convert to fixed point automatically must comply with code generation requirements and rules. To view the subset of the MATLAB

language that is supported for code generation, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%#codegen` pragma to the top of your MATLAB file. The MATLAB Code Analyzer flags functions and constructs that are not available in the subset of the MATLAB language supported for code generation. This advice appears in real time as you edit your code in the MATLAB editor. For more information, see “Check Code Using the MATLAB Code Analyzer” on page 14-103. The software provides a link to a report that identifies calls to functions and the use of data types that are not supported for code generation. For more information, see “Check Code Using the Code Generation Readiness Tool” on page 14-102.

Check for Fixed-Point Support for Functions Used in Your Algorithm

The app flags unsupported function calls found in your algorithm on the **Function Replacements** tab. For example, if you use the `fft` function, which is not supported for fixed point, the tool adds an entry to the table on this tab and indicates that you need to specify a replacement function to use for fixed-point operations.

Function Replacements	
Enter a function to replace	
Function or Operator	Replacement
<ul style="list-style-type: none"> Custom Function 	Function Name
fft	Replacement required to use fixed-point

You can specify additional replacement functions. For example, functions like `sin`, `cos`, and `sqrt` might support fixed point, but for better efficiency, you might want to consider an alternative implementation like a lookup table or CORDIC-based algorithm. The app provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. See “Replacing Functions Using Lookup Table Approximations” on page 8-62.

Manage Data Types and Control Bit Growth

The automated fixed-point conversion process automatically manages data types and controls bit growth. It controls bit growth by using subscripted assignments, that

is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. In addition to preventing bit growth, subscripted assignment reduces the number of casts in the generated fixed-point code and makes the code more readable.

Convert to Fixed Point

What Are Your Goals for Converting to Fixed Point?

Before you start the conversion, consider your goals for converting to fixed point. Are you implementing your algorithm in C or HDL? What are your target constraints? The answers to these questions determine many fixed-point properties such as the available word length, fraction length, and math modes, as well as available math libraries.

To set up these properties, use the **Advanced** settings.


Setting	Value
Basic	
Fixed-point type proposal mode	Propose fraction lengths for specified word length
Default word length	16
Default fraction length	4
Advanced	
When proposing types	use all collected data
Propose target container types	No
Optimize whole numbers	Yes
Signedness	Automatic
Safety margin for sim min/max (%)	0
fimath	
Rounding method	Floor

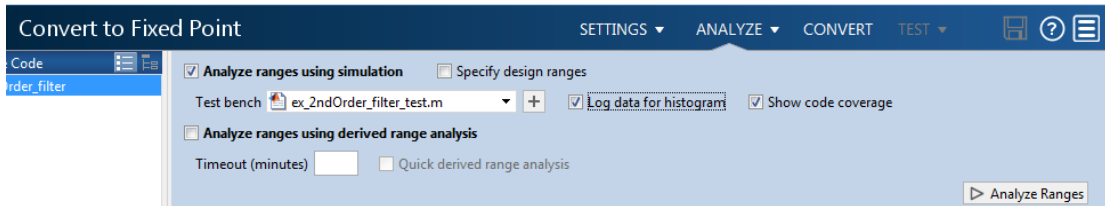
For more information, see “Specify Type Proposal Options” on page 9-2.

Run With Fixed-Point Types and Compare Results

Create a test file to validate that the floating-point algorithm works as expected before converting it to fixed point. You can use the same test file to propose fixed-point data types, and to compare fixed-point results to the floating-point baseline after the conversion. For more information, see “Running a Simulation” on page 8-9 and “Log Data for Histogram” on page 8-20 .

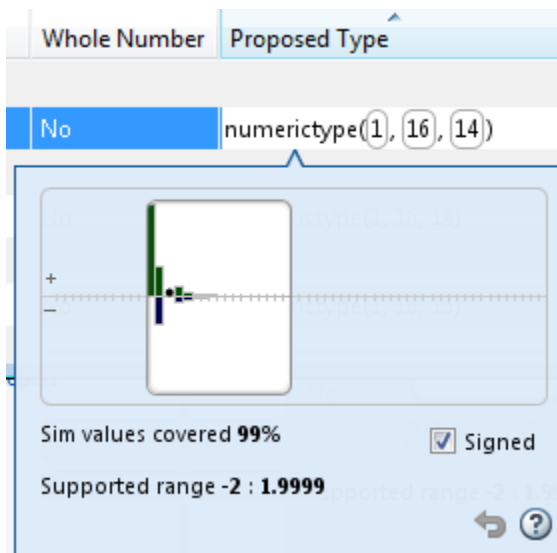
Use the Histogram to Fine-Tune Data Type Settings

To fine-tune fixed-point type settings, use the histogram. To log data for histograms, in the app, click the **Analyze** arrow  and select **Log data for histogram**.



After simulation and static analysis:

- To view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.



You can view the effect of changing the proposed data types by dragging the edges of the bounding box in the histogram window to change the proposed data type and selecting or clearing the **Signed** option.

- If the values overflow and the range cannot fit the proposed type, the table shows proposed types in red.

When the tool applies data types, it generates an html report that provides overflow information and highlights overflows in red. Review the proposed data types.

Optimize Your Algorithm

Use `fimath` to Get Optimal Types for C or HDL

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. You can use the `fimath ProductMode` and `SumMode` properties to retain optimal data types for C or HDL. HDL can have arbitrary word length types in the generated HDL code whereas C requires container types (`uint8`, `uint16`, `uint32`). Use the **Advanced** settings, see “Specify Type Proposal Options” on page 9-2.

C

The `KeepLSB` setting for `ProductMode` and `SumMode` models the behavior of integer operations in the C language, while `KeepMSB` models the behavior of many DSP devices. Different rounding methods require different amounts of overhead code. Setting the `RoundingMethod` property to `Floor`, which is equivalent to two's complement truncation, provides the most efficient rounding implementation. Similarly, the standard method for handling overflows is to wrap using modulo arithmetic. Other overflow handling methods create costly logic. Whenever possible, set `OverflowAction` to `Wrap`.

MATLAB Code	Best Practice	Generated C Code
Code being compiled <pre>function y = adder(a,b) y = a + b; end</pre>	Issue With the default word length set to 16 and the default <code>fimath</code> settings, additional code is generated to implement saturation overflow, nearest rounding, and full-precision arithmetic.	<pre>int adder(short a, short b) { int y; int i0; int i1; int i2; int i3; i0 = a; i1 = b; if ((i0 & 65536) != 0) { i2 = i0 -65536; } else { i2 = i0 & 65535; } }</pre>
Note: In the app, set Default word length to 16.		

MATLAB Code	Best Practice	Generated C Code												
		<pre> if ((i1 & 65536) != 0) { i3 = i1 -65536; } else { i3 = i1 & 65535; } i0 = i2 + i3; if ((i0 & 65536) != 0) { y = i0 -65536; } else { y = i0 & 65535; } return y; } </pre>												
	<p>Fix</p> <p>To make the generated C code more efficient, choose fixed-point math settings that match your processor types.</p> <p>To customize fixed-point type proposals, use the app Settings. Select fimath and then set:</p> <table border="1" data-bbox="457 1085 967 1381"> <tr> <td>Rounding method</td> <td>Floor</td> </tr> <tr> <td>Overflow action</td> <td>Wrap</td> </tr> <tr> <td>Product mode</td> <td>KeepLSB</td> </tr> <tr> <td>Sum mode</td> <td>KeepLSB</td> </tr> <tr> <td>Product word length</td> <td>32</td> </tr> <tr> <td>Sum word length</td> <td>32</td> </tr> </table>	Rounding method	Floor	Overflow action	Wrap	Product mode	KeepLSB	Sum mode	KeepLSB	Product word length	32	Sum word length	32	<pre> int adder(short a, short b) { return a + b; } </pre>
Rounding method	Floor													
Overflow action	Wrap													
Product mode	KeepLSB													
Sum mode	KeepLSB													
Product word length	32													
Sum word length	32													

HDL

For HDL code generation, set:

- ProductMode and SumMode to FullPrecision

- Overflow action to Wrap
- Rounding method to Floor

Replace Built-in Functions with More Efficient Fixed-Point Implementations

Some MATLAB built-in functions can be made more efficient for fixed-point implementation. For example, you can replace a built-in function with a Lookup table implementation, or a CORDIC implementation, which requires only iterative shift-add operations. For more information, see “Function Replacements” on page 8-22.

Reimplement Division Operations Where Possible

Often, division is not fully supported by hardware and can result in slow processing. When your algorithm requires a division, consider replacing it with one of the following options:

- Use bit shifting when the denominator is a power of two. For example, `bitsra(x,3)` instead of `x/8`.
- Multiply by the inverse when the denominator is constant. For example, `x*0.2` instead of `x/5`.
- If the divisor is not constant, use a temporary variable for the division. Doing so results in a more efficient data type proposal and, if overflows occur, makes it easier to see which expression is overflowing.

Eliminate Floating-Point Variables

For more efficient code, the automated fixed-point conversion process eliminates floating-point variables. The one exception to this is loop indices because they usually become integer types. It is good practice to inspect the fixed-point code after conversion to verify that there are no floating-point variables in the generated fixed-point code.

Avoid Explicit Double and Single Casts

For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts.

Instead of using casts, supply a replacement function. For more information, see “Function Replacements” on page 8-22.

Replacing Functions Using Lookup Table Approximations

The Fixed-Point Designer software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations in the Fixed-Point Converter app, see “Replace the exp Function with a Lookup Table” on page 9-47 and “Replace a Custom Function with a Lookup Table” on page 9-57.

To use lookup table approximations in the programmatic workflow, see `coder.approximation`, “Replace the exp Function with a Lookup Table” on page 10-19, and “Replace a Custom Function with a Lookup Table” on page 10-21.

Custom Plot Functions

The Fixed-Point Converter app provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Converter app facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations.

Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.

Use this information to:

- Customize plot headings and axes.
- Choose which variables to plot.
- Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.

- A cell array to hold the logged values for the variable after fixed-point conversion.

This cell array contains values observed during fixed-point simulation of the converted design.

For example, function `customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Converter app, select **Advanced**, and then set **Custom plot function** to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 9-65.

In the programmatic workflow, set the `coder.FixptConfig` configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 10-23.

Generate Fixed-Point MATLAB Code for Multiple Entry-Point Functions

When your end goal is to generate fixed-point C/C++ library functions, generating a single C/C++ library for more than one entry-point MATLAB function allows you to:

- Create C/C++ libraries containing multiple, compiled MATLAB files to integrate with larger C/C++ applications. Generating C/C++ code requires a MATLAB Coder license.
- Share code efficiently between library functions.
- Communicate between library functions using shared memory.

Note: If any of the entry-point functions in a project share memory (for example, persistent data), an error will occur. In this case, you should rewrite your code to avoid invoking functions with persistent data from multiple entry-points.

Convert Two Entry-Point Functions to Fixed-Point Using the Fixed-Point Converter App

In this example, you convert two entry-point functions, `ep1` and `ep2`, to fixed point.

- 1 In a local writable folder, create the functions `ep1.m` and `ep2.m`.

```
function y = ep1(u) %#codegen
y = u;
end

function y = ep2(u, v) %#codegen
y = u + v;
end
```

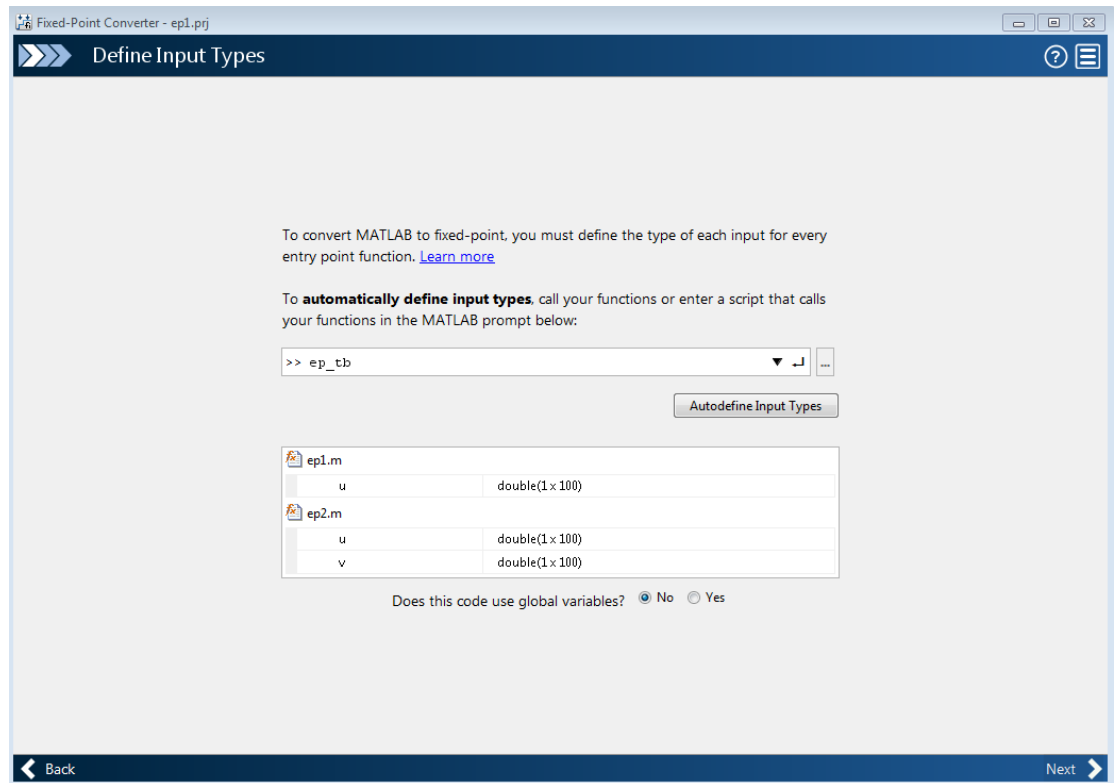
- 2 In the same folder, create a test file, `ep_tb.m`, that calls both functions.

```
% test file for ep1 and ep2
u = 1:100;
v = 5:104;
z = ep1(u);
y = ep2(v, z);
```

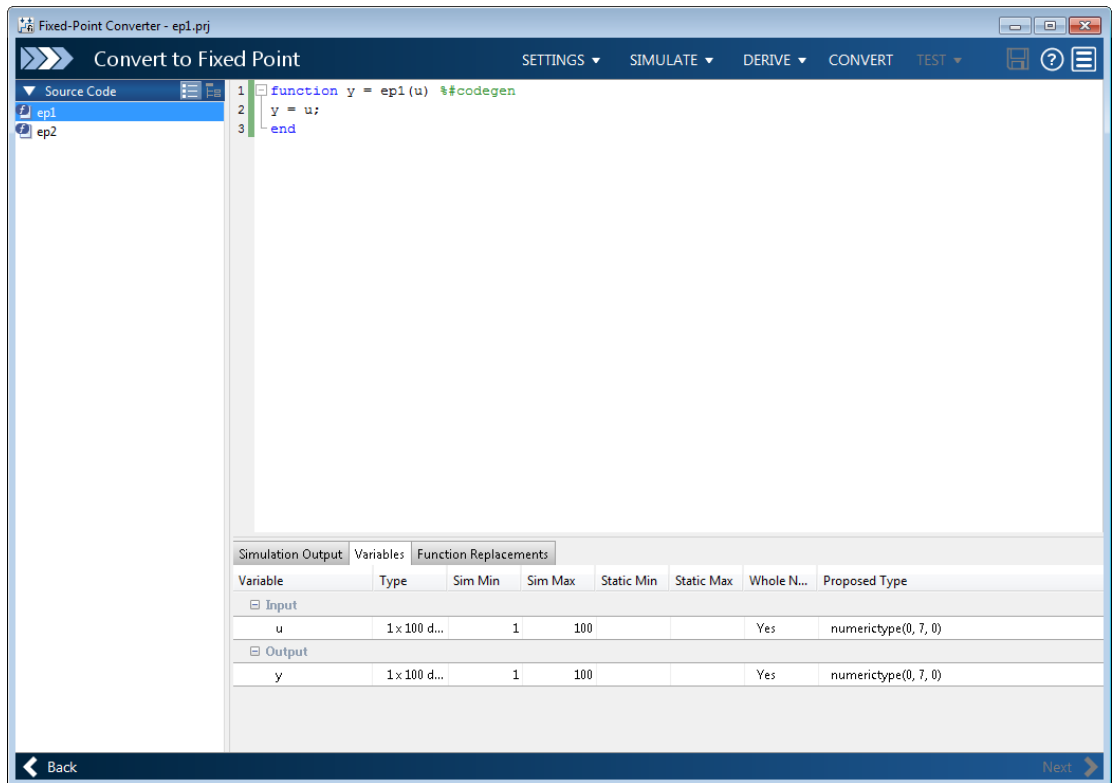
- 3 From the apps gallery, open the Fixed-Point Converter app.
- 4 To add the first entry-point function, `ep1`, to the project, on the **Select Source Files** page, browse to the `ep1` file, and click **Open**.

By default, the app uses the name of the first entry-point function as the name of the project.

- 5 Click **Add Entry-Point Function** and add the second entry-point function, **ep2**. Click **Next**.
- 6 On the **Define Input Types** page, enter a test file that exercises your two entry-point functions. Browse to select the **ep_tb** file. Click **Autodefine Input Types**.

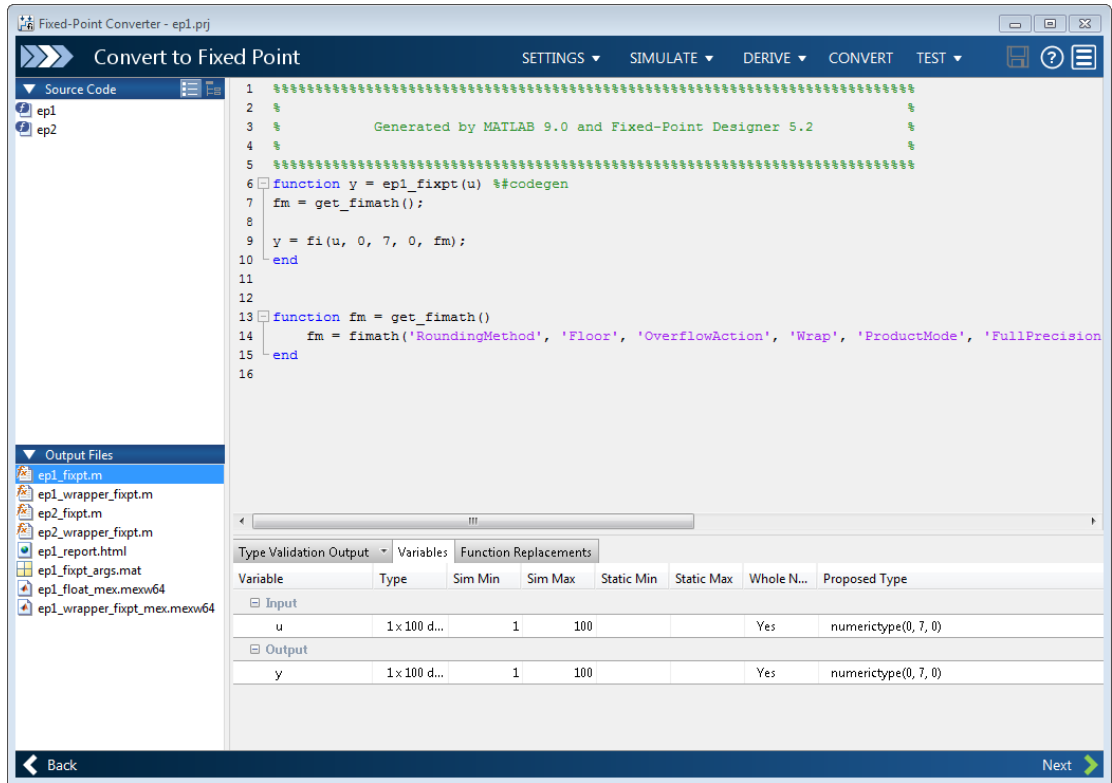


- 7 Click **Next**. The app generates an instrumented MEX function for your entry-point MATLAB function. On the **Convert to Fixed-Point** page, click **Simulate** to simulate the entry-point functions, gather range information, and get proposed data types.



8 Click **Convert**.

The entry-point functions **ep1** and **ep2** convert to fixed point. The **Output Files** pane lists the generated fixed-point and wrapper files for both entry-point functions.



- 9 Click **Next**. The **Finish Workflow** page contains the project summary. The generated Fixed-Point Conversion Report contains the reports for both entry-point functions. The app stores the generated files in the subfolder `codegen/ep1/fixpt`.

Convert Code Containing Global Data to Fixed Point

In this section...

“Workflow” on page 8-69

“Declare Global Variables” on page 8-69

“Define Global Data” on page 8-70

“Define Constant Global Data” on page 8-71

“Limitations of Using Global Data” on page 8-73

Workflow

To convert MATLAB code that uses global data to fixed-point:

- 1 Declare the variables as global in your code.

For more information, see “Declare Global Variables” on page 8-69

- 2 Before using the global data, define and initialize it.

For more information, see “Define Global Data” on page 8-70.

- 3 Convert code to fixed-point from the Fixed-Point Converter or using `fiaccel`.

The Fixed-Point Converter always synchronizes global data between MATLAB and the generated MEX function.

Declare Global Variables

When using global data, you must first declare the global variables in your MATLAB code. This code shows the `use_globals` function, which uses two global variables, `AR` and `B`.

```
function y = use_globals(u)
%#codegen
% Declare AR and B as global variables
global AR;
global B;
AR(1) = u + B(1);
```

```
y = AR * 2;
```

Define Global Data

You can define global data in the MATLAB global workspace, in a Fixed-Point Converter project, or at the command line. If you do not initialize global data in a project or at the command line, the software looks for the variable in the MATLAB global workspace.

Define Global Data in the MATLAB Global Workspace

To convert the `use_globals` function described in “Declare Global Variables” on page 8-69, you must first define and initialize the global data.

```
global AR B;  
AR = ones(4);  
B=[1 2 3];
```

Define Global Data in a Fixed-Point Converter Project

- 1 On the **Define Input Types** page, after selecting and running a test file, select **Yes** next to **Does this code use global variables**.

By default, the Fixed-Point Converter names the first global variable in a project `g`.

- 2 Enter the names of the global variables used in your code. After adding a global variable, specify its type.
- 3 Click **Add global** to enter more global variables.

Note: If you do not specify the type, you must create a variable with the same name in the global workspace.

Define Global Data at the Command Line

To define global data at the command line, use the `fiaccl -globals` option. For example, to convert the `use_globals` function described in “Declare Global Variables” on page 8-69 to fixed-point, specify two global inputs, `AR` and `B`, at the command line. Use the `-args` option to specify that the input `u` is a real, scalar double.

```
fiaccl -float2fixed cfg -global {'AR',ones(4),'B',[1 2 3]} use_globals -args {0}  
Alternatively, specify the type and initial value with the -globals flag using the format -globals {'g', {type, initial_value}}.
```


To provide initial values for variable-size global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
fiaccel -float2fixed cfg -global {'g', {coder.typeof(0, [2 2],1),[1 1]}} myfunction
```

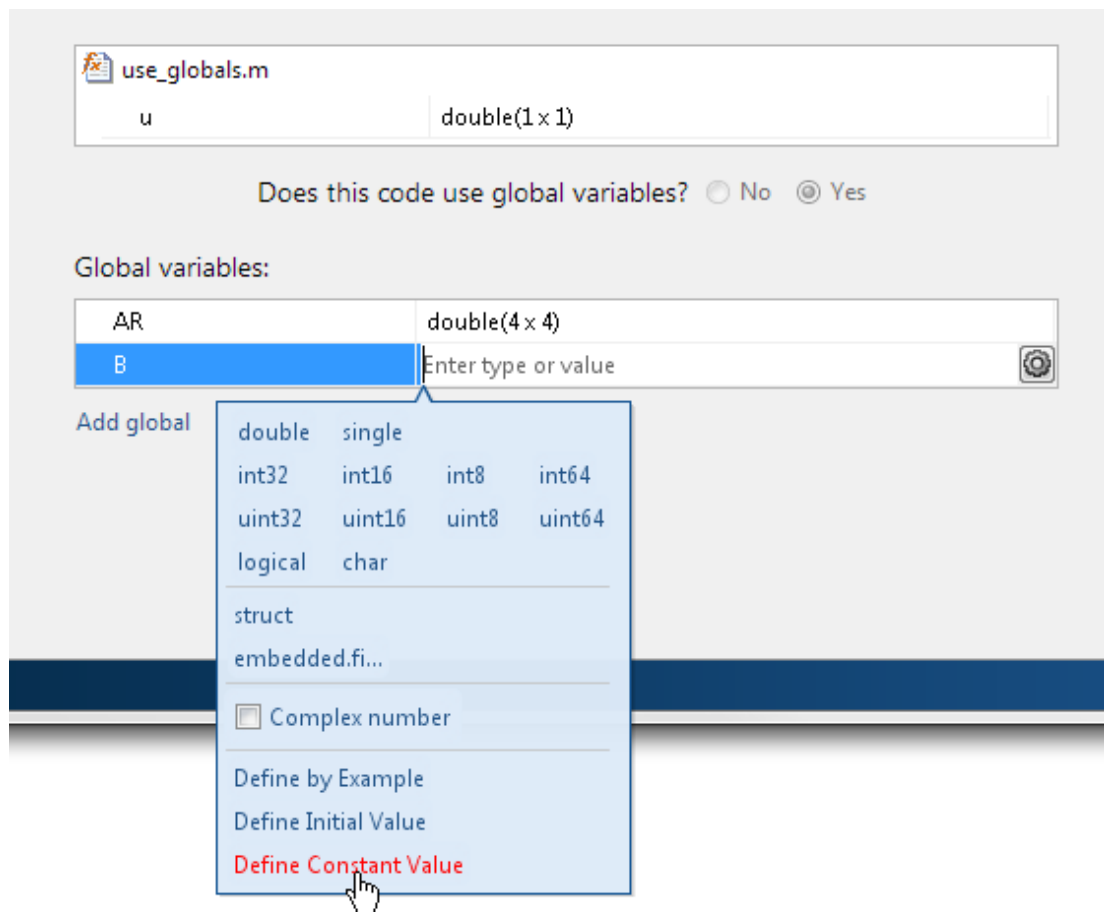
For a detailed explanation of the syntax, see `coder.typeof`.

Define Constant Global Data

If you know that the value of a global variable does not change at run time, you can reduce overhead in the fixed-point code by specifying that the global variable has a constant value. You cannot write to the constant global variable.

Define Constant Global Data in the Fixed-Point Converter

- 1 On the **Define Input Types** page, after selecting and running a test file, select **Yes** next to **Does this code use global variables**.
- 2 Enter the name of the global variables used in your code.
- 3 Click the field to the right of the global variable.
- 4 Select **Define Constant Value**.



- 5 In the field to the right of the constant global variable, enter a MATLAB expression.

Define Constant Global Data at the Command Line

To specify that a global variable is constant using the `fiaccl` command, use the `-globals` option with the `coder.Constant` class.

- 1 Define a fixed-point conversion configuration object.

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

- 2 Use `coder.Constant` to specify that a global variable has a constant value. For example, this code specifies that the global variable `g` has an initial value 4 and that global variable `gc` has the constant value 42.

```
global_values = {'g', 4, 'gc', coder.Constant(42)};
```

- 3 Convert the code to fixed-point using the `-globals` option. For example, convert `myfunction` to fixed-point, specifying that the global variables are defined in the cell array `global_values`.

```
fiaccel -float2fixed cfg -global global_values myfunction
```

Constant Global Data in a Code Generation Report

The code generation report provides this information about a constant global variable:

- Type of `Global` on the **Variables** tab.
- Highlighted variable name in the **Function** pane.

Limitations of Using Global Data

You cannot use global data with the `coder.cstructname` function.

Related Examples

- “Convert Code Containing Global Variables to Fixed-Point” on page 8-74

Convert Code Containing Global Variables to Fixed-Point

This example shows how to convert a MATLAB algorithm containing global variables to fixed point using the Fixed-Point Converter app.

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

```
function y = use_globals(u)
%#codegen
% Declare AR and B as global variables
global AR;
global B;
AR(1) = u + B(1);
y = AR * 2;
```

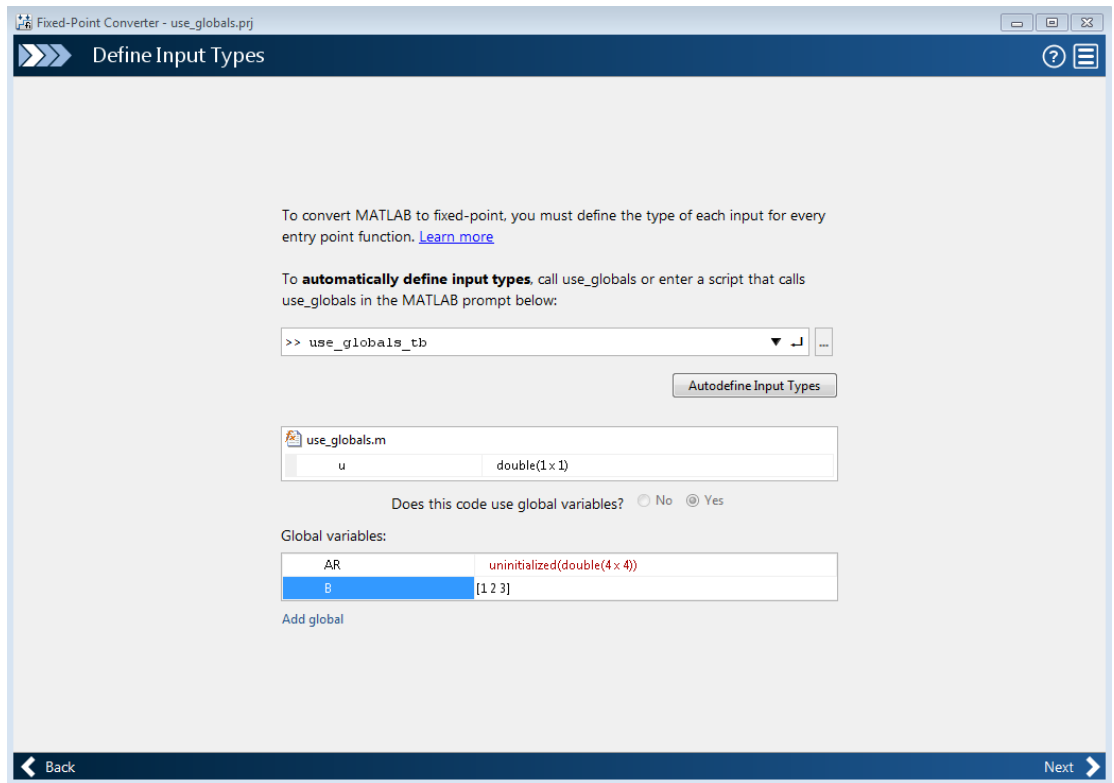
- 2 In the same folder, create a test file, `use_globals_tb.m` that calls the function.

```
u = 55;
global AR B;
AR = ones(4);
B=[1 2 3];
y = use_globals(u);
```

- 3 On the MATLAB toolstrip, in the **Apps** tab, under **Code Generation**, click the Fixed-Point Converter app icon.
- 4 To add the entry-point function, `use_globals.m` to the project, on the **Select Source Files** page, browse to the file, and click **Open**. Click **Next**.
- 5 On the **Define Input Types** page, add `use_globals_tb.m` as the test file. Click **Autodefine Input Types**.

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

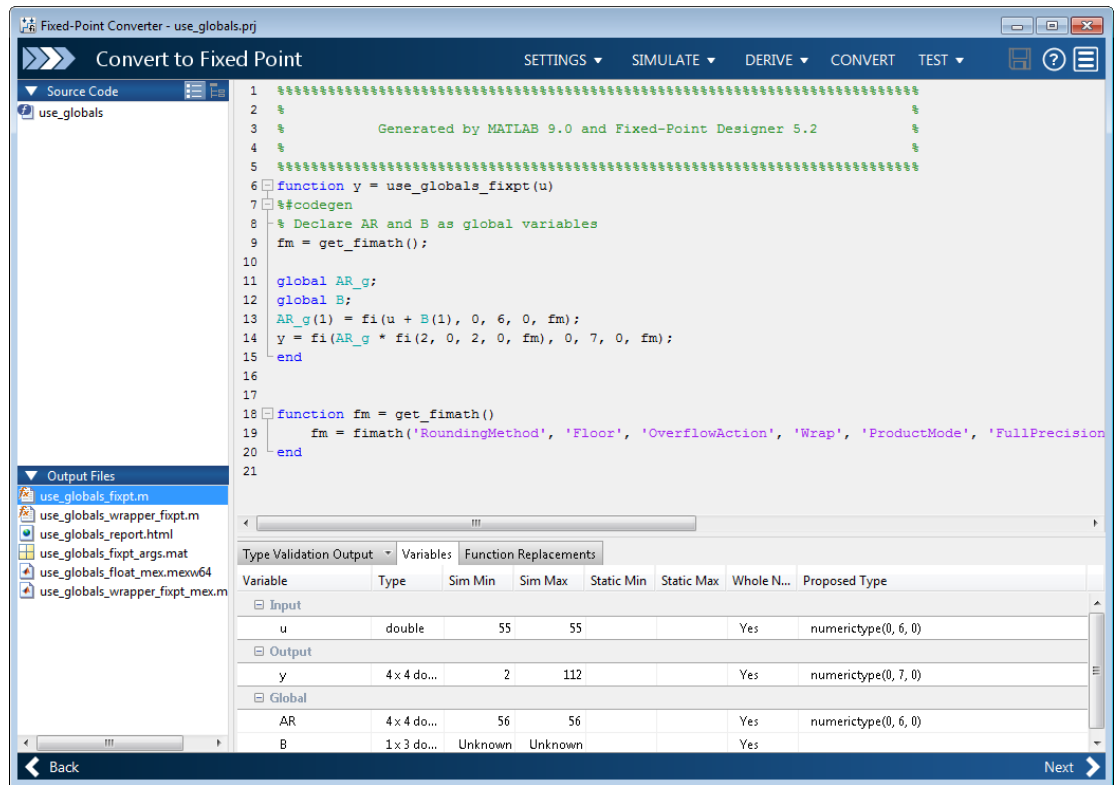
- 6 Select **Yes** next to **Does this code use global variables**. By default, the Fixed-Point Converter app names the first global variable in a project `g`.
- 7 Type in the names of the global variables in your code. In the field to the right of the global variable `AR`, specify its type as `double(4x4)`.
- 8 The global variable `B` is not assigned in the `use_globals` function. Define this variable as a global constant by clicking the field to the right of the constant and selecting **Define Constant Value**. Type in the value of `B` as it is defined in the test file, `[1 2 3]`. The app indicates that `B` has the value `[1 2 3]`. The app indicates that `AR` is not initialized.



- 9 Click **Next**. The app generates an instrumented MEX function for your entry-point MATLAB function. On the **Convert to Fixed-Point** page, click **Simulate** to simulate the function, gather range information, and get proposed data types.
- 10 Click **Convert** to accept the proposed data types and convert the function to fixed-point.

In the generated fixed-point code, the global variable AR is now AR_g.

The wrapper function contains three global variables: AR, AR_g, and B, where AR_g is set equal to a fi-casted AR, and AR is set equal to a double casted AR_g. The global variable B does not have a separate variable in the fixed-point code because it is a constant.



```
function y = use_globals_fixpt(u)
%#codegen
% Declare AR and B as global variables
fm = get_fimath();

global AR_g;
global B;
AR_g(1) = fi(u + B(1), 0, 6, 0, fm);
y = fi(AR_g * fi(2, 0, 2, 0, fm), 0, 7, 0, fm);
end
```

```
function fm = get_fimath()
fm = fimath('RoundingMethod', 'Floor',...
'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision',...
```

```
'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', ...  
'MaxSumWordLength', 128);  
end
```

More About

- “Convert Code Containing Global Data to Fixed Point” on page 8-69

Convert Code Containing Structures to Fixed Point

This example shows how to convert a MATLAB algorithm containing structures to fixed point using the Fixed-Point Converter app.

- 1 In a local writable folder, create the functions `struct_fcn.m`

```
function [out, y] = struct_fcn(in)
    % create a nested struct
    z = struct('prop1', struct('subprop1', 0, 'subprop2', [3 4 45]));
    % copy over the struct
    y = z;
    y.prop1.subprop1 = y.prop1.subprop1 + in;
    out = y.prop1.subprop1;
end
```

- 2 In the same folder, create a test file, `struct_fcn_tb.m`, that calls the function.

```
for ii = 1:10
    struct_fcn(ii);
end
```

- 3 From the apps gallery, open the Fixed-Point Converter app.
- 4 On the **Select Source Files** page, browse to the `struct_fcn.m` file, and click **Open**.
- 5 Click **Next**. On the **Define Input Types** page, enter the test file that exercises the `struct_fcn` function. Browse to select the `struct_fcn_tb.m` file. Click **Autodefine Input Types**.
- 6 Click **Next**. The app generates an instrumented MEX function for your entry-point MATLAB function. On the **Convert to Fixed-Point** page, click **Simulate** to simulate the function, gather range information, and propose data types.

When the names, number, and types of fields of two or more structures match, the Fixed-Point Converter app proposes a unified type. In this example, the range of `z.prop1.subprop1` is `[0, 0]`, while the range of `y.prop1.subprop1` is `[0, 10]`. The app proposes a data type of `numeric(0, 4, 0)` for both `z.prop1.subprop1` and `y.prop1.subprop1` based on the union of the ranges of the two fields.

The screenshot shows the 'Fixed-Point Converter - struct_fcn.prj' window. The main area displays the source code for a function named `struct_fcn`. The code defines a function that takes an input `in` and returns `out` and `y`. It creates a nested structure `z` with fields `prop1` and `subprop2`, copies it to `y`, and then increments the `subprop1` field of `y` by the value of `in`.

Below the source code, the 'Function Replacements' table is displayed, showing the conversion of variables to fixed-point types. The table has columns for Variable, Type, Sim Min, Sim Max, Static Min, Static Max, Whole N..., and Proposed Type.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole N...	Proposed Type
Input							
in	double	1	10			Yes	numerictype(0, 4, 0)
Output							
out	double	1	10			Yes	numerictype(0, 4, 0)
y	struct						
prop1							
subprop1	double	0	10			Yes	numerictype(0, 4, 0)
subprop2	1x3 dou...	3	45			Yes	numerictype(0, 6, 0)
Local							
z	struct						
prop1							
subprop1	double	0	0			Yes	numerictype(0, 4, 0)
subprop2	1x3 dou...	3	45			Yes	numerictype(0, 6, 0)

7 Click **Convert**.

The Fixed-Point Converter converts the function containing the structures to fixed point and generates the `struct_fcn_fixpt.m` file.

The screenshot displays the 'Fixed-Point Converter - struct_fcn.prj' window. The main area shows the source code for 'struct_fcn' with the following content:

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  %       Generated by MATLAB 9.0 and Fixed-Point Designer 5.2
4  %
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6  %#codegen
7  function [out, y] = struct_fcn_fixpt(in)
8      % create a nested struct
9      fm = get_fimath();
10
11     z = struct('prop1', struct('subprop1', fi(0, 0, 4, 0, fm), 'subprop2', fi([3 4 45], 0, 6, 0, f
12     % copy over the struct
13     y = z;
14     y.prop1.subprop1 = fi(y.prop1.subprop1 + in, 0, 4, 0, fm);
15     out = fi(y.prop1.subprop1, 0, 4, 0, fm);
16 end
17
18
19 function fm = get_fimath()
20     fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision
21 end
22

```

On the left, the 'Output Files' list includes:

- struct_fcn_fixpt.m
- struct_fcn_wrapper_fixpt.m
- struct_fcn_report.html
- struct_fcn_fixpt_args.mat
- struct_fcn_float_mex.mexw64
- struct_fcn_wrapper_fixpt_mex.mexw64

At the bottom, the 'Type Validation Output' table is visible:


Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole N...	Proposed Type
Input							
in	double	1	10			Yes	numerictype(0, 4, 0)
Output							
out	double	1	10			Yes	numerictype(0, 4, 0)
y							
prop1	struct						

Data Type Issues in Generated Code

Within the fixed-point conversion HTML report, you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

Enable the Highlight Option in the Fixed-Point Converter App

- 1 On the **Convert to Fixed Point** page, to open the **Settings** dialog box, click the **Settings** arrow .
- 2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to **Yes**.

When conversion is complete, open the fixed-point conversion HTML report to view the highlighting. Click **View report** in the **Type Validation Output** tab.

Enable the Highlight Option at the Command Line

- 1 Create a fixed-point code configuration object:

```
cfg = coder.config('fixpt');
```
- 2 Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.

```
cfg.HighlightPotentialDataTypeIssues = true;
```

Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to **C99 (ISO)**.

Stowaway Singles

This check highlights all expressions that result in a single operation.

Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient” on page 46-11.

Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated

code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

System Objects Supported by Fixed-Point Converter App

You can use the Fixed-Point Converter app to automatically propose and apply data types for commonly used system objects. The proposed data types are based on simulation data from the System object™.

Automated conversion is available for these DSP System Toolbox System Objects:

- `dsp.ArrayVectorAdder`
- `dsp.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.LUFactor`
- `dsp.UpperTriangularSolver`
- `dsp.VariableFractionalDelay`
- `dsp.Window`

The Fixed-Point Converter app can display simulation minimum and maximum values, whole number information, and histogram data.

- You cannot propose data types for these System objects based on static range data.
- You must configure the System object to use 'Custom' fixed-point settings.
- The app applies the proposed data types only if the input signal is floating point, not fixed-point.

The app treats scaled doubles as fixed-point. The scaled doubles workflow for System objects is the same as that for regular variables.

- The app ignores the **Default word length** setting in the **Settings** menu. The app also ignores specified rounding and overflow modes. Data-type proposals are based on the settings of the System object.

Related Examples

- “Convert `dsp.FIRFilter` Object to Fixed-Point Using the Fixed-Point Converter App” on page 8-85

Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App

This example converts a `dsp.FIRFilter` System object, which filters a high-frequency sinusoid signal, to fixed-point using the Fixed-Point Converter app. This example requires Fixed-Point Designer and DSP System Toolbox licenses.

Create DSP Filter Function and Test Bench

Create a `myFIRFilter` function from a `dsp.FIRFilter` System object.

By default, System objects are configured to use full-precision fixed-point arithmetic. To gather range data and get data type proposals from the Fixed-Point Converter app, configure the System object to use 'Custom' settings.

Save the function to a local writable folder.

```
function output = myFIRFilter(input, num)

    persistent lowpassFIR;
    if isempty(lowpassFIR)
        lowpassFIR = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype(1,16,4), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1,8,2));
    end
    output = lowpassFIR(input, num);
end
```

`end`

Create a test bench, `myFIRFilter_tb`, for the filter. The test bench generates a signal that gathers range information for conversion. Save the test bench.

```
% Test bench for myFIRFilter
% Remove high-frequency sinusoid using an FIR filter.

% Initialize
f1 = 1000;
f2 = 3000;
```

```
Fs = 8000;
Fcutoff = 2000;

% Generate input
SR = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1024);

% Filter coefficients
num = fir1(130,Fcutoff/(Fs/2));

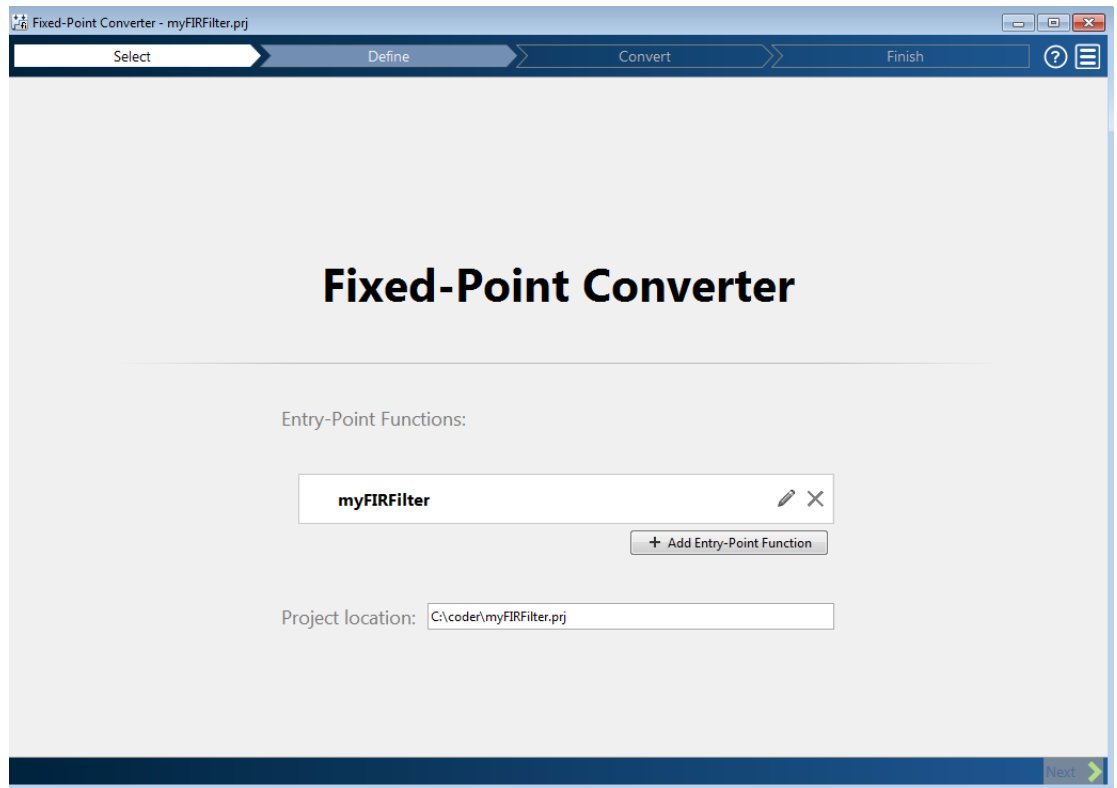
% Visualize input and output spectra
plot = dsp.SpectrumAnalyzer('SampleRate',Fs,'PlotAsTwoSidedSpectrum',...
    false,'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Input Signal (Channel 1) Output Signal (Channel 2)');

% Stream
for k = 1:100
    input = sum(SR(),2); % Add the two sinusoids together
    filteredOutput = myFIRFilter(input, num); % Filter
    plot([input,filteredOutput]); % Visualize
end
```

Convert the Function to Fixed-Point

- 1 Open the Fixed-Point Converter app.
 - MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
 - MATLAB command prompt: Enter
`fixedPointConverter`
- 2 To add the entry-point function `myFIRFilter` to the project, browse to the file `myFIRFilter.m`, and then click **Open**.

By default, the app saves information and settings for this project in the current folder in a file named `myFirFilter.prj`.

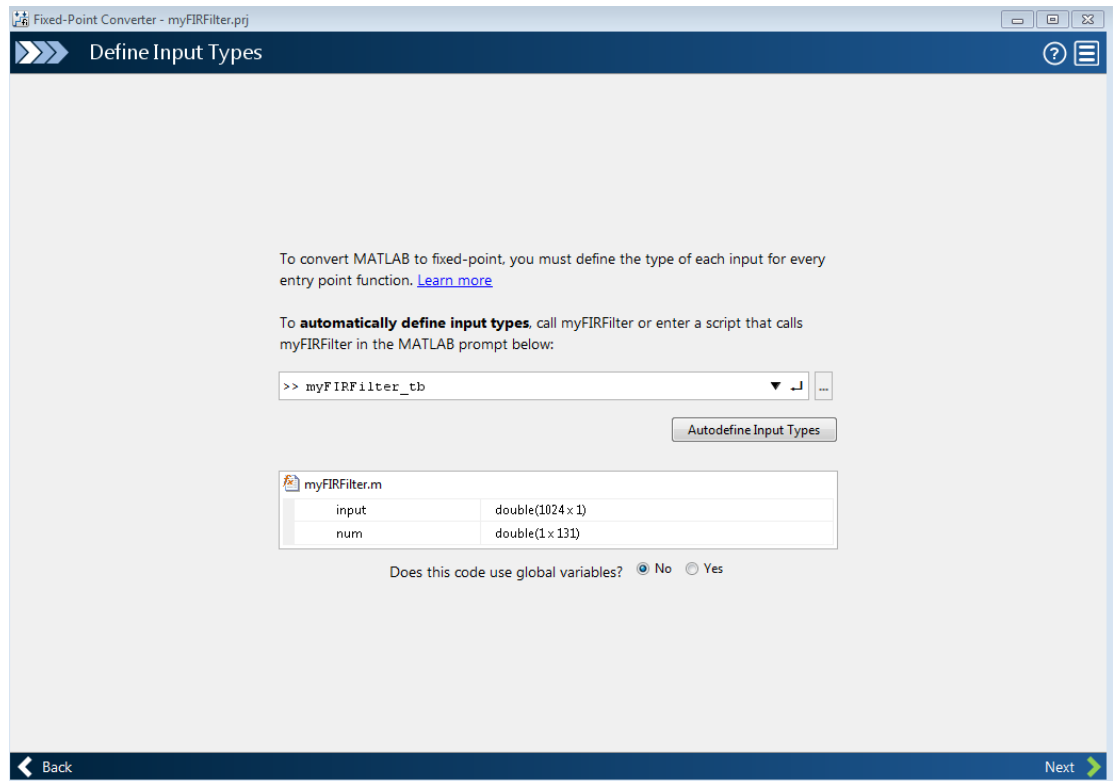


- 3 Click **Next** to go to the **Define Input Types** step.

The app screens `myFIRFilter.m` for code violations and fixed-point conversion readiness issues. The app does not find issues in `myFIRFilter.m`.

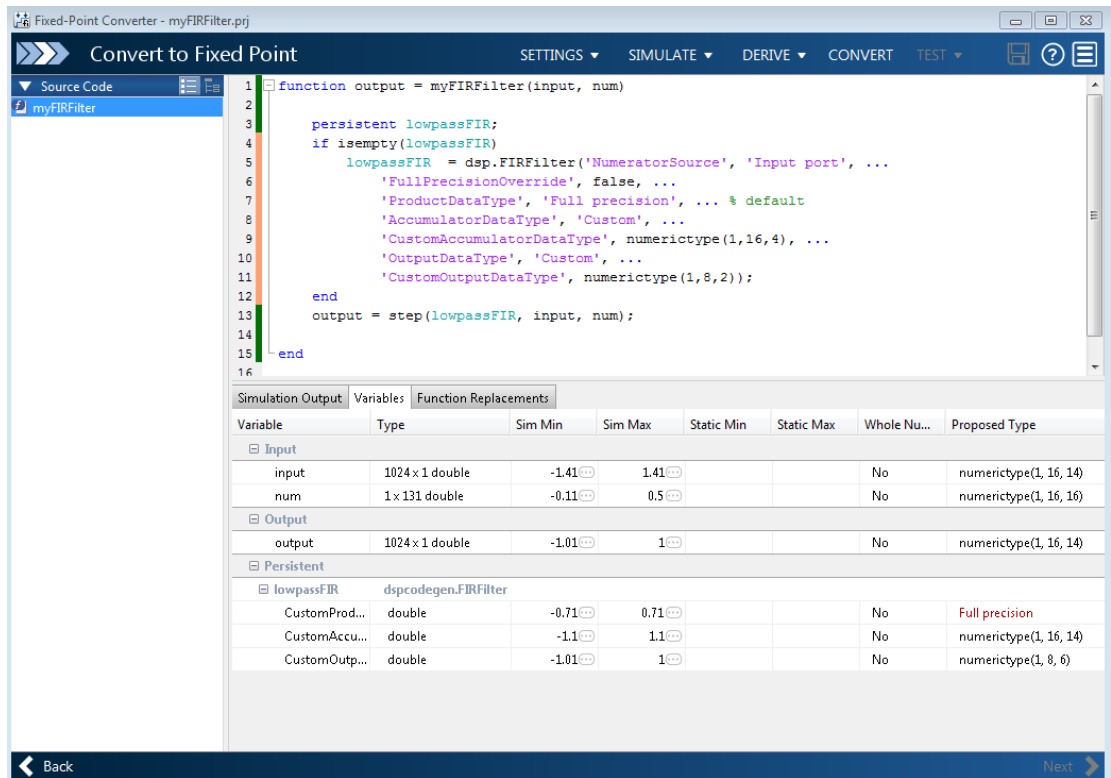
- 4 On the **Define Input Types** page, to add `myFIRFilter_tb` as a test file, browse to `myFIRFilter_tb.m`, and then click **Autodefine Input Types**.

The app determines from the test file that the type of `input` is `double(1024 x 1)` and the type of `num` is `double(1 x 131)`.



- 5 Click **Next** to go to the **Convert to Fixed Point** step.
- 6 On the **Convert to Fixed Point** page, click **Simulate** to collect range information.

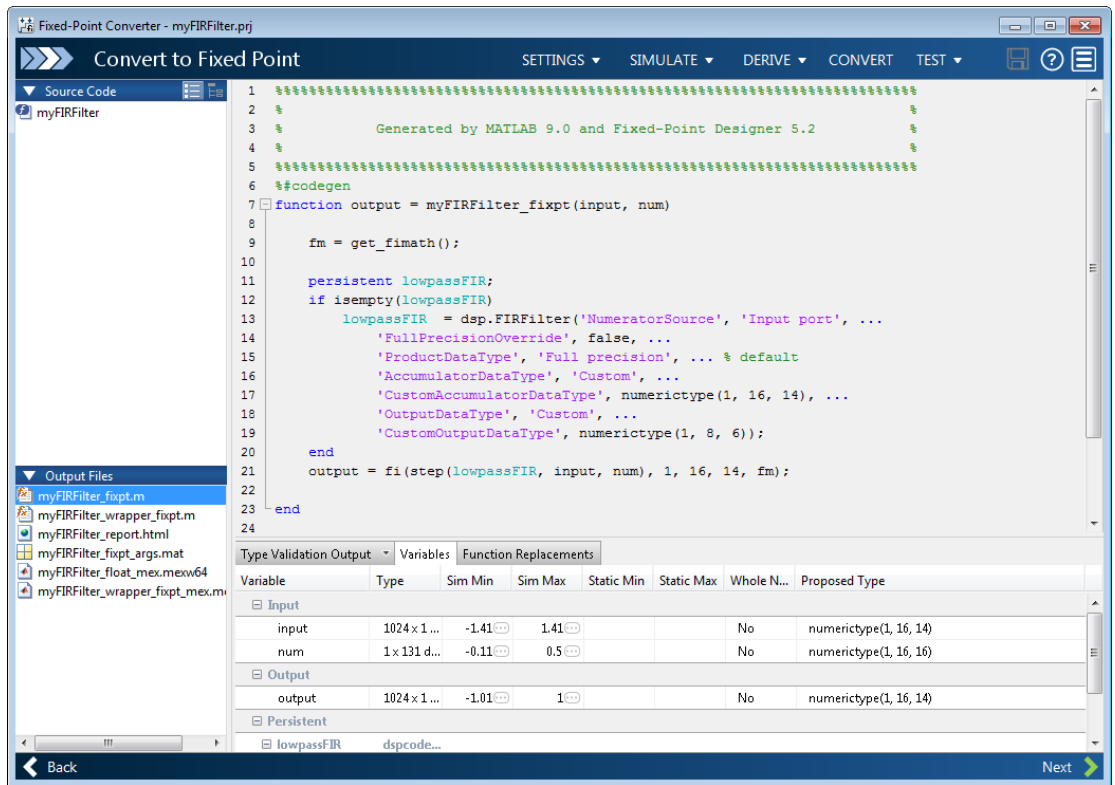
The **Variables** tab displays the collected range information and type proposals. Manually edit the data type proposals as needed.



In the **Variables** tab, the **Proposed Type** field for `lowpassFIR.CustomProductDataType` is Full Precision. The Fixed-Point Converter app did not propose a data type for this field because its 'ProductDataType' setting is not set to 'Custom'.

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

The Fixed-Point Converter app applies the proposed data types and generates a fixed-point function, `myFIRFilter_fixpt`.



```
function output = myFIRFilter_fixpt(input, num)
```

```
    fm = get_fimath();
```

```
    persistent lowpassFIR;
```

```
    if isempty(lowpassFIR)
```

```
        lowpassFIR = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numeric(1, 16, 14), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numeric(1, 8, 6));
```

```
    end
```

```
    output = fi(lowpassFIR(input, num), 1, 16, 14, fm);
```

```
end
```

```
function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode',...
        'FullPrecision', 'MaxSumWordLength', 128);
end
```

More About

- “System Objects Supported by Fixed-Point Converter App” on page 8-84

Automated Conversion Using Fixed-Point Converter App

- “Specify Type Proposal Options” on page 9-2
- “Detect Overflows” on page 9-6
- “Propose Data Types Based on Simulation Ranges” on page 9-15
- “Propose Data Types Based on Derived Ranges” on page 9-29
- “View and Modify Variable Information” on page 9-43
- “Replace the exp Function with a Lookup Table” on page 9-47
- “Convert Fixed-Point Conversion Project to MATLAB Scripts” on page 9-55
- “Replace a Custom Function with a Lookup Table” on page 9-57
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 9-65
- “Enable Plotting Using the Simulation Data Inspector” on page 9-76
- “Add Global Variables by Using the App” on page 9-77
- “Automatically Define Input Types by Using the App” on page 9-78
- “Define Constant Input Parameters Using the App” on page 9-79
- “Define or Edit Input Parameter Type by Using the App” on page 9-80
- “Define Input Parameter by Example by Using the App” on page 9-88
- “Specify Global Variable Type and Initial Value Using the App” on page 9-96
- “Specify Properties of Entry-Point Function Inputs Using the App” on page 9-100
- “Detect Dead and Constant-Folded Code” on page 9-101

Specify Type Proposal Options

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

The following options are available.

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	16 (default)	Default word length to use when Fixed-point type proposal mode is set to Propose fraction lengths for specified word lengths
Default fraction length	4 (default)	Default fraction length to use when Fixed-point type proposal mode is set to Propose word lengths for specified fraction lengths

Advanced Type Proposal Settings	Values	Description
When proposing types	ignore simulation ranges	Propose data types based on derived ranges.
Note: Manually-entered static ranges always take precedence over simulation ranges.	ignore derived ranges	Propose data types based on simulation ranges.

Advanced Type Proposal Settings	Values	Description
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code generation (8,16,32, 64 ...). For example, for a variable with range [0 . . 7], propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.

Advanced Type Proposal Settings	Values	Description
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values. The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.
Search paths	' ' (default)	Add paths to the list of paths to search for MATLAB files. Separate list items with a semicolon.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the <code>fimath</code> properties for the generated fixed-point data types. The default fixed-point math properties use the <code>Floor</code> rounding and <code>Wrap</code> overflow. These settings generate the most efficient code but might cause problems with overflow.
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	

fimath Settings	Values	Description
	KeepLSB	For more information on <code>fimath</code> properties, see “ <code>fimath</code> Object Properties” on page 4-5.
	KeepMSB	
	SpecifyPrecision	

Generated File Settings	Value	Description
Generated fixed-point file name suffix	<code>_fixpt</code> (default)	Specify the suffix to add to the generated fixed-point file names.

Plotting and Reporting Settings	Values	Description
Custom plot function	<code>' '</code> (default)	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	Specify whether to use the Simulation Data Inspector for comparison plots.
	Yes	
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

Detect Overflows

This example shows how to detect overflows using the Fixed-Point Converter app. At the numerical testing stage in the conversion process, you choose to simulate the fixed-point code using scaled doubles. The app then reports which expressions in the generated code produce values that overflow the fixed-point data type.

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

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

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

It is a best practice is to create a separate test script to do pre- and post-processing, such as:

- Loading inputs.
- Setting up input values.
- Outputting test results.

For more information, see “Create a Test File” on page 12-4.

Type	Name	Description
Function code	<code>overflow.m</code>	Entry-point MATLAB function

Type	Name	Description
Test file	overflow_test.m	MATLAB script that tests overflow.m

The overflow Function

```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end
```

The overflow_test Function

You use this test file to define input types for **b**, **x**, and **reset**, and, later, to verify the fixed-point version of the algorithm.

```
function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    % b = fir1(11,0.25);
    b = [-0.004465461051254
        -0.004324228005260
        +0.012676739550326
```

```
+0.074351188907780
+0.172173206073645
+0.249588554524763
+0.249588554524763
+0.172173206073645
+0.074351188907780
+0.012676739550326
-0.004324228005260
-0.004465461051254]';

% Input signal
nx = 256;
t = linspace(0,10*pi,nx)';

% Impulse
x_impulse = zeros(nx,1); x_impulse(1) = 1;

% Max Gain
% The maximum gain of a filter will occur when the inputs line up with the
% signs of the filter's impulse response.
x_max_gain = sign(b)';
x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
x_max_gain = x_max_gain(1:nx);

% Sums of sines
f0=0.1; f1=2;
x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

% Chirp
f_chirp = 1/16; % Target frequency
x_chirp = sin(pi*f_chirp*t.^2); % Linear chirp

x = [x_impulse, x_max_gain, x_sines, x_chirp];
titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
y = zeros(size(x));

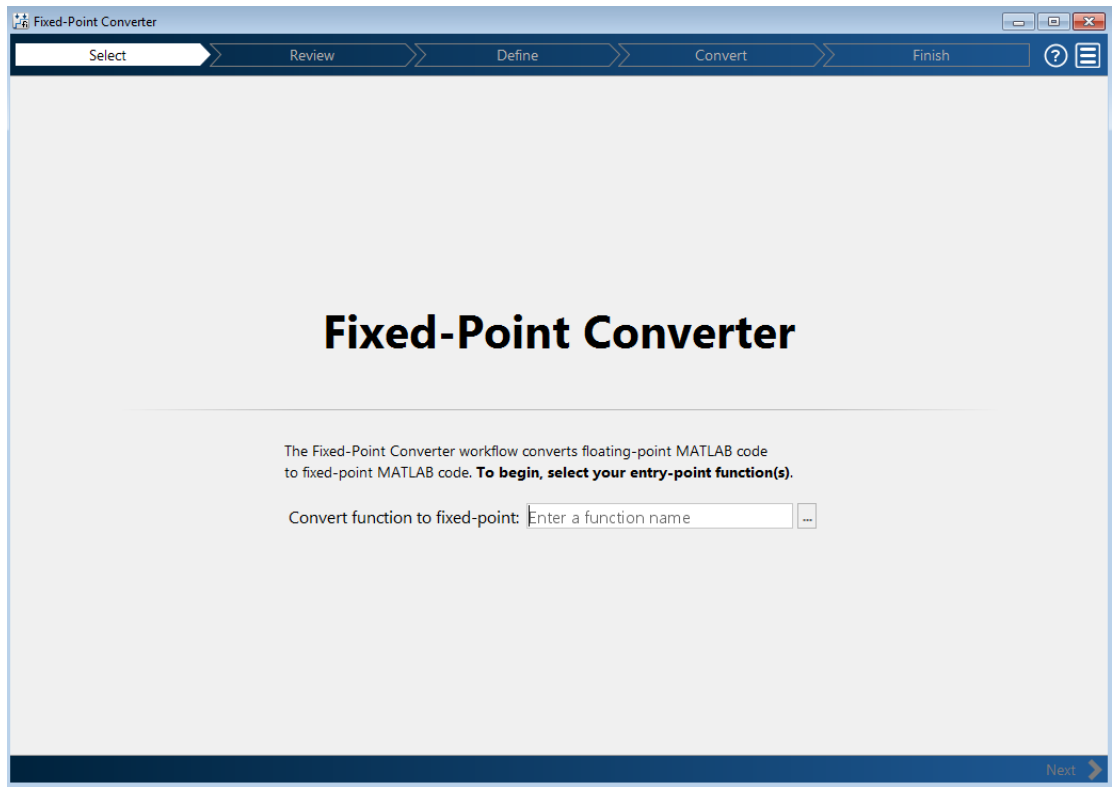
for i=1:size(x,2)
    reset = true;
    y(:,i) = overflow(b,x(:,i),reset);
end

test_plot(1,titles,t,x,y)
```

```
end
function test_plot(fig,titles,t,x,y1)
    figure(fig)
    clf
    sub_plot = 1;
    font_size = 10;
    for i=1:size(x,2)
        subplot(4,1,sub_plot)
        sub_plot = sub_plot+1;
        plot(t,x(:,i),'c',t,y1(:,i),'k')
        axis('tight')
        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end
```

Open the Fixed-Point Converter App

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



Select Source Files

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

The app screens `overflow.m` for code violations and fixed-point conversion readiness issues. The app does not find issues in `overflow.m`.

Define Input Types

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

2 Click **Autodefine Input Types**.

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

To **automatically define input types**, call `overflow` or enter a script that calls `overflow` in the MATLAB prompt below:

```
>> overflow_test
```

Autodefine Input Types

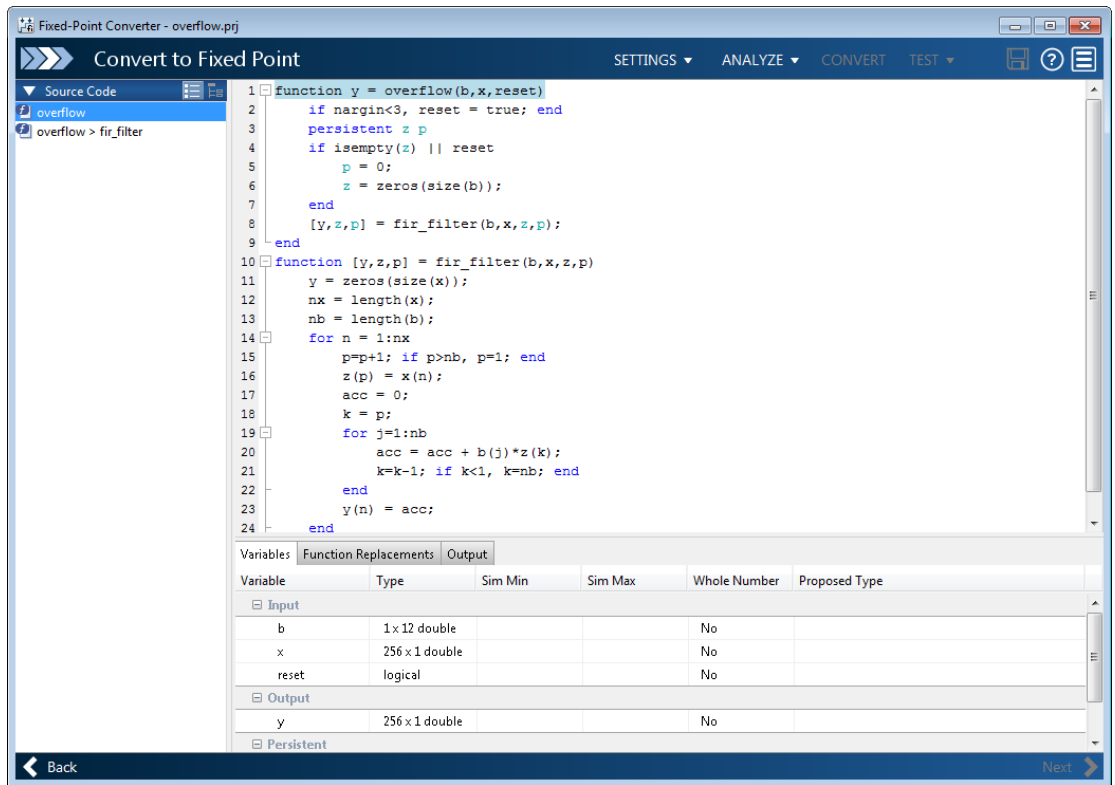
Variable	Type
<code>b</code>	<code>double(1 x 12)</code>
<code>x</code>	<code>double(256 x 1)</code>
<code>reset</code>	<code>logical(1 x 1)</code>

[Add global](#)


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

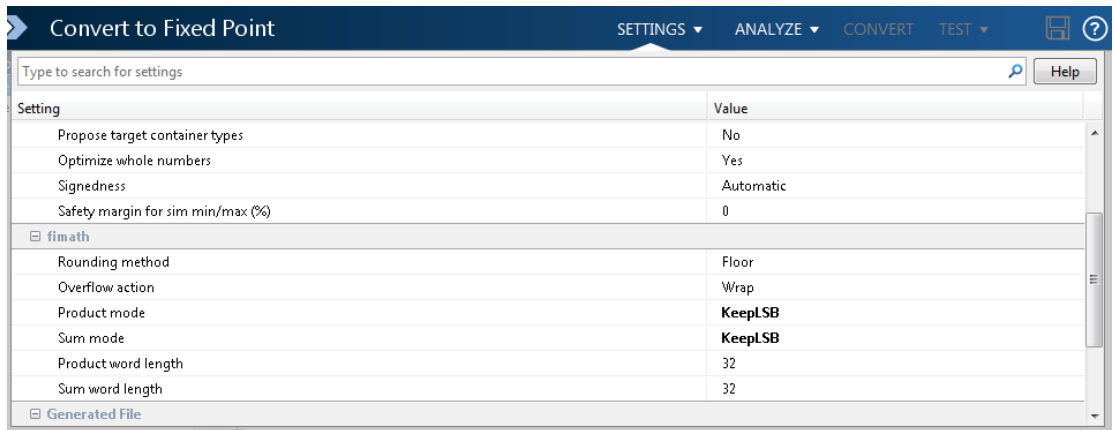
Convert to Fixed Point

- 1 The app generates an instrumented MEX function for your entry-point MATLAB function. The app displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information” on page 9-43.



On the **Function Replacements** tab the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 8-9.

- 2 To view the fimath settings, click the **Settings** arrow . Set the fimath **Product mode** and **Sum mode** to **KeepLSB**. These settings model the behavior of integer operations in the C language.



3 Click **Analyze**.

The test file, `overflow_test`, runs. The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.


Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
b	1 x 12 double	0	0.25	No	numerictype(1, 16, 17)
x	256 x 1 double	-1.1	1.09	No	numerictype(1, 16, 14)
reset	logical	1	1	Yes	numerictype(0, 1, 0)
Output					
y	256 x 1 double	-1	1.04	No	numerictype(1, 16, 14)
Persistent					
z	1 x 12 double	-1	1	No	numerictype(1, 16, 14)
p	double	0	4	Yes	numerictype(0, 3, 0)

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

The software validates the proposed types and generates a fixed-point version of the entry-point function.

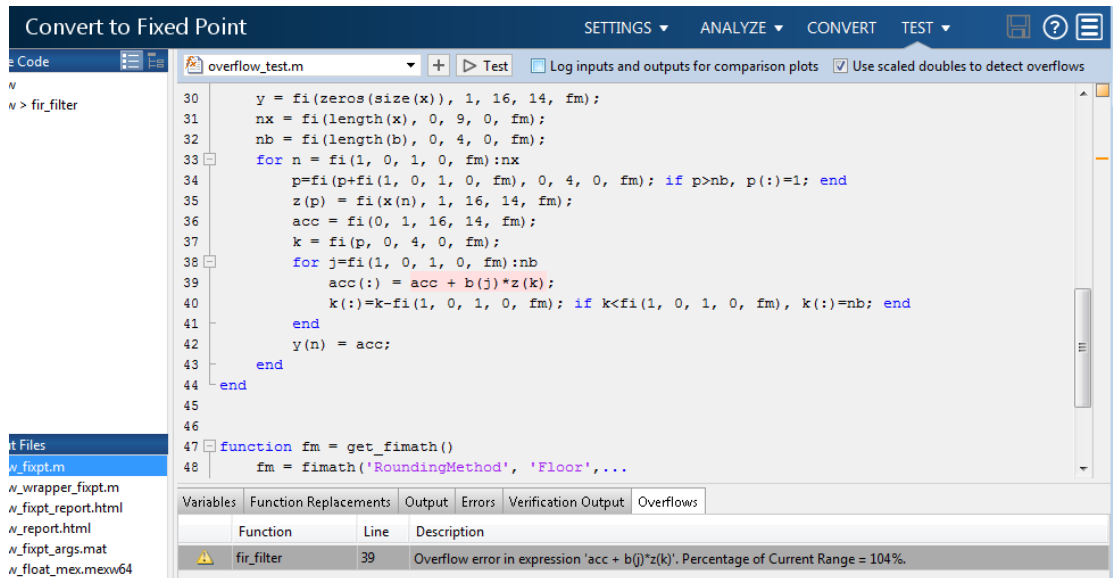
If errors and warnings occur during validation, the app displays them on the **Output** tab. See “Validating Types” on page 8-23.

Test Numerics and Check for Overflows

- 1 Click the **Test** arrow . Verify that the test file is `overflow_test.m`. Select **Use scaled doubles to detect overflows**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to detect overflows, it also runs the simulation using scaled double versions of the proposed fixed-point types. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. Because they retain their fixed-point settings, they can report when a computation goes out of the range of the fixed-point type.

The simulation runs. The app detects an overflow. The app reports the overflow on the **Overflow** tab. To highlight the expression that overflowed, click the overflow.



The screenshot shows the 'Convert to Fixed Point' app interface. The main window displays the MATLAB code for 'overflow_test.m'. The code includes a function 'fir_filter' and a function 'fm' that uses 'fimath' for fixed-point arithmetic. The 'TEST' tab is active, and the 'Use scaled doubles to detect overflows' checkbox is checked. Below the code editor, the 'Overflows' tab is selected, showing a table with the following data:

Function	Line	Description
fir_filter	39	Overflow error in expression 'acc + b(j)*z(k)'. Percentage of Current Range = 104%.

- 2 Determine whether it was the sum or the multiplication that overflowed.

In the **fimath** settings, set **Product mode** to **FullPrecision**, and then repeat the conversion and test the fixed-point code again.

The overflow still occurs, indicating that it is the addition in the expression that is overflowing.

Propose Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data using the Fixed-Point Converter app.

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

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

- 3 Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

It is a best practice is to create a separate test script to do pre- and post-processing, such as:

- Loading inputs.
- Setting up input values.
- Outputting test results.

See “Create a Test File” on page 12-4.

Type	Name	Description
Function code	<code>ex_2ndOrder_filter.m</code>	Entry-point MATLAB function
Test file	<code>ex_2ndOrder_filter_test.m</code>	MATLAB script that tests <code>ex_2ndOrder_filter.m</code>

The ex_2ndOrder_filter Function

```

function y = ex_2ndOrder_filter(x) %#codegen
    persistent z
    if isempty(z)
        z = zeros(2,1);
    end
    % [b,a] = butter(2, 0.25)
    b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
    a = [1, -0.942809041582063, 0.3333333333333333];

    y = zeros(size(x));
    for i = 1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i) - a(3) * y(i);
    end
end

```

The ex_2ndOrder_filter_test Script

The test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```

% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = ex_2ndOrder_filter(x(i,:));
end

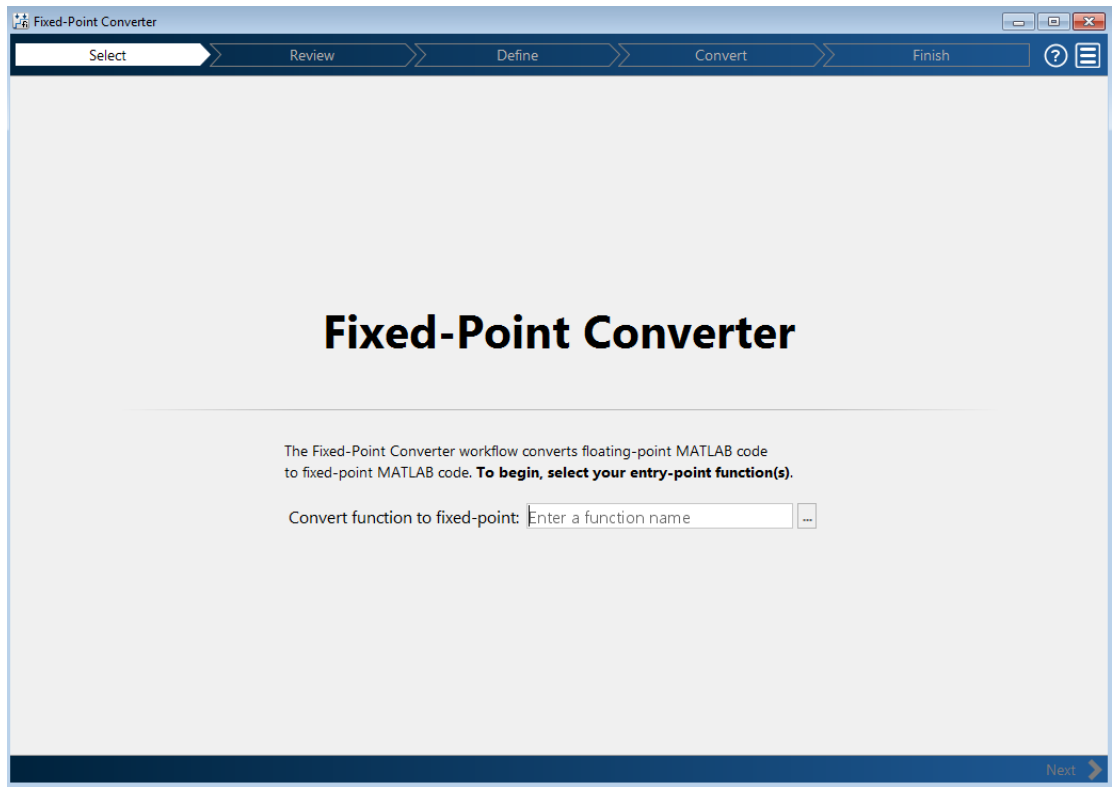
```

```
% Plot the results
titles = {'Chirp', 'Step', 'Impulse'}
clf
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input', 'Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Open the Fixed-Point Converter App

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



Select Source Files

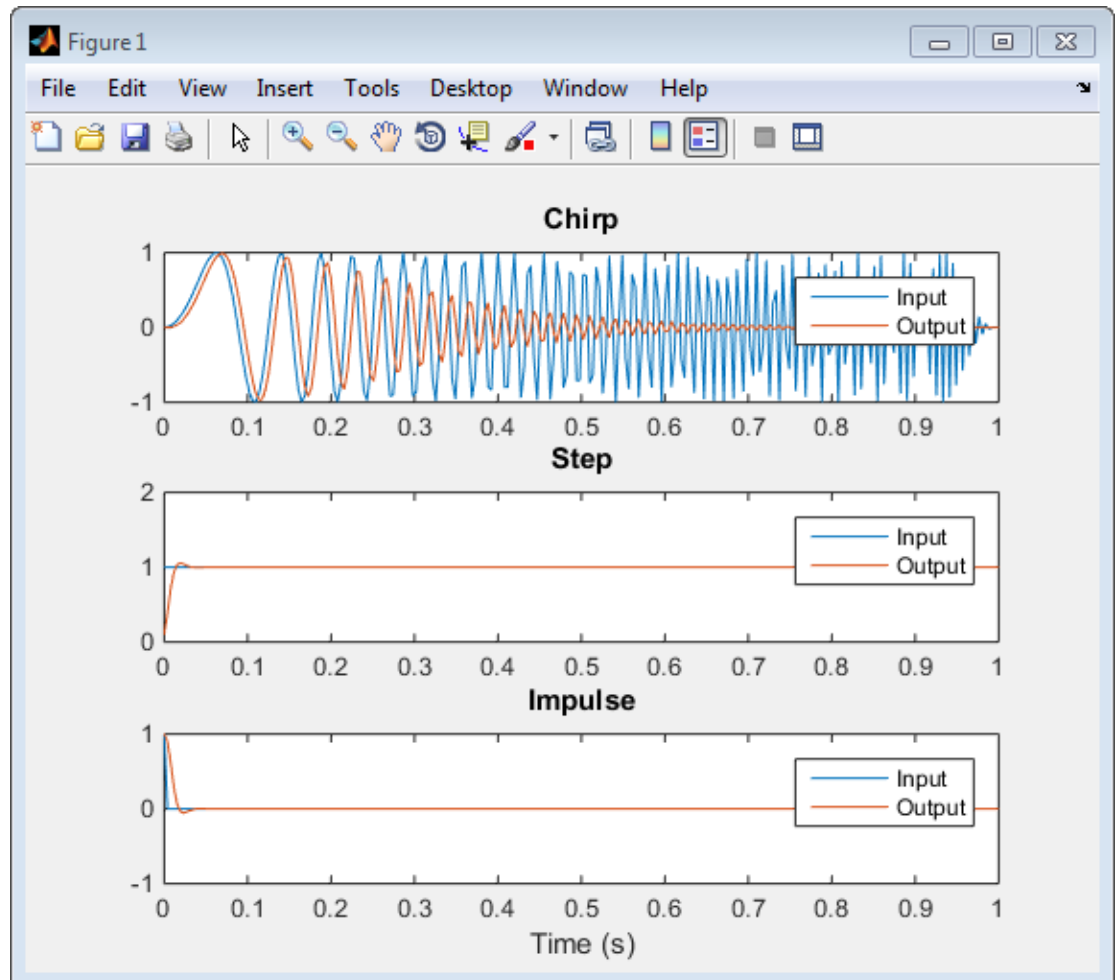
- 1 To add the entry-point function `ex_2ndOrder_filter` to the project, browse to the file `ex_2ndOrder_filter.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `ex_2ndOrder_filter.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `ex_2ndOrder_filter.m` for code violations and fixed-point conversion readiness issues. The app does not find issues in `ex_2ndOrder_filter.m`.

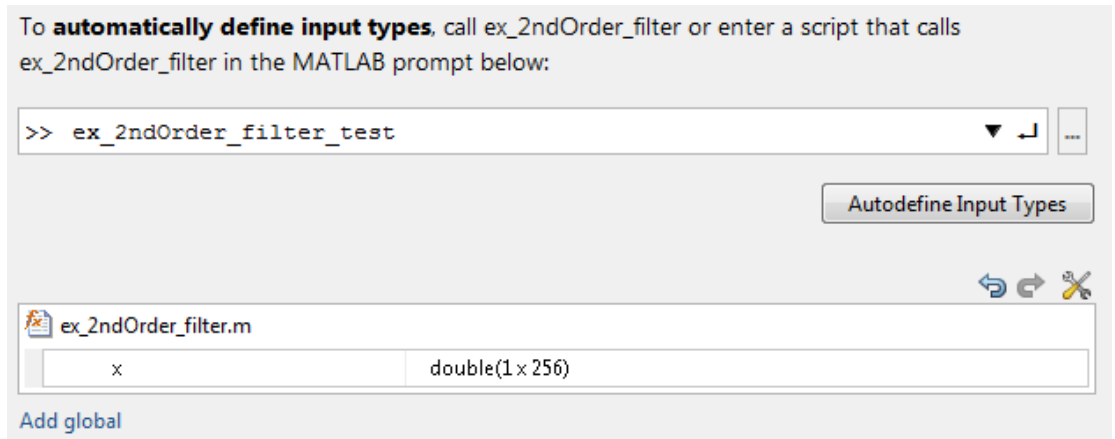
Define Input Types

- 1 On the **Define Input Types** page, to add `ex_2ndOrder_filter_test` as a test file, browse to `ex_2ndOrder_filter_test`, and then click **Open**.
- 2 Click **Autodefine Input Types**.

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



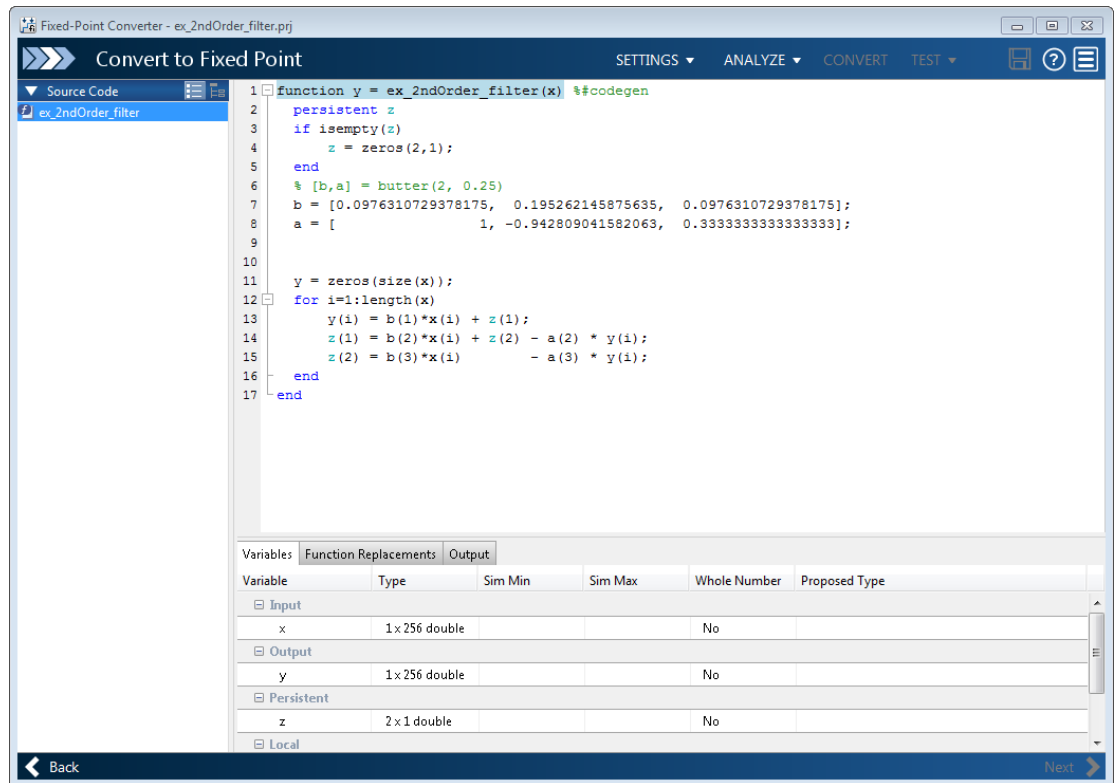
The app determines from the test file that the input type of `x` is `double(1x256)`.




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

Convert to Fixed Point

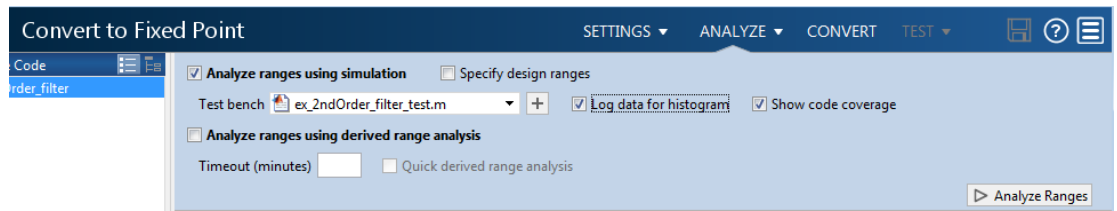
- 1 The app generates an instrumented MEX function for your entry-point MATLAB function. The app displays compiled information—type, size, and complexity—for variables in your code. See “View and Modify Variable Information” on page 9-43.



On the **Function Replacements** tab, the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 8-9.

- 2 Click the **Analyze** arrow . Verify that **Analyze ranges using simulation** is selected and that the test bench file is `ex_2ndOrder_filter_test`. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app merges the simulation results.
- 3 Select **Log data for histogram**.

By default, the **Show code coverage** option is selected. This option provides code coverage information that helps you verify that your test file is testing your algorithm over the intended operating range.



4 Click **Analyze**.

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

The screenshot shows the 'Fixed-Point Converter - ex_2ndOrder_filter.prj' window. The main area displays the source code for a MATLAB function named 'ex_2ndOrder_filter'. The code includes a persistent variable 'z', a Butterworth filter design, and a loop that processes the input 'x' to produce the output 'y'.

Below the code, the 'Variables' tab is active, showing a table of simulation ranges and proposed fixed-point types for variables 'x', 'y', and 'z'.

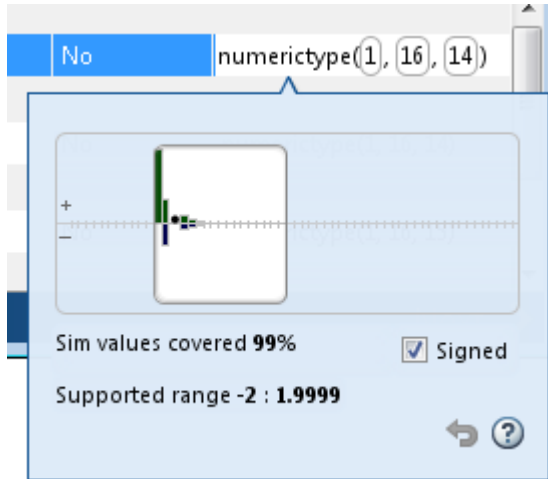
Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
x	1 x 256 double	-1	1	No	numerictype(1, 16, 14)
Output					
y	1 x 256 double	-0.97	1.06	No	numerictype(1, 16, 14)
Persistent					
z	2 x 1 double	-0.89	0.96	No	numerictype(1, 16, 15)
Local					

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

The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The app enables the **Convert** option.

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

- 5 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.



To modify the proposed data types, either enter the required type into the **Proposed Type** field or use the histogram controls. For more information about the histogram, see “Log Data for Histogram” on page 8-20.

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

During the fixed-point conversion process, the software validates the proposed types and generates the following files in the `codegen\ex_2ndOrder_filter\fixpt` folder in your local working folder.

- `ex_2ndOrder_filter_fixpt.m` — the fixed-point version of `ex_2ndOrder_filter.m`.
- `ex_2ndOrder_filter_wrapper_fixpt.m` — this file converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during conversion. These fixed-point values are fed into the converted fixed-point design, `ex_2ndOrder_filter_fixpt.m`.
- `ex_2ndOrder_filter_fixpt_report.html` — this report shows the generated fixed-point code and the fixed-point instrumentation results.
- `ex_2ndOrder_filter_report.html` — this report shows the original algorithm and the fixed-point instrumentation results.

- `ex_2ndOrder_filter_fixpt_args.mat` — MAT-file containing a structure for the input arguments, a structure for the output arguments and the name of the fixed-point file.

If errors or warnings occur during validation, you see them on the **Type Validation Output** tab. See “Validating Types” on page 8-23.

- 7 In the **Output Files** list, select `ex_2ndOrder_filter_fixpt.m`. The app displays the generated fixed-point code.

The screenshot shows the Fixed-Point Converter app interface. The main window displays the generated fixed-point code for a Butterworth filter. The code includes variable declarations, filter coefficients, and a loop for signal processing. The Output Files list shows the generated files, and the Variables table shows the data types and sizes for the input, output, and persistent variables.


```

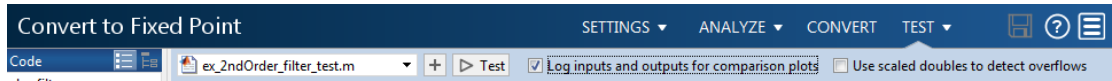
function y = ex_2ndOrder_filter_fixpt(x) %codegen
7   fm = get_fimath();
8
9   persistent z
10  if isempty(z)
11     z = fi(zeros(2,1), 1, 16, 15, fm);
12  end
13  % [b,a] = butter(2, 0.25)
14  b = fi([0.0976310729378175, 0.195262145875635, 0.0976310729378175], 0, 16, 18, fm);
15  a = fi([1, -0.942809041582063, 0.333333333333333], 1, 16, 14, fm);
16
17
18  y = fi(zeros(size(x)), 1, 16, 14, fm);
19  for i=1:length(x)
20     y(i) = b(1)*x(i) + z(1);
21     z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
22     z(2) = fi_signed(b(3)*x(i)) - a(3) * y(i);
23  end
24 end
25
26
27
28 function y = fi_signed(a)
29     coder.inline( 'always' );
--

```

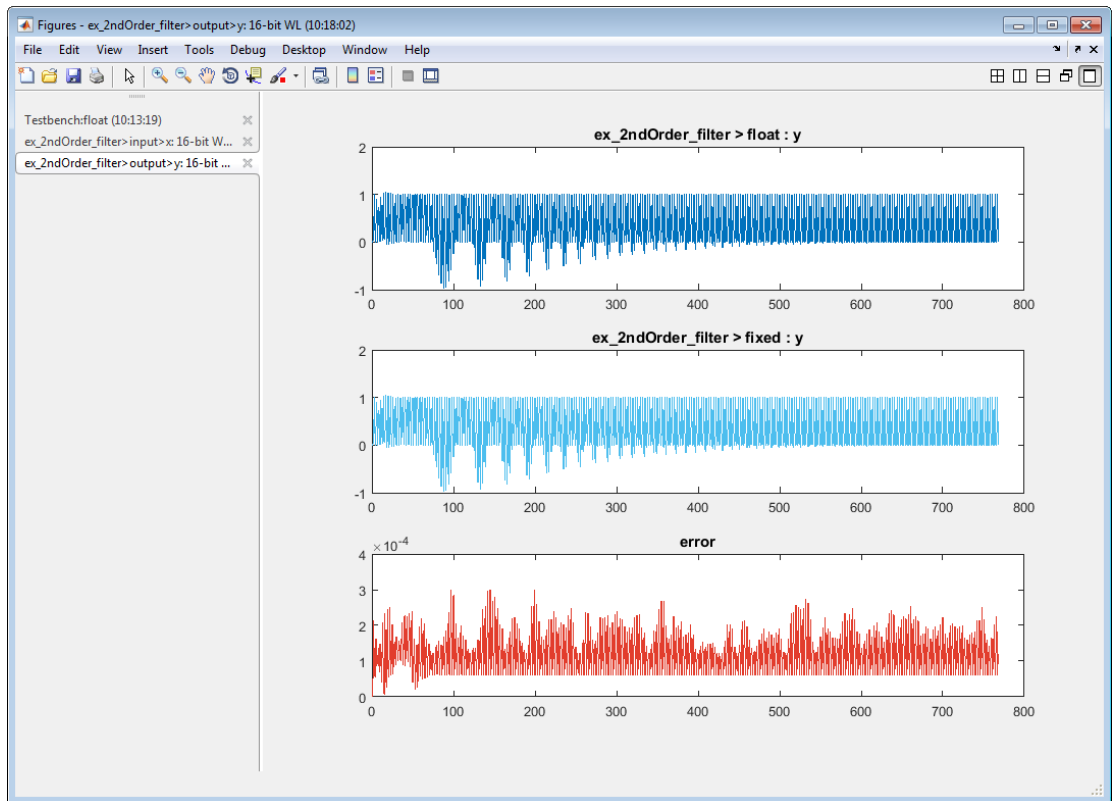
Variable	Type	Size	Signed	Word Length	Fraction Length
Input					
x	embedded.fi	1 x 256	Yes	16	14
Output					
y	embedded.fi	1 x 256	Yes	16	14
Persistent					
z	embedded.fi	2 x 1	Yes	16	15
Local					

Validation succeeded

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



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



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

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics” on page 8-23.

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

Fixed-Point Report *ex_2ndOrder_filter_fixpt*

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
    fm = get_fimath();

    persistent z
    if isempty(z)
        z = fi(zeros(2,1), 1, 16, 15, fm);
    end
    % [b,a] = butter(2, 0.25)
    b = fi([0.0976310729378175, 0.195262145875635, 0.0976310729378175], 0, 16, 18, fm);
    a = fi([
        1, -0.942809041582063, 0.3333333333333333], 1, 16, 14, fm);

    y = fi(zeros(size(x)), 1, 16, 14, fm);
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
        z(2) = fi_signed(b(3)*x(i)) - a(3) * y(i);
    end
end
```

Variable Name	Type	Sim Min	Sim Max
a	numerictype(1, 16, 14) 1 x 3	-0.94281005859375	1
b	numerictype(0, 16, 18) 1 x 3	0.09762954711914063	0.19525909423828125
i	double	1	256
x	numerictype(1, 16, 14) 1 x 256	-1	1
y	numerictype(1, 16, 14) 1 x 256	-0.9698486328125	1.0552978515625
z	numerictype(1, 16, 15) 2 x 1	-0.890869140625	0.957672119140625

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

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

Integrate Fixed-Point Code

To integrate the fixed-point version of the code into system-level simulations, generate a MEX function to accelerate the fixed-point algorithm. Call this MEX function instead of the original MATLAB algorithm.

- 1 Copy `ex_2ndOrder_filter_fixpt.m` to your local working folder.
- 2 Generate a MEX function for `ex_2ndOrder_filter_fixpt.m`. Look at the `get_fimath` function in the `ex_2ndOrder_filter_fixpt.m` file to get the `fimath`, and use the type proposal report to get fixed-point data type for input `x`.

```
fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', ...  
           'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
fiaccel ex_2ndOrder_filter_fixpt -args {fi( 0, 1, 16, 14, fm )}
```

`fiaccel` generates a MEX function, `ex_2ndOrder_filter_fixpt_mex`, in the current folder.

- 3 You can now call this MEX function in place of the original MATLAB algorithm.

Propose Data Types Based on Derived Ranges

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

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

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

- 3 Copy the `dti.m` and `dti_test.m` files to your local working folder.

It is a best practice is to create a separate test script to do pre- and post-processing, such as:

- Loading inputs.
- Setting up input values.
- Outputting test results.

Type	Name	Description
Function code	<code>dti.m</code>	Entry-point MATLAB function

Type	Name	Description
Test file	dti_test.m	MATLAB script that tests dti.m

The dti Function

The dti function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```

```
% Update State
tprod = gain_val * u_in;
u_state = y + tprod;
```

The dti_test Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10);

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);
end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

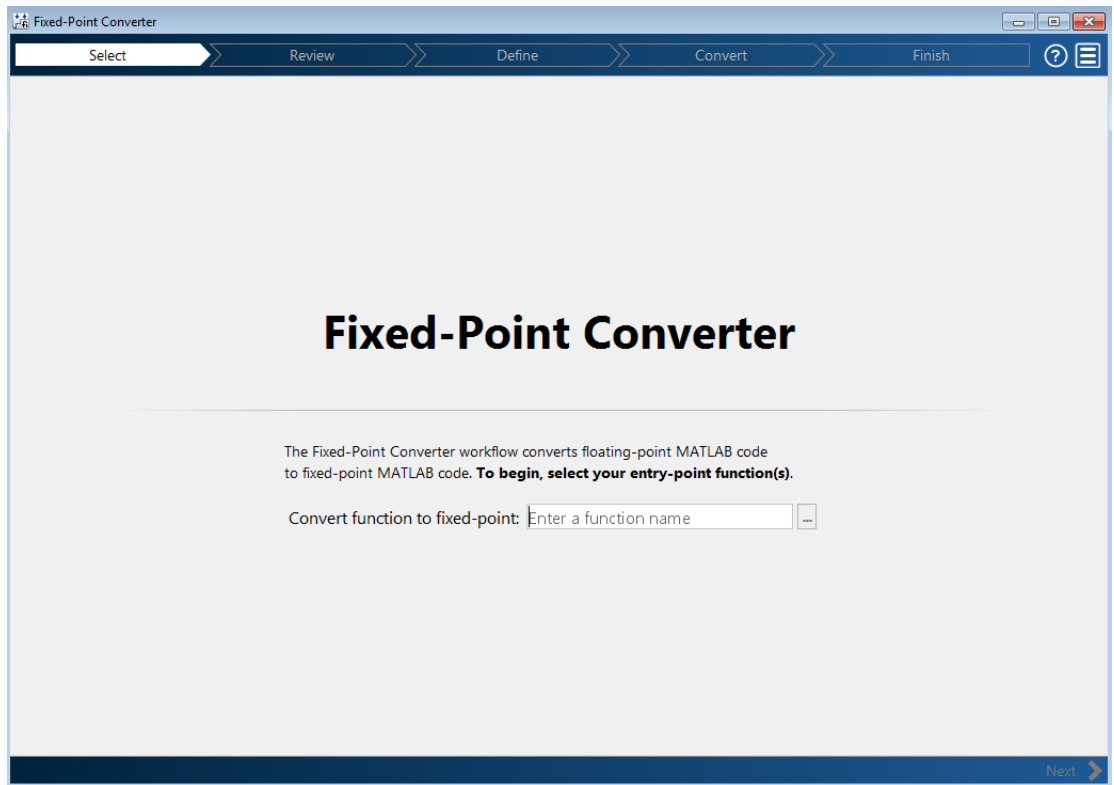
subplot(2,1,2)
plot(1:len,y_out)
xlabel('Time')
```

```
ylabel('Amplitude')
title('Output Signal (DTI)')

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

Open the Fixed-Point Converter App

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



Select Source Files

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

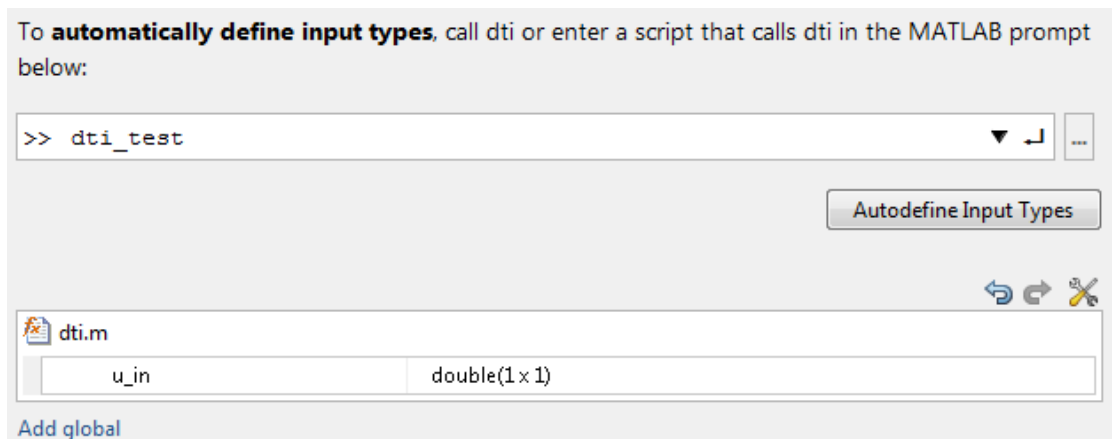
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `dti.m` for code violations and fixed-point conversion readiness issues. The app does not find issues in `dti.m`.

Define Input Types

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

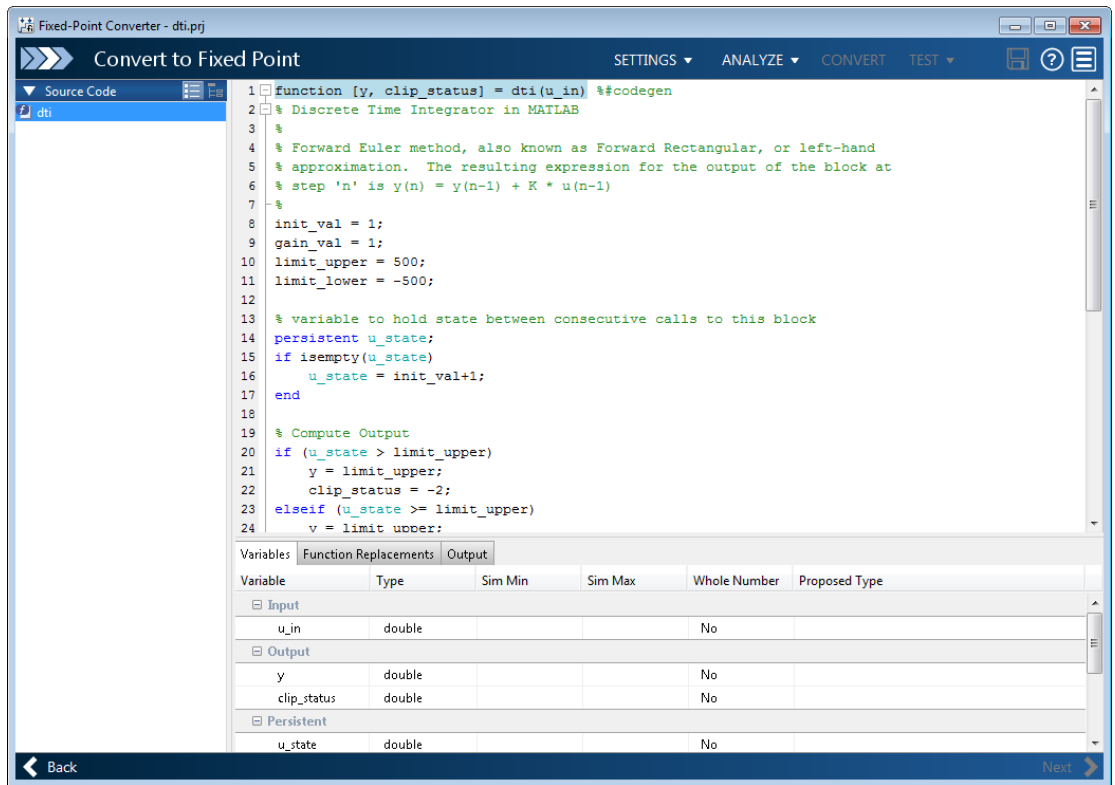
The test file runs. The app determines from the test file that the input type of `u_in` is `double(1x1)`.



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

Convert to Fixed Point

- 1 The app generates an instrumented MEX function for your entry-point MATLAB function. The app displays compiled information—type, size, and complexity—for variables in your code. For more information, see “View and Modify Variable Information” on page 9-43.

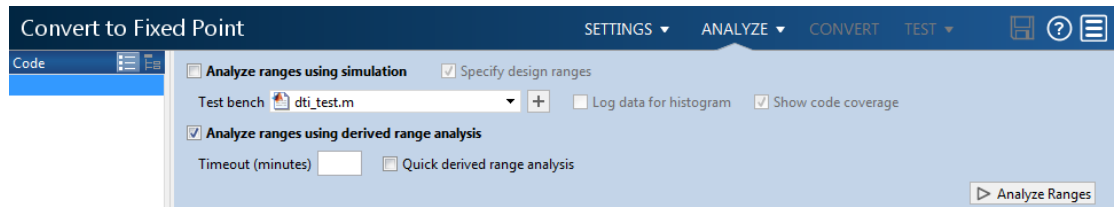


If functions are not supported for fixed-point conversion, the app displays them on the **Function Replacements** tab.

2 Click the **Analyze** arrow .

- a Select **Analyze ranges using derived range analysis**.
- b Clear the **Analyze ranges using simulation** check box.

Design ranges are required to use derived range analysis.



- 3 On the **Convert to Fixed Point** page, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to -1. Set **Static Max** to 1.

To compute derived range information, at a minimum you must specify static minimum and maximum values or proposed data types for all input variables.

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

- 4 Click **Analyze**.

Range analysis computes the derived ranges and displays them in the **Variables** tab. Using these derived ranges, the analysis proposes fixed-point types for each variable based on the default type proposal settings. The app displays them in the **Proposed Type** column.

In the `dti` function, the `clip_status` output has a minimum value of -2 and a maximum of 2.

```
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
```

```

    y = u_state;
    clip_status = 0;
end

```

When you derive ranges, the app analyzes the function and computes these minimum and maximum values for `clip_status`.

```

1 function [y, clip_status] = dti(u_in) %#codegen
2 % Discrete Time Integrator in MATLAB
3 %
4 % Forward Euler method, also known as Forward Rectangular, or left-hand
5 % approximation. The resulting expression for the output of the block at
6 % step 'n' is y(n) = y(n-1) + K * u(n-1)
7 %
8 init_val = 1;
9 gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;
15 if isempty(u_state)
16     u_state = init_val+1;

```

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


The app provides a **Quick derived range analysis** option and the option to specify a timeout in case the analysis takes a long time. See “Computing Derived Ranges” on page 8-10.

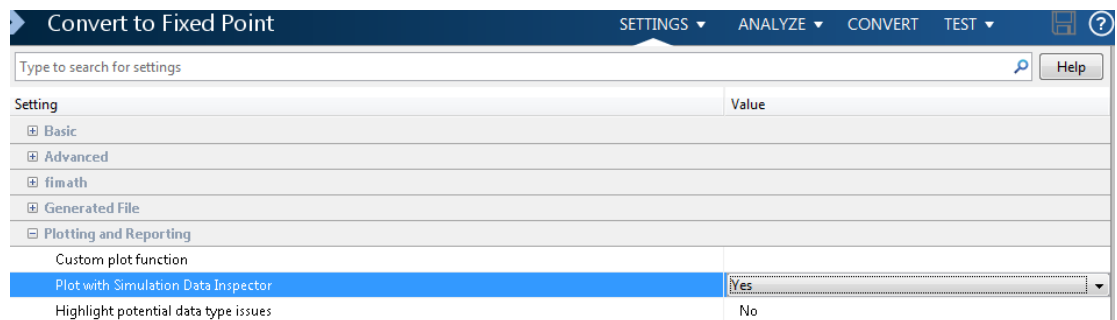
- To convert the floating-point algorithm to fixed point, click **Convert**.


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

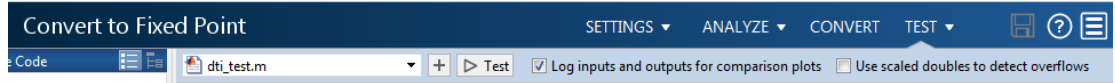
- `dti_fixpt.m` — the fixed-point version of `dti.m`.
- `dti_wrapper_fixpt.m` — this file converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during conversion. The app feeds these fixed-point values into the converted fixed-point design,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

If errors or warnings occur during validation, they show on the **Output** tab. See “Validating Types” on page 8-23.

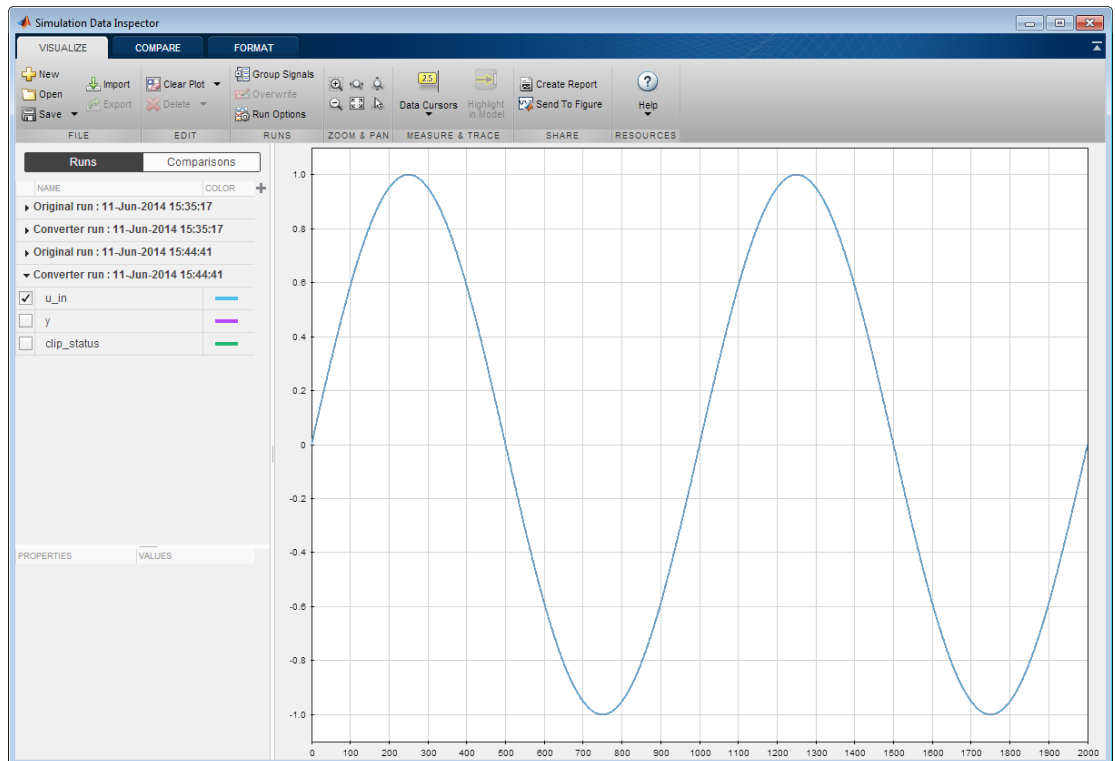
- 6 In the **Output Files** list, select `dti_fixpt.m`. The app displays the generated fixed-point code.
- 7 Use the Simulation Data Inspector to plot the floating-point and fixed-point results.
 - a Click the **Settings** arrow .
 - b Expand the **Plotting and Reporting** settings and set **Plot with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



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

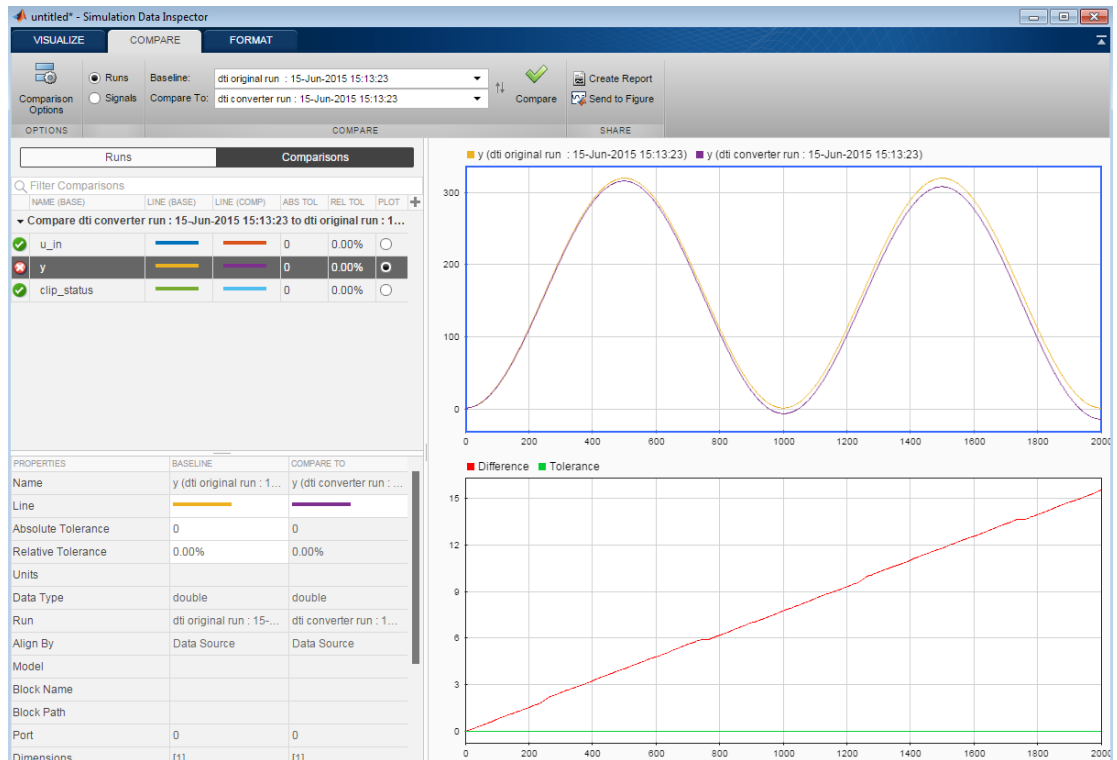


The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable y . Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.

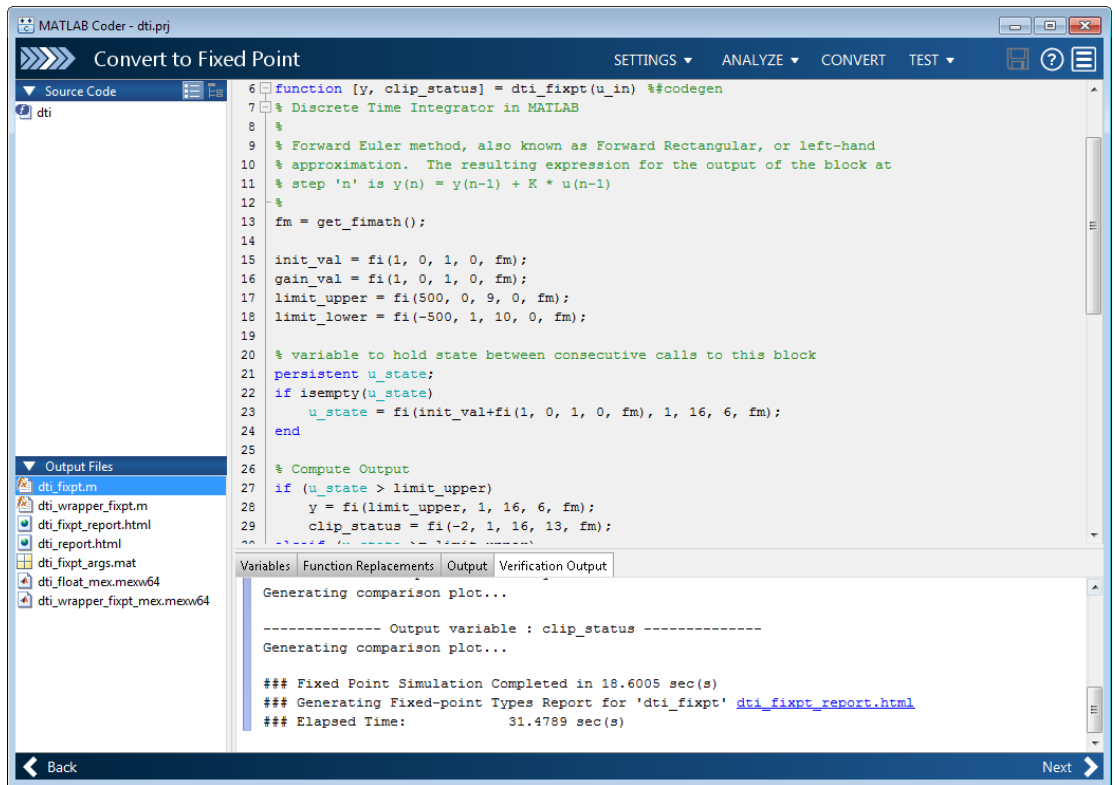


- d You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output *y*, on the **Compare** tab, select *y*. Select **Runs** and then click **Compare**.

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



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



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

```

% Compute Output
if (u_state > limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
elseif (u_state >= limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-1, 1, 16, 13, fm);
elseif (u_state < limit_lower)
    y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(2, 1, 16, 13, fm);
elseif (u_state <= limit_lower)
    y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(1, 1, 16, 13, fm);
else
    y = fi(u_state, 1, 16, 6, fm);
    clip_status = fi(0, 1, 16, 13, fm);
end

% Update State
uprod = fi(gain_val * u_in, 1, 16, 14, fm);
u_state(+) = y + uprod;
end

```

Variable Name	Type	Sim Min	Sim Max
clip_status	numerictype(1, 16, 13)	0	0
gain_val	numerictype(0, 1, 0)	1	1
init_val	numerictype(0, 1, 0)	1	1
limit_lower	numerictype(1, 10, 0)	-500	-500
limit_upper	numerictype(0, 9, 0)	500	500
uprod	numerictype(1, 16, 14)	-1	1
u_in	numerictype(1, 16, 14)	-1	1
u_state	numerictype(1, 16, 6)	-13.59375	316.28125
y	numerictype(1, 16, 6)	-13.578125	316.28125

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

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

Integrate Fixed-Point Code

To integrate the fixed-point version of the code into system-level simulations, generate a MEX function to accelerate the fixed-point algorithm. Call this MEX function instead of the original MATLAB algorithm.

- 1 Copy `dti_fixpt.m` to your local working folder.
- 2 To get the `fimath` properties for the input argument, look at the `get_fimath` function in `dti_fixpt.m`.

```

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision',...
        'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);
end

```

- 3 To get the fixed-point data type for input `u_in`, look at the type proposal report.

- 4 Generate a MEX function for `dti_fixpt.m`.

```
fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision',...  
           'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
fiaccel dti_fixpt -args {fi( 0, 1, 16, 14, fm )}
```

`fiaccel` generates a MEX function, `dti_fixpt_mex`, in the current folder.

- 5 You can now call this MEX function in place of the original MATLAB algorithm.

View and Modify Variable Information

View Variable Information

On the **Convert to Fixed Point** page of the Fixed-Point Converter app, you can view information about the variables in the MATLAB functions. To view information about the variables that you select in the **Source Code** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 8-18.

You can view the variable information:

- **Variable**

Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.

- **Type**

The original size, type, and complexity of each variable.

- **Sim Min**

The minimum value assigned to the variable during simulation.

- **Sim Max**

The maximum value assigned to the variable during simulation.

To search for a variable in the MATLAB code window and on the **Variables** tab, use Ctrl+F. The app highlights occurrences of the variable in the code.

Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. You can modify the following fields:

- **Static Min**

You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 9-46.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Static Max**

You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 9-46.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Whole Number**

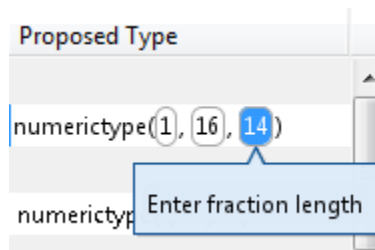
The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually:

- On the **Variables** tab, modify the value in the **ProposedType** field.



- In the code window, select a variable, and then modify the **ProposedType** field.

```
function y = fun_with_matlab(x) %#codegen
persistent z
if isempty(z)
    z = zeros(2,1);
end
% [b,a] = b...
b = [0.0976 3.1177 0.4781 75.5235];
a = [
y = zeros(size(x));
```

If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Log Data for Histogram” on page 8-20.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.

Simulation Output	Variables	Function Replacements						
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type	
Input								
x	1x256 double	-1	1				numeric(1, 16, 14)	
Output								
y	1x256 double	-0.97	1.06				numeric(1, 16, 14)	
Persistent								
z	2x1 double	-0.89	0.96				numeric(1, 16, 15)	
Local								
b	1x3 double	0.1	0.2				numeric(0, 16, 18)	
a	1x3 double	-0.94	1				numeric(1, 16, 14)	
i	double	1	256				numeric(0, 9, 0)	

- To revert the type of a selected variable to the type computed by the app, right-click the field and select **Undo changes**.
- To revert changes to variables, right-click the field and select **Undo changes for all variables**.

- To clear a static range value, right-click an edited field and select **Clear** this static range.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select **Clear all manually entered static ranges**.

Promote Sim Min and Sim Max Values

With the Fixed-Point Converter app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.

Simulation Output	Variables	Function Replacements	Static Min	Static Max	Whole Number	Decimal	Type
Input							
x	1x256 double		-1	1			type(1, 16, 14)
Output							
y	1x256 double		-0.97	1.06			type(1, 16, 14)
Persistent							

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select **Copy sim range**.
- Simulation ranges for top-level inputs, right-click the **Static Min** or **Static Max** column, and then select **Copy sim ranges for all top-level inputs**.
- Simulation ranges for persistent variables, right-click the **Static Min** or **Static Max** column, and then select **Copy sim ranges for all persistent variables**.

Replace the exp Function with a Lookup Table

This example shows how to replace the `exp` function with a lookup table approximation in fixed-point code generated using the Fixed-Point Converter app.

Prerequisites

To complete this example, you must install the following products:

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

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

Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

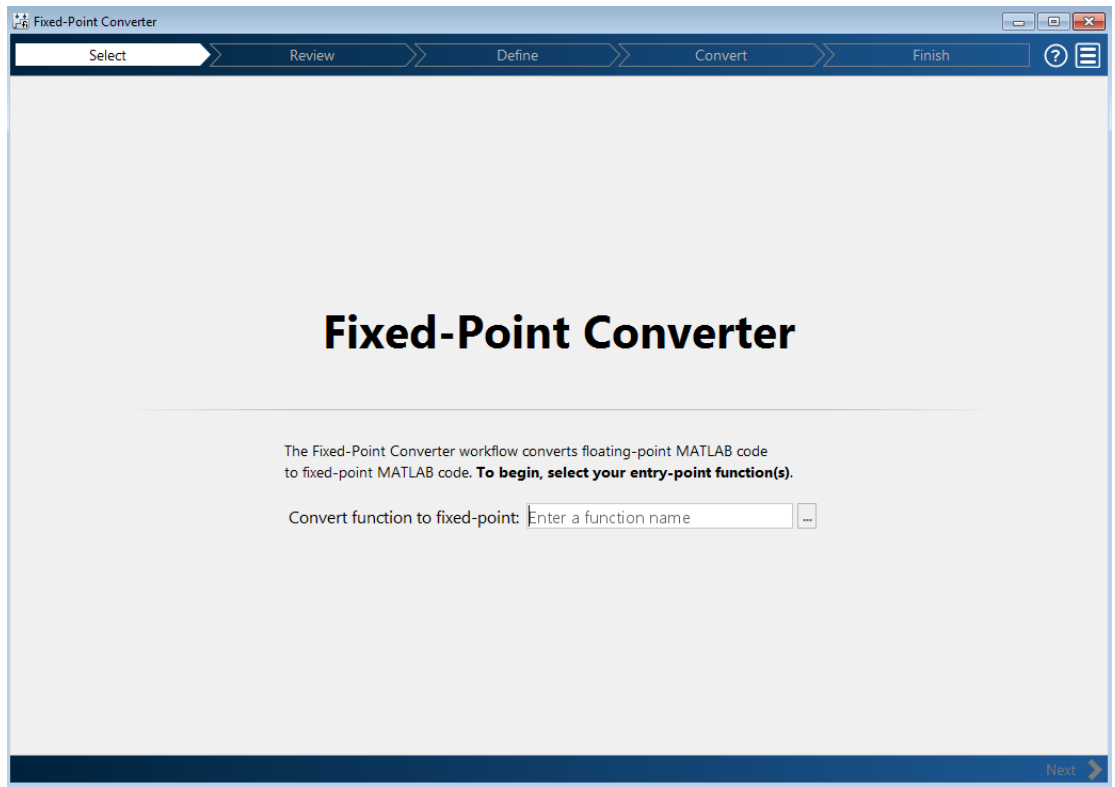
- 2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Open the Fixed-Point Converter App

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



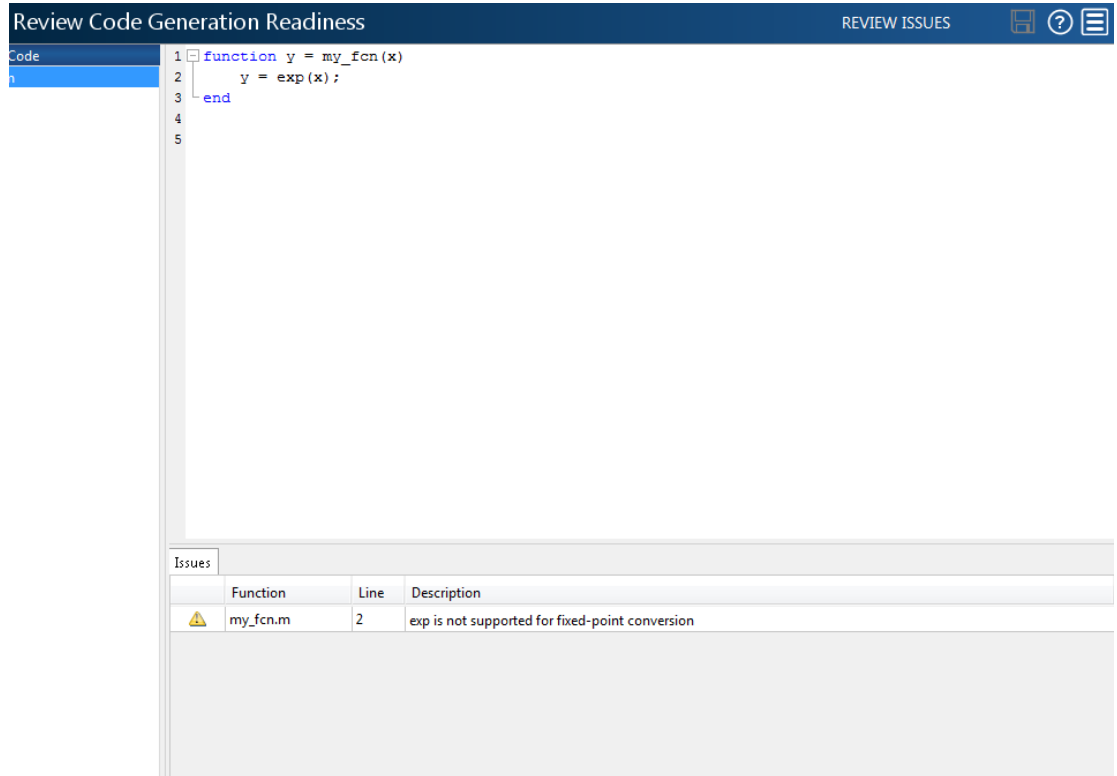
Select Source Files

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

The app screens `my_fcn.m` for code violations and fixed-point conversion readiness issues. The app opens the **Review Code Generation Readiness** page.

Review Code Generation Readiness


- 1 Click **Review Issues**. The app indicates that the `exp` function is not supported for fixed-point conversion. In a later step, you specify a lookup table replacement for this function.



The screenshot shows the 'Review Code Generation Readiness' interface. At the top, there is a dark blue header with the text 'Review Code Generation Readiness' on the left and 'REVIEW ISSUES' on the right, along with icons for a folder, a question mark, and a list. Below the header is a code editor with the following code:

```
1 function y = my_fcn(x)
2     y = exp(x);
3 end
4
5
```

Below the code editor is an 'Issues' table with the following data:

Function	Line	Description
 my_fcn.m	2	exp is not supported for fixed-point conversion

- 2 Click **Next** to go to the **Define Input Types** step.

Define Input Types

- 1 Add `my_fcn_test` as a test file and then click **Autodefine Input Types**.

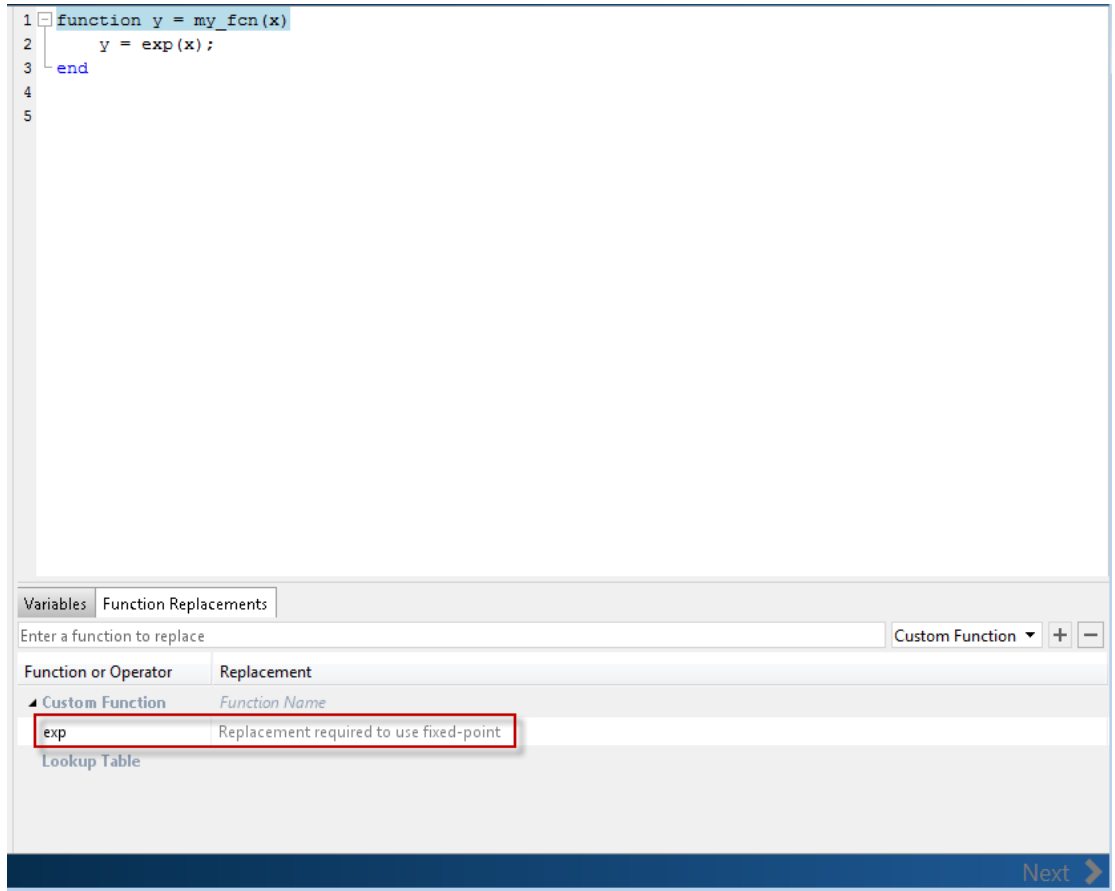
The test file runs. The app determines from the test file that `x` is a scalar double.

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

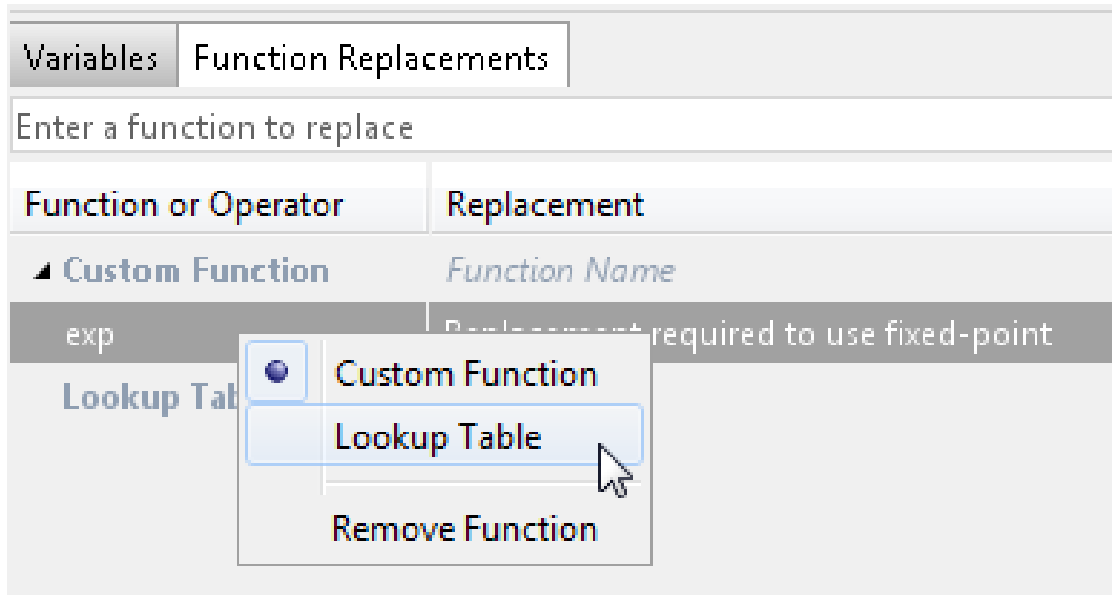
Replace exp Function with Lookup Table

- 1 Select the **Function Replacements** tab.

The app indicates that you must replace the `exp` function.



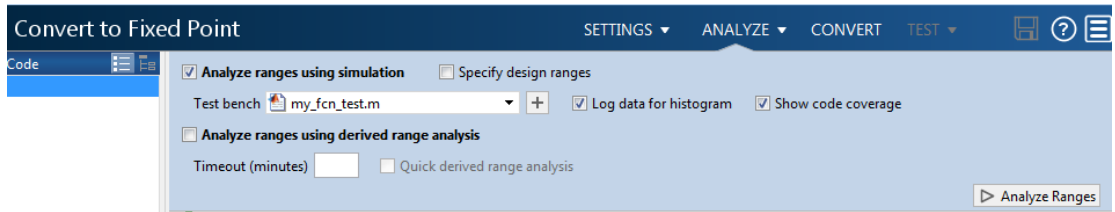
- 2 On the **Function Replacements** tab, right-click the `exp` function and select Lookup Table.



The app moves the `exp` function to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. **Design Min** and **Design Max** are set to `Auto` which means that the app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



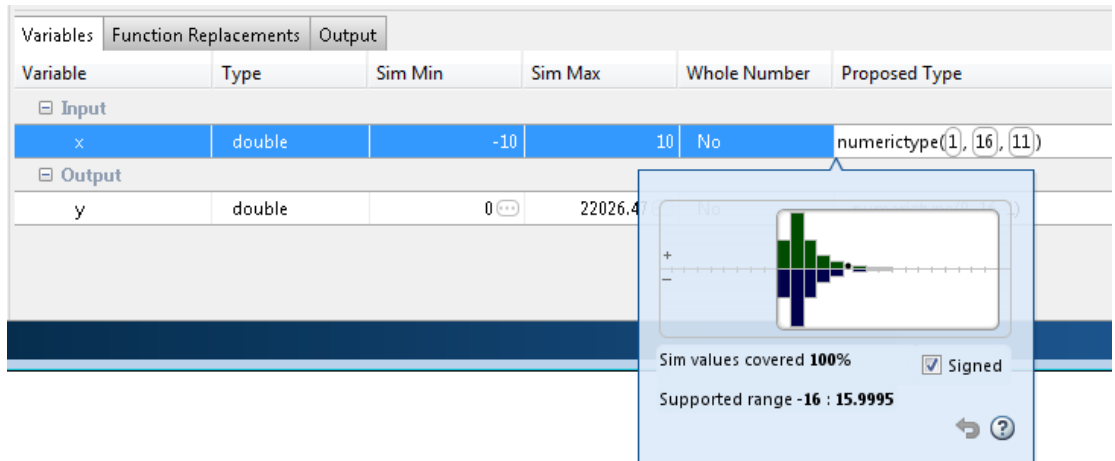
- 3 Click the **Analyze** arrow , select **Log data for histogram**, and verify that the test file is `my_fcn_test`.



4 Click **Analyze**.

The simulation runs. On the **Variables** tab, the app displays simulation minimum and maximum ranges. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The app enables the **Convert** option.

5 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The histogram provides range information and the percentage of simulation range covered by the proposed data type.



Convert to Fixed Point

1 Click **Convert**.

The app validates the proposed types, and generates a fixed-point version of the entry-point function, `my_fcn_fixpt.m`.

- In the Output Files list, select `my_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_exp`, for the `exp` function.

```

7 function y = my_fcn_fixpt(x)
8     fm = get_fimath();
9
10    y = fi(replacement_exp(x), 0, 16, 1, fm);
11 end
12
13
14 %
15 % Copyright 2017 The MathWorks, Inc.
16
17 % calculate replacement_exp via lookup table between extents x = fi([-10,10]),
18 % interpolation degree = 1, number of points = 1000
19 function y = replacement_exp ( x )
20     persistent LUT
21     if ( isempty(LUT) )
22         T = numerictype( false, 16, 1);
23         LUT = fi([4.53999297624848e-05, 4.63179964587419e-05, 4.72546280836935e-05, ...
24 4.82102000529591e-05, 4.91850953737242e-05, 5.01797047982559e-05, ...
25 5.11944269805214e-05, 5.22296686359748e-05, 5.32858447045743e-05, ...
26 5.43633785170962e-05, 5.54627019648123e-05, 5.6584255672598e-05, ...
27 5.77284891755413e-05, 5.88958610991229e-05, 6.00868393430401e-05, ...
28 6.13019012687477e-05, 6.25415338907916e-05, 6.38062340720112e-05, ...
29 6.50965087226892e-05, 6.6412875003729e-05, 6.77558605339399e-05, ...
30 6.91260036015147e-05, 7.05238533797832e-05, 7.19499701473294e-05, ...

```

Variable	Type	Size	Signed	Word Length	Fraction Length
Input					
x	embedded.fi	1x1	Yes	16	11
Output					
y	embedded.fi	1x1	No	16	1

The generated fixed-point function, `my_fcn_fixpt.m`, calls this approximation instead of calling `exp`. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

```

function y = my_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_exp(x), 0, 16, 1, fm);
end

```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table. Then, regenerate the code.

More About

- “Replacing Functions Using Lookup Table Approximations” on page 8-62

Convert Fixed-Point Conversion Project to MATLAB Scripts

This example shows how to convert a Fixed-Point Converter app project to a MATLAB script. You can use the `-tocode` option of the `fixedPointConverter` command to create a script for fixed-point conversion. You can use the script to repeat the project workflow in a command-line workflow. Before you convert the project to a script, you must complete the **Test** step of the fixed-point conversion process.

Prerequisites

This example uses the following files:

- Project file `ex_2ndOrder_filter.prj`
- Entry-point file `ex_2ndOrder_filter.m`
- Test bench file `ex_2ndOrder_filter_test.m`
- Generated fixed-point MATLAB file `ex_2ndOrder_filter_fixpt.m`

To obtain these files, complete the example “Propose Data Types Based on Simulation Ranges” on page 9-15, including the **Test** step.

Generate the Scripts

- 1 Change to the folder that contains the project file `ex_2ndOrder_filter.prj`.
- 2 Use the `-tocode` option of the `fixedPointConverter` command to convert the project to a script. Use the `-script` option to specify the file name for the script.

```
fixedPointConverter -tocode ex_2ndOrder_filter -script ex_2ndOrder_filter_script.m
```

The `fixedPointConverter` command generates a script in the current folder. `ex_2ndOrder_filter_script.m` contains the MATLAB commands to:

- Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
- Run the `fiaccl` command to convert the MATLAB function `ex_2ndOrder_filter` to the fixed-point MATLAB function `ex_2ndOrder_filter_fixpt`.

The `fiaccl` command overwrites existing files that have the same name as the generated script. If you omit the `-script` option, the `fiaccl` command returns the script in the Command Window.

Run Script That Generates Fixed-Point MATLAB Code

If you want to regenerate the fixed-point function, use the generated script.

- 1 Make sure that the current folder contains the entry-point function `ex_2ndOrder_filter.m` and the test bench file `ex_2ndOrder_filter_test.m`.
- 2 Run the script.

```
ex_2ndOrder_filter_script
```

The script generates `ex_2ndOrder_filter_fixpt.m` in the folder `codegen\ex_2ndOrder_filter\fixpt`. The variables `cfg` and `ARGS` appear in the base workspace.

See Also

`coder.FixptConfig` | `fiaccel`

Related Examples

- “Propose Data Types Based on Simulation Ranges” on page 9-15

Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the Fixed-Point Converter app.

Prerequisites

This example requires the following products:

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

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

Create Algorithm and Test Files

In a local, writable folder:

- 1 Create a MATLAB function, `custom_fcn.m` which is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

- 2 Create a wrapper function, `call_custom_fcn.m`, that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

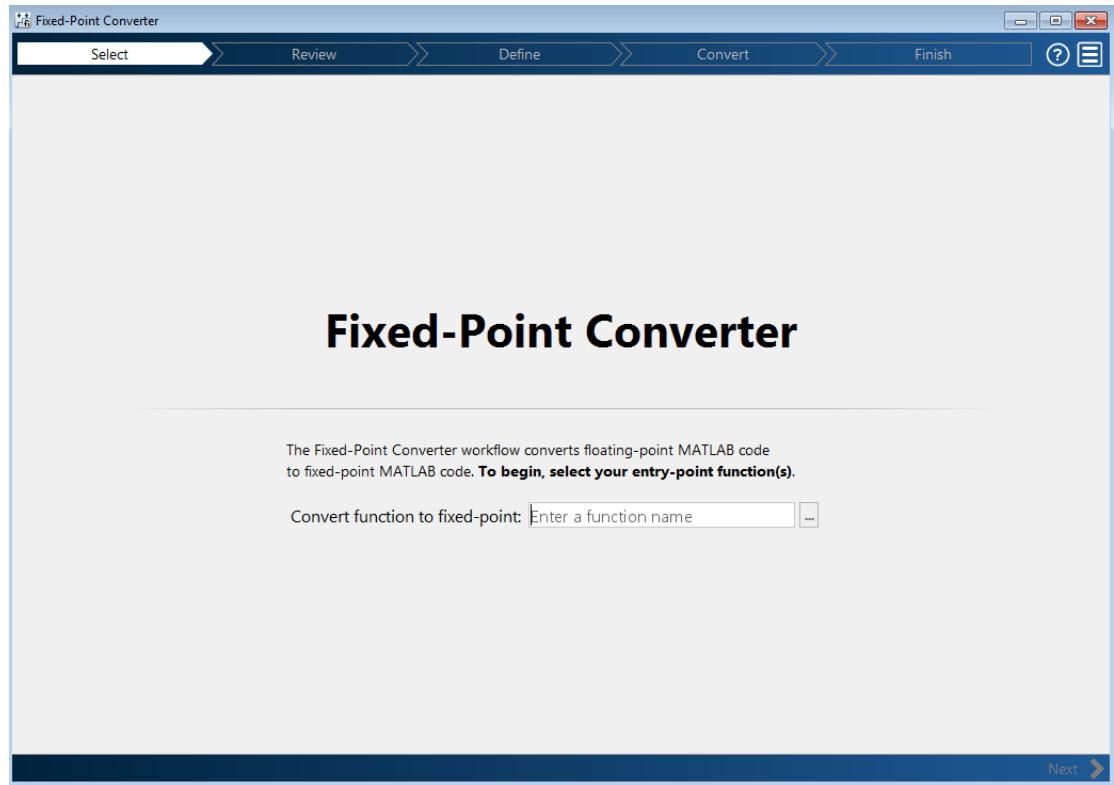
- 3 Create a test file, `custom_test.m`, that uses `call_custom_fcn`.

```
close all
clear all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Open the Fixed-Point Converter App

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



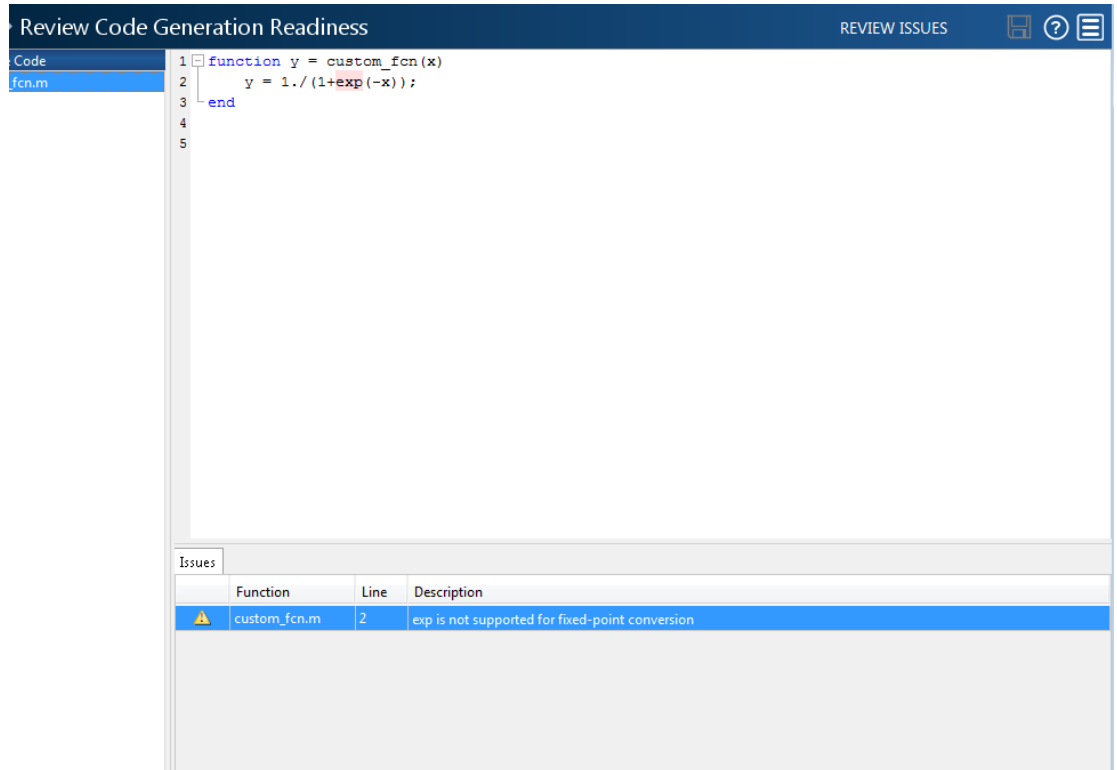
Select Source Files

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

The app screens `call_custom_fcn.m` for code violations and fixed-point conversion issues. The app opens the **Review Code Generation Readiness** page.

Review Code Generation Readiness

- 1 Click **Review Issues**. The app indicates that the `exp` function is not supported for fixed-point conversion. You can ignore this warning because you are going to replace `custom_fcn`, which is the function that calls `exp`.



- 2 Click **Next** to go to the **Define Input Types** step.

Define Input Types

- 1 Add `custom_test` as a test file and then click **Autodefine Input Types**.

The test file runs. The app determines from the test file that `x` is a scalar double.

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

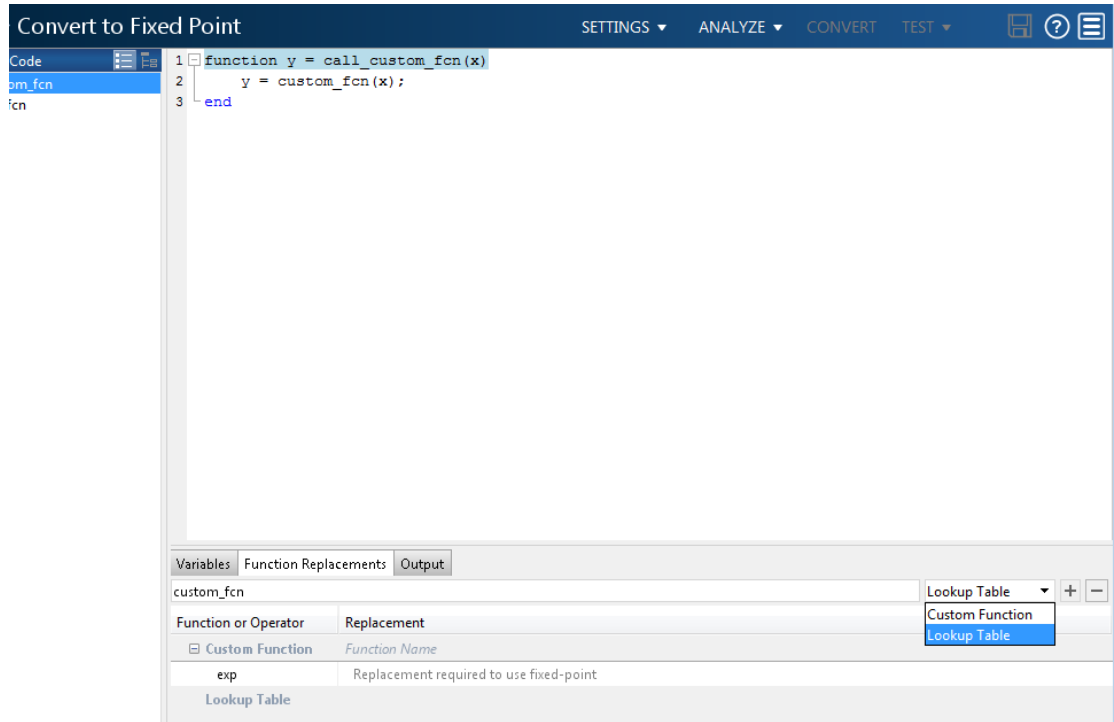
Replace custom_fcn with Lookup Table

- 1 Select the **Function Replacements** tab.

The app indicates that you must replace the `exp` function.


Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
x	double			No	
Output					
y	double			No	

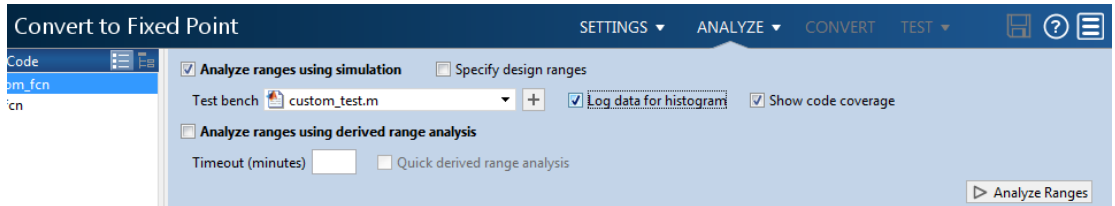
- 2 Enter the name of the function to replace, `custom_fcn`, select **Lookup Table**, and then click **+**.



The app adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. The app sets **Design Min** and **Design Max** to **Auto** which means that app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.

Function Replacements					
custom_fcn		Lookup Table + -			
Function or Operator	Replacement				
Custom Function <i>Function Name</i>					
exp	Replacement required to use fixed-point				
Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points	
custom_fcn	Linear	Auto	Auto	1000	

- Click the **Analyze** arrow , select **Log data for histogram**, and verify that the test file is `call_custom_test`.



- Click **Analyze**.

The simulation runs. The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Convert** option is now enabled.

- Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The histogram provides range information and the percentage of simulation range covered by the proposed data type.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 16, 11)
Output							
y	double	0	1				

Convert to Fixed Point

- Click **Convert**.

The app validates the proposed types and generates a fixed-point version of the entry-point function, `call_custom_fcn_fixpt.m`.

The screenshot shows the MATLAB IDE interface. On the left, the 'Source Code' pane displays the file structure with 'call_custom_fcn' and 'custom_fcn' selected. Below it, the 'Output Files' pane lists generated files, with 'call_custom_fcn_fixpt.m' highlighted. The main editor window shows the source code for three functions: `call_custom_fcn_fixpt`, `custom_fcn`, and `divideType`. Below the code, a table summarizes the function's variables and their properties.

```

7 function y = call_custom_fcn_fixpt(x)
8     fm = get_fimath();
9
10    y = fi(replacement_custom_fcn(x), 0, 16, 16, fm);
11 end
12
13 function y = custom_fcn(x)
14     fm = get_fimath();
15
16    y = fi(fi_dotdiv(fi(1, 0, 1, 0, fm), (fi(1, 0, 1, 0, fm)+exp(fi_uminus(x)))), 0, 16, 16,
17 end
18
19
20
21 function ntype = divideType(a,b)
22     coder.inline( 'always' );
23     nt1 = numerictype( a );
24     nt2 = numerictype( b );
25     maxFL = max( [ min( nt1.WordLength, nt1.FractionLength ), min( nt2.WordLength, nt2.Fracti
26     FL = max( maxFL, 24 );
27     extraBits = (FL - maxFL);
28     WL = nt1.WordLength + nt2.WordLength;
29     WL = min( WL, 124 );
    
```

Variable	Function Replacements	Output	Type	Size	Signed	Word Length	Fraction Length
Input							
x			embedded.fi	1x1	Yes	16	11
Output							
y			embedded.fi	1x1	No	16	16

- 2 In the Output Files list, select `call_custom_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```

function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_custom_fcn(x), 0, 16, 16, fm);
end
    
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

More About

- “Replacing Functions Using Lookup Table Approximations” on page 8-62

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the Fixed-Point Converter app to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the **Log inputs and outputs for comparison plots** option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```
- 3 Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

It is a best practice is to create a separate test script to do pre- and post-processing, such as:

- Loading inputs.

- Setting up input values.
- Outputting test results.

For more information, see “Create a Test File” on page 12-4.

Type	Name	Description
Function code	myFilter.m	Entry-point MATLAB function
Test file	myFilterTest.m	MATLAB script that tests myFilter.m
Plotting function	plotDiff.m	Custom plot function
MAT-file	filterData.mat	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
    b = complex(zeros(1,16));
    h = complex(zeros(1,16));
    h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;
```



```

for idx = 1:4000
    y = myFilter(d(idx));
end

```

The plotDiff Function

```

% varInfo - structure with information about the variable. It has the following fields
%         i) name
%         ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%             Fixed-Point Conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexprep(varName, '_', '\\_');
    escapedFcnName = regexprep(fcnName, '_', '\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

```

```
        % plot fixed point values
        y_vec = flatFixedVals;
        subplot(1, 2, 2);
        plotScatter(x_vec, y_vec, 100, fixedTitle);

    otherwise
        % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

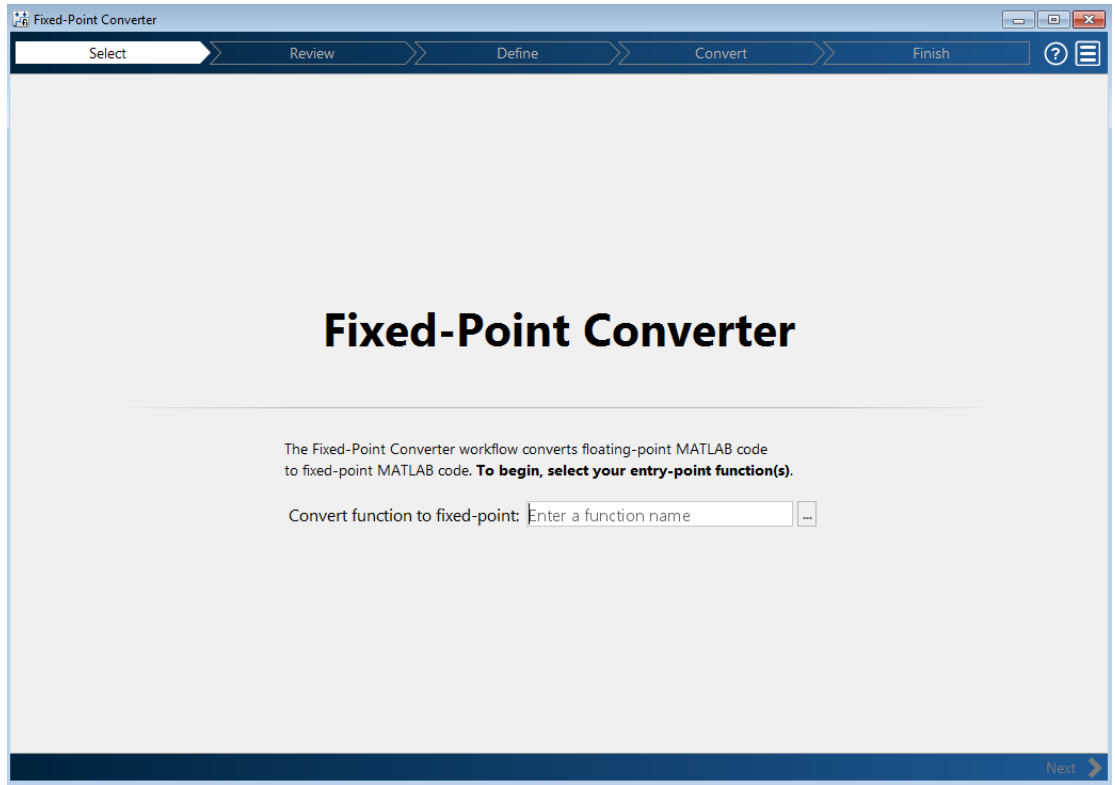
    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Open the Fixed-Point Converter App

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



Select Source Files

- 1 To add the entry-point function `myFilter` to the project, browse to the file `myFilter.m`, and then click **Open**.

By default, the app saves information and settings for this project in the current folder in a file named `myFilter.prj`.

- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `myFilter.m` for code violations and fixed-point conversion readiness issues. The app does not find issues in `myFilter.m`.

Define Input Types

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

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

To **automatically define input types**, call `myFilter` or enter a script that calls `myFilter` in the MATLAB prompt below:

The screenshot shows the MATLAB prompt with the command `>> myFilterTest`. Below the prompt, there is a table with one row: `in` and `complex(double(1 x 1))`. A button labeled "Autodefine Input Types" is visible to the right of the prompt. At the bottom left, there is a link "Add global".

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

Convert to Fixed Point

- 1 The app generates an instrumented MEX function for your entry-point MATLAB function. The app displays compiled information for variables in your code. For more information, see “View and Modify Variable Information” on page 9-43.



The screenshot shows the 'Convert to Fixed Point' dialog box. The code editor contains the following MATLAB code:

```

1 function [y, ho] = myFilter(in)
2
3 persistent b h;
4 if isempty(b)
5     b = complex(zeros(1,16));
6     h = complex(zeros(1,16));
7     h(8) = 1;
8 end
9
10 b = [in, b(1:end-1)];
11 y = b*h.';
12
13 errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
14 update = 0.001*conj(b)*y*errf;
15
16 h = h + update;
17 h(8) = 1;
18 ho = h;
19
20 end
    
```

Below the code editor is a table with the following data:

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
in	complex double			No	
Output					
y	complex double			No	
ho	1 x 16 complex...			No	
Persistent					
b	1 x 16 complex...			No	

- 2 To open the settings dialog box, click the **Settings** arrow .
 - a Verify that **Default word length** is set to 16.
 - b Under **Advanced**, set **Signedness** to Signed
 - c Under **Plotting and Reporting**, set **Custom plot function** to plotDiff.
- 3 Click the **Analyze** arrow . Verify that the test file is myFilterTest.
- 4 Click **Analyze**.

The test file, myFilterTest, runs and the app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

Convert to Fixed Point

SETTINGS ANALYZE CONVERT TEST

```

1 function [y, ho] = myFilter(in)
2
3 persistent b h;
4 if isempty(b)
5     b = complex(zeros(1,16));
6     h = complex(zeros(1,16));
7     h(8) = 1;
8 end
9
10 b = [in, b(1:end-1)];
11 y = b*h.';
12
13 errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
14 update = 0.001*conj(b)*y*errf;
15
16 h = h + update;
17 h(8) = 1;
18 ho = h;
19
20 end
    
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
in	complex double	-0.95	0.95	No	numerictype(1, 16, 15)
Output					
y	complex double	-0.95	0.95	No	numerictype(1, 16, 15)
ho	1 x 16 complex...	-0.13	1	No	numerictype(1, 16, 14)
Persistent					
b	1 x 16 complex...	-0.95	0.95	No	numerictype(1, 16, 15)

Next

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

The software validates the proposed types and generates a fixed-point version of the entry-point function.

```

7 function [y, ho] = myFilter_fixpt(in)
8
9 fm = get_fimath();
10
11 persistent b h;
12 if isempty(b)
13     b = fi(complex(zeros(1,16)), 1, 16, 15, fm);
14     h = fi(complex(zeros(1,16)), 1, 16, 14, fm);
15     h(8) = 1;
16 end
17
18 b(:) = [fi(in, 1, 16, 15, fm), b(1:end-1)];
19 y = fi(b*h.', 1, 16, 15, fm);
20
21 errf = fi(fi_signed(fi(1, 1, 2, 0, fm))-sqrt(real(y)*real(y) + imag(y)*imag(y)), 1, 16, 14, fm);
22 update = fi(fi(0.001, 1, 16, 24, fm)*conj(b)*y*errf, 1, 16, 25, fm);
23
24 h(:) = h + update;
25 h(8) = 1;
26 ho = fi(h, 1, 16, 14, fm);
27
28 end
29


```

Variable	Type	Size	Signed	Word Length	Fraction Length
Input					
in	embedded.fi	1 x 1	Yes	16	15
Output					
y	embedded.fi	1 x 1	Yes	16	15
ho	embedded.fi	1 x 16	Yes	16	14
Persistent					
b	embedded	16	Yes	16	15

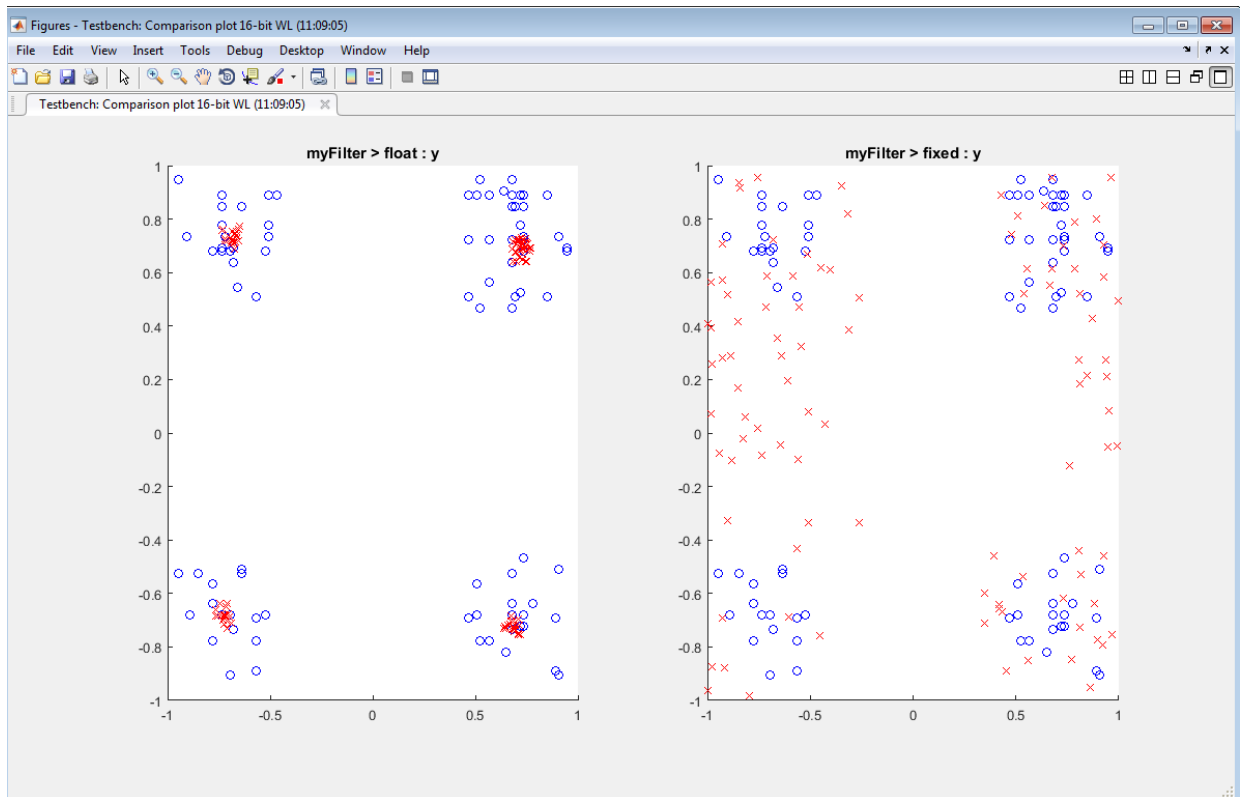
Validation succeeded

Next

Test Numerics and View Comparison Plots

- 1 Click **Test** arrow , select **Log inputs and outputs for comparison plots**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the app uses this function to generate the comparison plot. The plot shows that the fixed-point results do not closely match the floating-point results.

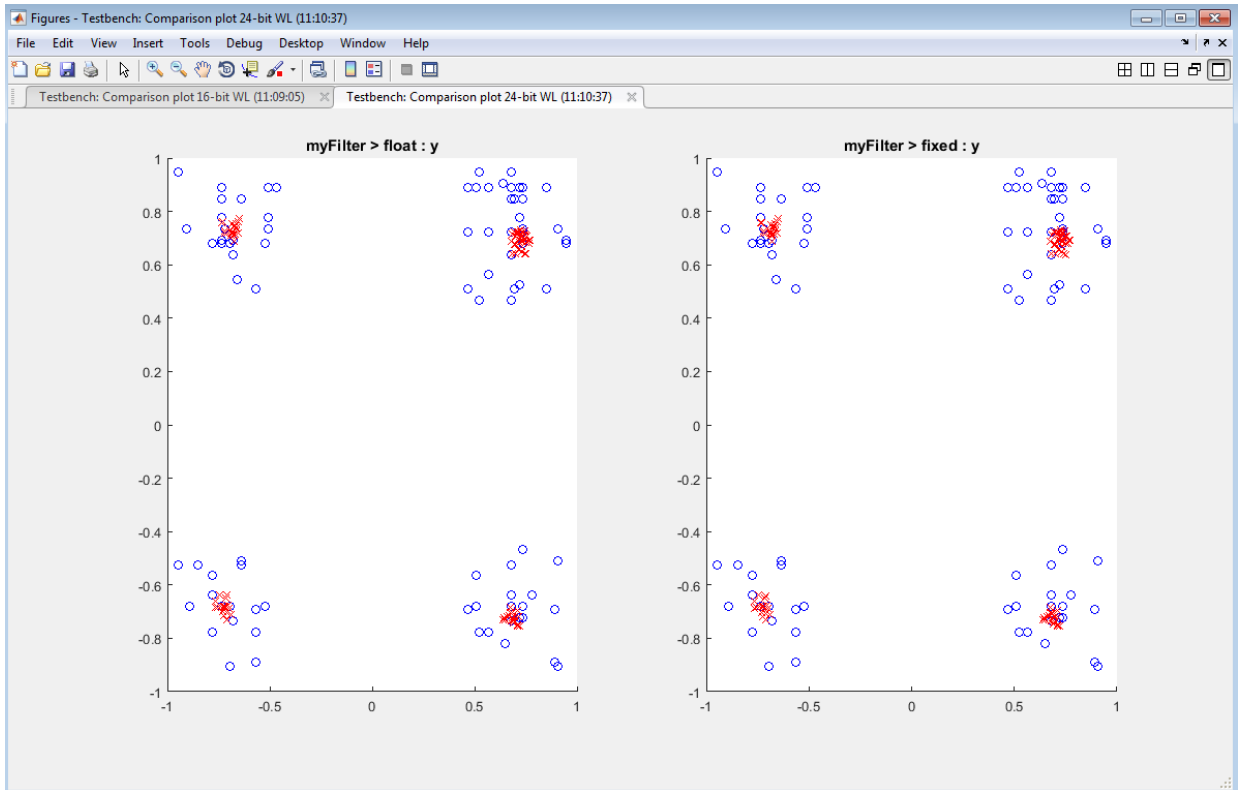


- 2 In the settings, increase the **DefaultWordLength** to 24 and then convert to fixed point again.

The app converts `myFilter.m` to fixed point and proposes fixed-point data types using the new default word length.

- 3 Run the test numerics step again.

The increased word length improves the results. This time, the plot shows that the fixed-point results match the floating-point results.




More About

- “Custom Plot Functions” on page 8-63

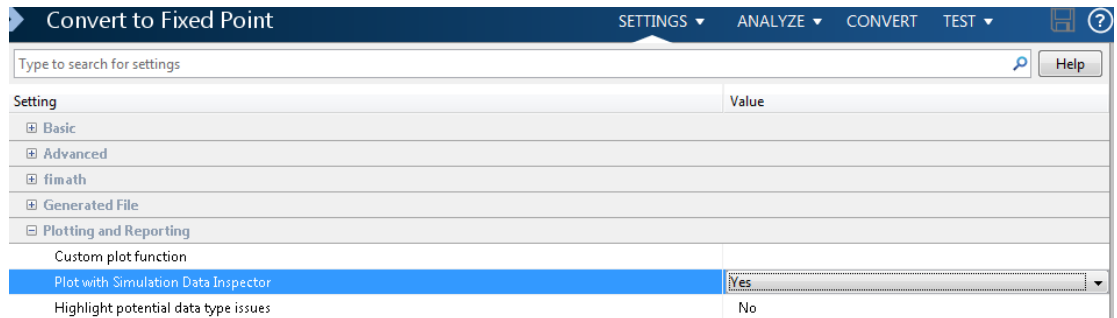
Enable Plotting Using the Simulation Data Inspector


You can use the Simulation Data Inspector with the Fixed-Point Converter app to inspect and compare floating-point and fixed-point logged input and output data.

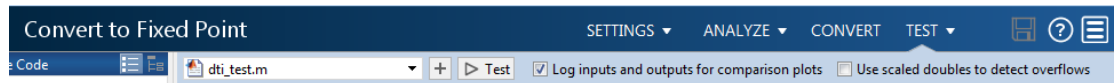
- 1 On the **Convert to Fixed Point** page,

Click the **Settings** arrow .

- 2 Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to **Yes**.



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



For an example, see “Propose Data Types Based on Derived Ranges” on page 9-29.

More About

- “Inspecting Data Using the Simulation Data Inspector” on page 13-2

Add Global Variables by Using the App

To add global variables to the project:

- 1 On the **Define Input Types** page, automatically define input types or click **Let me enter input or global types directly**.

The app displays a table of entry-point inputs.

- 2 To add a global variable, click **Add global**.

By default, the app names the first global variable in a project **g**, and subsequent global variables **g1**, **g2**, and so on.

- 3 Under **Global variables**, enter a name for the global variable.
- 4 After adding a global variable, but before generating code, specify its type and initial value. Otherwise, you must create a variable with the same name in the global workspace. See “Specify Global Variable Type and Initial Value Using the App” on page 9-96.

Automatically Define Input Types by Using the App

If you specify a test file that calls the project entry-point functions, the Fixed-Point Converter app can infer the input argument types by running the test file. If a test file calls an entry-point function multiple times with different size inputs, the app takes the union of the inputs. The app infers that the inputs are variable size, with an upper bound equal to the size of the largest input.

Before using the app to automatically define function input argument types, you must add at least one entry-point file to your project. You must also specify code that calls your entry-point functions with the expected input types. It is a best practice to provide a test file that calls your entry-point functions. The test file can be either a MATLAB function or a script. The test file must call the entry-point function at least once.

To automatically define input types:

- 1 On the **Define Input Types** page, specify a test file. Alternatively, you can enter code directly.
- 2 Click **Autodefine Input Types**.

The app runs the test file and infers the types for entry-point input arguments. The app displays the inferred types.

Note: If you automatically define the input types, the entry-point functions must be in a writable folder.

If your test file does not call an entry-point function with different size inputs, the resulting type dimensions are fixed-size. After you define the input types, you can specify and apply rules for making type dimensions variable-size when they meet a size threshold. See “Make Dimensions Variable-Size When They Meet Size Threshold” (MATLAB Coder).

Define Constant Input Parameters Using the App

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter name.
- 3 Select **Define Constant**.
- 4 In the field to the right of the parameter name, enter the value of the constant or a MATLAB expression that represents the constant.

The app uses the value of the specified MATLAB expression as a compile-time constant.

Define or Edit Input Parameter Type by Using the App

In this section...
“Define or Edit an Input Parameter Type” on page 9-80
“Specify an Enumerated Type Input Parameter by Type” on page 9-81
“Specify a Fixed-Point Input Parameter by Type” on page 9-82
“Specify a Structure Input Parameter” on page 9-82
“Specify a Cell Array Input Parameter” on page 9-83

Define or Edit an Input Parameter Type

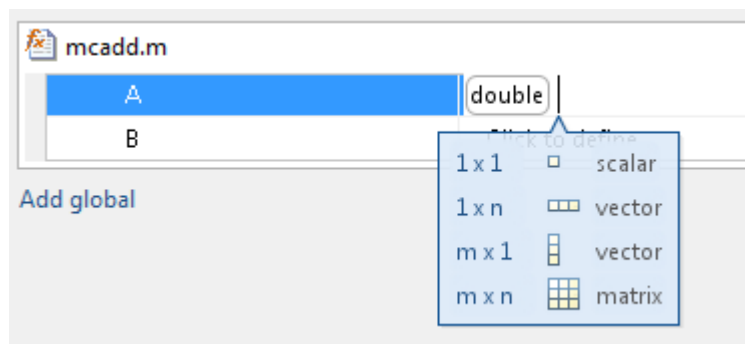
The following procedure shows you how to define or edit `double`, `single`, `int64`, `int32`, `int16`, `int8`, `uint64`, `uint32`, `uint16`, `uint8`, `logical`, and `char` types.

For more information about defining other types, see the following table.

Input Type	Link
A structure (struct)	“Specify a Structure Input Parameter” on page 9-82
A cell array (cell (Homogeneous) or cell (Heterogeneous))	“Specify a Cell Array Input Parameter” on page 9-83
A fixed-point data type (embedded.fi)	“Specify a Fixed-Point Input Parameter by Type” on page 9-82
An input by example (Define by Example)	“Define Input Parameter by Example by Using the App” on page 9-88
A constant (Define Constant)	“Define Constant Input Parameters Using the App” on page 9-79

- 1 Click the field to the right of the input parameter name.
- 2 Optionally, for numeric types, to make the parameter a complex type, select the **Complex number** check box.
- 3 Select the input type.

The app displays the selected type. It displays the size options.



- 4 From the list, select whether your input is a scalar, a $1 \times n$ vector, a $m \times 1$ vector, or a $m \times n$ matrix. By default, if you do not select a size option, the app defines inputs as scalars.
- 5 Optionally, if your input is not scalar, enter sizes m and n . You can specify:
 - Fixed size, for example, 10.
 - Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to 10, enter `:10`.
 - Unbounded variable size by entering `:Inf`.

You can edit the size of each dimension.

Specify an Enumerated Type Input Parameter by Type

To specify that an input uses the enumerated type `MyColors`:

- 1 Suppose that the enumeration `MyColors` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

- 2 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 3 In the field to the right of the input parameter, enter `MyColors`.


Specify a Fixed-Point Input Parameter by Type

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **embedded.fi**.
- 4 Select the size. If you do not specify the size, the size defaults to 1x1.
- 5 Specify the input parameter `numerictype` and `fimath` properties.

If you do not specify a local `fimath`, the app uses the default `fimath`. See “Default `fimath` Usage to Share Arithmetic Rules” on page 4-23.

To modify the `numerictype` or `fimath` properties, open the properties dialog box. To open the properties dialog box, click to the right of the fixed-point type definition.

Optionally, click .

Specify a Structure Input Parameter

When a primary input is a structure, the app treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:

- For each field of an input structure, specify class, size, and complexity.
- For each field that is a fixed-point class, also specify `numerictype`, and `fimath`.

Specify Structures by Type

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **struct**.

The app displays the selected type, `struct`. The app displays the size options.

- 4 Specify that your structure is a scalar, $1 \times n$ vector, $m \times 1$ vector, or $m \times n$ matrix. By default, if you do not select a size option, the app defines inputs as scalars.
- 5 If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size 10:

- To specify fixed size, select 10.
 - To specify variable size with an upper bound of 10, select :10.
 - To specify unbounded variable size, select :Inf.
- 6 Add fields to the structure. Specify the class, size, and complexity of the fields. See “Add a Field to a Structure” on page 9-83.

Rename a Field in a Structure

Select the name field of the structure that you want to rename. Enter the new name.

Add a Field to a Structure

- 1 To the right of the structure, click **+**
- 2 Enter the field name. Specify the class, size, and complexity of the field.

Insert a Field into a Structure

- 1 Select the structure field below which you want to add another field.
 - 2 Right-click the structure field.
 - 3 Select **Insert Field Below**.
- The app adds the field after the field that you selected.
- 4 Enter the field name. Specify the class, size, and complexity of the field.

Remove a Field from a Structure

- 1 Right-click the field that you want to remove.
- 2 Select **Remove Field**.

Specify a Cell Array Input Parameter

Note: The Fixed-Point Converter app does not support cell arrays.

For code generation, cell arrays are homogeneous or heterogeneous. . A homogeneous cell array is represented as an array in the generated code. All elements have the same properties. A heterogeneous cell array is represented as a structure in the generated code. Elements can have different properties.

Specify a Homogeneous Cell Array

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **cell (Homogeneous)**.

The app displays the selected type, `cell`. The app displays the size options.

- 4 From the list, select whether your input is a scalar, a $1 \times n$ vector, a $m \times 1$ vector, or a $m \times n$ matrix. By default, if you do not select a size option, the app defines inputs as scalars.
- 5 If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size 10:
 - To specify fixed size, select 10.
 - To specify variable size with an upper bound of 10, select :10.
 - To specify unbounded variable size, select :Inf.

Below the cell array variable, a colon inside curly braces `{:}` indicates that the cell array elements have the same properties (class, size, and complexity).

- 6 To specify the class, size, and complexity of the elements in the cell array, click the field to the right of `{:}`.

Specify a Heterogeneous Cell Array

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **cell (Heterogeneous)**.

The app displays the selected type, `cell`. The app displays the size options.


- 4 Specify that your structure is a scalar, $1 \times n$ vector, $m \times 1$ vector, or $m \times n$ matrix. By default, if you do not select a size option, the app defines inputs as scalars.
- 5 Optionally, if your input is not scalar, enter sizes m and n . A heterogeneous cell array is fixed size.

The app lists the cell array elements. It uses indexing notation to specify each element. For example, `{1,2}` indicates the element in row 1, column 2.

- 6 Specify the class, size, and complexity for each cell array element.
- 7 Optionally, add elements. See “Add an Element to a Heterogeneous Cell Array” on page 9-87

Set Structure Properties for a Heterogeneous Cell Array

A heterogeneous cell array is represented as a structure in the generated code. You can specify the properties for the structure that represents the cell array.

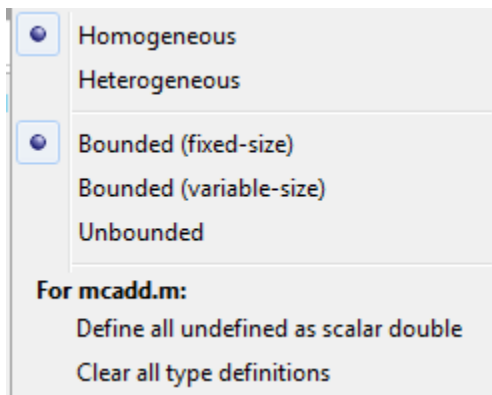
- 1 Click to the right of the cell array definition. Optionally click .
- 2 In the dialog box, specify properties for the structure in the generated code.

Property	Description
C type definition name	Name for the structure type in the generated code.
Type definition is externally defined	<p>Default: No — type definition is not externally defined.</p> <p>If you select Yes to declare an externally defined structure, the app does not generate the definition of the structure type. You must provide it in a custom include file.</p> <p>Dependency: C type definition name enables this option.</p>
C type definition header file	<p>Name of the header file that contains the external definition of the structure, for example, "mystruct.h". Specify the path to the file using the Additional include directories parameter on the project settings dialog box Custom Code tab.</p> <p>By default, the generated code contains #include statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation</p>

Property	Description
	<p>fails. If you specify the C type definition header file, the app includes that header file exactly at the point where it is required.</p> <p>Dependency: When Type definition is externally defined is set to Yes, this option is enabled.</p>
Data alignment boundary	<p>The run-time memory alignment of structures of this type in bytes.</p> <p>Alignment must be either - 1 or a power of 2 that is no more than 128.</p> <p>Default: 0</p> <p>Dependency: When Type definition is externally defined is set to Yes, this option is enabled.</p>

Change Classification as Homogeneous or Heterogeneous

To change the classification as homogeneous or heterogeneous, right-click the variable. Select **Homogeneous** or **Heterogeneous**.



The app clears the definitions of the elements.

Change the Size of the Cell Array

- 1 In the definition of the cell array, click a dimension. Specify the size.
- 2 For a homogeneous cell array, specify whether the dimension is variable size and whether the dimension is bounded or unbounded. Alternatively, right-click the variable. Select **Bounded (fixed-size)**, **Bounded (variable-size)**, or **Unbounded**.
- 3 For a heterogeneous cell array, the app adds elements so that the cell array has the specified size and shape.

Add an Element to a Heterogeneous Cell Array

- 1 In the definition of the cell array, click a dimension. Specify the size. For example, enter 1 for the first dimension and 4 for the second dimension.

The app adds elements so that the cell array has the specified size and shape. For example for a 1x4 heterogeneous cell array, the app lists four elements: {1, 1}, {1, 2}, {1, 3}, and {1, 4}.

- 2 Specify the properties of the new elements.

Define Input Parameter by Example by Using the App

In this section...

“Define an Input Parameter by Example” on page 9-88

“Specify Input Parameters by Example” on page 9-89

“Specify a Structure Type Input Parameter by Example” on page 9-90

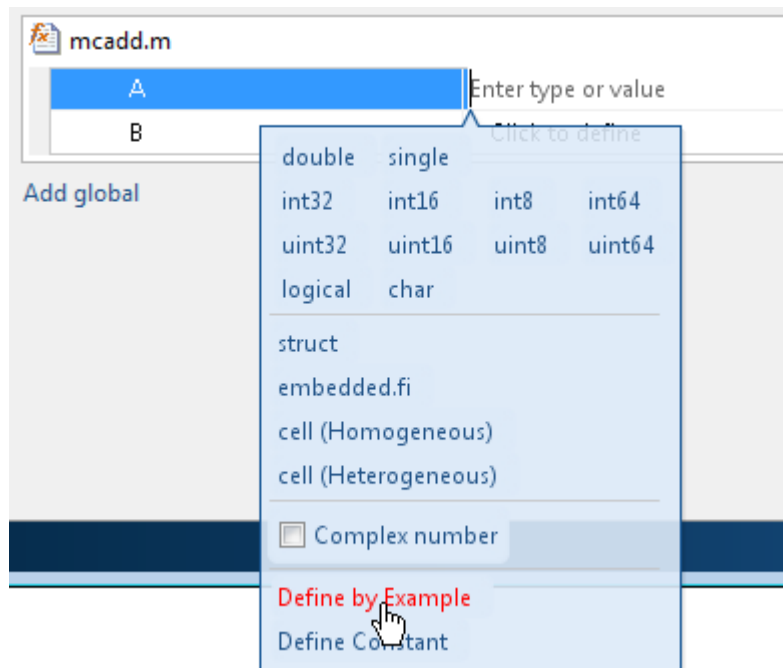
“Specify a Cell Array Type Input Parameter by Example” on page 9-91

“Specify an Enumerated Type Input Parameter by Example” on page 9-93

“Specify a Fixed-Point Input Parameter by Example” on page 9-94

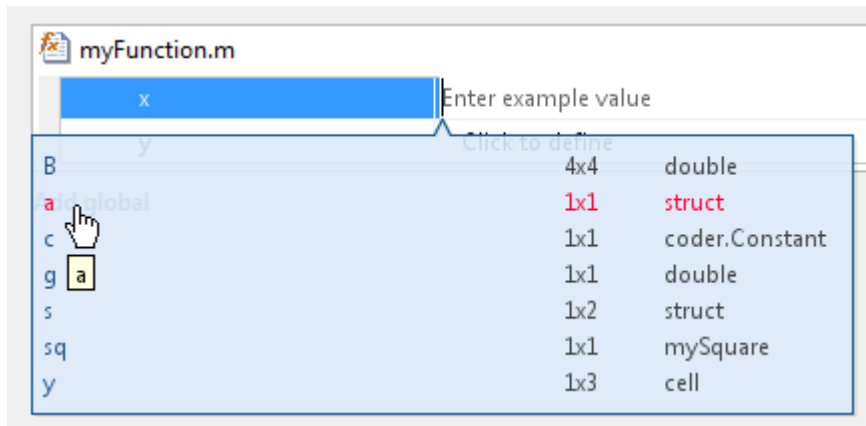
Define an Input Parameter by Example

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.



- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter a MATLAB expression. The variable has the class, size, and complexity of the value of the expression.

Alternatively, you can select a variable from the list of workspace variables that displays.



Specify Input Parameters by Example

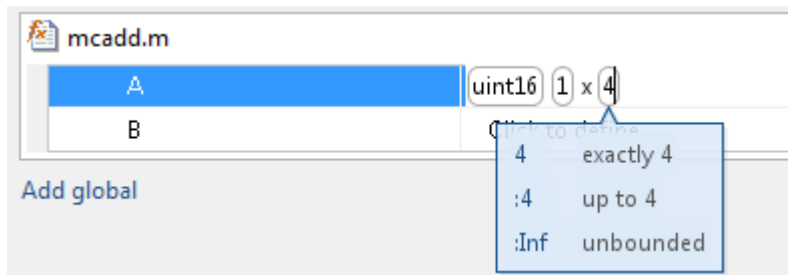
This example shows how to specify a 1-by-4 vector of unsigned 16-bit integers.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter:

```
zeros(1,4,'uint16')
```

The input type is `uint16(1x4)`.

- 5 Optionally, after you specify the input type, you can specify that the input is variable size. For example, select the second dimension.



- 6 To specify that the second dimension is variable size with an upper bound of 4, select :4. Alternatively, to specify that the second dimension is unbounded, select :Inf.

Alternatively, you can specify that the input is variable size by using the `coder.newtype` function. Enter the MATLAB expression:

```
coder.newtype('uint16',[1 4],[0 1])
```

Note: To specify that an input is a double-precision scalar, enter 0.

Specify a Structure Type Input Parameter by Example

This example shows how to specify a structure with two fields `a` and `b`. The input type of `a` is scalar double. The input type of `b` is scalar char.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter:

```
struct('a', 1, 'b', 'x')
```

The type of the input parameter is `struct(1x1)`. The type of field `a` is `double(1x1)`. The type of field `b` is `char(1x1)`

- 5 For an array of structures, to specify the size of each dimension, click the dimension and specify the size. For example, enter 4 for the first dimension.
- 6 To specify that the second dimension is variable size with an upper bound of 4, select :4. Alternatively, to specify that the second dimension is unbounded select :Inf.

Alternatively, specify the size of the array of structures in the `struct` function call. For example, `struct('a', { 1 2}, 'b', {'x', 'y'})` specifies a 1x2 array of structures with fields `a` and `b`. The type of field `a` is `double(1x1)`. The type of field `b` is `char(1x1)`.

To modify the type definition, see “Specify a Structure Input Parameter” (MATLAB Coder).

Specify a Cell Array Type Input Parameter by Example

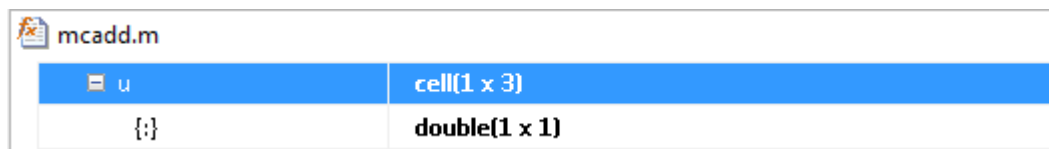
Note: The Fixed-Point Converter app does not support cell arrays.

This example shows how to specify a cell array input by example. When you define a cell array by example, the app determines whether the cell array is homogeneous or heterogeneous. . If you want to control whether the cell array is homogeneous or heterogeneous, specify the cell array by type. See “Specify a Cell Array Input Parameter” (MATLAB Coder).

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter an example cell array.
 - If all cell array elements have the same properties, the cell array is homogeneous. For example, enter:

```
{1 2 3}
```

The input is a 1x3 cell array. The type of each element is `double(1x1)`.



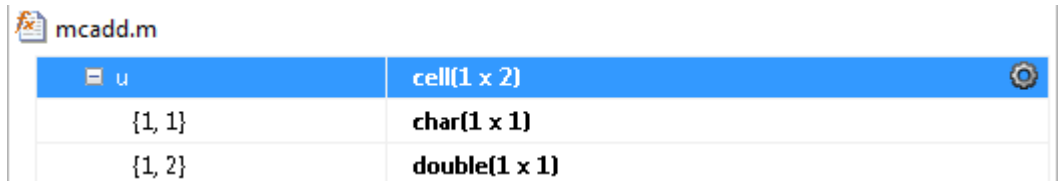
mcadd.m	
u	cell(1 x 3)
{:}	double(1 x 1)

The colon inside curly braces `{:}` indicates that all elements have the same properties.

- If elements of the cell array have different classes, the cell array is heterogeneous. For example, enter:

{'a', 1}

The input is a 1x2 cell array. For a heterogeneous cell array, the app lists each element. The type of the first element is `char(1x1)`. The type of the second element is `double(1x1)`.

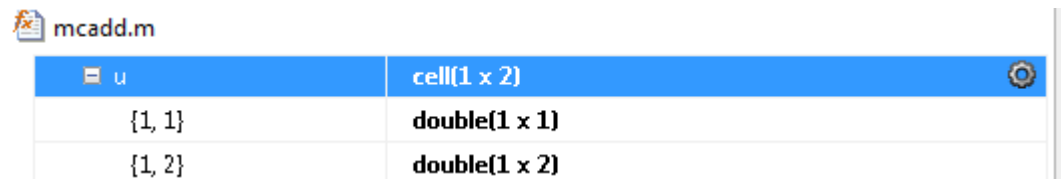


mcadd.m	
u	cell(1 x 2)
{1, 1}	char(1 x 1)
{1, 2}	double(1 x 1)

- For some example cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For these cell arrays, the app uses heuristics to determine whether the cell array is homogeneous or heterogeneous. For example, for the example cell array, enter:

{1 [2 3]}

The elements have the same class, but different sizes. The app determines that the input is a 1x2 heterogeneous cell array. The type of the first element is `double(1x1)`. The type of the second element is `double(1x2)`.



mcadd.m	
u	cell(1 x 2)
{1, 1}	double(1 x 1)
{1, 2}	double(1 x 2)

However, the example cell array, {1 [2 3]}, can also be a homogeneous cell array whose elements are 1x:2 double. If you want this cell array to be homogeneous, do one of the following:

- Specify the cell array input by type. Specify that the input is a homogeneous cell array. Specify that the elements are 1x:2 double. See “Specify a Cell Array Input Parameter” (MATLAB Coder).
- Right-click the variable. Select **Homogeneous**. Specify that the elements are 1x:2 double.

If you use `coder.typeof` to specify that the example cell array is variable size, the app makes the cell array homogeneous. For example, for the example input, enter:

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

The app determines that the input is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

To modify the type definition, see “Specify a Cell Array Input Parameter” (MATLAB Coder).

Specify an Enumerated Type Input Parameter by Example

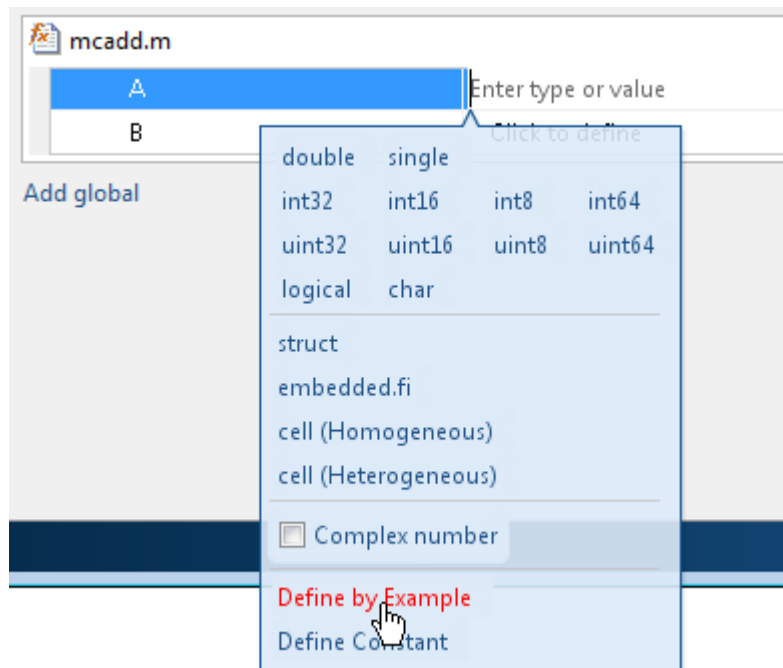
This example shows how to specify that an input uses the enumerated type `MyColors`.

Suppose that `MyColors.m` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

To specify that an input has the enumerated type `MyColors`:

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.



- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter the MATLAB expression:
`MyColors.red`

Specify a Fixed-Point Input Parameter by Example

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

This example shows how to specify a signed fixed-point type with a word length of eight bits, and a fraction length of three bits.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter:

```
fi(10, 1, 8, 3)
```

The app sets the type of input `u` to `fi(1x1)`. By default, if you do not specify a local `fi` object, the app uses the default `fi` object. See “`fi` for Sharing Arithmetic Rules” on page 4-23.

Optionally, modify the fixed-point properties or the size of the input. See “Specify a Fixed-Point Input Parameter by Type” on page 9-82 and “Define or Edit Input Parameter Type by Using the App” on page 9-80.

Specify Global Variable Type and Initial Value Using the App

In this section...

“Why Specify a Type Definition for Global Variables?” on page 9-96

“Specify a Global Variable Type” on page 9-96

“Define a Global Variable by Example” on page 9-96

“Define or Edit Global Variable Type” on page 9-97

“Define Global Variable Initial Value” on page 9-98

“Define Global Variable Constant Value” on page 9-99

“Remove Global Variables” on page 9-99

Why Specify a Type Definition for Global Variables?

If you use global variables in your MATLAB algorithm, before building the project, you must add a global type definition and initial value for each global variable. If you do not initialize the global data, the app looks for the variable in the MATLAB global workspace. If the variable does not exist, the app generates an error.

For MEX functions, if you use global data, you must also specify whether to synchronize this data between MATLAB and the MEX function.

Specify a Global Variable Type

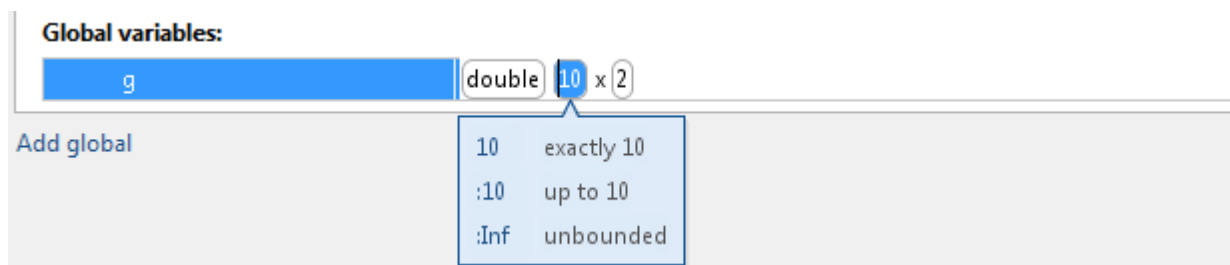
- 1 Specify the type of each global variable using one of the following methods:
 - Define by example
 - Define type
- 2 Define an initial value for each global variable.

If you do not provide a type definition and initial value for a global variable, create a variable with the same name and suitable class, size, complexity, and value in the MATLAB workspace.

Define a Global Variable by Example

- 1 Click the field to the right of the global variable that you want to define.

- 2 Select **Define by Example**.
- 3 In the field to the right of the global name, enter a MATLAB expression that has the required class, size, and complexity. MATLAB Coder software uses the class, size, and complexity of the value of this expression as the type for the global variable.
- 4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, **10**.



You can specify:

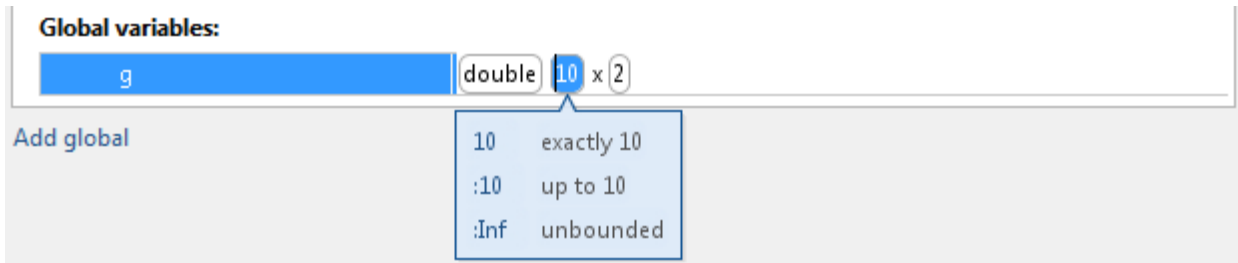
- Fixed size. In this example, select **10**.
- Variable size, up to a specified limit, by using the **:** prefix. In this example, to specify that your input can vary in size up to **10**, select **:10**.
- Unbounded variable size by selecting **:Inf**.

Define or Edit Global Variable Type

- 1 Click the field to the right of the global variable that you want to define.
- 2 Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.
- 3 Select the type for the global variable. For example, **double**.

By default, the global variable is a scalar.

- 4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, **10**.



You can specify:

- Fixed size. In this example, select 10.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to 10, select `:10`.
- Unbounded variable size by selecting `:Inf`.

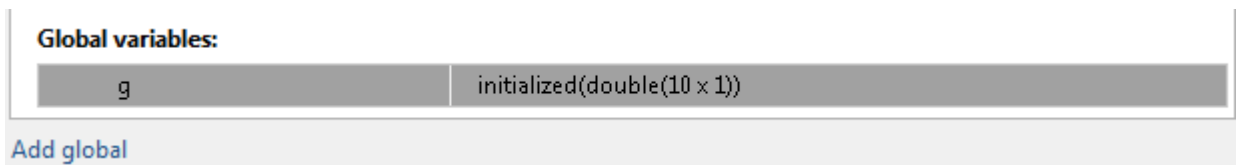
Define Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 9-98
- “Define Initial Value After Defining Type” on page 9-99

Define Initial Value Before Defining Type

- 1 Click the field to the right of the global variable.
- 2 Select **Define Initial Value**.
- 3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable. Because you did not define the type of the global variable before you defined its initial value, MATLAB Coder uses the initial value type as the global variable type.

The project shows that the global variable is initialized.



If you change the type of a global variable after defining its initial value, you must redefine the initial value.

Define Initial Value After Defining Type

- Click the type field of a predefined global variable.
- Select **Define Initial Value**.
- Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

The project shows that the global variable is initialized.

Global variables:

g	initialized(double(10 x 1))
---	-----------------------------

[Add global](#)

Define Global Variable Constant Value

- 1 Click the field to the right of the global variable.
- 2 Select **Define Constant Value**.
- 3 In the field to the right of the global variable, enter a MATLAB expression.

Remove Global Variables

- 1 Right-click the global variable.
- 2 From the menu, select **Remove Global**.

Specify Properties of Entry-Point Function Inputs Using the App

Why Specify Input Properties?

Fixed-Point Designer must determine the properties of all variables in the MATLAB files. To infer variable properties in MATLAB files, Fixed-Point Designer must identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs to Fixed-Point Designer. If your primary function has no input parameters, you do not need to specify properties of inputs to local functions or external functions called by the primary function.

Unless you use the tilde (~) character to specify unused function inputs, you must specify the same number and order of inputs as the MATLAB function . If you use the tilde character, the inputs default to real, scalar doubles.

See Also

- “Properties to Specify” on page 29-2

Specify an Input Definition Using the App

Specify an input definition using one of the following methods:

- Autodefine Input Types
- Define Type
- Define by Example
- Define Constant

Detect Dead and Constant-Folded Code

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

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

Coverage Bar Color	Indicates
Green	<p>One of the following situations:</p> <ul style="list-style-type: none"> • The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. <p>Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.</p>
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

What Is Dead Code?

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

- Defensive code containing intended corner cases that are not reached
- Human error in the code, resulting in code that cannot be reached by any execution path
- Inadequate test bench range
- Constant folding

Detect Dead Code

This example shows how to detect dead code in your algorithm by using the Fixed-Point Converter .

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

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

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

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

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

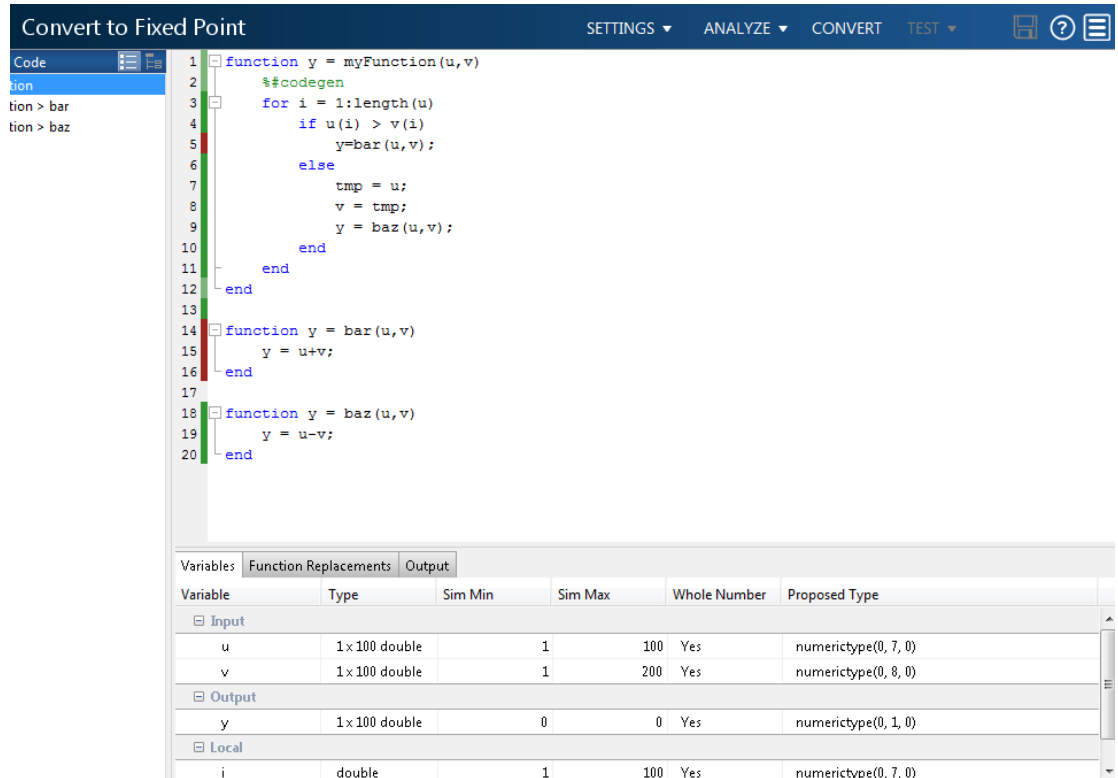
```
u = 1:100;
v = 101:200;
```

```
myFunction(u,v);
```

- 3 From the apps gallery, open the Fixed-Point Converter .
- 4 On the **Select Source Files** page, browse to the `myFunction` file, and click **Open**.
- 5 Click **Next**. On the **Define Input Types** page, browse to select the test file that you created, `myFunction_tb`. Click **Autodefine Input Types**.
- 6 Click **Next**. On the **Convert to Fixed-Point** page, click **Analyze** to simulate the entry-point functions, gather range information, and get proposed data types.

The color-coded bar on the left side of the edit window indicates whether the code executes. The code in the first condition of the if-statement does not execute during simulation because u is never greater than v . The `bar` function never executes

because the if-statement never executes. These parts of the algorithm are marked with a red bar, indicating that they are dead code.

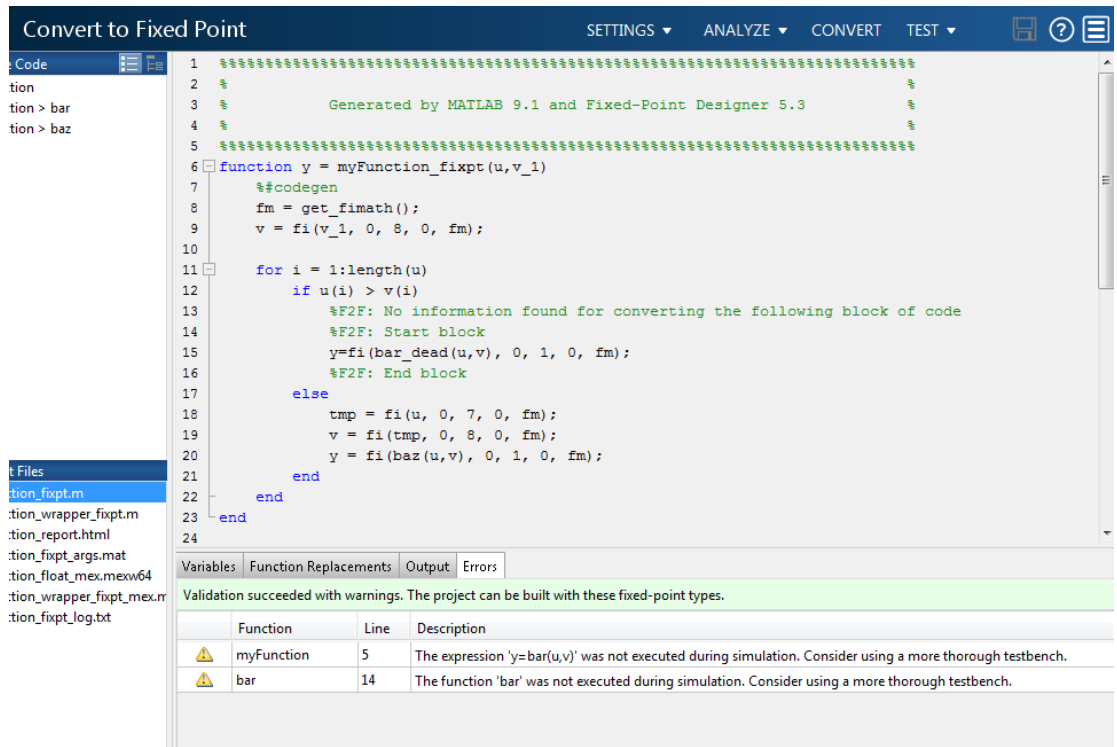


The screenshot shows the 'Convert to Fixed Point' tool interface. The top pane displays MATLAB code with a red bar on lines 4-9, indicating dead code. The bottom pane shows a table of variable types and simulation ranges.

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	1 x 100 double	1	100	Yes	numerictype(0, 7, 0)
v	1 x 100 double	1	200	Yes	numerictype(0, 8, 0)
Output					
y	1 x 100 double	0	0	Yes	numerictype(0, 1, 0)
Local					
i	double	1	100	Yes	numerictype(0, 7, 0)

- To apply the proposed data types to the function, click **Convert**.

The Fixed-Point Converter generates a fixed-point function, `myFunction_fixpt`. The generated fixed-point code contains comments around the pieces of code identified as dead code. The **Validation Results** pane proposes that you use a more thorough test bench.



When the Fixed-Point Converter detects dead code, consider editing your test file so that your algorithm is exercised over its full range. If your test file already reflects the full range of the input variables, consider editing your algorithm to eliminate the dead code.

- 8 Close the Fixed-Point Converter .

Fix Dead Code

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

```

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

```

- 2 Reopen the Fixed-Point Converter .

- 3 Using the same function and the edited test file, go through the conversion process again.
- 4 After you click **Analyze**, this time the code coverage bar shows that all parts of the algorithm execute with the new test file input ranges.

The screenshot shows the 'Convert to Fixed Point' tool interface. The top bar contains the title 'Convert to Fixed Point' and navigation buttons: 'SETTINGS', 'ANALYZE', 'CONVERT', and 'TEST'. The main area displays MATLAB code for three functions: `myFunction`, `bar`, and `baz`. A green coverage bar on the left indicates that all code is executed. Below the code is a table showing variable analysis for input, output, and local variables.

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	1 x 100 double	1	100	Yes	numerictype(0, 7, 0)
v	1 x 100 double	-50	200	Yes	numerictype(1, 9, 0)
Output					
y	1 x 100 double	-49	248	Yes	numerictype(1, 9, 0)
Local					
i	double	1	100	Yes	numerictype(0, 7, 0)

To finish the conversion process and convert the function to fixed point, click **Convert**.

Automated Conversion Using Programmatic Workflow

- “Propose Data Types Based on Simulation Ranges” on page 10-2
- “Propose Data Types Based on Derived Ranges” on page 10-7
- “Detect Overflows” on page 10-14
- “Replace the exp Function with a Lookup Table” on page 10-19
- “Replace a Custom Function with a Lookup Table” on page 10-21
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 10-23
- “Enable Plotting Using the Simulation Data Inspector” on page 10-29

Propose Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data using the `fiaccel` function.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

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

- 3 Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

It is best practice to 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.

Type	Name	Description
Function code	<code>ex_2ndOrder_filter.m</code>	Entry-point MATLAB function
Test file	<code>ex_2ndOrder_filter_test.m</code>	MATLAB script that tests <code>ex_2ndOrder_filter.m</code>

The `ex_2ndOrder_filter` Function

```
function y = ex_2ndOrder_filter(x) %#codegen
```

```

persistent z
if isempty(z)
    z = zeros(2,1);
end
% [b,a] = butter(2, 0.25)
b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
a = [1, -0.942809041582063, 0.3333333333333333];

y = zeros(size(x));
for i = 1:length(x)
    y(i) = b(1)*x(i) + z(1);
    z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
    z(2) = b(3)*x(i) - a(3) * y(i);
end
end

```

The ex_2ndOrder_filter_test Script

The test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```

% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = ex_2ndOrder_filter(x(i,:));
end

% Plot the results
titles = {'Chirp', 'Step', 'Impulse'}
clf

```

```
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```
cfg = coder.config('fixpt');
cfg.TestBenchName = 'ex_2ndOrder_filter_test';
```

Collect Simulation Ranges and Generate Fixed-Point Code

Use the `fiaccel` function to convert the floating-point MATLAB function, `ex_2ndOrder_filter`, to fixed-point MATLAB code. Set the default word length for the fixed-point data types to 16.

```
cfg.ComputeSimulationRanges = true;
cfg.DefaultWordLength = 16;

% Derive ranges and generate fixed-point code
fiaccel -float2fixed cfg ex_2ndOrder_filter
```

`fiaccel` analyzes the floating-point code. Because you did not specify the input types for the `ex_2ndOrder_filter` function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

View Range Information

Click the link to the type proposal report for the `ex_2ndOrder_filter` function, `ex_2ndOrder_filter_report.html`.

The report opens in a web browser.

Fixed-Point Report *ex_2ndOrder_filter*

Simulation Coverage	Code
100%	function y = ex_2ndOrder_filter(x) %#codegen persistent z
Once	if isempty(z) z = zeros(2,1); end
100%	% [b,a] = butter(2, 0.25) b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175]; a = [1, -0.942809041582063, 0.333333333333333]; y = zeros(size(x)); for i=1:length(x) y(i) = b(1)*x(i) + z(1); z(1) = b(2)*x(i) + z(2) - a(2) * y(i); z(2) = b(3)*x(i) - a(3) * y(i); end end

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
a	double 1 x 3	-0.942809041582063	1			No	numericType(1, 16, 14)
b	double 1 x 3	0.0976310729378175	0.195262145875635			No	numericType(0, 16, 18)
i	double	1	256			Yes	numericType(0, 9, 0)
x	double 1 x 256	-0.9999756307053946	1			No	numericType(1, 16, 14)
y	double 1 x 256	-0.9696817930434206	1.0553496057969345			No	numericType(1, 16, 14)
z	double 2 x 1	-0.8907046852192462	0.957718532859117			No	numericType(1, 16, 15)

View Generated Fixed-Point MATLAB Code

fiaccl generates a fixed-point version of the *ex_2ndOrder_filter.m* function, *ex_2ndOrder_filter_fixpt.m*, and a wrapper function that calls *ex_2ndOrder_filter_fixpt*. These files are generated in the *codegen\ex_2ndOrder_filter\fixpt* folder in your local working folder.

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
    fm = get_fimath();

    persistent z
    if isempty(z)
        z = fi(zeros(2,1), 1, 16, 15, fm);
    end
    % [b,a] = butter(2, 0.25)
    b = fi([0.0976310729378175, 0.195262145875635, ...
    0.0976310729378175], 0, 16, 18, fm);
```

```

a = fi([
    1, -0.942809041582063,...
    0.3333333333333333], 1, 16, 14, fm);

y = fi(zeros(size(x)), 1, 16, 14, fm);
for i=1:length(x)
    y(i) = b(1)*x(i) + z(1);
    z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
    z(2) = fi_signed(b(3)*x(i))          - a(3) * y(i);
end
end

function y = fi_signed(a)
    coder.inline( 'always' );
    if isfi( a ) && ~(issigned( a ))
        nt = numerictype( a );
        new_nt = numerictype( 1, nt.WordLength + 1, nt.FractionLength );
        y = fi( a, new_nt, fimath( a ) );
    else
        y = a;
    end
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
        'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision',...
        'MaxSumWordLength', 128);
end

```

Propose Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges using the `fiaccel` function. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a New Folder and Copy Relevant Files

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

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

- 3 Copy the `dti.m` and `dti_test.m` files to your local working folder.

It is best practice to 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.

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

The dti Function

The `dti` function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;

function b = subFunction(a)
```



```
b = a*a;
```

The `dti_test` Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10)

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);
end

figure('Name', [mfilename, '_plot'])
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2)
plot(1:len,y_out)
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')
```

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

Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'dti_test';
```

Specify Design Ranges

Specify design range information for the `dti` function input parameter `u_in`.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
```

Enable Plotting Using the Simulation Data Inspector

Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;  
fixptcfg.LogIOForComparisonPlotting = true;  
fixptcfg.PlotWithSimulationDataInspector = true;
```

Derive Ranges and Generate Fixed-Point Code

Use the `fiaccel` function to convert the floating-point MATLAB function, `dti`, to fixed-point MATLAB code. Set the default word length for the fixed-point data types to 16.

```
fixptcfg.ComputeDerivedRanges = true;  
fixptcfg.ComputeSimulationRanges = false;  
fixptcfg.DefaultWordLength = 16;  
  
% Derive ranges and generate fixed-point code  
fiaccel -float2fixed fixptcfg dti
```

`fiaccel` analyzes the floating-point code. Because you did not specify the input types for the `dti` function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

View Derived Range Information

Click the link to the type proposal report for the `dti` function, [dti_report.html](#).

The report opens in a web browser.

Fixed Point Report dti

```
function [y,clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;
% variable to hold state between consecutive calls to this block
persistent u_state
if isempty( u_state )
    u_state = init_val + 1;
end
% Compute Output
if (u_state>limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state<=limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state
```

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
clip_status	double			-2	2	No	numerictype(1, 16, 13)
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
init_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)
u_in	double			-1	1	No	numerictype(1, 16, 14)
u_state	double			-501	501	No	numerictype(1, 16, 6)
y	double			-500	500	No	numerictype(1, 16, 6)

View Generated Fixed-Point MATLAB Code

`fiaccel` generates a fixed-point version of the `dti` function, `dti_fxpt.m`, and a wrapper function that calls `dti_fxpt`. These files are generated in the `codegen\dti\fixpt` folder in your local working folder.

```
function [y, clip_status] = dti_fixpt(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
```

```
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
fm = get_fimath();

init_val = fi(1, 0, 1, 0, fm);
gain_val = fi(1, 0, 1, 0, fm);
limit_upper = fi(500, 0, 9, 0, fm);
limit_lower = fi(-500, 1, 10, 0, fm);

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = fi(init_val+fi(1, 0, 1, 0, fm), 1, 16, 6, fm);
end

% Compute Output
if (u_state > limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
elseif (u_state >= limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-1, 1, 16, 13, fm);
elseif (u_state < limit_lower)
    y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(2, 1, 16, 13, fm);
elseif (u_state <= limit_lower)
    y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(1, 1, 16, 13, fm);
else
    y = fi(u_state, 1, 16, 6, fm);
    clip_status = fi(0, 1, 16, 13, fm);
end

% Update State
tprod = fi(gain_val * u_in, 1, 16, 14, fm);
u_state(:) = y + tprod;
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
        'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision',...
```

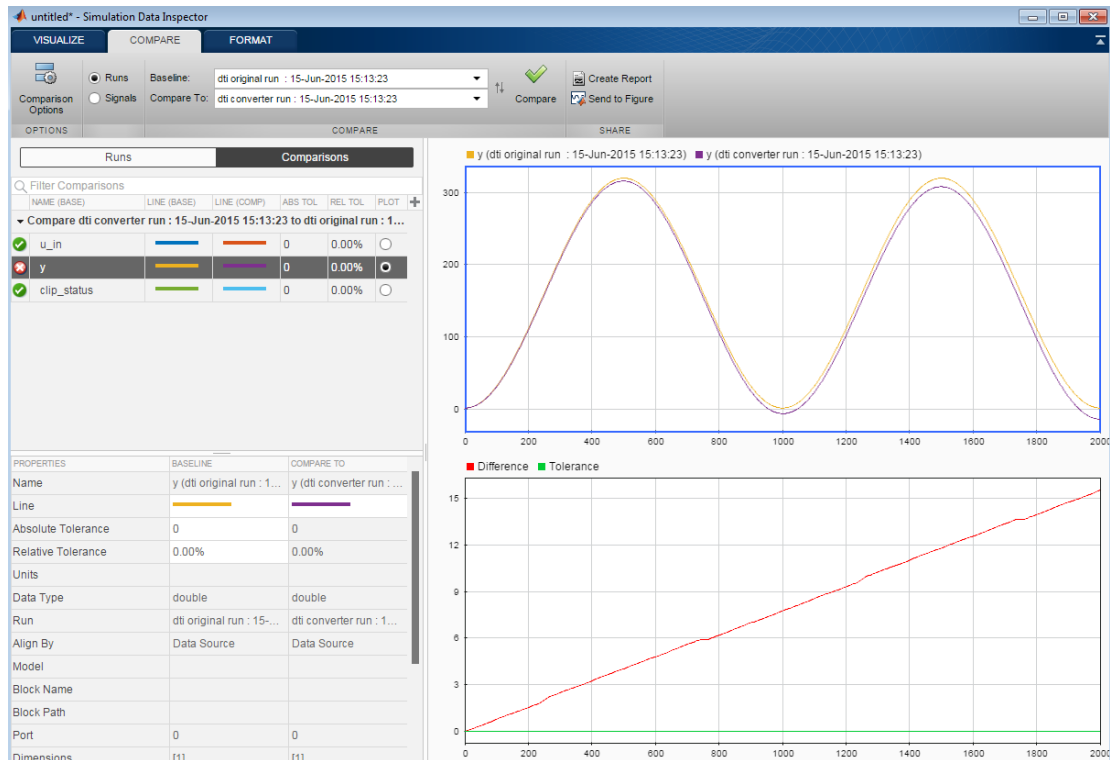
```
'MaxSumWordLength', 128);
end
```

Compare Floating-Point and Fixed-Point Runs

Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.

You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output **y**, on the **Compare** tab, select **y**, and then click **Compare Runs**.

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



Detect Overflows

This example shows how to detect overflows using the `fiaccl` function. At the numerical testing stage in the conversion process, the tool simulates the fixed-point code using scaled doubles. It then reports which expressions in the generated code produce values that would overflow the fixed-point data type.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a New Folder and Copy Relevant Files

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

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

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

It is best practice to 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.

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

The `overflow` Function

```
function y = overflow(b,x,reset)
```

```

    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end

```

The overflow_test Function

```

function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    % b = fir1(11,0.25);
    b = [-0.004465461051254
        -0.004324228005260
        +0.012676739550326
        +0.074351188907780
        +0.172173206073645
        +0.249588554524763
        +0.249588554524763
        +0.172173206073645
        +0.074351188907780
        +0.012676739550326
        -0.004324228005260
        -0.004465461051254]';

    % Input signal

```

```
nx = 256;
t = linspace(0,10*pi,nx)';

% Impulse
x_impulse = zeros(nx,1); x_impulse(1) = 1;

% Max Gain
% The maximum gain of a filter will occur when the inputs line up with the
% signs of the filter's impulse response.
x_max_gain = sign(b)';
x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
x_max_gain = x_max_gain(1:nx);

% Sums of sines
f0=0.1; f1=2;
x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

% Chirp
f_chirp = 1/16; % Target frequency
x_chirp = sin(pi*f_chirp*t.^2); % Linear chirp

x = [x_impulse, x_max_gain, x_sines, x_chirp];
titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
y = zeros(size(x));

for i=1:size(x,2)
    reset = true;
    y(:,i) = overflow(b,x(:,i),reset);
end

test_plot(1,titles,t,x,y)

end
function test_plot(fig,titles,t,x,y1)
figure(fig)
clf
sub_plot = 1;
font_size = 10;
for i=1:size(x,2)
    subplot(4,1,sub_plot)
    sub_plot = sub_plot+1;
    plot(t,x(:,i),'c',t,y1(:,i),'k')
    axis('tight')
```



```

        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end

```

Set Up Configuration Object

- 1 Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

- 2 Set the test bench name. In this example, the test bench function name is `overflow_test`.

```
fixptcfg.TestBenchName = 'overflow_test';
```

- 3 Set the default word length to 16.

```
fixptcfg.DefaultWordLength = 16;
```

Enable Overflow Detection

```
fixptcfg.TestNumerics = true;
fixptcfg.DetectFixptOverflows = true;
```

Set `fimath` Options

Set the `fimath` `Product` mode and `Sum` mode to `KeepLSB`. These settings models the behavior of integer operations in the C language.

```
fixptcfg.fimath = ...
'fimath(''RoundingMethod'', ''Floor'', ''OverflowAction'', ''Wrap'', ...
''ProductMode'', ''KeepLSB'', ''SumMode'', ''KeepLSB'');
```

Convert to Fixed Point

Convert the floating-point MATLAB function, `overflow`, to floating-point MATLAB code. You do not need to specify input types for the `fiaccel` command because it infers the types from the test file.

```
fiaccel -float2fixed fixptcfg overflow
```

The numerics testing phase reports an overflow.

```
Overflow error in expression 'acc + b( j )*z( k )'. Percentage of Current Range = 104%.
```

Review Results

Determine if the addition or the multiplication in this expression overflowed. Set the `fimath` `ProductMode` to `FullPrecision` so that the multiplication will not overflow, and then run the `fiaccel` command again.

```
fixptcfg.fimath = ...  
'fimath('RoundingMethod','Floor','OverflowAction','Wrap',...  
        'ProductMode','FullPrecision','SumMode','KeepLSB');  
fiaccel -float2fixed fixptcfg overflow
```

The numerics testing phase still reports an overflow, indicating that it is the addition in the expression that is overflowing.

Replace the exp Function with a Lookup Table

This example shows how to replace the `exp` function with a lookup table approximation in the generated fixed-point code using the `fiaccel` function.

Prerequisites

To complete this example, you must install the following products:

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

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

Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

- 2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'my_fcn_test';  
fixptcfg.TestNumerics = true;  
fixptcfg.DefaultWordLength = 16;  
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
fiaccel -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_exp`, for the `exp` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)  
    fm = get_fimath();  
  
    y = fi(replacement_exp(x), 0, 16, 1, fm);  
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

More About

- “Replacing Functions Using Lookup Table Approximations” on page 8-62

Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the `fiaccl` function.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...  
                       'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'custom_test';  
fixptcfg.TestNumerics = true;  
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
fiaccel -float2fixed fixptcfg call_custom_fcn
```

`fiaccel` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)  
    fm = get_fimath();  
  
    y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);  
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

More About

- “Replacing Functions Using Lookup Table Approximations” on page 8-62

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `fiaccel` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a New Folder and Copy Relevant Files

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

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```
- 3 Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

It is best practice to 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.

Type	Name	Description
Function code	myFilter.m	Entry-point MATLAB function
Test file	myFilterTest.m	MATLAB script that tests myFilter.m
Plotting function	plotDiff.m	Custom plot function
MAT-file	filterData.mat	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
    b = complex(zeros(1,16));
    h = complex(zeros(1,16));
    h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```


The plotDiff Function

```

% varInfo - structure with information about the variable. It has the following fields
%         i) name
%         ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%             Fixed-Point conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexp(varName, '_', '\\_');
    escapedFcnName = regexp(fcnName, '_', '\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
            y_vec = flatFixedVals;
            subplot(1, 2, 2);

```

```
        plotScatter(x_vec, y_vec, 100, fixedTitle);

    otherwise
        % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Set Up Configuration Object

- 1 Create a `coder.FixptConfig` object.

```
fxptcfg = coder.config('fixpt');
```
- 2 Specify the test file name and custom plot function name. Enable logging and numerics testing.

```
fxptcfg.TestBenchName = 'myFilterTest';
fxptcfg.PlotFunction = 'plotDiff';
fxptcfg.TestNumerics = true;
fxptcfg.LogIOForComparisonPlotting = true;
fxptcfg.DefaultWordLength = 16;
```

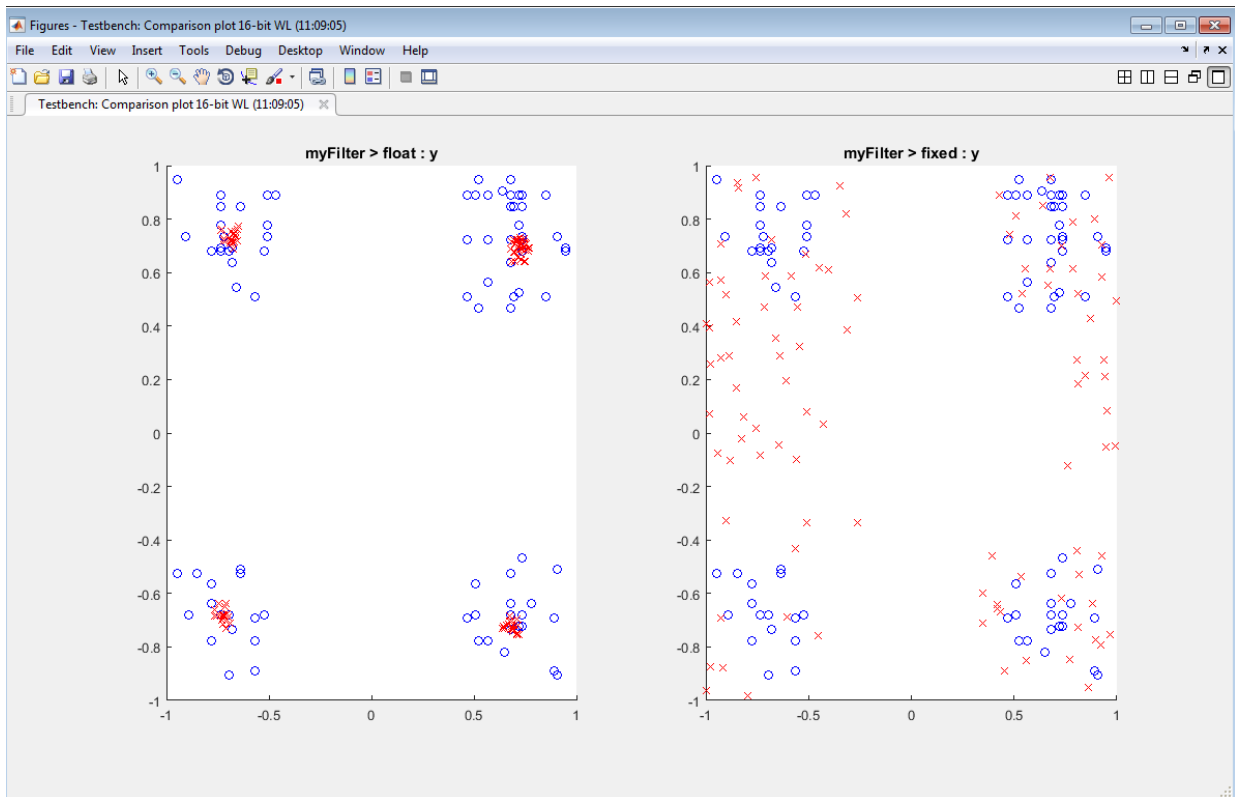
Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `fiaccl` command because it infers the types from the test file.

```
fiaccl -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of 16 and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

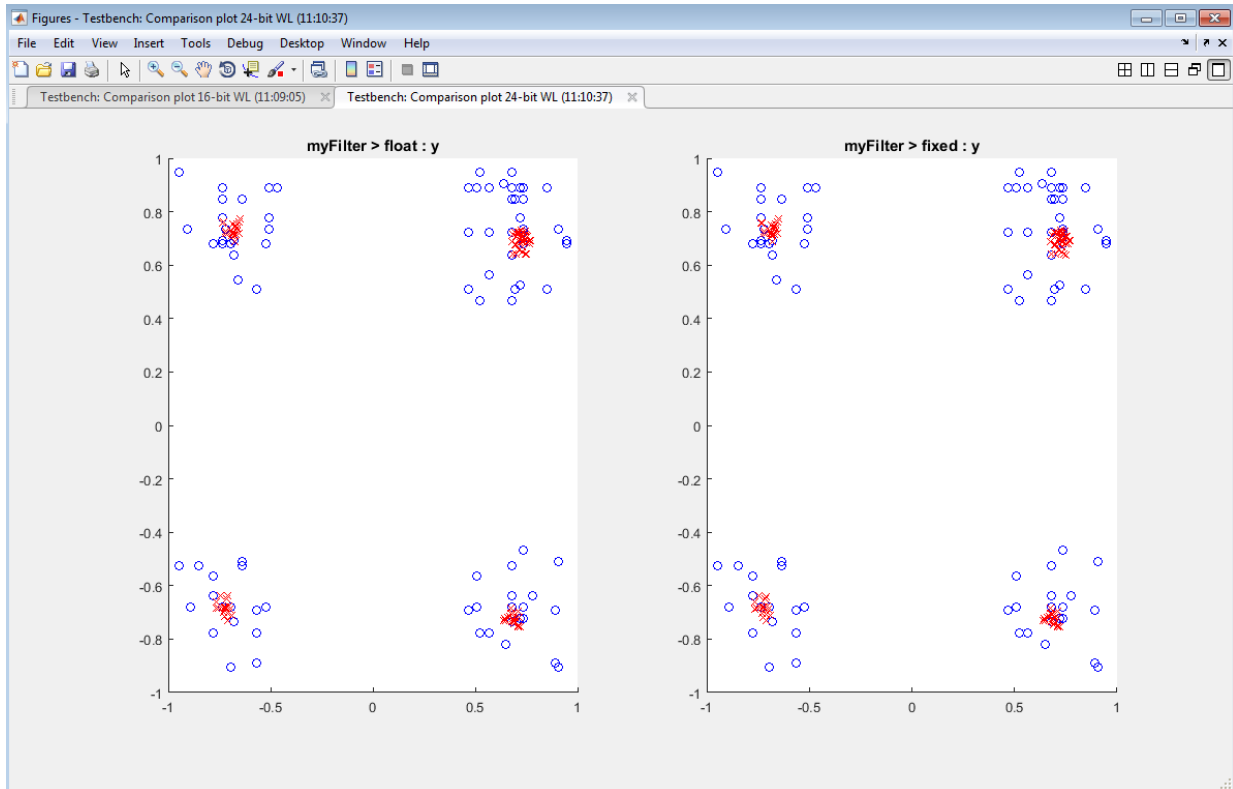


The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to 24 and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;
fiaccl -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.



More About

- “Custom Plot Functions” on page 8-63

Enable Plotting Using the Simulation Data Inspector

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the `fiaccl` function. At the MATLAB command line:

- 1 Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'dti_test';
```

- 2 Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;  
fixptcfg.LogIOForComparisonPlotting = true;  
fixptcfg.PlotWithSimulationDataInspector = true;
```

- 3 Generate fixed-point MATLAB code using `fiaccl`.

```
fiaccl -float2fixed fixptcfg dti
```

For an example, see “Propose Data Types Based on Derived Ranges” on page 10-7.

More About

- “Inspecting Data Using the Simulation Data Inspector” on page 13-2

Single-Precision Conversion

- “Generate Single-Precision MATLAB Code” on page 11-2
- “MATLAB Language Features Supported for Single-Precision Conversion” on page 11-10
- “Single-Precision Conversion Best Practices” on page 11-12

Generate Single-Precision MATLAB Code

This example shows how to generate single-precision MATLAB code from double-precision MATLAB code.

Prerequisites

To complete this example, install the following products:

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

To change the default compiler, you can use `mex -setup`. See “Change Default Compiler” (MATLAB).

Create a Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

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

- 3 Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>ex_2ndOrder_filter.m</code>	Entry-point MATLAB function
Test file	<code>ex_2ndOrder_filter_test.m</code>	MATLAB script that tests <code>ex_2ndOrder_filter.m</code>

The `ex_2ndOrder_filter` Function

```
function y = ex_2ndOrder_filter(x) %#codegen
    persistent z
    if isempty(z)
        z = zeros(2,1);
```



```

end
% [b,a] = butter(2, 0.25)
b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
a = [1, -0.942809041582063, 0.333333333333333];

y = zeros(size(x));
for i = 1:length(x)
    y(i) = b(1)*x(i) + z(1);
    z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
    z(2) = b(3)*x(i) - a(3) * y(i);
end
end

```

The `ex_2ndOrder_filter_test` Script

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse. The script then plots the outputs.

```

% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = ex_2ndOrder_filter(x(i,:));
end

```

```
end

% Plot the results
titles = {'Chirp', 'Step', 'Impulse'}
clf
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input', 'Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Set Up the Single-Precision Configuration Object

Create a single-precision configuration object. Specify the test file name. Verify the single-precision code using the test file. Plot the error between the double-precision code and single-precision code. Use the default values for the other properties.

```
scfg = coder.config('single');
scfg.TestBenchName = 'ex_2ndOrder_filter_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Generate Single-Precision MATLAB Code

To convert the double-precision MATLAB function, `ex_2ndOrder_filter`, to single-precision MATLAB code, use the `convertToSingle`

```
convertToSingle -config scfg ex_2ndOrder_filter
```

`convertToSingle` analyzes the double-precision code. The conversion process infers types by running the test file because you did not specify the input types for the `ex_2ndOrder_filter` function. The conversion process selects single-precision types for the double-precision variables. It selects `int32` for index variables. When the conversion is complete, `convertToSingle` generates a type proposal report.

View the Type Proposal Report

To see the types that the conversion process selected for the variables, open the type proposal report for the `ex_2ndOrder_filter` function. Click the link `ex_2ndOrder_filter_report.html`.

The report opens in a web browser. The conversion process converted:

- Double-precision variables to `single`.
- The index `i` to `int32`. The conversion process casts index and dimension variables to `int32`.

Single-Precision Report `ex_2ndOrder_filter`

Simulation Coverage	Code
100%	<code>function y = ex_2ndOrder_filter(x) %#codegen</code>
Once	<code>persistent z</code>
	<code>if isempty(z)</code>
	<code>z = zeros(2,1);</code>
	<code>end</code>
100%	<code>% [b,a] = butter(2, 0.25)</code>
	<code>b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];</code>
	<code>a = [</code>
	<code>1, -0.942809041582063, 0.333333333333333];</code>
	<code>y = zeros(size(x));</code>
	<code>for i=1:length(x)</code>
	<code>y(i) = b(1)*x(i) + z(1);</code>
	<code>z(1) = b(2)*x(i) + z(2) - a(2) * y(i);</code>
	<code>z(2) = b(3)*x(i) - a(3) * y(i);</code>
	<code>end</code>
	<code>end</code>

Variable Name	Type	Sim Min	Sim Max	Whole Number	ProposedType
a	double 1 x 3	-0.942809041582063	1	No	single
b	double 1 x 3	0.0976310729378175	0.195262145875635	No	single
i	double	1	256	Yes	int32
x	double 1 x 256	-0.9999756307053946	1	No	single
y	double 1 x 256	-0.9696817930434206	1.0553496057969345	No	single
z	double 2 x 1	-0.8907046852192462	0.957718532859117	No	single

View Generated Single-Precision MATLAB Code

To view the report for the generation of the single-precision MATLAB code, in the Command Window:

- 1 Scroll to the **Generate Single-Precision Code** step. Click the **View report** link.
- 2 On the **MATLAB code** tab, under **Functions**, click `ex_2ndOrder_filter_single`.

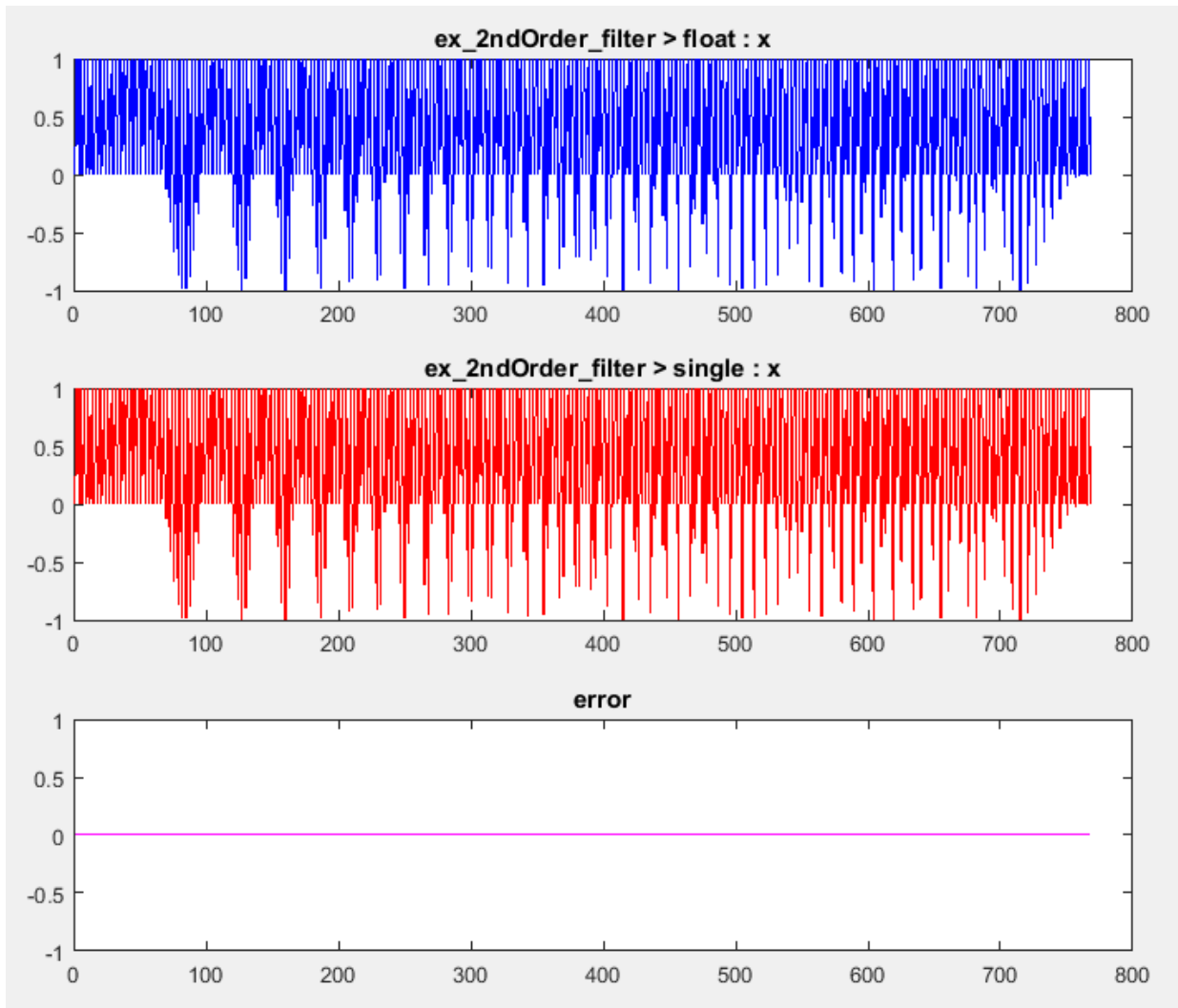
The code generation report displays the single-precision MATLAB code for `ex_2ndOrder_filter`.

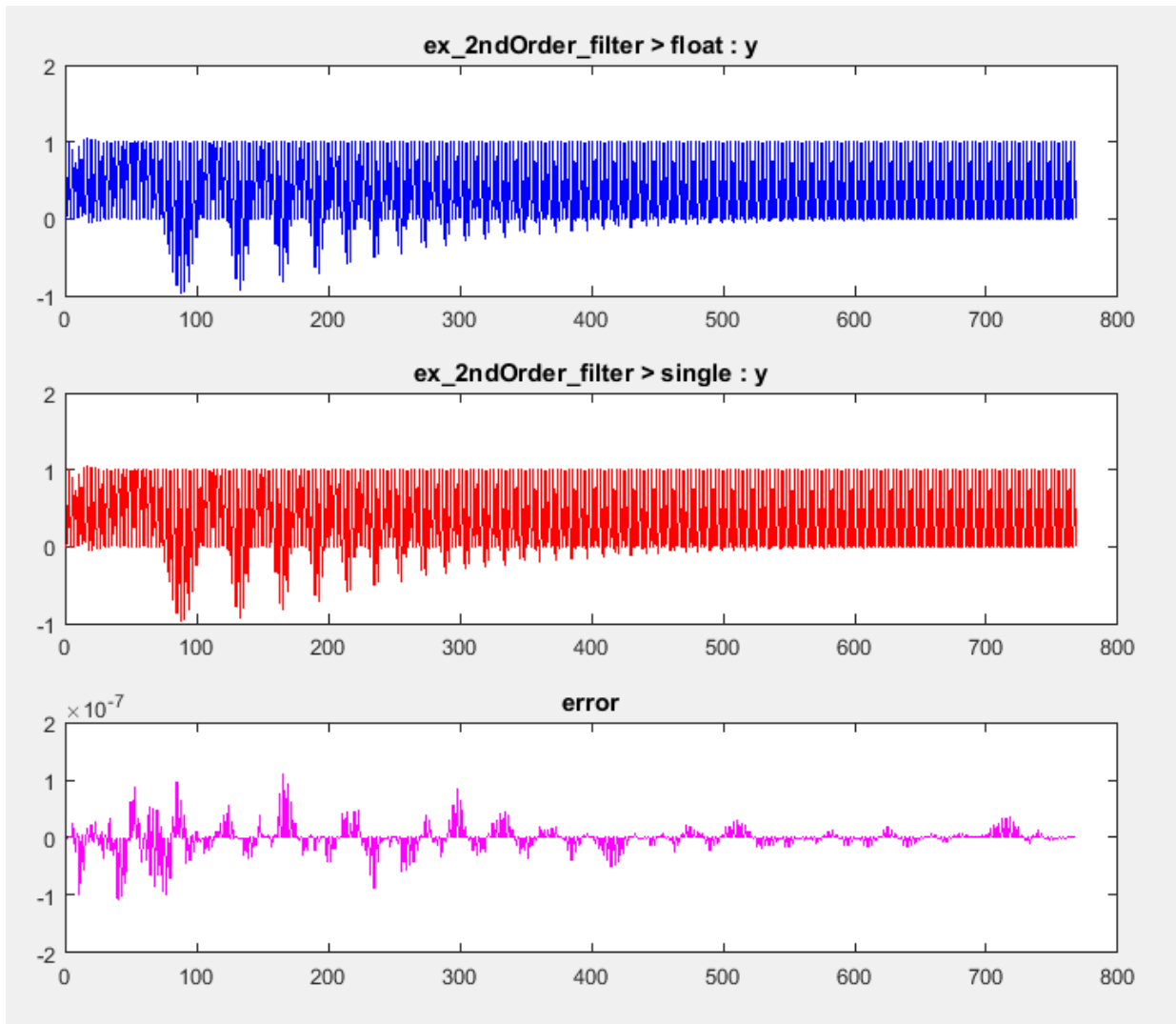
View Potential Data Type Issues

When you generate single-precision code, `convertToSingle` enables highlighting of potential data type issues in code generation reports. If `convertToSingle` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation. Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

Compare the Double-Precision and Single-Precision Variables

You can see the comparison plots for the input `x` and output `y` because you selected to log inputs and outputs for comparison plots .





See Also

`coder.SingleConfig` | `coder.config` | `convertToSingle`

More About

- “Single-Precision Conversion Best Practices” on page 11-12

MATLAB Language Features Supported for Single-Precision Conversion

In this section...
“MATLAB Language Features Supported for Single-Precision Conversion” on page 11-10
“MATLAB Language Features Not Supported for Single-Precision Conversion” on page 11-11

MATLAB Language Features Supported for Single-Precision Conversion

Single-precision conversion supports the following MATLAB language features:

- N-dimensional arrays.
- Matrix operations, including deletion of rows and columns.
- Variable-size data. Comparison plotting does not support variable-size data.
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 27-30).
- Complex numbers (see “Code Generation for Complex Data” on page 18-4).
- Numeric classes (see “Supported Variable Types” on page 20-16).
- Program control statements `if`, `switch`, `for`, `while`, and `break`.
- Arithmetic, relational, and logical operators.
- Local functions.
- Global variables.
- Persistent variables.
- Structures.
- Characters.

Single-precision conversion does not support the complete set of Unicode characters. Characters are restricted to 8 bits of precision in generated code. Many mathematical operations require more than 8 bits of precision. If you intend to convert your MATLAB algorithm to single precision, it is a best practice not to perform arithmetic with characters.

- MATLAB classes. Single-precision conversion supports:

- Class properties
- Constructors
- Methods
- Specializations

It does not support class inheritance or packages.

- Function calls (see “Resolution of Function Calls for Code Generation” on page 16-2)

MATLAB Language Features Not Supported for Single-Precision Conversion

Single-precision conversion does not support the following features:

- Anonymous functions
- Cell arrays
- Function handles
- Java
- Nested functions
- Recursion
- Sparse matrices
- `try/catch` statements
- `varargin` and `varargout`

Single-Precision Conversion Best Practices

In this section...

“Use Integers for Index Variables” on page 11-12

“Limit Use of assert Statements” on page 11-12

“Initialize MATLAB Class Properties in Constructor” on page 11-12

“Provide a Test File That Calls Your MATLAB Function” on page 11-12

“Prepare Your Code for Code Generation” on page 11-13

“Use the -args Option to Specify Input Properties” on page 11-13

“Test Numerics and Log I/O Data” on page 11-13

Use Integers for Index Variables

In MATLAB code that you want to convert to single precision, it is a best practice to use integers for index variables. However, if the code does not use integers for index variables, when possible `convertToSingle` tries to detect the index variables and select `int32` types for them.

Limit Use of assert Statements

- Do not use `assert` statements to define the properties of input arguments.
- Do not use `assert` statements to test the type of a variable. For example, do not use `assert(isa(a, 'double'))`

Initialize MATLAB Class Properties in Constructor

Do not initialize MATLAB class properties in the `properties` block. Instead, use the constructor to initialize the class properties.

Provide a Test File That Calls Your MATLAB Function

Separate your core algorithm from other code that you use to test and verify the results. Create a test file that calls your double-precision MATLAB algorithm. You can use the test file to:

- Automatically define properties of the top-level function inputs.

- Verify that the double-precision algorithm behaves as you expect. The double-precision behavior is the baseline against which you compare the behavior of the single-precision versions of your algorithm.
- Compare the behavior of the single-precision version of your algorithm to the double-precision baseline.

For best results, the test file must exercise the algorithm over its full operating range.

Prepare Your Code for Code Generation

MATLAB code that you want to convert to single precision must comply with code generation requirements. See “MATLAB Language Features Supported for C/C++ Code Generation” on page 21-21.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%#codegen` pragma to the top of your MATLAB file. When you edit your code in the MATLAB editor, the MATLAB Code Analyzer flags functions and constructs that are not supported for code generation. See “Check Code Using the MATLAB Code Analyzer” on page 14-103. When you use the MATLAB Coder app, the app screens your code for code generation readiness. At the function line, you can use the Code Generation Readiness Tool. See “Check Code Using the Code Generation Readiness Tool” on page 14-102.

Use the `-args` Option to Specify Input Properties

When you generate single-precision MATLAB code, if you specify a test file, you do not have to specify argument properties with the `-args` option. In this case, the code generator runs the test file to determine the properties of the input types. However, running the test file can slow the code generation. It is a best practice to pass the properties to the `-args` option so that `convertToSingle` does not run the test file to determine the argument properties. If you have a MATLAB Coder license, you can use `coder.getArgTypes` to determine the argument properties. For example:

```
types = coder.getArgTypes('myfun_test', 'myfun');
scfg = coder.config('single');
convertToSingle -config scfg -args types myfun
```

Test Numerics and Log I/O Data

When you use the `convertToSingle` function to generate single-precision MATLAB code, enable numerics testing and I/O data logging for comparison plots. To use numerics

testing, you must provide a test file that calls your MATLAB function. To enable numerics testing and I/O data logging, create a `coder.SingleConfig` object. Set the `TestBenchName`, `TestNumerics`, and `LogIOForComparisonPlotting` properties. For example:

```
scfg = coder.config('single');  
scfg.TestBenchName = 'mytest';  
scfg.TestNumerics = true;  
scfg.LogIOForComparisonPlotting = true;
```

Fixed-Point Conversion — Manual Conversion

- “Manual Fixed-Point Conversion Workflow” on page 12-2
- “Manual Fixed-Point Conversion Best Practices” on page 12-4
- “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros” on page 12-19
- “Fixed-Point Design Exploration in Parallel” on page 12-44

Manual Fixed-Point Conversion Workflow

- 1 Implement your algorithm in MATLAB.
- 2 Write a test file that calls your original MATLAB algorithm to validate the behavior of your algorithm.

Create a test file to validate that the algorithm works as expected in floating point before converting it to fixed point. Use the same test file to propose fixed-point data types. After the conversion, use this test file to compare fixed-point results to the floating-point baseline.

- 3 Prepare algorithm for instrumentation.
- 4 Write an entry-point function.

For instrumentation and code generation, it is convenient to have an entry-point function that calls the function to be converted to fixed point. You can cast the function inputs to different data types, and add calls to different variations of the algorithm for comparison. By using an entry-point function, you can run both fixed-point and floating-point variants of your algorithm. You can also run different variants of fixed-point. This approach allows you to iterate on your code more quickly to arrive at the optimal fixed-point design.

- 5 Build instrumented MEX for original MATLAB algorithm.
- 6 Run your original MATLAB algorithm to log min/max data. View this data in the instrumentation report.
- 7 Separate data types from algorithm.

Convert functions to use types tables and update entry-point function.

- 8 Validate modified function.
 - a Create fixed-point types table based on proposed data types.
 - b Build MEX function.
 - c Run and compare MEX function behavior against baseline.
- 9 Use proposed fixed-point data types.

Create fixed-point types table based on proposed data types, build mex, run, and then compare against baseline.

- 10 Optionally, if have a MATLAB Coder license, generate code.

Start by testing native C-types.

11 Iterate, tune algorithm.

For example, tune the algorithm to avoid overflow or eliminate bias.

Manual Fixed-Point Conversion Best Practices

In this section...

“Create a Test File” on page 12-4

“Prepare Your Algorithm for Code Acceleration or Code Generation” on page 12-5

“Check for Fixed-Point Support for Functions Used in Your Algorithm” on page 12-6

“Manage Data Types and Control Bit Growth” on page 12-7

“Separate Data Type Definitions from Algorithm” on page 12-8

“Convert to Fixed Point” on page 12-10

“Optimize Data Types” on page 12-12

“Optimize Your Algorithm” on page 12-15

Fixed-Point Designer software helps you design and convert your algorithms to fixed point. Whether you are simply designing fixed-point algorithms in MATLAB or using Fixed-Point Designer in conjunction with MathWorks code generation products, these best practices help you get from generic MATLAB code to an efficient fixed-point implementation. These best practices are also covered in this webinar: Manual Fixed-Point Conversion Best Practices Webinar

Create a Test File

A best practice for structuring your code is to separate your core algorithm from other code that you use to test and verify the results. Create a test file to call your original MATLAB algorithm and fixed-point versions of the algorithm. For example, as shown in the following table, you might set up some input data to feed into your algorithm, and then, after you process that data, create some plots to verify the results. Since you need to convert only the algorithmic portion to fixed-point, it is more efficient to structure your code so that you have a test file, in which you create your inputs, call your algorithm, and plot the results, and one (or more) algorithmic files, in which you do the core processing.

Original code	Best Practice	Modified code
<pre>% TEST INPUT x = randn(100,1); % ALGORITHM</pre>	<p>Issue</p> <p>Generation of test input and verification of results</p>	<p>Test file</p> <pre>% TEST INPUT x = randn(100,1);</pre>

Original code	Best Practice	Modified code
<pre> y = zeros(size(x)); y(1) = x(1); for n=2:length(x) y(n)=y(n-1) + x(n); end % VERIFY RESULTS yExpected=cumsum(x); plot(y-yExpected) title('Error') </pre>	<p>are intermingled with the algorithm code.</p> <p>Fix</p> <p>Create a test file that is separate from your algorithm. Put the algorithm in its own function.</p>	<pre> % ALGORITHM y = cumulative_sum(x); % VERIFY RESULTS yExpected = cumsum(x); plot(y-yExpected) title('Error') Algorithm in its own function function y = cumulative_sum(x) y = zeros(size(x)); y(1) = x(1); for n=2:length(x) y(n) = y(n-1) + x(n); end end </pre>

You can use the test file to:

- Verify that your floating-point algorithm behaves as you expect before you convert it to fixed point. The floating-point algorithm behavior is the baseline against which you compare the behavior of the fixed-point versions of your algorithm.
- Propose fixed-point data types.
- Compare the behavior of the fixed-point versions of your algorithm to the floating-point baseline.

Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. For example, for a filter, realistic inputs are impulses, sums of sinusoids, and chirp signals. With these inputs, using linear theory, you can verify that the outputs are correct. Signals that produce maximum output are useful for verifying that your system does not overflow. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want.

Prepare Your Algorithm for Code Acceleration or Code Generation

Using Fixed-Point Designer, you can:

- Instrument your code and provide data type proposals to help you convert your algorithm to fixed point, using the following functions:

- `buildInstrumentedMex`, which generates compiled C code that includes logging instrumentation.
- `showInstrumentationResults`, which shows the results logged by the instrumented, compiled C code.
- `clearInstrumentationResults`, which clears the logged instrumentation results from memory.
- Accelerate your fixed-point algorithms by creating a MEX file using the `fiaccl` function.

Any MATLAB algorithms that you want to instrument using `buildInstrumentedMex` and any fixed-point algorithms that you want to accelerate using `fiaccl` must comply with code generation requirements and rules. To view the subset of the MATLAB language that is supported for code generation, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2.

To help you identify unsupported functions or constructs in your MATLAB code, use one of the following tools.

- Add the `%#codegen` pragma to the top of your MATLAB file. The MATLAB code analyzer flags functions and constructs that are not available in the subset of the MATLAB language supported for code generation. This advice appears in real-time as you edit your code in the MATLAB editor.

For more information, see “Check Code Using the MATLAB Code Analyzer” on page 14-103.

- Use the Code Generation Readiness tool to generate a static report on your code. The report identifies calls to functions and the use of data types that are not supported for code generation. To generate a report for a function, `myFunction1`, at the command line, enter `coder.screener('myFunction1')`.

For more information, see “Check Code Using the Code Generation Readiness Tool” on page 14-102.

Check for Fixed-Point Support for Functions Used in Your Algorithm

Before you start your fixed-point conversion, identify which functions used in your algorithm are not supported for fixed point. Consider how you might replace them or otherwise modify your implementation to be more optimized for embedded targets. For

example, you might need to find (or write your own) replacements for functions like `log2`, `fft`, and `exp`. Other functions like `sin`, `cos`, and `sqrt` may support fixed point, but for better efficiency, you may want to consider an alternative implementation like a lookup table or CORDIC-based algorithm.

If you cannot find a replacement immediately, you can continue converting the rest of your algorithm to fixed point by simply insulating any functions that don't support fixed-point with a cast to double at the input, and a cast back to a fixed-point type at the output.

Original Code	Best Practice	Modified Code
<code>y = 1/exp(x);</code>	<p>Issue</p> <p>The <code>exp()</code> function is not defined for fixed-point inputs.</p> <p>Fix</p> <p>Cast the input to double until you have a replacement. In this case, <code>1/exp(x)</code> is more suitable for fixed-point growth than <code>exp(x)</code>, so replace the whole expression with a <code>1/exp</code> function, possibly as a lookup table.</p>	<code>y = 1/exp(double(x));</code>

Manage Data Types and Control Bit Growth

The `(:)=` syntax is known as subscripted assignment. When you use this syntax, MATLAB overwrites the value of the left-hand side argument, but retains the existing data type and array size. This is particularly important in keeping fixed-point variables fixed point (as opposed to inadvertently turning them into doubles), and for preventing bit growth when you want to maintain a particular data type for the output.

Original Code	Best Practice	Modified Code
<code>acc = 0; for n = 1:numel(x) acc = acc + x(n); end</code>	<p>Issue</p>	<code>acc = 0; for n = 1:numel(x) acc(:) = acc + x(n); end</code>

Original Code	Best Practice	Modified Code
	<p><code>acc = acc + x(n)</code> overwrites <code>acc</code> with <code>acc + x(n)</code>. When you are using all double types, this behavior is fine. However, when you introduce fixed-point data types in your code, if <code>acc</code> is overwritten, the data type of <code>acc</code> might change.</p> <p>Fix</p> <p>To preserve the original data type of <code>acc</code>, assign into <code>acc</code> using <code>acc(:) =</code>. Using subscripted assignment casts the right-hand-side value into the same data type as <code>acc</code> and prevents bit growth.</p>	

For more information, see “Controlling Bit Growth”.

Separate Data Type Definitions from Algorithm

For instrumentation and code generation, create an entry-point function that calls the function that you want to convert to fixed point. You can then cast the function inputs to different data types. You can add calls to different variations of the function for comparison. By using an entry-point function, you can run both fixed-point and floating-point variants of your algorithm. You can also run different variants of fixed-point. This approach allows you to iterate on your code more quickly to arrive at the optimal fixed-point design.

This method of fixed-point conversion makes it easier for you to compare several different fixed-point implementations, and also allows you to easily retarget your algorithm to a different device.

To separate data type definitions from your algorithm:

- 1 When a variable is first defined, use `cast(x, 'like', y)` or `zeros(m, n, 'like', y)` to cast it to your desired data type.

- 2 Create a table of data type definitions, starting with original data types used in your code. Before converting to fixed point, create a data type table that uses all single data types to find type mismatches and other problems.
- 3 Run your code connected to each table and look at the results to verify the connection.

Original code	Best Practice	Modified code
<pre>% Algorithm n = 128; y = zeros(size(n));</pre>	<p>Issue</p> <p>The default data type in MATLAB is double-precision floating-point.</p> <p>Fix</p> <ol style="list-style-type: none"> 1 Use <code>cast(..., 'like', ...)</code> and <code>zeros(..., 'like', ...)</code> to programmatically specify types that are defined in a separate table. 2 Create an original types table, usually in a separate function. 3 Add single data types to your table to help verify the connection with your code. 	<pre>% Algorithm T = mytypes('double'); n = cast(128, 'like', T.n); y = zeros(size(n), 'like', T.y); function T = mytypes(dt) switch(dt) case 'double' T.n = double([]); T.y = double([]); case 'single' T.n = single([]); T.y = single([]); end end</pre>

Separating data type specifications from algorithm code enables you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.

- Switch between variations of fixed-point settings without changing the algorithm code.

Convert to Fixed Point

What Are Your Goals for Converting to Fixed Point?

Before you start the conversion, consider your goals for converting to fixed point. Are you implementing your algorithm in C or HDL? What are your target constraints? The answers to these questions determine many fixed-point properties such as the available word length, fraction length, and math modes, as well as available math libraries.

Build and Run an Instrumented MEX Function

Build and run an instrumented MEX function to get fixed-point types proposals using the `buildInstrumentedMex` and `showInstrumentationResults` functions. Test files should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test file covers the operating range of the algorithm with the accuracy that you want. A simple set of test vectors may not exercise the full range of types, so use the proposals as a guideline for choosing an initial set of fixed-point types, and use your best judgement and experience in adjusting the types. If loop indices are used only as index variables, they are automatically converted to integer types, so you do not have to explicitly convert them to fixed point.

Algorithm Code	Test File
<pre>function [y,z] = myfilter(b,x,z) y = zeros(size(x)); for n = 1:length(x) z(:) = [x(n); z(1:end-1)]; y(n) = b * z; end end</pre>	<pre>% Test inputs b = fir1(11,0.25); t = linspace(0,10*pi,256)'; x = sin((pi/16)*t.^2); % Linear chirp z = zeros(size(b')); % Build buildInstrumentedMex myfilter ... -args {b,x,z} -histogram % Run [y,z] = myfilter_mex(b,x,z); % Show showInstrumentationResults myfilter_mex ... -defaultDT numericType(1,16) -proposeFL</pre>

```

Function: myfilter
1 function [y,z] = myfilter(b,x,z)
2   y = zeros(size(x));
3   for n=1:length(x)
4     z(:) = [x(n); z(1:end-1)];
5     y(n) = b * z;
6   end
7 end

```

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax	Histogram
1	y	Output	256 x 1	double	No	Signed	16	15	No	-0.9514825238809547	0.9831716442570267	
2	z	I/O	12 x 1	double	No	Signed	16	14	No	-0.999998881428792	0.9999956940701684	
3	b	Input	1 x 12	double	No	Signed	16	17	No	-0.004465461051253895	0.24958855452476303	
4	x	Input	256 x 1	double	No	Signed	16	14	No	-0.999998881428792	0.9999956940701684	
5	n	Local	1 x 1	double	No	Signed	16	0	Yes	1	256	

Create a Types Table

Create a types table using a structure with prototypes for the variables. The proposed types are computed from the simulation runs. A long simulation run with a wide range of expected data produces better proposals. You can use the proposed types or use your knowledge of the algorithm and implementation constraints to improve the proposals.

Because the data types, not the values, are used, specify the prototype values as empty ([]).

In some cases, it might be more efficient to leave some parts of the code in floating point. For example, when there is high dynamic range or that part of the code is sensitive to round-off errors.

Algorithm Code

```

function [y,z]=myfilter(b,x,z,T)
  y = zeros(size(x), 'like', T.y);
  for n = 1:length(x)
    z(:) = [x(n); z(1:end-1)];
    y(n) = b * z;
  end
end

```

Types Tables

```

function T = mytypes(dt)
  switch dt
    case 'double'

```

Types Tables

```

T.b = double([]);
T.x = double([]);
T.y = double([]);

case 'fixed16'
    T.b = fi([],true,16,15);
    T.x = fi([],true,16,15);
    T.y = fi([],true,16,14);
end
end

```

Test File

```

% Test inputs
b = fir1(11,0.25);
t = linspace(0,10*pi,256)';
x = sin((pi/16)*t.^2);
% Linear chirp

% Cast inputs
T=mytypes('fixed16');
b=cast(b,'like',T.b);
x=cast(x,'like',T.x);
z=zeros(size(b),'like',T.x);

% Run
[y,z] = myfilter(b,x,z,T);

```

Run With Fixed-Point Types and Compare Results

Create a test file to validate that the floating-point algorithm works as expected before converting it to fixed point. You can use the same test file to propose fixed-point data types, and to compare fixed-point results to the floating-point baseline after the conversion.

Optimize Data Types**Use Scaled Doubles**

Use scaled doubles to detect potential overflows. Scaled doubles are a hybrid between floating-point and fixed-point numbers. Fixed-Point Designer stores them as doubles


with the scaling, sign, and word length information retained. To use scaled doubles, you can use the data type override (DTO) property or you can set the 'DataType' property to 'ScaledDouble' in the `fi` or `numericType` constructor.

To...	Use...	Example
Set data type override locally	<code>numericType</code> <code>DataType</code> property	<pre>T.a = fi([],1,16,13,'DataType', 'ScaledDouble'); a = cast(pi, 'like', T.a) a = 3.1416 DataTypeMode: Scaled double: binary point scaling Signedness: Signed WordLength: 16 FractionLength: 13</pre>
Set data type override globally	<code>fipref</code> <code>DataTypeOverride</code> property	<pre>fipref('DataTypeOverride','ScaledDoubles') T.a = fi([],1,16,13); a = 3.1416 DataTypeMode:Scaled double: binary point scaling Signedness: Signed WordLength:16 FractionLength:13</pre>

For more information, see “Scaled Doubles” on page 32-22.

Use the Histogram to Fine-Tune Data Type Settings

To fine-tune fixed-point type settings, run the `buildInstrumentedMex` function with the `-histogram` flag and then run the generated MEX function with your desired test inputs. When you use the `showInstrumentationResults` to display the code generation report, the report displays a Histogram icon. Click the icon to open the `NumericTypeScope` and view the distribution of values observed in your simulation for the selected variable.

Overflows indicated in red in the Code Generation Report show in the "outside range" bin in the `NumericTypeScope`. Launch the `NumericTypeScope` for an associated variable or expression by clicking on the histogram view icon .

Explore Design Tradeoffs

Once you have your first set of fixed-point data types, you can then add different variations of fixed-point values to your types table. You can modify and iterate to avoid overflows, adjust fraction lengths, and change rounding methods to eliminate bias.

Algorithm Code

```
function [y,z] = myfilter(b,x,z,T)
    y = zeros(size(x),'like',T.y);
    for n = 1:length(x)
        z(:) = [x(n); z(1:end-1)];
        y(n) = b * z;
    end
end
```

Types Tables

```
function T = mytypes(dt)
    switch dt
        case 'double'
            T.b = double([]);
            T.x = double([]);
            T.y = double([]);

        case 'fixed8'
            T.b = fi([],true,8,7);
            T.x = fi([],true,8,7);
            T.y = fi([],true,8,6);

        case 'fixed16'
            T.b = fi([],true,16,15);
            T.x = fi([],true,16,15);
            T.y = fi([],true,16,14);

    end
end
```

Test File

```
function mytest
    % Test inputs
    b = fir1(11,0.25);
    t = linspace(0,10*pi,256)';
    x = sin((pi/16)*t.^2); % Linear chirp

    % Run
```

Test File

```

y0 = entrypoint('double',b,x);
y8 = entrypoint('fixed8',b,x);
y16 = entrypoint('fixed16',b,x);

% Plot
subplot(3,1,1)
plot(t,x,'c',t,y0,'k')
legend('Input','Baseline output')
title('Baseline')

subplot(3,2,3)
plot(t,y8,'k')
title('8-bit fixed-point output')
subplot(3,2,4)
plot(t,y0-double(y8),'r')
title('8-bit fixed-point error')

subplot(3,2,5)
plot(t,y16,'k')
title('16-bit fixed-point output')
xlabel('Time (s)')
subplot(3,2,6)
plot(t,y0-double(y16),'r')
title('16-bit fixed-point error')
xlabel('Time (s)')
end

function [y,z] = entrypoint(dt,b,x)
    T = mytypes(dt);
    b = cast(b,'like',T.b);
    x = cast(x,'like',T.x);
    z = zeros(size(b),'like',T.x);
    [y,z] = myfilter(b,x,z,T);
end

```

Optimize Your Algorithm

Use `fimath` to Get Natural Types for C or HDL

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. You can use the `fimath`

ProductMode and SumMode properties to retain natural data types for C and HDL. The KeepLSB setting for ProductMode and SumMode models the behavior of integer operations in the C language, while KeepMSB models the behavior of many DSP devices. Different rounding methods require different amounts of overhead code. Setting the RoundingMethod property to Floor, which is equivalent to two's complement truncation, provides the most efficient rounding implementation. Similarly, the standard method for handling overflows is to wrap using modulo arithmetic. Other overflow handling methods create costly logic. Whenever possible, set the OverflowAction to Wrap.

MATLAB Code	Best Practice	Generated C Code
<pre>% Code being compiled function y = adder(a,b) y = a + b; end With types defined with default fimath settings: T.a = fi([],1,16,0); T.b = fi([],1,16,0); a = cast(0,'like',T.a); b = cast(0,'like',T.b);</pre>	<p>Issue</p> <p>Additional code is generated to implement saturation overflow, nearest rounding, and full-precision arithmetic.</p>	<pre>int adder(short a, short b) { int y; int i0; int i1; int i2; int i3; i0 = a; i1 = b; if ((i0 & 65536) != 0) { i2 = i0 -65536; } else { i2 = i0 & 65535; } if ((i1 & 65536) != 0) { i3 = i1 -65536; } else { i3 = i1 & 65535; } i0 = i2 + i3; if ((i0 & 65536) != 0) { y = i0 -65536; } else { y = i0 & 65535; } return y; }</pre>
Code being compiled	Fix	int adder(short a, short b)

MATLAB Code	Best Practice	Generated C Code
<pre>function y = adder(a,b) y = a + b; end</pre> <p>With types defined with <code>fimath</code> settings that match your processor types:</p> <pre>F = fimath(... 'RoundingMethod','Floor', ... 'OverflowAction','Wrap', ... 'ProductMode','KeepLSB', ... 'ProductWordLength',32, ... 'SumMode','KeepLSB', ... 'SumWordLength',32);</pre> <pre>T.a = fi([],1,16,0,F); T.b = fi([],1,16,0,F); a = cast(0,'like',T.a); b = cast(0,'like',T.b);</pre>	<p>To make the generated code more efficient, choose fixed-point math settings that match your processor types.</p>	<pre>{ return a + b; }</pre>

Replace Built-in Functions With More Efficient Fixed-Point Implementations

Some MATLAB built-in functions can be made more efficient for fixed-point implementation. For example, you can replace a built-in function with a Lookup table implementation, or a CORDIC implementation, which requires only iterative shift-add operations.

Re-implement Division Operations Where Possible

Often, division is not fully supported by hardware and can result in slow processing. When your algorithm requires a division, consider replacing it with one of the following options:

- Use bit shifting when the denominator is a power of two. For example, `bitsra(x,3)` instead of `x/8`.
- Multiply by the inverse when the denominator is constant. For example, `x*0.2` instead of `x/5`.

Eliminate Floating-Point Variables

For more efficient code, eliminate floating-point variables. The one exception to this is loop indices because they usually become integer types.

Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros

This example shows you how to convert a finite impulse-response (FIR) filter to fixed point by separating the fixed-point type specification from the algorithm code.

Separating data type type specification from algorithm code allows you to:

- Re-use your algorithm code with different data types
- Keep your algorithm uncluttered with data type specification and switch statements for different data types
- Keep your algorithm code more readable
- Switch between fixed point and floating point to compare baselines
- Switch between variations of fixed point settings without changing the algorithm code

Original Algorithm

This example converts MATLAB® code for a finite impulse response (FIR) filter to fixed point.

The formula for the n'th output $y(n)$ of an (FIR) filter, given filter coefficients b , and input x is:

$$y(n) = b(1)*x(n) + b(2)*x(n-1) + \dots + b(\text{end})*x(n-\text{length}(b)+1)$$

Linear Buffer Implementation

There are several different ways to write an FIR filter. One way is with a linear buffer like in the following function, where b is a row vector and z is a column vector the same length as b .

```
function [y,z] = fir_filt_linear_buff(b,x,z)
    y = zeros(size(x));
    for n=1:length(x)
        z = [x(n); z(1:end-1)];
        y(n) = b * z;
    end
end
```

The linear buffer implementation takes advantage of MATLAB's convenient matrix syntax and is easy to read and understand. However, it introduces a full copy of the state buffer for every sample of the input.

Circular Buffer Implementation

To implement the FIR filter more efficiently, you can store the states in a circular buffer, z , whose elements are $z(p) = x(n)$, where $p = \text{mod}(n-1, \text{length}(b))+1$, for $n=1, 2, 3, \dots$

For example, let $\text{length}(b) = 3$, and initialize p and z to:

$$p = 0, z = [0 \quad 0 \quad 0]$$

Start with the first sample and fill the state buffer z in a circular manner.

$$n = 1, p = 1, z(1) = x(1), z = [x(1) \quad 0 \quad 0] \\ y(1) = b(1)*z(1) + b(2)*z(3) + b(3)*z(2)$$

$$n = 2, p = 2, z(2) = x(2), z = [x(1) \quad x(2) \quad 0] \\ y(2) = b(1)*z(2) + b(2)*z(1) + b(3)*z(3)$$

$$n = 3, p = 3, z(3) = x(3), z = [x(1) \quad x(2) \quad x(3)] \\ y(3) = b(1)*z(3) + b(2)*z(2) + b(3)*z(1)$$

$$n = 4, p = 1, z(1) = x(4), z = [x(4) \quad x(2) \quad x(3)] \\ y(4) = b(1)*z(1) + b(2)*z(3) + b(3)*z(2)$$

$$n = 5, p = 2, z(2) = x(5), z = [x(4) \quad x(5) \quad x(3)] \\ y(5) = b(1)*z(2) + b(2)*z(1) + b(3)*z(3)$$

$$n = 6, p = 3, z(3) = x(6), z = [x(4) \quad x(5) \quad x(6)] \\ y(6) = b(1)*z(3) + b(2)*z(2) + b(3)*z(1)$$

...

You can implement the FIR filter using a circular buffer like the following MATLAB function.

```
function [y,z,p] = fir_filt_circ_buff_original(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n=1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
```



```
        acc = acc + b(j)*z(k);  
        k=k-1; if k<1, k=nb; end  
    end  
    y(n) = acc;  
end  
end
```

Test File

Create a test file to validate that the floating-point algorithm works as expected before converting it to fixed point. You can use the same test file to propose fixed-point data types, and to compare fixed-point results to the floating-point baseline after the conversion.

The test vectors should represent realistic inputs that exercise the full range of values expected by your system. Realistic inputs are impulses, sums of sinusoids, and chirp signals, for which you can verify that the outputs are correct using linear theory. Signals that produce maximum output are useful for verifying that your system does not overflow.

Set up

Run the following code to capture and reset the current state of global fixed-point math settings and fixed-point preferences.

```
resetglobalfimath;  
FIPREF_STATE = get(fipref);  
resetfipref;
```

Run the following code to copy the test functions into a temporary folder so this example doesn't interfere with your own work.

```
tempdirObj = fidemo.fiTempdir('fir_filt_circ_buff_fixed_point_conversion_example');  
copyfile(fullfile(matlabroot, 'toolbox', 'fixedpoint', 'fidemos', '+fidemo', ...  
    'fir_filt_*.m'), '.', 'f');
```

Filter coefficients

Use the following low-pass filter coefficients that were computed using the `fir1` function from Signal Processing Toolbox.

```
b = fir1(11,0.25);
```

```
b = [-0.004465461051254
     -0.004324228005260
     +0.012676739550326
     +0.074351188907780
     +0.172173206073645
     +0.249588554524763
     +0.249588554524763
     +0.172173206073645
     +0.074351188907780
     +0.012676739550326
     -0.004324228005260
     -0.004465461051254]';
```

Time vector

Use this time vector to create the test signals.

```
nx = 256;
t = linspace(0,10*pi,nx)';
```

Impulse input

The response of an FIR filter to an impulse input is the filter coefficients themselves.

```
x_impulse = zeros(nx,1); x_impulse(1) = 1;
```

Signal that produces the maximum output

The maximum output of a filter occurs when the signs of the inputs line up with the signs of the filter's impulse response.

```
x_max_output = sign(fliplr(b))';
x_max_output = repmat(x_max_output,ceil(nx/length(b)),1);
x_max_output = x_max_output(1:nx);
```

The maximum magnitude of the output is the 1-norm of its impulse response, which is $\text{norm}(b,1) = \text{sum}(\text{abs}(b))$.

```
maximum_output_magnitude = norm(b,1) %#ok<*NOPTS>
```

```
maximum_output_magnitude =
```

```
1.0352
```

Sum of sines

A sum of sines is a typical input for a filter and you can easily see the high frequencies filtered out in the plot.

```
f0=0.1; f1=2;  
x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);
```

Chirp

A chirp gives a good visual of the low-pass filter action of passing the low frequencies and attenuating the high frequencies.

```
f_chirp = 1/16; % Target frequency  
x_chirp = sin(pi*f_chirp*t.^2); % Linear chirp  
  
titles = {'Impulse', 'Max output', 'Sum of sines', 'Chirp'};  
x = [x_impulse, x_max_output, x_sines, x_chirp];
```

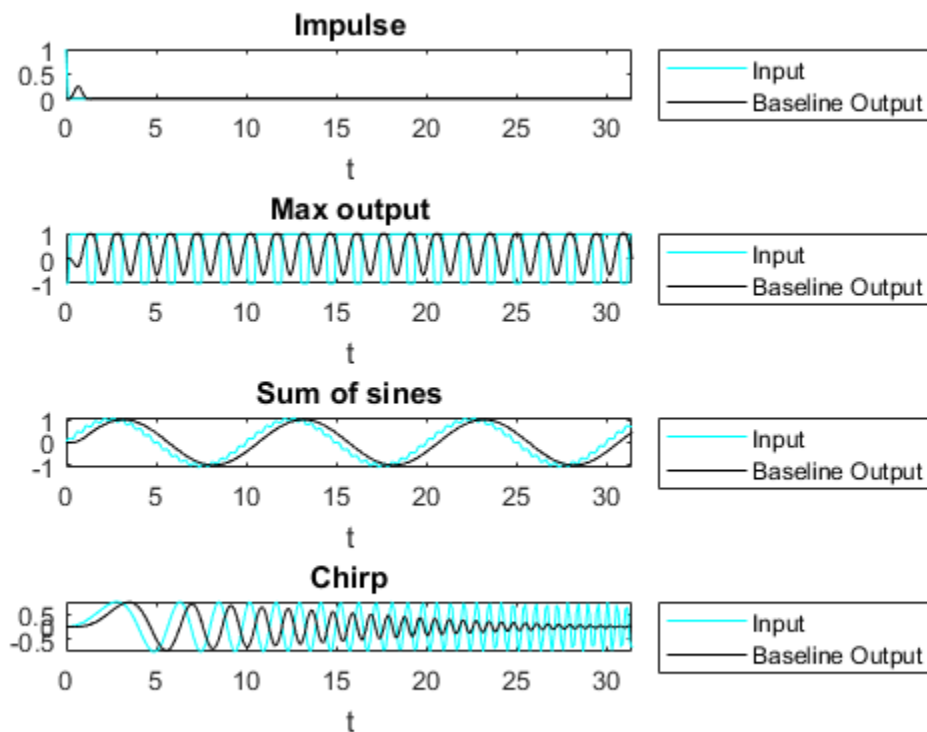
Call the original function

Before starting the conversion to fixed point, call your original function with the test file inputs to establish a baseline to compare to subsequent outputs.

```
y0 = zeros(size(x));  
for i=1:size(x,2)  
    % Initialize the states for each column of input  
    p = 0;  
    z = zeros(size(b));  
    y0(:,i) = fir_filt_circ_buff_original(b,x(:,i),z,p);  
end
```

Baseline Output

```
fir_filt_circ_buff_plot(1,titles,t,x,y0)
```



Prepare for Instrumentation and Code Generation

The first step after the algorithm works in MATLAB is to prepare it for instrumentation, which requires code generation. Before the conversion, you can use the `coder.screener` function to analyze your code and identify unsupported functions and language features.

Entry-point function

When doing instrumentation and code generation, it is convenient to have an entry-point function that calls the function to be converted to fixed point. You can cast the FIR filter's inputs to different data types, and add calls to different variations of the filter for comparison. By using an entry-point function you can run both fixed-point and floating-

point variants of your filter, and also different variants of fixed-point. This allows you to iterate on your code more quickly to arrive at the optimal fixed-point design.

```
function y = fir_filt_circ_buff_original_entry_point(b,x,reset)
    if nargin<3, reset = true; end
    % Define the circular buffer z and buffer position index p.
    % They are declared persistent so the filter can be called in a streaming
    % loop, each section picking up where the last section left off.
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filt_circ_buff_original(b,x,z,p);
end
```

Test file

Your test file calls the compiled entry-point function.

```
function y = fir_filt_circ_buff_test(b,x)

    y = zeros(size(x));

    for i=1:size(x,2)
        reset = true;
        y(:,i) = fir_filt_circ_buff_original_entry_point_mex(b,x(:,i),reset);
    end

end
```

Build original function

Compile the original entry-point function with buildInstrumentedMex. This instruments your code for logging so you can collect minimum and maximum values from the simulation and get proposed data types.

```
reset = true;
buildInstrumentedMex fir_filt_circ_buff_original_entry_point -args {b, x(:,1), reset}
```

Run original function

Run your test file inputs through the algorithm to log minimum and maximum values.

```
y1 = fir_filt_circ_buff_test(b,x);
```

Show types

Use `showInstrumentationResults` to view the data types of all your variables and the minimum and maximum values that were logged during the test file run. Look at the maximum value logged for the output variable `y` and accumulator variable `acc` and note that they attained the theoretical maximum output value that you calculated previously.

```
showInstrumentationResults fir_filt_circ_buff_original_entry_point_mex
```

To see these results in the instrumented Code Generation Report:

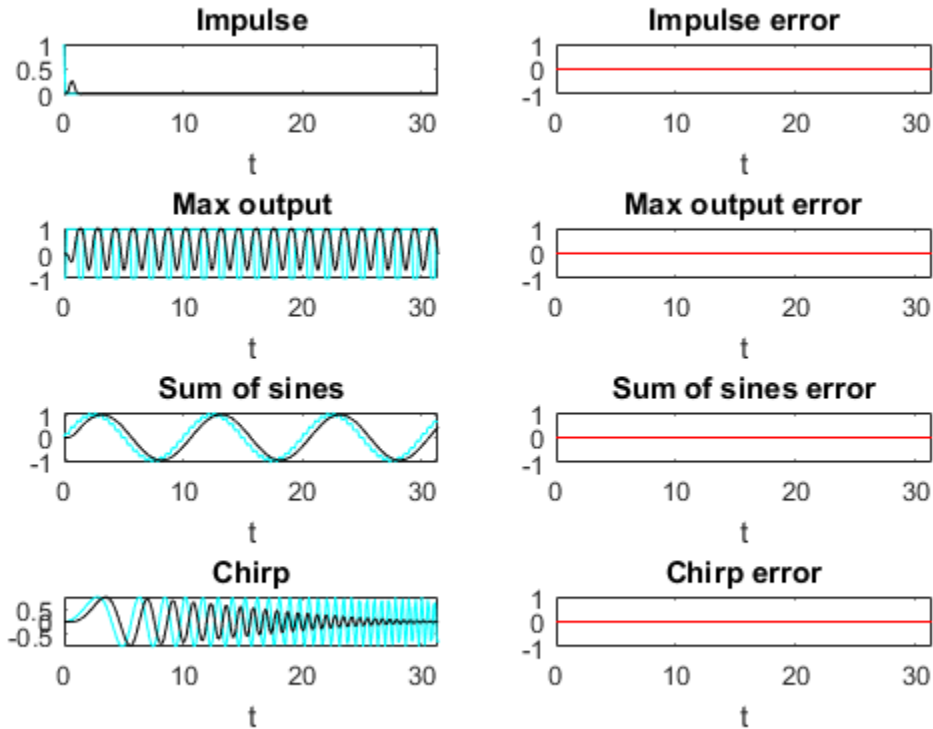
- Select function `fir_filt_circ_buff_original`
- Select the Variables tab

Variable	Type	Size	Class	Complex	Always Whole Number	SimMin	SimMax
acc	Local	1 x 1	double	No	No	-1.0045281391112986	1.035158756226056
b	Input	1 x 12	double	No	No	-0.004465461051254	0.249588554524763
j	Local	1 x 1	double	No	Yes	1	12
k	Local	1 x 1	double	No	Yes	0	12
n	Local	1 x 1	double	No	Yes	1	256
nb	Local	1 x 1	double	No	Yes	12	12
nx	Local	1 x 1	double	No	Yes	256	256
p	I/O	1 x 1	double	No	Yes	0	13
x	Input	256 x 1	double	No	No	-1.0966086573451541	1.0874250377226369
y	Output	256 x 1	double	No	No	-0.9959923266057762	1.035158756226056
z	I/O	1 x 12	double	No	No	-1.0966086573451541	1.0874250377226369

Validate original function

Every time you modify your function, validate that the results still match your baseline.

```
fir_filt_circ_buff_plot2(2,titles,t,x,y0,y1)
```



Convert Functions to use Types Tables

To separate data types from the algorithm, you:

- 1 Create a table of data type definitions.
- 2 Modify the algorithm code to use data types from that table.

This example shows the iterative steps by creating different files. In practice, you can make the iterative changes to the same file.

Original types table

Create a types table using a structure with prototypes for the variables set to their original types. Use the baseline types to validate that you made the initial conversion

correctly, and also use it to programatically toggle your function between floating point and fixed point types. The index variables j , k , n , nb , nx are automatically converted to integers by MATLAB Coder™, so you don't need to specify their types in the table.

Specify the prototype values as empty (`[]`) since the data types are used, but not the values.

```
function T = fir_filt_circ_buff_original_types()
    T.acc=double([]);
    T.b=double([]);
    T.p=double([]);
    T.x=double([]);
    T.y=double([]);
    T.z=double([]);
end
```

Type-aware filter function

Prepare the filter function and entry-point function to be type-aware by using the `cast` and `zeros` functions and the types table.

Use subscripted assignment `acc(:)=...`, `p(:)=1`, and `k(:)=nb` to preserve data types during assignment. See the "Cast fi Objects" section in the Fixed-Point Designer documentation for more details about subscripted assignment and preserving data types.

The function call `y = zeros(size(x),'like',T.y)` creates an array of zeros the same size as `x` with the properties of variable `T.y`. Initially, `T.y` is a double defined in function `fir_filt_circ_buff_original_types`, but it is re-defined as a fixed-point type later in this example.

The function call `acc = cast(0,'like',T.acc)` casts the value `0` with the same properties as variable `T.acc`. Initially, `T.acc` is a double defined in function `fir_filt_circ_buff_original_types`, but it is re-defined as a fixed-point type later in this example.

```
function [y,z,p] = fir_filt_circ_buff_typed(b,x,z,p,T)
    y = zeros(size(x),'like',T.y);
    nx = length(x);
    nb = length(b);
    for n=1:nx
        p(:)=p+1; if p>nb, p(:)=1; end
        z(p) = x(n);
        acc = cast(0,'like',T.acc);
    end
end
```



```

        k = p;
        for j=1:nb
            acc(:) = acc + b(j)*z(k);
            k(:)=k-1; if k<1, k(:)=nb; end
        end
        y(n) = acc;
    end
end

```

Type-aware entry-point function

The function call `p1 = cast(0,'like',T1.p)` casts the value 0 with the same properties as variable `T1.p`. Initially, `T1.p` is a double defined in function `fir_filt_circ_buff_original_types`, but it is re-defined as an integer type later in this example.

The function call `z1 = zeros(size(b),'like',T1.z)` creates an array of zeros the same size as `b` with the properties of variable `T1.z`. Initially, `T1.z` is a double defined in function `fir_filt_circ_buff_original_types`, but it is re-defined as a fixed-point type later in this example.

```

function y1 = fir_filt_circ_buff_typed_entry_point(b,x,reset)
    if nargin<3, reset = true; end
    %
    % Baseline types
    %
    T1 = fir_filt_circ_buff_original_types();
    % Each call to the filter needs to maintain its own states.
    persistent z1 p1
    if isempty(z1) || reset
        p1 = cast(0,'like',T1.p);
        z1 = zeros(size(b),'like',T1.z);
    end
    b1 = cast(b,'like',T1.b);
    x1 = cast(x,'like',T1.x);
    [y1,z1,p1] = fir_filt_circ_buff_typed(b1,x1,z1,p1,T1);
end

```

Validate modified function

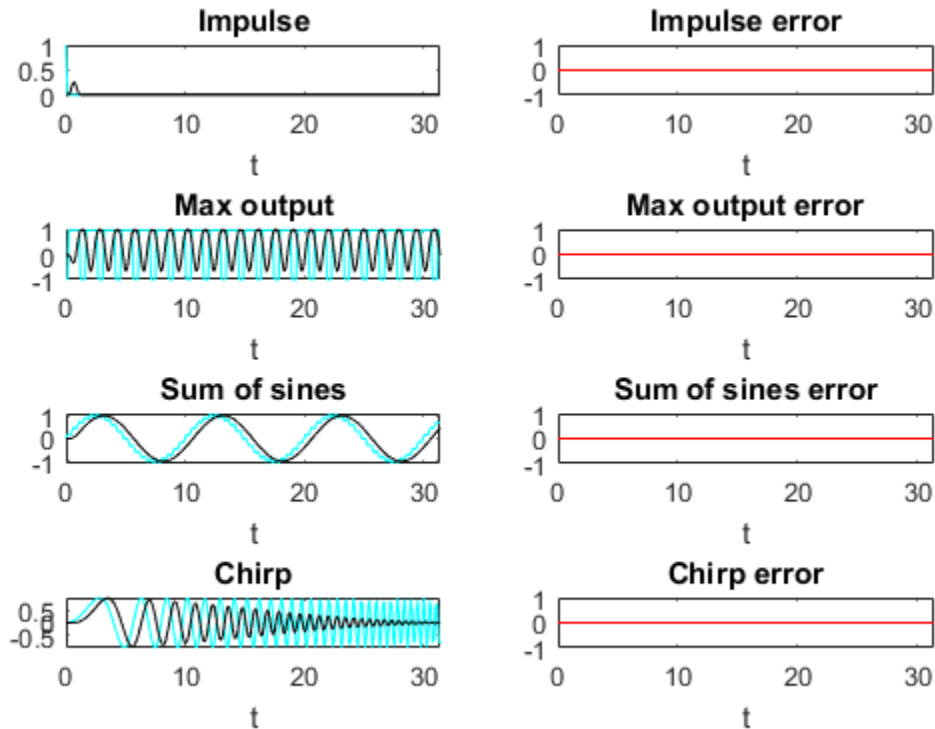
Every time you modify your function, validate that the results still match your baseline. Since you used the original types in the types table, the outputs should be identical. This validates that you made the conversion to separate the types from the algorithm correctly.

```

buildInstrumentedMex fir_filt_circ_buff_typed_entry_point -args {b, x(:,1), reset}

y1 = fir_filt_circ_buff_typed_test(b,x);
fir_filt_circ_buff_plot2(3,titles,t,x,y0,y1)

```



Propose data types from simulation min/max logs

Use the `showInstrumentationResults` function to propose fixed-point fraction lengths, given a default signed fixed-point type and 16-bit word length.

```

showInstrumentationResults fir_filt_circ_buff_original_entry_point_mex ...
    -defaultDT numerictype(1,16) -proposeFL

```

In the instrumented Code Generation Report, select function `fir_filt_circ_buff_original` and the Variables tab to see these results.

Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax
acc	Local	1 x 1	double	No	Signed	16	14	No	-1.0045281391112986	1.035158756226056
b	Input	1 x 12	double	No	Signed	16	17	No	-0.004465461051254	0.249588554524763
j	Local	1 x 1	double	No	Signed	16	0	Yes	1	12
k	Local	1 x 1	double	No	Signed	16	0	Yes	0	12
n	Local	1 x 1	double	No	Signed	16	0	Yes	1	256
nb	Local	1 x 1	double	No	Signed	16	0	Yes	12	12
nx	Local	1 x 1	double	No	Signed	16	0	Yes	256	256
p	I/O	1 x 1	double	No	Signed	16	0	Yes	0	13
x	Input	256 x 1	double	No	Signed	16	14	No	-1.0966086573451541	1.0874250377226369
y	Output	256 x 1	double	No	Signed	16	14	No	-0.9959923266057762	1.035158756226056
z	I/O	1 x 12	double	No	Signed	16	14	No	-1.0966086573451541	1.0874250377226369

Create a fixed-point types table

Use the proposed types from the Code Generation Report to guide you in choosing fixed-point types and create a fixed-point types table using a structure with prototypes for the variables.

Use your knowledge of the algorithm to improve on the proposals. For example, you are using the `acc` variable as an accumulator, so make it 32-bits. From the Code Generation Report, you can see that `acc` needs at least 2 integer bits to prevent overflow, so set the fraction length to 30.

Variable `p` is used as an index, so you can make it a builtin 16-bit integer.

Specify the prototype values as empty (`[]`) since the data types are used, but not the values.

```
function T = fir_filt_circ_buff_fixed_point_types()
    T.acc=fi([],true,32,30);
    T.b=fi([],true,16,17);
    T.p=int16([]);
    T.x=fi([],true,16,14);
    T.y=fi([],true,16,14);
    T.z=fi([],true,16,14);
end
```

Add fixed point to entry-point function

Add a call to the fixed-point types table in the entry-point function:

```
T2 = fir_filt_circ_buff_fixed_point_types();
```

```

persistent z2 p2
if isempty(z2) || reset
    p2 = cast(0, 'like', T2.p);
    z2 = zeros(size(b), 'like', T2.z);
end
b2 = cast(b, 'like', T2.b);
x2 = cast(x, 'like', T2.x);
[y2,z2,p2] = fir_filt_circ_buff_typed(b2,x2,z2,p2,T2);

```

Build and run algorithm with fixed-point data types

```
buildInstrumentedMex fir_filt_circ_buff_typed_entry_point -args {b, x(:,1), reset}
```

```
[y1,y2] = fir_filt_circ_buff_typed_test(b,x);
```

```
showInstrumentationResults fir_filt_circ_buff_typed_entry_point_mex
```

To see these results in the instrumented Code Generation Report:

- Select the entry-point function, `fir_filt_circ_buff_typed_entry_point`
- Select `fir_filt_circ_buff_typed` in the following line of code:

```
[y2,z2,p2] = fir_filt_circ_buff_typed(b2,x2,z2,p2,T2);
```

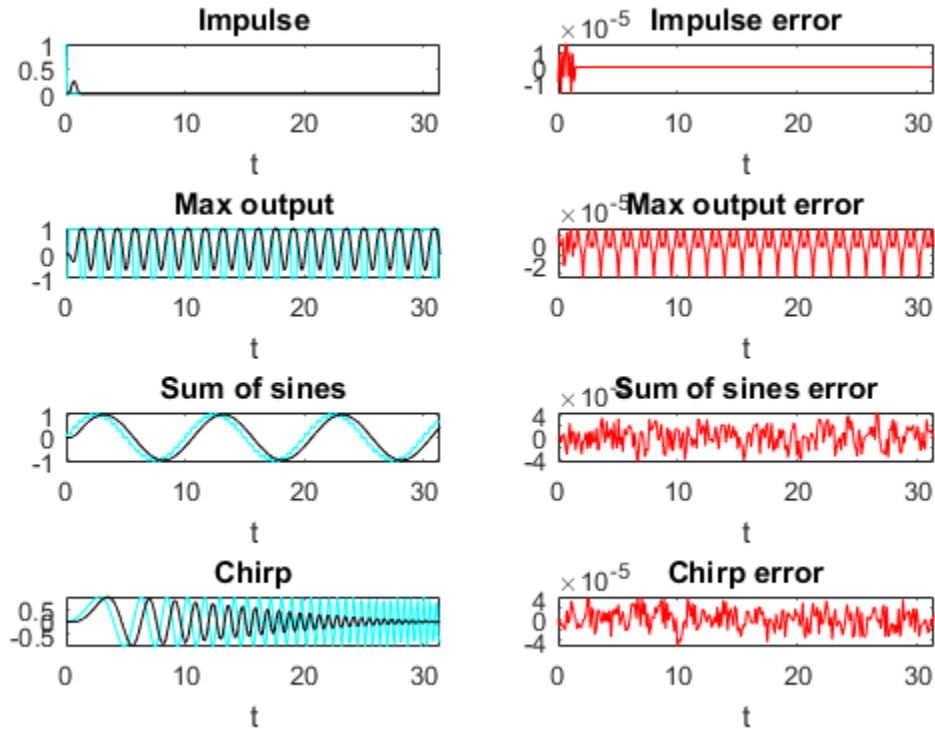
- Select the Variables tab

Variable	Type	Size	Class	Complex	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
acc	Local	1 x 1	embedded.fi	No	Signed	32	30	52	No	-1.004551738500595	1.03515625
b	Input	1 x 12	embedded.fi	No	Signed	16	17	100	No	-0.00446319580078125	0.2495880126953125
j	Local	1 x 1	double	No	-	-	-	-	Yes	1	12
k	Local	1 x 1	int16	No	-	-	-	-	Yes	0	12
n	Local	1 x 1	double	No	-	-	-	-	Yes	1	256
nb	Local	1 x 1	double	No	-	-	-	-	Yes	12	12
nx	Local	1 x 1	double	No	-	-	-	-	Yes	256	256
p	I/O	1 x 1	int16	No	-	-	-	-	Yes	0	13
T	Input	1 x 1	struct	-	-	-	-	-	-	-	-
x	Input	256 x 1	embedded.fi	No	Signed	16	14	55	No	-1.09661865234375	1.08740234375
y	Output	256 x 1	embedded.fi	No	Signed	16	14	52	No	-0.9959716796875	1.03515625
z	I/O	1 x 12	embedded.fi	No	Signed	16	14	55	No	-1.09661865234375	1.08740234375

16-bit word length, full precision math

Validate that the results are within an acceptable tolerance of your baseline.

```
fir_filt_circ_buff_plot2(4,titles,t,x,y1,y2);
```



Your algorithm has now been converted to fixed-point MATLAB code. If you also want to convert to C-code, then proceed to the next section.

Generate C-Code

This section describes how to generate efficient C-code from the fixed-point MATLAB code from the previous section.

Required products

You need MATLAB Coder™ to generate C-code, and you need Embedded Coder® for the hardware implementation settings used in this example.

Algorithm tuned for most efficient C-code

The output variable `y` is initialized to zeros, and then completely overwritten before it is used. Therefore, filling `y` with all zeros is unnecessary. You can use the `coder.nullcopy` function to declare a variable without actually filling it with values, which makes the code in this case more efficient. However, you have to be very careful when using `coder.nullcopy` because if you access an element of a variable before it is assigned, then you are accessing uninitialized memory and its contents are unpredictable.

A rule of thumb for when to use `coder.nullcopy` is when the initialization takes significant time compared to the rest of the algorithm. If you are not sure, then the safest thing to do is to not use it.

```
function [y,z,p] = fir_filt_circ_buff_typed_codegen(b,x,z,p,T)
    % Use coder.nullcopy only when you are certain that every value of
    % the variable is overwritten before it is used.
    y = coder.nullcopy(zeros(size(x),'like',T.y));
    nx = length(x);
    nb = length(b);
    for n=1:nx
        p(:)=p+1; if p>nb, p(:)=1; end
        z(p) = x(n);
        acc = cast(0,'like',T.acc);
        k = p;
        for j=1:nb
            acc(:) = acc + b(j)*z(k);
            k(:)=k-1; if k<1, k(:)=nb; end
        end
        y(n) = acc;
    end
end
```

Native C-code types

You can set the fixed-point math properties to match the native actions of C. This generates the most efficient C-code, but this example shows that it can create problems with overflow and produce less accurate results which are corrected in the next section. It doesn't always create problems, though, so it is worth trying first to see if you can get the cleanest possible C-code.

Set the fixed-point math properties to use floor rounding and wrap overflow because those are the default actions in C.

Set the fixed-point math properties of products and sums to match native C 32-bit integer types, and to keep the least significant bits (LSBs) of math operations.

Add these settings to a fixed-point types table.

```
function T = fir_filt_circ_buff_dsp_types()
    F = fimath('RoundingMethod','Floor',...
             'OverflowAction','Wrap',...
             'ProductMode','KeepLSB',...
             'ProductWordLength',32,...
             'SumMode','KeepLSB',...
             'SumWordLength',32);
    T.acc=fi([],true,32,30,F);
    T.p=int16([]);
    T.b=fi([],true,16,17,F);
    T.x=fi([],true,16,14,F);
    T.y=fi([],true,16,14,F);
    T.z=fi([],true,16,14,F);
end
```

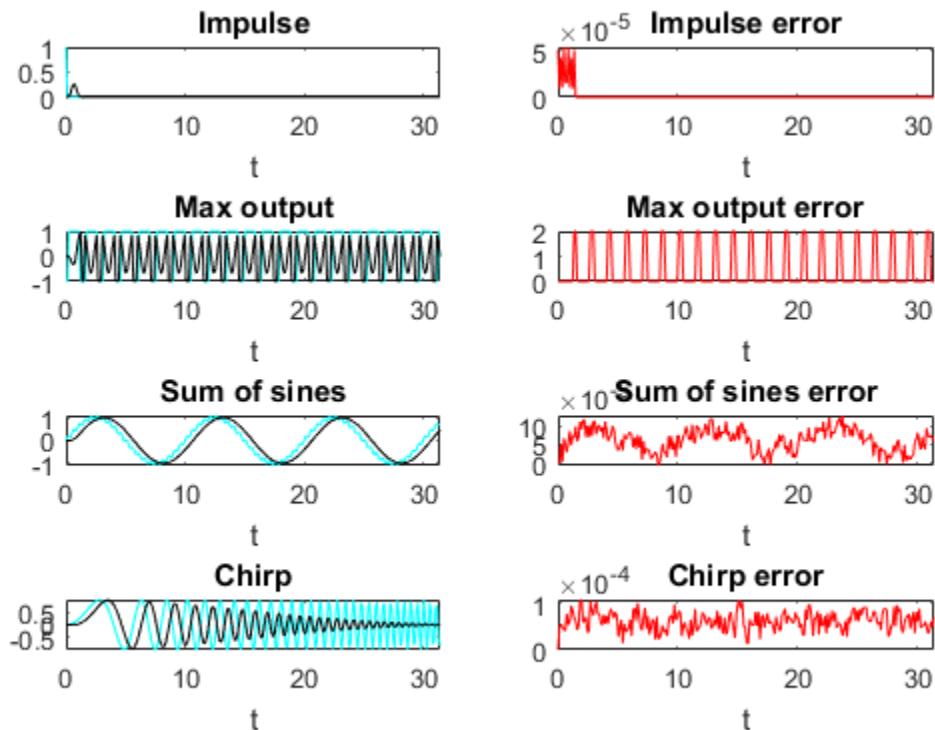
Test the native C-code types

Add a call to the types table in the entry-point function and run the test file.

```
[y1,y2,y3] = fir_filt_circ_buff_typed_test(b,x); %#ok<*ASGLU>
```

In the second row of plots, you can see that the maximum output error is twice the size of the input, indicating that a value that should have been positive overflowed to negative. You can also see that the other outputs did not overflow. This is why it is important to have your test file exercise the full range of values in addition to other typical inputs.

```
fir_filt_circ_buff_plot2(5,titles,t,x,y1,y3);
```



Scaled Double types to find overflows

Scaled double variables store their data in double-precision floating-point, so they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type.

Change the data types to scaled double, and add these settings to a scaled-double types table.

```
function T = fir_filt_circ_buff_scaled_double_types()
    F = fimath('RoundingMethod','Floor',...
              'OverflowAction','Wrap',...
              'ProductMode','KeepLSB',...
              'ProductWordLength',32,...
```



```

        'SumMode', 'KeepLSB', ...
        'SumWordLength', 32);
DT = 'ScaledDouble';
T.acc=fi([],true,32,30,F,'DataType',DT);
T.p=int16([]);
T.b=fi([],true,16,17,F,'DataType',DT);
T.x=fi([],true,16,14,F,'DataType',DT);
T.y=fi([],true,16,14,F,'DataType',DT);
T.z=fi([],true,16,14,F,'DataType',DT);
end

```

Add a call to the the scaled-double types table to the entry-point function and run the test file.

```
[y1,y2,y3,y4] = fir_filt_circ_buff_typed_test(b,x); %#ok<*NASGU>
```

Show the instrumentation results with the scaled-double types.

```
showInstrumentationResults fir_filt_circ_buff_typed_entry_point_mex
```

To see these results in the instrumented Code Generation Report:

- Select the entry-point function, `fir_filt_circ_buff_typed_entry_point`
- Select `fir_filt_circ_buff_typed_codegen` in the following line of code:

```
[y4,z4,p4] = fir_filt_circ_buff_typed_codegen(b4,x4,z4,p4,T4);
```

- Select the Variables tab.
- Look at the variables in the table. None of the variables overflowed, which indicates that the overflow occurred as the result of an operation.
- Hover over the operators in the report (+, -, *, =).
- Hover over the "+" in this line of MATLAB code in the instrumented Code Generation Report:

```
acc(:) = acc + b(j)*z(k);
```

The report shows that the sum overflowed:

```

% Use coder.nullcopy only when you are certain that even
% the variable will be overwritten before it is used.
y = coder.nullcopy(zeros(size(x),'like',T.y));
nx = length(x);
nb = length(b);
for n=1:nx
    p(:)=p+1; if p>nb, p(:)=1; end
    z(p) = x(n);
    acc = cast(0,'like',T.acc);
    k = p;
    for j=1:nb
        acc(:) = acc + b(j)*z(k);
        k(:)=k-1; if k
    end
    y(n) = acc;
end

```

Primary	All Messages (0)	Variable
Order	Variable	Type
	y	Output
	z	I/O
	p	I/O

Information for the selected expression:	
Size	1 x 1
Class	embedded.fi
Complex	No
DT Mode	ScaledDouble
Signedness	Signed
WL	32
FL	31
Percent of Current Range	104
Always Whole Number	No
SimMin	-1.0045281391112986
SimMax	1.035158756226056

The reason the sum overflowed is that a full-precision product for $b(j)*z(k)$ produces a `numerictype(true,32,31)` because b has `numerictype(true,16,17)` and z has

numerictype(true,16,14). The sum type is set to "keep least significant bits" (KeepLSB), so the sum has numerictype(true,32,31). However, 2 integer bits are necessary to store the minimum and maximum simulated values of -1.0045 and +1.035, respectively.

Adjust to avoid the overflow

Set the fraction length of b to 16 instead of 17 so that $b(j)*z(k)$ is numerictype(true,32,30), and so the sum is also numerictype(true,32,30) following the KeepLSB rule for sums.

Leave all other settings the same, and set

```
T.b=fir([],true,16,16,F);
```

Then the sum in this line of MATLAB code no longer overflows:

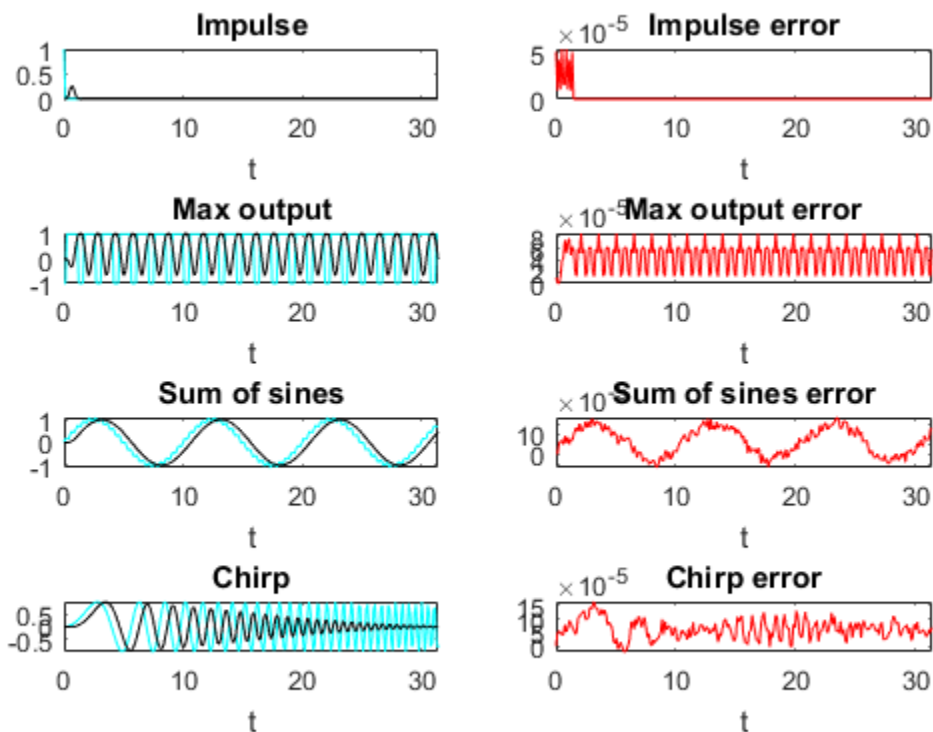
```
acc(:) = acc + b(j)*z(k);
```

Run the test file with the new settings and plot the results.

```
[y1,y2,y3,y4,y5] = fir_filt_circ_buff_typed_test(b,x);
```

You can see that the overflow has been avoided. However, the plots show a bias and a larger error due to using C's natural floor rounding. If this bias is acceptable to you, then you can stop here and the generated C-code is very clean.

```
fir_filt_circ_buff_plot2(6,titles,t,x,y1,y5);
```



Eliminate the bias

If the bias is not acceptable in your application, then change the rounding method to 'Nearest' to eliminate the bias. Rounding to nearest generates slightly more complicated C-code, but it may be necessary for you if you want to eliminate the bias and have a smaller error.

The final fixed-point types table with nearest rounding and adjusted coefficient fraction length is:

```
function T = fir_filt_circ_buff_dsp_nearest_types()
    F = fimath('RoundingMethod','Nearest',...
              'OverflowAction','Wrap',...
              'ProductMode','KeepLSB',...
              ...)
```

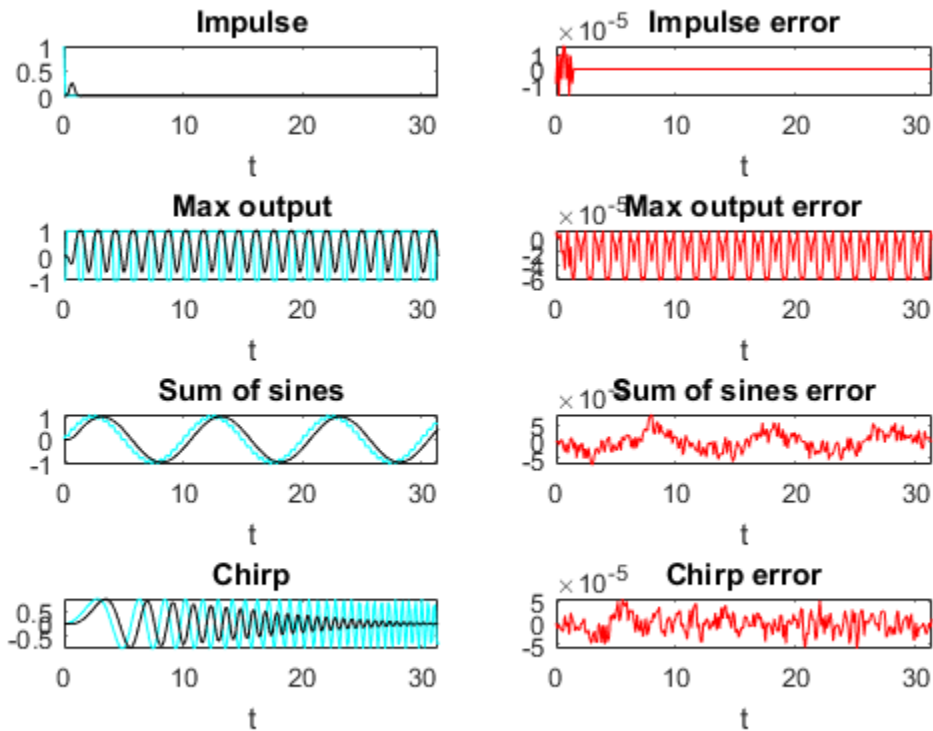
```

        'ProductWordLength',32,...
        'SumMode','KeepLSB',...
        'SumWordLength',32);
    T.acc=fi([],true,32,30,F);
    T.p=int16([]);
    T.b=fi([],true,16,16,F);
    T.x=fi([],true,16,14,F);
    T.y=fi([],true,16,14,F);
    T.z=fi([],true,16,14,F);
end
    
```

Call this types table from the entry-point function and run and plot the output.

```

[y1,y2,y3,y4,y5,y6] = fir_filt_circ_buff_typed_test(b,x);
fir_filt_circ_buff_plot2(7,titles,t,x,y1,y6);
    
```



Code generation command

Run this build function to generate C-code. It is a best practice to create a build function so you can generate C-code for your core algorithm without the entry-point function or test file so the C-code for the core algorithm can be included in a larger project.

```
function fir_filt_circ_buff_build_function()
    %
    % Declare input arguments
    %
    T = fir_filt_circ_buff_dsp_nearest_types();
    b = zeros(1,12,'like',T.b);
    x = zeros(256,1,'like',T.x);
    z = zeros(size(b),'like',T.z);
    p = cast(0,'like',T.p);
    %
    % Code generation configuration
    %
    h = coder.config('lib');
    h.PurelyIntegerCode = true;
    h.SaturateOnIntegerOverflow = false;
    h.SupportNonFinite = false;
    h.HardwareImplementation.ProdBitPerShort = 8;
    h.HardwareImplementation.ProdBitPerInt = 16;
    h.HardwareImplementation.ProdBitPerLong = 32;
    %
    % Generate C-code
    %
    codegen fir_filt_circ_buff_typed_codegen -args {b,x,z,p,T} -config h -launchreport
end
```

Generated C-Code

Using these settings, MATLAB Coder generates the following C-code:

```
void fir_filt_circ_buff_typed_codegen(const int16_T b[12], const int16_T x[256],
    int16_T z[12], int16_T *p, int16_T y[256])
{
    int16_T n;
    int32_T acc;
    int16_T k;
    int16_T j;
    for (n = 0; n < 256; n++) {
        (*p)++;
        if (*p > 12) {
```

```
    *p = 1;
}
z[*p - 1] = x[n];
acc = 0L;
k = *p;
for (j = 0; j < 12; j++) {
    acc += (int32_T)b[j] * z[k - 1];
    k--;
    if (k < 1) {
        k = 12;
    }
}
y[n] = (int16_T)((acc >> 16) + ((acc & 32768L) != 0L));
}
}
```

Run the following code to restore the global states.

```
fipref(FIPREF_STATE);
clearInstrumentationResults fir_filt_circ_buff_original_entry_point_mex
clearInstrumentationResults fir_filt_circ_buff_typed_entry_point_mex
clear fir_filt_circ_buff_original_entry_point_mex
clear fir_filt_circ_buff_typed_entry_point_mex
```

Run the following code to delete the temporary folder.

```
tempdirObj.cleanup;
```

Fixed-Point Design Exploration in Parallel

In this section...

“Overview” on page 12-44

“Setup” on page 12-44

“Using Parallel for-Loops For Design Exploration” on page 12-44

“Description of System Under Test” on page 12-46

Overview

This example shows how to explore and test fixed-point designs by distributing tests across many computers in parallel. The example uses a `parfor` loop to test the accuracy of a QRS detector algorithm.

Setup

To run this example, copy the example files to a local working directory.

```
copyfile(fullfile(matlabroot, 'help', 'toolbox', 'fixpoint', 'examples', 'parallel_examples', 'test_heart_rate_detector_in_parallel.m'), 'test_heart_rate_detector_in_parallel.m');
```

In your local working directory, open and run the test file that runs the example.

```
edit test_heart_rate_detector_in_parallel.m  
test_heart_rate_detector_in_parallel
```

Using Parallel for-Loops For Design Exploration

Like a standard `for`-loop, a `parfor`-loop executes a series of statements over a range of values. Using the `parfor` command, you can set up a parallel `for`-loop in your code to explore fixed-point designs by distributing the tests across many computers. In a `parfor` loop, loop iterations execute in parallel which can provide better performance than standard `for`-loops.

This script sets up the system under test, and initializes the arrays that will contain the results outside of the `parfor`-loop. It then uses a `parfor` loop to test each record in parallel. The `parfor`-loop loads the data, runs the system, then classifies and saves the results in parallel. When the `parfor`-loop finishes, the script displays the results. For

more information on the system being tested in this example, see “Description of System Under Test” on page 12-46.

```

%% Run test of all records in a database in parallel
record_names = {'ecg_01','ecg_02','ecg_03','ecg_04','ecg_05','ecg_06',...
               'ecg_07','ecg_08','ecg_09','ecg_10','ecg_11','ecg_12','ecg_13'};

%% Set up the system under test
data_type = 'fixedwrap';
T = heart_rate_detector_types(data_type);
[mex_function_name,Fs_target] = setup_heart_rate_detector(record_names,data_type,T);

%% Initialize array to contain results
results_file_names = cell(size(record_names));

%% Test each record in the database in parallel
parfor record_number = 1:length(record_names);
    % Load data
    record_name = record_names{record_number};
    [ecg,tm,ann,Fs] = load_ecg_data(record_name,Fs_target);

    % Run system under test
    detector_outputs = run_heart_rate_detector(mex_function_name,ecg,T);

    % Classify results
    [qrs_struct,qrs_stats] = classify_qrs(ann, Fs, detector_outputs);

    % Save results
    results_file_names{record_number} = save_heart_rate_data(...
        mex_function_name,record_name,...
        data_type,ecg,tm,ann,Fs,...
        detector_outputs,...
        qrs_struct,qrs_stats);
end

%% Display results
display_ecg_results(record_names, results_file_names);

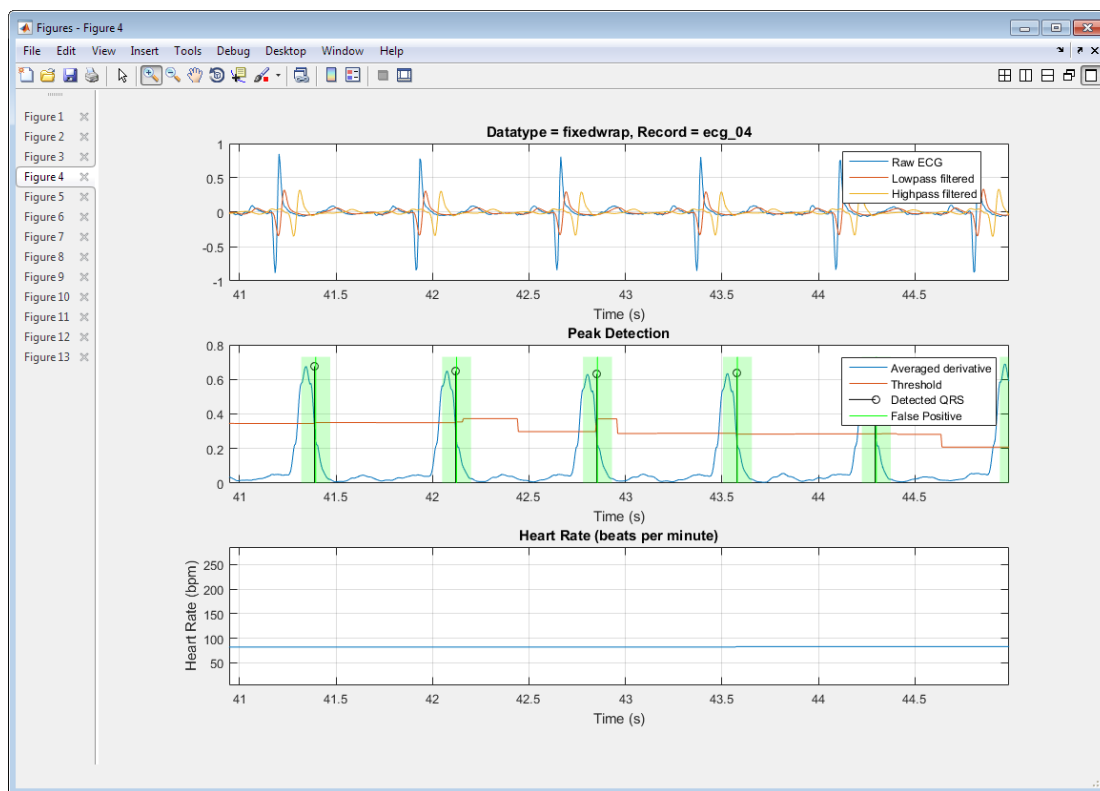
```

Because loop iterations must be completely independent of each other, you cannot call the save and load commands directly inside a `parfor`-loop. You can, however, call functions that call these commands. In this example, the functions `load_ecg_data` and `save_heart_rate_data` load and save the necessary data. For more information on limitations, see “Parallel for-Loops (`parfor`)” (Parallel Computing Toolbox).

Description of System Under Test

The system under test in this example tests a simple QRS detector that measures the time difference between QRS detections to compute heart rate. The `test_heart_rate_detector_in_parallel` script passes ECG recordings to the detection algorithm.

The following plot is an example when the detector algorithm correctly identifies the QRS detections to compute the heartbeat.



The detection algorithm is simplified for this example. Examining the plots and results that are displayed when the example runs shows that the algorithm is not always very accurate.

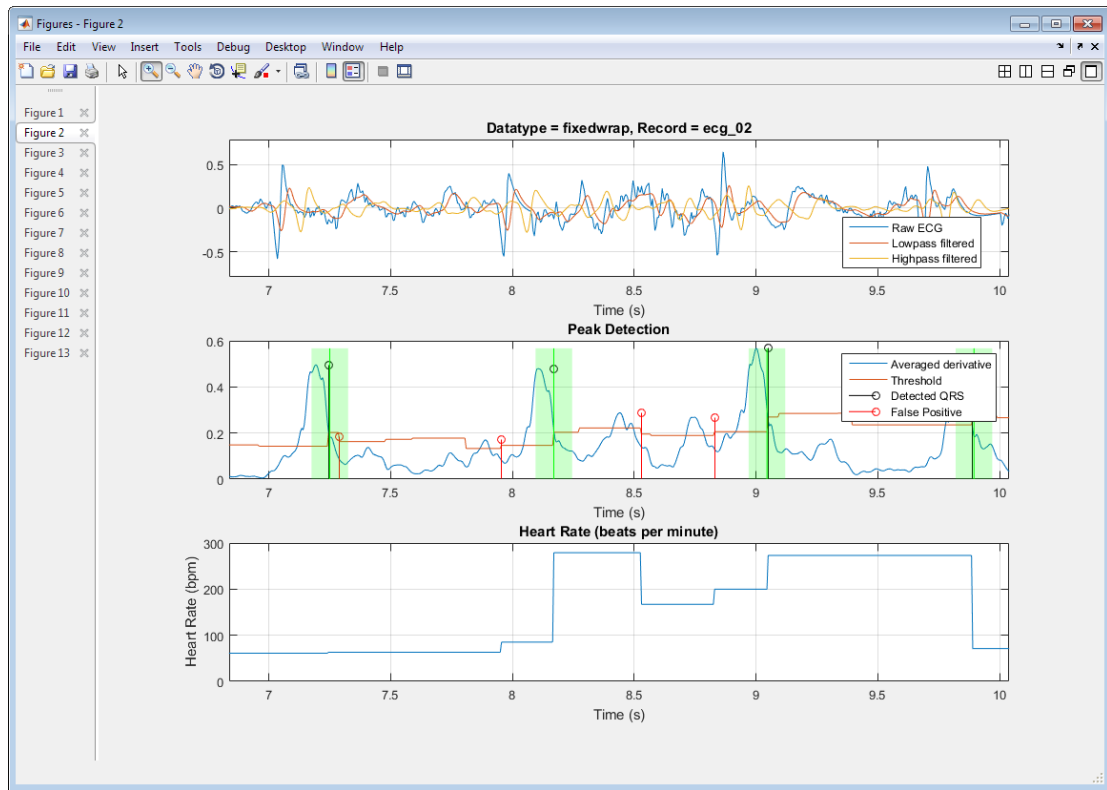
Record	#QRS	TP	FP	FN	DER	Se	+P
ecg_01	253	195	1	58	23.32	77.08	99.49

ecg_02	133	133	18	0	13.53	100.00	88.08
ecg_03	94	94	1	0	1.06	100.00	98.95
ecg_04	92	91	0	1	1.09	98.91	100.00
ecg_05	93	91	1	2	3.23	97.85	98.91
ecg_06	131	131	22	0	16.79	100.00	85.62
ecg_07	174	173	2	0	1.15	100.00	98.86
ecg_08	117	116	10	1	9.40	99.15	92.06
ecg_09	137	137	1	0	0.73	100.00	99.28
ecg_10	96	96	3	0	3.13	100.00	96.97
ecg_11	73	73	1	0	1.37	100.00	98.65
ecg_12	146	145	71	0	48.63	100.00	67.13
ecg_13	144	144	5	0	3.47	100.00	96.64
Totals	1683	1619	136	62	11.76	96.31	92.25

Legend:

#QRS: Total number of QRS Complexes
TP: Number of true positive
FP: Number of false positive
FN: Number of false negative
DER: Detection error rate in percent
Se: Sensitivity in percent
+P: Positive prediction in percent

The following plot is an example when the detector algorithm identifies false positives due to noise in the recording.



Note:

- All ECG recordings used in this example were measured on hobbyist equipment.
You can use the PhysioNet database of recorded physiological signals to do a similar analysis on your own.
- The annotations on these recordings were not verified by doctors.

References

[1] Patrick S. Hamilton, Open Source ECG Analysis Software (OSEA), E.P. Limited, Somerville, MA, <http://www.eplimited.com>, 2002.

- [2] Gari D Clifford, Francisco Azuaje, and Patrick E. McSharry. *Advanced Methods and Tools for ECG Data Analysis*, Artech House, 2006.
- [3] American National Standard ANSI/AAMI EC38:2007 *Medical electrical equipment — Part 2–47: Particular requirements for the safety, including essential performance, of ambulatory electrocardiographic systems*, Association for the Advancement of Medical Instrumentation, 2008.
- [4] George B. Moody, "Evaluating ECG Analyzers", *WaveForm DataBase Applications Guide*, Harvard-MIT Division of Health Sciences and Technology, Cambridge, MA, WFDB10.5.23, 13 March 2014.
- [5] Ida Laila binti Ahmad, Masnani binti Mohamed, Norul Ain binti Ab Ghani, "Development of a Concept Demonstrator for QRS Complex Detection using Combined Algorithms", 2012 IEEE EMBS International Conference on Biomedical Engineering and Sciences, Langkawi, 17th–19th December 2012.
- [6] R. Harikumar, S.N. Shivappriya, "Analysis of QRS Detection Algorithm for Cardiac Abnormalities—A Review", *International Journal of Soft Computing and Engineering (IJSCE)*, ISSN: 2231–2307, Volume–1, Issue–5, November 2011.

Viewing Test Results With Simulation Data Inspector

Inspecting Data Using the Simulation Data Inspector

In this section...

“What Is the Simulation Data Inspector?” on page 13-2

“Import Logged Data” on page 13-2

“Export Logged Data” on page 13-2

“Group Signals” on page 13-3

“Run Options” on page 13-3

“Create Report” on page 13-3

“Comparison Options” on page 13-3

“Enabling Plotting Using the Simulation Data Inspector” on page 13-3

“Save and Load Simulation Data Inspector Sessions” on page 13-4

What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs at top**.

- Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector in the Fixed-Point Converter app, see “Enable Plotting Using the Simulation Data Inspector” on page 9-76.

To enable the Simulation Data Inspector in the programmatic workflow, see “Enable Plotting Using the Simulation Data Inspector” on page 10-29.

Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

Save a Session to a MAT-File

- 1 On the **Visualize** tab, click **Save**.
- 2 Browse to where you want to save the MAT-file to, name the file, and click **Save**.

Load a Saved Simulation Data Inspector Simulation

- 1 On the **Visualize** tab, click **Open**.
- 2 Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
- 3 If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

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

- “Code Acceleration and Code Generation from MATLAB” on page 14-3
- “Requirements for Generating Compiled C Code Files” on page 14-4
- “Functions Supported for Code Acceleration or C Code Generation” on page 14-5
- “Workflow for Fixed-Point Code Acceleration and Generation” on page 14-11
- “Set Up C Compiler” on page 14-12
- “Accelerate Code Using fiaccel” on page 14-13
- “File Infrastructure and Paths Setup” on page 14-19
- “Detect and Debug Code Generation Errors” on page 14-23
- “Set Up C Code Compilation Options” on page 14-25
- “MEX Configuration Dialog Box Options” on page 14-27
- “Best Practices for Accelerating Fixed-Point Code” on page 14-32
- “Use Fixed-Point Code Generation Reports” on page 14-36
- “Generate C Code from Code Containing Global Data” on page 14-41
- “Define Input Properties Programmatically in MATLAB File” on page 14-46
- “Specify Cell Array Inputs at the Command Line” on page 14-55
- “Specify Global Cell Arrays at the Command Line” on page 14-62
- “Control Run-Time Checks” on page 14-64
- “Fix Run-Time Stack Overflows” on page 14-66
- “Code Generation with MATLAB Coder” on page 14-67
- “Code Generation with MATLAB Function Block” on page 14-68
- “Generate Fixed-Point FIR Code Using MATLAB Function Block” on page 14-77

- “Fixed-Point FIR Code Example Parameter Values” on page 14-81
- “Accelerate Code for Variable-Size Data” on page 14-83
- “Accelerate Fixed-Point Simulation” on page 14-93
- “Code Generation Readiness Tool” on page 14-96
- “Check Code Using the Code Generation Readiness Tool” on page 14-102
- “Check Code Using the MATLAB Code Analyzer” on page 14-103
- “Fix Errors Detected at Code Generation Time” on page 14-104
- “Avoid Multiword Operations in Generated Code” on page 14-105
- “Find Potential Data Type Issues in Generated Code” on page 14-108

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 Designer 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 Designer 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” (MATLAB Coder) in the MATLAB Coder documentation
MATLAB Function	Use MATLAB code in your Simulink models that generate embeddable C/C++ code	Simulink license	“What Is a MATLAB Function Block?” (Simulink) 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 “Code Generation for Variable-Size Arrays” on page 27-2.

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 C Code Generation

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

- `fipref` and `quantizer` objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given `fi` variable after that variable has been created.
- The `boolean` value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- You can use parallel `for` (`parfor`) loops in code compiled with `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.

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

<code>abs</code>
<code>accumneg</code>
<code>accumpos</code>
<code>add*</code>
<code>all</code>
<code>any</code>
<code>atan2</code>
<code>bitand*</code>
<code>bitandreduce</code>
<code>bitcmp</code>
<code>bitconcat</code>
<code>bitget</code>

bitor*
bitorreduce
bitreplicate
bitrol
bitror
bitset
bitshift
bitsliceget
bitsll*
bitsra*
bitsrl*
bitxor*
bitxorreduce
ceil
complex
conj
conv*
convergent
cordicabs*
cordicangle*
cordicatan2*
cordiccart2pol*
cordicexp*
cordiccos*
cordicpol2cart*
cordicrotate*
cordicsin*
cordicsincos*

cordicsqrt*
cos
ctranspose
diag*
divide*
double*
end
eps*
eq*
fi*
filter*
fimath*
fix
fixed.Quantizer
flip*
fliplr
flipud
floor
for
ge*
get*
getlsb
getmsb
gt*
horzcat
imag
int8, int16, int32, int64
ipermute

iscolumn
isempty
isequal
isfi*
isfimath
isfimathlocal
isfinite
isinf
isnan
isnumeric
isnumericitype
isreal
isrow
isscalar
issigned
isvector
le*
length
logical
lowerbound
lsb*
lt*
max
mean
median
min
minus*
mpower*

mpy*
mrdivide
mtimes*
ndims
ne*
nearest
numberofelements*
numel
numericity*
permute*
plus*
pow2
power*
qr
quantize
range
rdivide
real
realmax
realmin
reinterprecast
removefimath
repmat*
rescale
reshape
rot90*
round
setfimath

sfi*
shiftdim*
sign
sin
single*
size
sort*
sqrt*
squeeze
storedInteger
storedIntegerToDouble
sub*
subsasgn
subsref
sum*
times*
transpose
tril*
triu*
ufi*
uint8, uint16, uint32, uint64
uminus
uplus
upperbound
vertcat

Workflow for Fixed-Point Code Acceleration and Generation

Step	Action	Details
1	Set up your C compiler.	See “Set Up C Compiler” on page 14-12.
2	Set up your file infrastructure.	See “File Infrastructure and Paths Setup” on page 14-19.
3	Make your MATLAB algorithm suitable for code generation	See “Best Practices for Accelerating Fixed-Point Code” on page 14-32.
4	Set compilation options.	See “Set Up C Code Compilation Options” on page 14-25.
5	Specify properties of primary function inputs.	See “Specify Properties of Entry-Point Function Inputs” on page 29-2.
6	Run <code>fiaccel</code> with the appropriate command-line options.	See “Recommended Compilation Options for <code>fiaccel</code> ” on page 14-32.

Set Up C Compiler

Fixed-Point Designer automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

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

Accelerate Code Using `fiaccel`

In this section...

“Speeding Up Fixed-Point Execution with `fiaccel`” on page 14-13

“Running `fiaccel`” on page 14-13

“Generated Files and Locations” on page 14-14

“Data Type Override Using `fiaccel`” on page 14-16

“Specifying Default `fimath` Values for MEX Functions” on page 14-17

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 14-19.
- 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 14-35.
- 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 14-14.
- 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.

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
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 `fiaccl` 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

% fiaccl
fiaccl 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 `fiaccl` 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 `fiaccl` 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 Designer 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 **MATLAB Examples** under Fixed-Point Designer, 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(fioref)` command in that section:

```
fioref('DataTypeOverride','TrueDoubles')
```

This command tells Fixed-Point Designer software to create all `fi` objects with type `fi double`. When you compile the code using the `fiaccl` command in the “Compile the M-File into a MEX File” section of the example, the resulting MEX-function uses floating-point data.

Specifying Default `fimath` Values for MEX Functions

MEX functions generated with `fiaccl` use the MATLAB default global `fimath`. The MATLAB factory default global `fimath` has the following properties:

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

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, MATLAB generates a warning, alerting you to a mismatch between the compile-time and run-time `fimath` values. For example, create the following MATLAB function:

```
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.

Generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`.

```
resetglobalfimath;
fiaccl test
```

`fiaccl` generates a MEX function, `test_mex`, in the current folder.

Run `test_mex`.

```
test_mex
```

```
ans =  
    0  
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 15
```

Modify the MATLAB default `fimath` value so it no longer matches the setting used at compile time.

```
F = fimath('RoundingMethod', 'Floor');  
globalfimath(F);
```

Clear the MEX function from memory and rerun it.

```
clear test_mex  
test_mex
```

The mismatch is detected and MATLAB generates a warning.

```
testglobalfimath_mex  
Warning: This function was generated with a different default fimath than the current c  
ans =  
    0
```

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

To avoid this issue, separate the `fimath` properties from your algorithm by using types tables. For more information, see “Separate Data Type Definitions from Algorithm” on page 12-8.

File Infrastructure and Paths Setup

In this section...

“Compile Path Search Order” on page 14-19

“Naming Conventions” on page 14-19

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.

Naming Conventions

MATLAB enforces naming conventions for functions and generated files.

- “Reserved Prefixes” on page 14-19
- “Reserved Keywords” on page 14-19
- “Conventions for Naming Generated files” on page 14-22

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 14-20
- “C++ Reserved Keywords” on page 14-20
- “Reserved Keywords for Code Generation” on page 14-21

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to

MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains reserved keywords that the code generator cannot rename, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the “C++ Reserved Keywords” on page 14-20.

C Reserved Keywords

assert	extern	setjmp	string
auto	fenv	short	struct
break	float	signal	switch
case	for	signed	tgmath
char	goto	sizeof	threads
const	if	static	time
complex	int	stdalign	typedef
continue	inttypes	stdarg	uchar
ctype	iso646	stdatomic	union
default	limits	stdbool	unsigned
do	locale	stddef	void
double	long	stdint	volatile
else	math	stdio	wchar
enum	register	stdlib	wctype
errno	return	stdnoreturn	while

C++ Reserved Keywords

algorithm	cstddef	iostream	sstream
any	cstdint	istream	stack
array	cstdio	iterator	static_cast
atomic	cstdlib	limits	stdexcept

bitset	cstring	list	streambuf
cassert	ctgmath	locale	string_view
catch	ctime	map	stringstream
ccomplex	cuchar	memory	system_error
cctype	cwchar	memory_resource	template
cerrno	cwctype	mutable	this
cfenv	delete	mutex	thread
cfloat	deque	namespace	throw
chrono	dynamic_cast	new	try
cinttypes	exception	numeric	tuple
ciso646	execution	operator	typeid
class	explicit	optional	type_traits
climits	export	ostream	typeid
clocale	filesystem	private	typeinfo
cmath	forward_list	protected	typename
codecvt	friend	public	unordered_map
complex	fstream	queue	unordered_set
condition_variable	functional	random	using
const_cast	future	ratio	utility
csetjmp	initializer_list	regex	valarray
csignal	inline	reinterpret_cast	vector
cstdalign	iomani	scoped_allocator	virtual
cstdarg	ios	set	wchar_t
cstdbool	iosfwd	shared_mutex	

Reserved Keywords for Code Generation

abs	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen

boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true
cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_SIZE	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMPLES	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T
cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

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 14-23

“Error Detection at Design Time” on page 14-24

“Error Detection at Compile Time” on page 14-24

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	<ol style="list-style-type: none"> 1 Verify that your lowest-level (leaf) functions are suitable for code generation. 2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function. 	<ul style="list-style-type: none"> • Efficient • Safe • Easy to isolate syntax violations 	Requires application tests that work from the bottom up
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 one 	Lets you retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: <ul style="list-style-type: none"> • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values.

Debugging Strategy	What to Do	Pros	Cons
	b. Compile and verify each function, ending with the leaf functions.		

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.

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 14-23.

`fiaccl` checks for all potential syntax violations at compile time. When `fiaccl` detects errors or warnings, it automatically produces a code generation report that describes the issues and provides links to the offending code. See “Use Fixed-Point Code Generation Reports” on page 14-36.

If your MATLAB code calls functions on the MATLAB path, `fiaccl` 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 14-25

“Compilation Options Modification at the Command Line Using Dot Notation” on page 14-25

“How fiaccel Resolves Conflicting Options” on page 14-26

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 14-25
- 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 `fiaccl` 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.
Dynamic memory allocation	DynamicMemoryAllocation 'off' , 'AllVariableSizeArrays'	Enable dynamic memory allocation for variable-size data. By default, dynamic memory allocation is disabled and <code>fiaccl</code> allocates memory

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
		<p>statically on the stack. When you select dynamic memory allocation, <code>fiaccel</code> allocates memory for all variable-size data dynamically on the heap.</p> <p>You <i>must</i> use dynamic memory allocation for all unbounded variable-size data.</p>
Enable variable sizing	EnableVariableSizing true , false	Enable support for variable-size arrays.
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>
Saturate on integer overflow	SaturateOnIntegerOverflow true , false	Add checks in the generated code to detect integer overflow or underflow.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Safety (disable for faster MEX)		
Ensure memory integrity	IntegrityChecks true , false	Detects violations of memory integrity while building MATLAB Function blocks and stops simulation 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> .
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 14-64
- “Code Generation for Variable-Size Arrays” on page 27-2
- “Generate C Code from Code Containing Global Data” on page 14-41

Best Practices for Accelerating Fixed-Point Code

In this section...

“Recommended Compilation Options for `fiaccel`” on page 14-32

“Build Scripts” on page 14-33

“Check Code Interactively Using MATLAB Code Analyzer” on page 14-34

“Separating Your Test Bench from Your Function Code” on page 14-34

“Preserving Your Code” on page 14-34

“File Naming Conventions” on page 14-35

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” (MATLAB Coder).

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 14-46.

- `-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 `fiaccl`.

Build Scripts

Use build scripts to call `fiaccl` 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 `fiaccl` to process `lms_02.m`:

```
close all;
clear all;
clc;

N = 73113;

fiaccl -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`
- `fiaccl -report lms_02.m -args { zeros(N,1) zeros(N,1) }` calls `fiaccl` 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 `fiaccl` 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 14-35. 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.

Use Fixed-Point Code Generation Reports

In this section...

“Code Generation Report Creation” on page 14-36

“Code Generation Report Opening” on page 14-37

“Viewing Your MATLAB Code” on page 14-37

“Viewing Variables in the Variables Tab” on page 14-39

“See Also” on page 14-40

Code Generation Report Creation

When you compile your code with the `fiaccl` 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.

This example shows how to create a code generation report.

- 1 In a local writable folder, write a MATLAB function, `moving_average.m`.

```
function [avg,z] = moving_average(x,z)
%#codegen
    if nargin < 2,
        z = fi(zeros(10,1),1,16,15);
    end
    z(2:end) = z(1:end-1); % Update buffer
    z(1) = x; % Add new value
    avg = mean(z); % Compute moving average
end
```

- 2 In the same folder, write a MATLAB function, `test_moving_average.m`.

```
function avg = test_moving_average(x)
%#codegen
    if nargin < 1,
        x = fi(rand(100,1),1,16,15);
    end
    z = fi(zeros(10,1),1,16,15);
    avg = x;
    for k = 1:length(x)
        [avg(k),z] = moving_average(x(k),z);
    end
end
```

```

    end
end

```

- 3 Use `fiaccel` to create a MEX function and accelerate the code. Specify the type of input `x` using the `-args` option. Specify the `-report` option to create a code generation report.

```

x = fi(rand(100,1),1,16,15);
fiaccel -report test_moving_average -args {x}

```

Code Generation Report Opening

If code generation completes, you receive the following message:

Code generation successful: [View report](#)

Click the **View report** link to open the report.

If code generation fails, you get a link to the error report:

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” (MATLAB Coder).

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.

Color	Meaning
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.

The screenshot shows the 'Code Generation Report' window for the function 'test_moving_average'. The MATLAB code is displayed with line numbers 1 through 11. Line 6, containing the assignment 'z = fi(zeros(10,1), 1, 16, 15);', is highlighted in green. A tooltip over this line provides the following information for the selected variable 'z':

Information for the selected variable:	
Size	10 x 1
Class	embedded.fi
Complex	No
Signedness	Signed
WL	16
FL	15

Below the code editor, a 'Summary' table lists the variables used in the function:

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
1	avg	Output	100 x 1	embedded.fi	No	Signed	16	15
2	x	Input	100 x 1	embedded.fi	No	Signed	16	15
3	z	Local	10 x 1	embedded.fi	No	Signed	16	15
4	k	Local	1 x 1	double	No	-	-	-

Expressions in your code that have data type information available appear highlighted in purple, as the next figure shows.

The screenshot shows the 'Code Generation Report' window. The 'MATLAB code' tab is active, displaying the code for the function `test_moving_average`. The code includes a `fi` object `z` and a call to `moving_average`. A mouse cursor is hovering over the `moving_average` call, which has triggered a popup window titled 'Information for the selected expression:'. This popup displays the following properties for the selected expression:

Size	1 x 1
Class	embedded.fi
Complex	No
Signedness	Signed
WL	16
FL	15

Below the code editor, the 'Summary' tab is selected, showing a table of variables:

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
1	avg	Output	100 x 1	embedded.fi	No	Signed	16	15
2	x	Input	100 x 1	embedded.fi	No	Signed	16	15
3	z	Local	10 x 1	embedded.fi	No	Signed	16	15
4	k	Local	1 x 1	double	No	-	-	-

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	Class	Complex	Signedness	WL	FL	
1	avg	Output	100 x 1	embedded.fi	No	Signed	16	15	
Global fimath:									
				Round mode: nearest	Sum mode: FullPrecision				
				Overflow mode: saturate	Maximum sum word length: 65535				
				Product mode: FullPrecision	Cast before sum: Yes				
				Maximum product word length: 65535					
2	x	Input	100 x 1	embedded.fi	No	Signed	16	15	
3	z	Local	10 x 1	embedded.fi	No	Signed	16	15	
4	k	Local	1 x 1	double	No	-	-	-	

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” (MATLAB Coder).

Generate C Code from Code Containing Global Data

In this section...

“Workflow Overview” on page 14-41

“Declaring Global Variables” on page 14-41

“Defining Global Data” on page 14-42

“Synchronizing Global Data with MATLAB” on page 14-43

“Limitations of Using Global Data” on page 14-45

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 14-42.

- 3 Compile your code using `fiaccl`.

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 14-43.

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 14-41 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 14-41, 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 14-44.

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	Yes. Use the <code>coder.extrinsic - sync:off</code> option to turn

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
	extrinsic calls (default)	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 14-43.

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.varsized` function. Instead, use a `coder.typeof` object to define variable-sized global data as described in “Defining Variable-Sized Global Data” on page 14-42.

Define Input Properties Programmatically in MATLAB File

In this section...

“How to Use `assert`” on page 14-46

“Rules for Using `assert` Function” on page 14-50

“Specifying Properties of Primary Fixed-Point Inputs” on page 14-51

“Specifying Properties of Cell Arrays” on page 14-51

“Specifying Class and Size of Scalar Structure” on page 14-53

“Specifying Class and Size of Structure Array” on page 14-54

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 14-49. You can also set its `fiMath` properties, see “Specify `fiMath` of Fixed-Point Input” on page 14-50.

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 14-49. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 14-50.

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 Cell Array Class

```
assert(iscell( param ))
assert(isa( param, 'cell' ))
```

Sets the input parameter *param* to the MATLAB class `cell` (cell array). For example, to set the class of input `C` to a `cell`, call:

```
...
assert(iscell(C));
...
```

or

```
...
assert(isa(C, 'cell'));
...
```

To specify the properties of cell array elements, see “Specifying Properties of Cell Arrays” on page 14-51.

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 `numericity`, see “Specify numericity of Fixed-Point Input” on page 14-49.

- You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 14-50.
- 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.

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>	<code>fimath</code> 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);
assert(isfi(x));
assert(isequal(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Specifying Properties of Cell Arrays

To specify the MATLAB class `cell` (cell array), use one of the following syntaxes:

```
assert(iscell(param))
assert(isa(param,'cell'))
```

For example, to set the class of input `C` to `cell`, use:

```
...  
assert(iscell(C));  
...
```

or

```
...  
assert(isa(C, 'cell'));  
...
```

You can also specify the size of the cell array and the properties of the cell array elements. The number of elements that you specify determines whether the cell array is homogeneous or heterogeneous. See “Code Generation for Cell Arrays” (MATLAB Coder).

If you specify the properties of the first element only, the cell array is homogeneous. For example, the following code specifies that `C` is a 1x3 homogeneous cell array whose elements are 1x1 double.

```
...  
assert(isa(C, 'cell'));  
assert(all(size(C) == [1 3]));  
assert(isa(C{1}, 'double'));  
...
```

If you specify the properties of the first element only, but also assign a structure type name to the cell array, the cell array is heterogeneous. Each element has the properties of the first element. For example, the following code specifies that `C` is a 1x3 heterogeneous cell array. Each element is a 1x1 double.

```
...  
assert(isa(C, 'cell'));  
assert(all(size(C) == [1 3]));  
assert(isa(C{1}, 'double'));  
coder.cstructname(C, 'myname');  
...
```

If you specify the properties of each element, the cell array is heterogeneous. For example, the following code specifies a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x3 double.

```
...  
assert(isa(C, 'cell'));
```

```

assert(all(size(C) == [1 2]));
assert(isa(C{1}, 'char'));
assert(all(size(C{2}) == [1 3]));
assert(isa(C{2}, 'double'));
...

```

If you specify more than one element, you cannot specify that the cell array is variable size, even if all elements have the same properties. For example, the following code specifies a variable-size cell array. Because the code specifies the properties of the first and second elements, code generation fails.

```

...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
assert(isa(C{2}, 'double'));
...

```

In the previous example, if you specify the first element only, you can specify that the cell array is variable-size. For example:

```

...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
...

```

Specifying Class and Size of Scalar Structure

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 = numerictype('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.

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 = numericity('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(numericity(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.

Specify Cell Array Inputs at the Command Line

To specify cell array inputs at the command line, use the same methods that you use for other types of inputs. You can:

- Provide an example cell array input to the `-args` option of the `fiaccel` command.
- Provide a `coder.CellType` object to the `-args` option of the `fiaccel` command. To create a `coder.CellType` object, use `coder.typeof`.
- Use `coder.Constant` to specify a constant cell array input.

For code generation, cell arrays are classified as homogeneous or heterogeneous. See “Code Generation for Cell Arrays” on page 28-2. When you provide an example cell array to `fiaccel` or `coder.typeof`, the function determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `coder.CellType` `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

Specify Cell Array Inputs by Example

To specify a cell array input by example, provide an example cell array in the `-args` option of the `fiaccel` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

```
fiaccel myfunction -args {{1 2 3}} -report
```

The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.

- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

```
fiaccel myfunction -args {'a', 1} -report
```

The input argument is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

Specify the Type of the Cell Array Input

To specify the type of a cell array input, use `coder.typeof` to create a `coder.CellType` object. Pass the `coder.CellType` object to the `-args` option of the `fiaccel` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

```
t = coder.typeof({1 2 3});  
fiaccel myfunction -args {t} -report
```

The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.

- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

```
t = coder.typeof({'a', 1});  
fiaccel myfunction -args {t}
```

The input argument is a 1x2 heterogeneous cell array whose first element is a 1x1 char and whose second element is a 1x1 double.

You can also use the advanced function `coder.newtype` to create a `coder.CellType` object.

Make a Homogeneous Copy of a Type

If `coder.typeof` returns a heterogeneous cell array type, but you want a homogeneous type, use the `makeHomogeneous` method to make a homogeneous copy of the type.

The following code creates a heterogeneous type.

```
t = coder.typeof({1 [2 3]})
t =
coder.CellType
  1x2 heterogeneous cell
  f0: 1x1 double
  f1: 1x2 double
```

To make a homogeneous copy of the type, use:

```
t = makeHomogeneous(t)
t =
coder.CellType
  1x2 locked homogeneous cell
  base: 1x:2 double
```

Alternatively, use this notation:

```
t = makeHomogeneous(coder.typeof({1 [2 3]}))
t =
coder.CellType
  1x2 locked homogeneous cell
  base: 1x:2 double
```

The classification as homogeneous is locked (permanent). You cannot later use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

If the elements of a type have different classes, such as `char` and `double`, you cannot use `makeHomogeneous` to make a homogeneous copy of the type.

If you use `coder.cstructname` to specify a name for the structure type that represents a type in the generated code, you cannot create a homogeneous copy of the type.

Make a Heterogeneous Copy of a Type

If `coder.typeof` returns a homogeneous cell array type, but you want a heterogeneous type, use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

The following code creates a homogeneous type.

```
t = coder.typeof({1 2 3})  
  
t =  
  
coder.CellType  
  1x3 homogeneous cell  
    base: 1x1 double
```

To make the type heterogeneous, use:

```
t = makeHeterogeneous(t)  
  
t =  
  
coder.CellType  
  1x3 locked heterogeneous cell  
    f1: 1x1 double  
    f2: 1x1 double  
    f3: 1x1 double
```

Alternatively, use this notation:

```
t = makeHeterogeneous(coder.typeof({1 2 3}))  
  
t =  
  
coder.CellType  
  1x3 locked heterogeneous cell  
    f1: 1x1 double  
    f2: 1x1 double  
    f3: 1x1 double
```

The classification as heterogeneous is locked (permanent). You cannot later use the `makeHomogeneous` method to make a homogeneous copy of the type.

If a type is variable size, you cannot use `makeHeterogeneous` to make a heterogeneous copy of it.

Specify Variable-Size Cell Array Inputs

You can specify variable-size cell array inputs in the following ways:

- In the `coder.typeof` call.

For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

```
t = coder.typeof({1}, [1 5], [0 1])
```

```
t =
```

```
coder.CellType
  1x:5 homogeneous cell
    base: 1x1 double
```

For elements with the same classes, but different sizes, you can use the `coder.typeof` `size` and `variable dimensions` arguments to create a variable-size homogeneous cell array type. For example, the following code does not use the `size` and `variable dimensions` arguments. This code creates a type for a heterogeneous cell array.

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

```
t =
```

```
coder.CellType
  1x2 heterogeneous cell
    f0: 1x1 double
    f1: 1x2 double
```

The following code, that uses the `size` and `dimensions` arguments, creates a type for a variable-size homogeneous type cell array:

```
t = coder.typeof({1 [2 3]}, [1 5], [0 1])
```

```
t =
```

```
coder.CellType
  1x:5 locked homogeneous cell
    base: 1x:2 double
```

- Use `coder.resize`.

For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

```
t = coder.typeof({1});
```

```
t = coder.resize(t, [1 5], [0,1])  
  
t =  
  
coder.CellType  
  1x5 homogeneous cell  
    base: 1x1 double
```

You cannot use `coder.resize` with a heterogeneous cell array type.

Specify Type Name for Heterogeneous Cell Array Inputs

A heterogeneous cell array is represented in the generated code as a structure. To specify the name of the structure type in the generated code, use `coder.cstructname`.

For example, to specify the name `myname` for the cell array type in the generated code:

```
t = coder.typeof({'a', 1})  
t = coder.cstructname(t, 'myname')  
  
t =  
  
coder.CellType  
  1x2 locked heterogeneous cell myname  
    f1: 1x1 char  
    f2: 1x1 double
```

If you use `coder.cstructname` with a homogeneous cell array type, `coder.cstructname` returns a heterogeneous copy of the type. However, it is a best practice to use the `makeHeterogeneous` method of the `coder.CellType` object to make a heterogeneous copy of a homogeneous cell array type. Then, you can use `coder.cstructname` with the heterogeneous copy of the type.

Specify Constant Cell Array Inputs

To specify that a cell array input is constant, use the `coder.Constant` function with the `-args` option of the `fiaccel` command. For example:

```
fiaccel myfunction -args {coder.Constant({'red', 1 'green', 2, 'blue', 3})} -report
```

The input is a 1x6 heterogeneous cell array. The sizes and classes of the elements are:

- 1x3 char

- 1x1 double
- 1x5 char
- 1x1 double
- 1x4 char
- 1x1 double

See Also

`coder.CellType` | `coder.newtype` | `coder.resize` | `coder.typeof`

Related Examples

- “Define Input Properties by Example at the Command Line” on page 29-5
- “Specify Constant Inputs at the Command Line” on page 29-7

More About

- “Code Generation for Cell Arrays” on page 28-2

Specify Global Cell Arrays at the Command Line

To specify global cell array inputs, use the `-globals` option of the `fiaccl` command with this syntax:

```
fiaccl myfunction -globals {global_var, {type_object, initial_value}}
```

For example:

- To specify that the global variable `g` is a 1x3 cell array whose elements have class `double` and whose initial value is `{1 2 3}`, use:

```
fiaccl myfunction -globals {'g', {coder.typeof({1 1 1}), {1 2 3}}}
```

Alternatively, use:

```
t = coder.typeof({1 1 1});  
fiaccl myfunction -globals {'g', {t, {1 2 3}}}
```

The global variable `g` is a 1x3 homogeneous cell array whose elements are 1x1 double.

To make `g` heterogeneous, use:

```
t = makeHeterogeneous(coder.typeof({1 1 1}));  
fiaccl myfunction -globals {'g', {t, {1 2 3}}}
```

- To specify that `g` is a cell array whose first element has type `char`, whose second element has type `double`, and whose initial value is `{'a', 1}`, use:

```
fiaccl myfunction -globals {'g', {coder.typeof({'a', 1}), {'a', 1}}}
```

The global variable `g` is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

- To specify that `g` is a cell array whose first element has type `double`, whose second element is a 1x2 double array, and whose initial value is `{1 [2 3]}`, use:

```
fiaccl myfunction -globals {'g', {coder.typeof({1 [2 3]}), {1 [2 3]}}}
```

Alternatively, use:

```
t = coder.typeof({1 [2 3]});  
fiaccl myfunction -globals {'g', {t, {1 [2 3]}}}
```


The global variable `g` is a 1x2 heterogeneous cell array whose first element is 1x1 double and whose second element is 1x2 double.

Global variables that are cell arrays cannot have variable size.

See Also

`coder.typeof` | `fiaccl`

Related Examples

- “Generate C Code from Code Containing Global Data” on page 14-41

Control Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 14-64
“When to Disable Run-Time Checks” on page 14-64
“How to Disable Run-Time Checks” on page 14-65

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 16-10.

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;
```

Fix Run-Time Stack Overflows

If your C compiler reports a run-time stack overflow, set the value of the maximum stack usage parameter to be less than the available stack size. Create a command-line configuration object, `coder.MexConfig` and then set the `StackUsageMax` parameter.

Code 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 Designer 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 a MATLAB Language Function in a Simulink Model” on page 14-68

“MATLAB Function Block with Data Type Override” on page 14-68

“Fixed-Point Data Types with MATLAB Function Block” on page 14-70

Composing a MATLAB Language Function in a 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
- “What Is a MATLAB Function Block?” (Simulink) in the Simulink documentation
- “Code Generation Workflow” (MATLAB Coder) 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 settings in the Simulink model

For more information about data type override, see `fxptdlg`.

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Applies To Setting	Override Data Type
Inherited single	Double	All numeric types or Floating-point	Built-in double
	Single	All numeric types or Floating-point	Built-in single
	Scaled double	All numeric types or Floating-point	fi scaled double
Specified single	Double	All numeric types or Floating-point	Built-in double
	Single	All numeric types or Floating-point	Built-in single
	Scaled double	All numeric types or Floating-point	fi scaled double
Inherited double	Double	All numeric types or Floating-point	Built-in double
	Single	All numeric types or Floating-point	Built-in single
	Scaled double	All numeric types or Floating-point	fi scaled double
Specified double	Double	All numeric types or Floating-point	Built-in double
	Single	All numeric types or Floating-point	Built-in single
	Scaled double	All numeric types or Floating-point	fi scaled double
Inherited Fixed	Double	All numeric types or Fixed-point	fi double
	Single	All numeric types or Fixed-point	fi single
	Scaled double	All numeric types or Fixed-point	fi scaled double

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Applies To Setting	Override Data Type
Specified Fixed	Double	All numeric types or Fixed-point	fi double
	Single	All numeric types or Fixed-point	fi single
	Scaled double	All numeric types or Fixed-point	fi scaled double

For more information about using the MATLAB Function block with data type override, see “Using Data Type Override with the MATLAB Function Block” (Simulink).

Fixed-Point Data Types with MATLAB Function Block

Code generation from MATLAB supports a significant number of Fixed-Point Designer functions. Refer to “Functions Supported for Code Acceleration or C Code Generation” on page 14-5 for information about which Fixed-Point Designer functions are supported.

For more information on working with fixed-point MATLAB Function blocks, see:

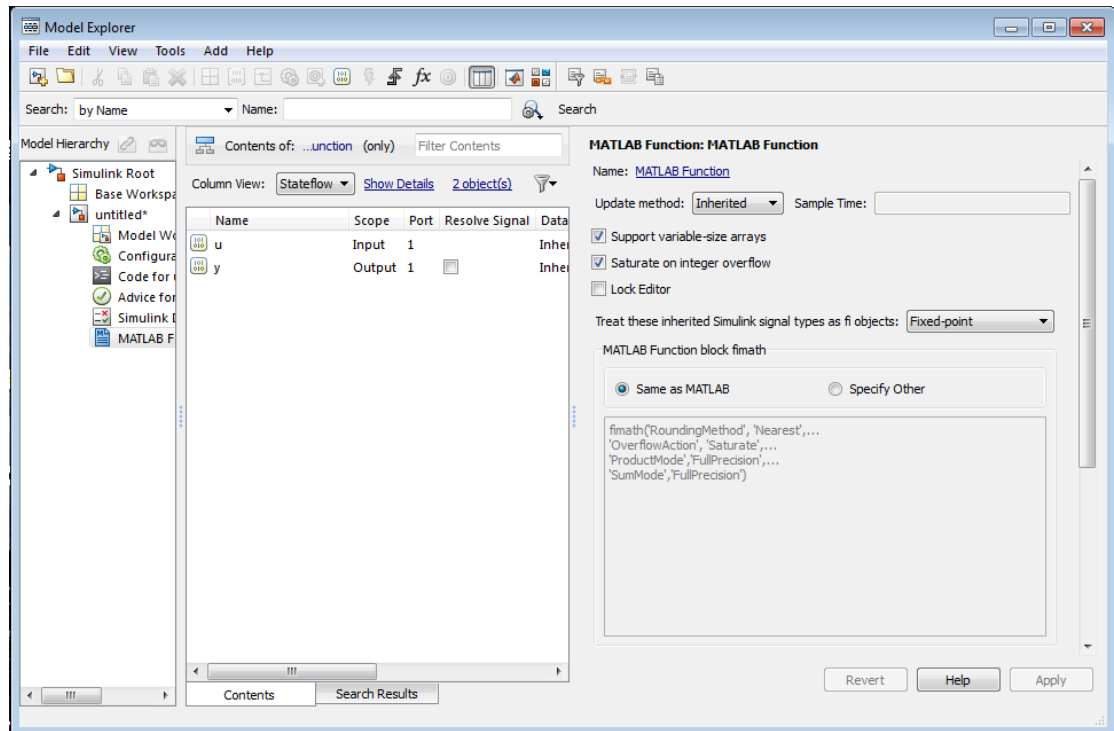
- “Specifying Fixed-Point Parameters in the Model Explorer” on page 14-70
- “Using fimath Objects in MATLAB Function Blocks” on page 14-72
- “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 14-74

Note: To simulate models using fixed-point data types in Simulink, you must have a Fixed-Point Designer license.

Specifying Fixed-Point Parameters in the Model Explorer

You can specify parameters for a 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 Designer **fi** objects.
- When you select **Fixed-point & Integer**, the MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Designer **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.

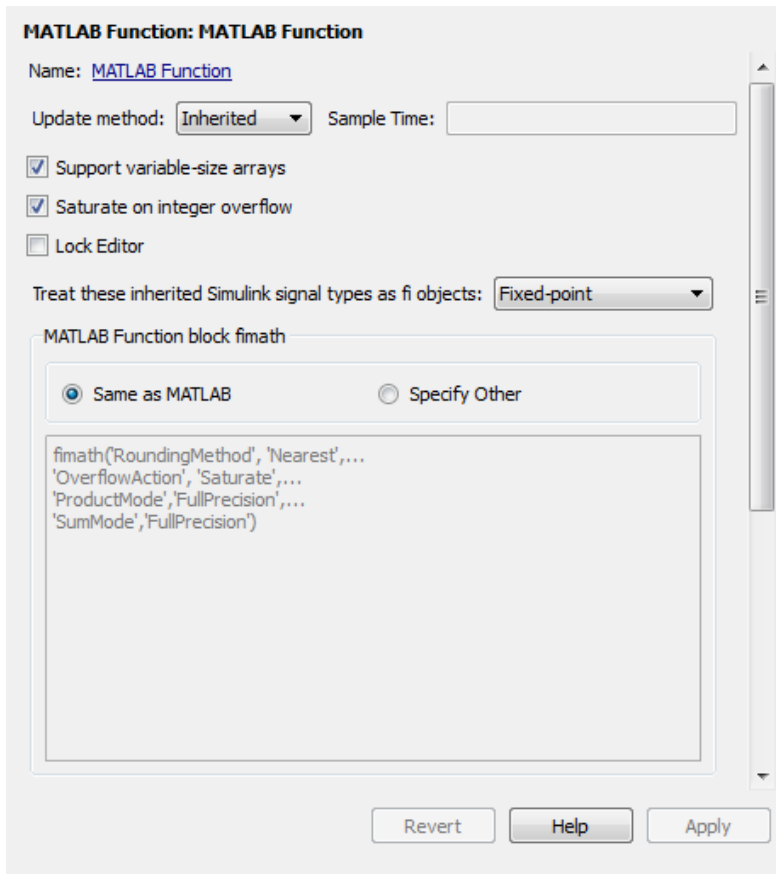
For more information on these parameters, see “Using `fimath` Objects in MATLAB Function Blocks” on page 14-72.

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 (Simulink) or the “Ports and Data Manager” (Simulink).



- 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 14-74 for more information on sharing models.

The Fixed-Point Designer `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 =
```

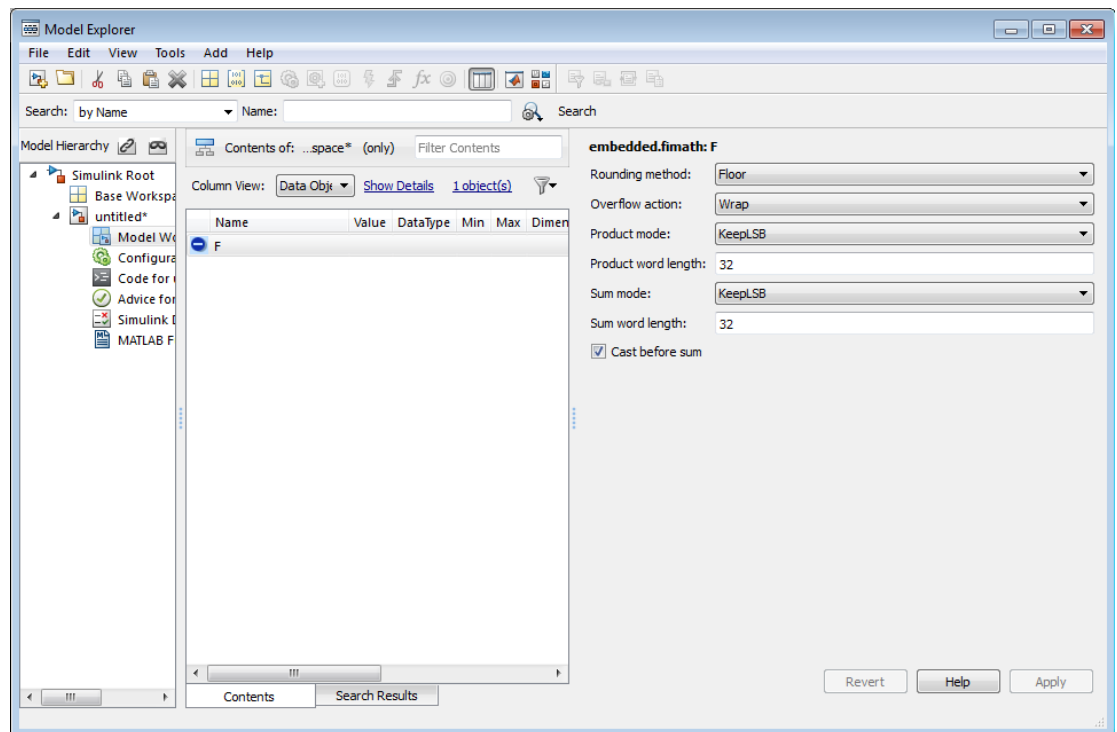
```
    RoundingMethod: Floor
    OverflowAction: Wrap
           ProductMode: KeepLSB
ProductWordLength: 32
           SumMode: KeepLSB
    SumWordLength: 32
    CastBeforeSum: true
```

- 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 14-77

“Prepare the Inputs” on page 14-78

“Create the Model” on page 14-78

“Define the fimath Object Using the Model Explorer” on page 14-80

“Run the Simulation” on page 14-80

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 Designer 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`.
- 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
%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.  
figure;  
subplot(211);plot(x); title('Noisy Signal');grid;  
subplot(212);plot(yout); title('Filtered Signal');grid;
```

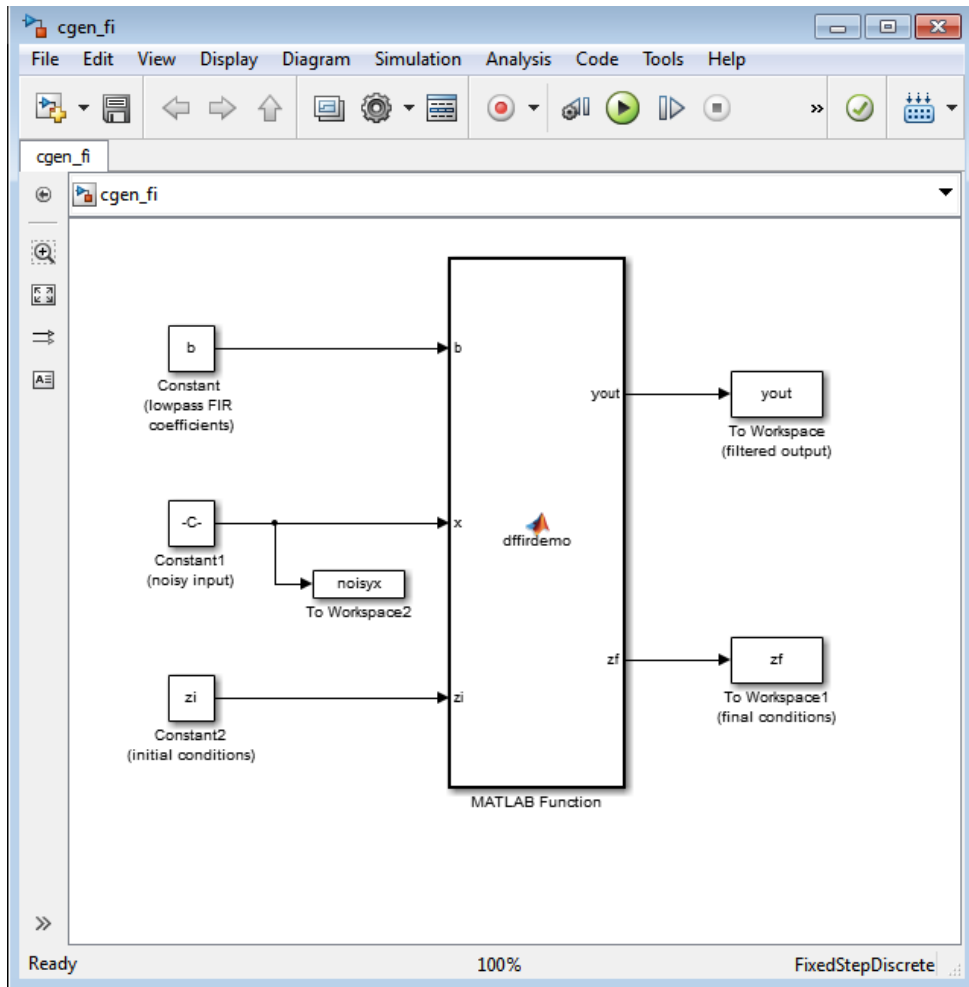
Prepare the Inputs

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 FIR Code Example Parameter Values” on page 14-81.
- 3 From the model menu, select **Simulation > Model Configuration Parameters** and set the following parameters.

Parameter	Value
Stop time	0

Parameter	Value
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` > `cgen_fi.c`. In this file, you can see the code generated from your model. Search for the following comment in your code:

```
/* codegen_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 FIR Code Example Parameter Values

Block	Parameter	Value
Constant	Constant value	b
	Interpret vector parameters as 1-D	Not selected
	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
Constant2	Constant value	zi
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf

Block	Parameter	Value
	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
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 14-83

“Control Dynamic Memory Allocation” on page 14-83

“Accelerate Code for MATLAB Functions with Variable-Size Data” on page 14-84

“Accelerate Code for a MATLAB Function That Expands a Vector in a Loop” on page 14-86

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 `fiaccl` :

```
fiaccl -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
<code>mexcfg.DynamicMemoryAllocation='Off';</code>	Dynamic memory allocation is disabled. All variable-size data is allocated statically on the stack.
<code>mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';</code>	Dynamic memory allocation is enabled for all variable-size arrays. All variable-size data is allocated dynamically on the heap.
<code>mexcfg.DynamicMemoryAllocation='Threshold';</code>	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 <code>Dynamic memory allocation threshold</code> 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 `fiaccel`. Use the following command-line options:

- `-args {coder.typeof...}` if you have variable-size inputs
- `-report` to generate a code generation report

For example:

```
fiaccel -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 27-20.

5 Fix upper bounds errors

Cause:	How To Fix:	For More Information:
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See “Specify Upper Bounds for Variable-Size Arrays” (Simulink) and “Diagnosing and Fixing Size Mismatch Errors” on page 27-20.
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See “Control Dynamic Memory Allocation” on page 14-83

- 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 14-86
- “Step 1: Add Compilation Directive for Code Generation” on page 14-87
- “Step 2: Address Issues Detected by the Code Analyzer” on page 14-87
- “Step 3: Generate MEX Code” on page 14-87
- “Step 4: Fix the Size Mismatch Error” on page 14-89
- “Step 5: Compare Execution Speed of MEX Function to Original Code” on page 14-91

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
```



```

    end
end

```

Step 1: Add Compilation Directive for Code Generation

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

```

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

1 Generate a MEX function for `uniquetol`:

```

T = numerictype(1, 16, 15);
fiaccl -report uniquetol -args {coder.typeof(fi(0,T),[1 100],1),coder.typeof(fi(0,T))}

```

What do these command-line options mean?

`T = numerictype(1, 16, 15)` creates a signed `numerictype` object with a 16-bit word length and 15-bit fraction length that you use to specify the data type of the input arguments for the function `uniquetol`.

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

- The first argument, `coder.typeof`, defines a variable-size input. The expression `coder.typeof(fi(0,T),[1 100],1)` defines input `A` as a vector of real, signed embedded.fi objects that have a 16-bit word length and 15-bit fraction length. The vector has 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” (MATLAB Coder).

- The second argument, `coder.typeof(fi(0,T))`, defines input `tol` as a real, signed embedded.fi object with a 16-bit word length and 15-bit fraction length.

The `-report` option instructs `fiaccel` to generate a code generation report, even if no errors or warnings occur.

For more information, see the `fiaccel` 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.

Function: `uniquetol`

```

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

```

Summary	All Messages (1)	Variables							
Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL	
+	1	B	Output	1 x 1	embedded.fi	No	Signed	16	15
+	2	A > 1	Input	1 x :100	embedded.fi	No	Signed	16	15
+	3	A > 2	Local	1 x :?	embedded.fi	No	Signed	16	15
+	4	tol	Input	1 x 1	embedded.fi	No	Signed	16	15
	5	k	Local	1 x 1	double	No	-	-	-
	6	i	Local	1 x 1	double	No	-	-	-

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:

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

In the current folder, `fiaccl` generates a MEX function for `uniquetol` named `uniquetol_mex` 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.

Function: `uniquetol`

```

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

```

Summary	All Messages (0)	Variables						
Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
1	B	Output	1x?	embedded.fi	No	Signed	16	15
2	A	Input	1x:100	embedded.fi	No	Signed	16	15
3	tol	Input	1x1	embedded.fi	No	Signed	16	15
4	k	Local	1x1	double	No	-	-	-
5	i	Local	1x1	double	No	-	-	-

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

Step 5: Compare Execution Speed of MEX Function to Original Code

Run the original MATLAB algorithm and MEX function with the same inputs for the same number of loop iterations and compare their execution speeds.

- 1 Create inputs of the correct class, complexity, and size to pass to the `uniquetol` MATLAB and MEX functions.

```
x = fi(rand(1,90), T);
tol = fi(0, T);
```

- 2 Run the original `uniquetol` function in a loop and time how long it takes to execute 10 iterations of the loop.

```
tic; for k=1:10, b = uniquetol(x,tol); end; tSim=toc
```

- 3 Run the generated MEX function with the same inputs for the same number of loop iterations.

```
tic; for k=1:10, b = uniquetol_mex(x,tol); end; tSim_mex=toc
```

- 4 Compare the execution times.

```
r = tSim/tSim_mex
```

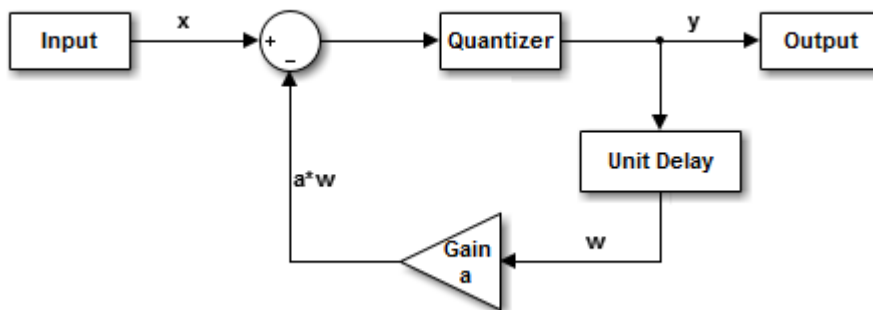
This example shows that generating a MEX function using `fiaccel` greatly accelerates the execution of the fixed-point algorithm.

Accelerate Fixed-Point Simulation

This example shows how to accelerate fixed-point algorithms using `fiaccel` function. You generate a MEX function from MATLAB® code, run the generated MEX function, and compare the execution speed with MATLAB code simulation.

Description of the Example

This example uses a first-order feedback loop. It also uses a quantizer to avoid infinite bit growth. The output signal is delayed by one sample and fed back to dampen the input signal.



Copy Required File

You need this MATLAB-file to run this example. Copy it to a temporary directory. This step requires write-permission to the system's temporary directory.

```

tempdirObj = fidemo.fiTempdir('fiaccelbasicsdemo');
fiaccedir = tempdirObj.tempDir;
fiaccesrc = ...
    fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','fiaccelFeedback.m');
copyfile(fiaccesrc,fiaccedir,'f');
  
```

Inspect the MATLAB Feedback Function Code

The MATLAB function that performs the feedback loop is in the file `fiaccelFeedback.m`. This code quantizes the input, and performs the feedback loop action :

```

type(fullfile(fiaccel_dir, 'fiaccelFeedback.m'))

function [y,w] = fiaccelFeedback(x,a,y,w)
%FIACCELFEEDBACK Quantizer and feedback loop used in FIACCELBASICSDEMO.

% Copyright 1984-2013 The MathWorks, Inc.
%#codegen

for n = 1:length(x)
    y(n) = quantize(x(n) - a*w, true, 16, 12, 'floor', 'wrap');
    w     = y(n);
end

```

The following variables are used in this function:

- **x** is the input signal vector.
- **y** is the output signal vector.
- **a** is the feedback gain.
- **w** is the unit-delayed output signal.

Create the Input Signal and Initialize Variables

```

rng('default'); % Random number generator
x = fi(2*rand(1000,1)-1,true,16,15); % Input signal
a = fi(.9,true,16,15); % Feedback gain
y = fi(zeros(size(x)),true,16,12); % Initialize output. Fraction length
% is chosen to prevent overflow
w = fi(0,true,16,12); % Initialize delayed output
A = coder.Constant(a); % Declare "a" constant for code
% generation

```

Run Normal Mode

```

tic,
y = fiaccelFeedback(x,a,y,w);
t1 = toc;

```

Build the MEX Version of the Feedback Code

```

fiaccel fiaccelFeedback -args {x,A,y,w} -o fiaccelFeedback_mex

```

Run the MEX Version

```

tic

```



```
y2 = fiaccelFeedback_mex(x,y,w);  
t2 = toc;
```

Acceleration Ratio

Code acceleration provides optimizations for accelerating fixed-point algorithms through MEX file generation. Fixed-Point Designer™ provides a convenience function `fiaccel` to convert your MATLAB code to a MEX function, which can greatly accelerate the execution speed of your fixed-point algorithms.

```
r = t1/t2
```

```
r =
```

```
8.5754
```

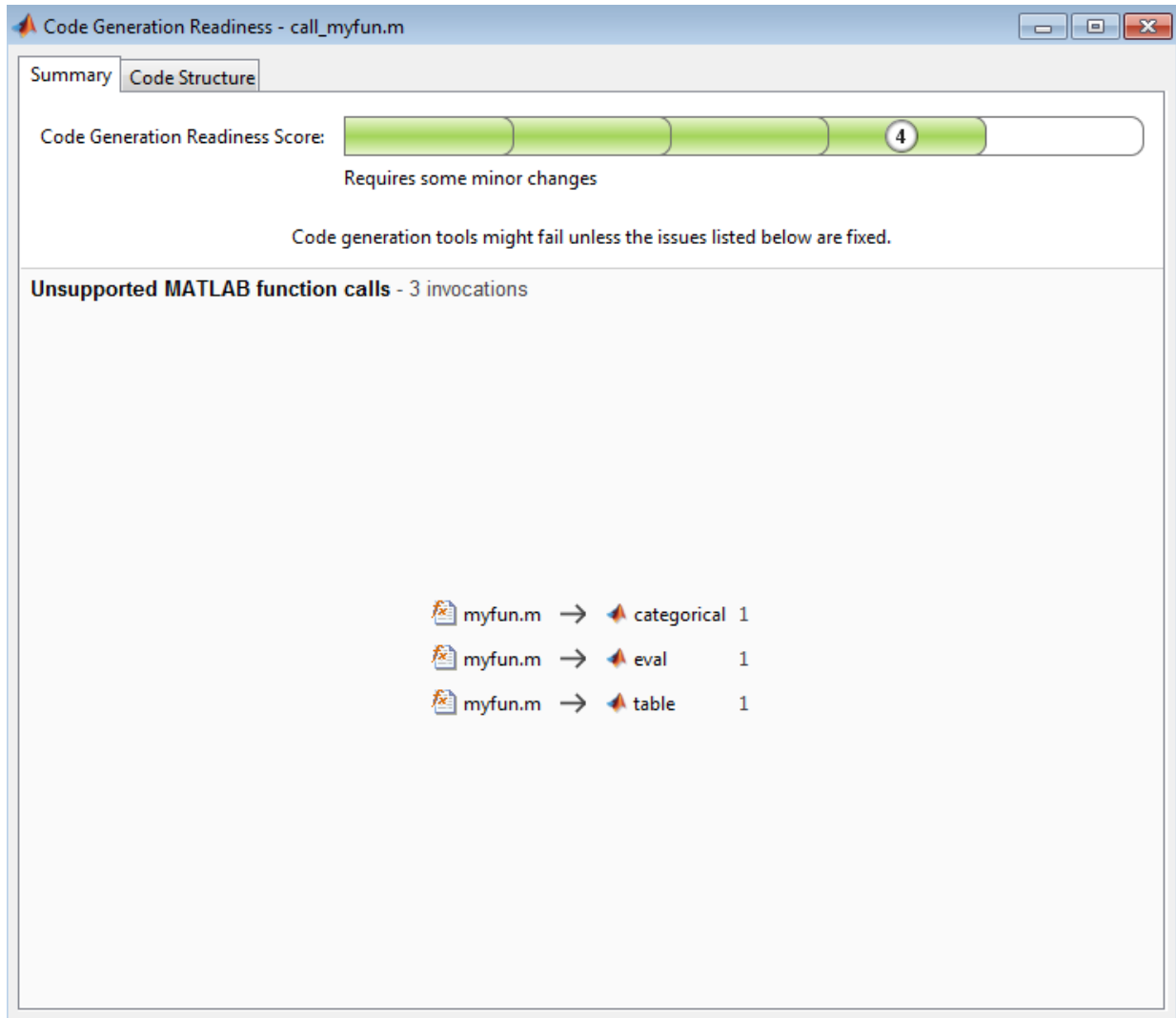
Clean up Temporary Files

```
clear fiaccelFeedback_mex;  
tempdirObj.cleanUp;
```

Code Generation Readiness Tool

The code generation readiness tool screens MATLAB code for features and functions that code generation does not support. The tool provides a report that lists the source files that contain unsupported features and functions. The report also indicates the amount of work required to make the MATLAB code suitable for code generation. It is possible that the tool does not detect all code generation issues. Under certain circumstances, it is possible that the tool can report false errors. Therefore, before you generate C code, verify that your code is suitable for code generation by generating a MEX function.

Summary Tab



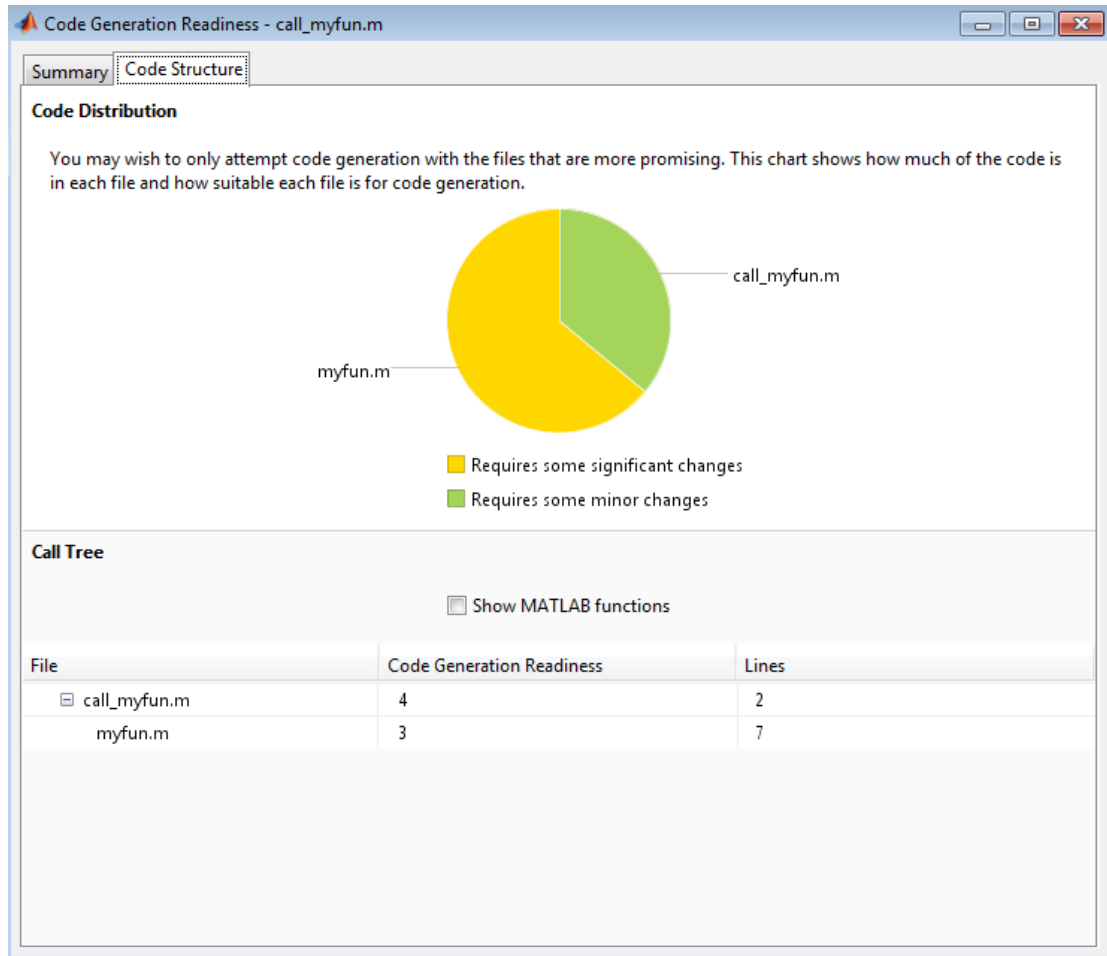
The **Summary** tab provides a **Code Generation Readiness Score**, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that

the tool does not detect code generation issues; the code is ready to use with minimal or no changes.

On this tab, the tool also displays information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. To learn more about the issues and how to fix them, use the Code Analyzer.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features.
- Unsupported data types.

Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab displays information about the relative size of each file and how suitable each file is for code generation.

Code Distribution

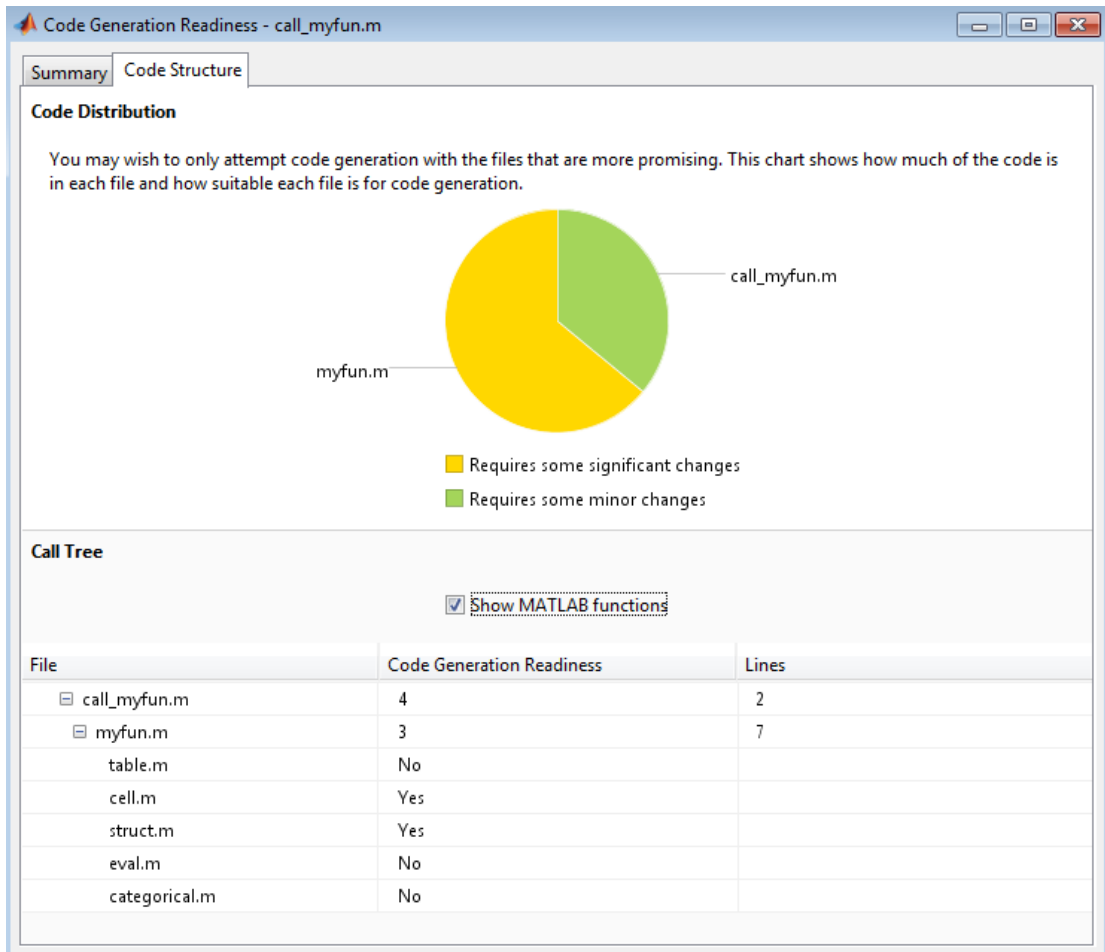
The **Code Distribution** pane displays a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. During the planning phase of a project, you can use this information for estimation and scheduling. If the report indicates that multiple files are not suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane displays information about the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues. The code is ready to use with minimal or no changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions

If you select **Show MATLAB Functions**, the report also lists the MATLAB functions that your function calls. For each of these MATLAB functions, if code generation supports the function, the report sets **Code Generation Readiness** to **Yes**.



Related Examples

- “Check Code Using the Code Generation Readiness Tool” on page 14-102

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 14-96

Check Code Using the MATLAB Code Analyzer

The code analyzer checks your code for problems and recommends modifications. You can use the code analyzer to check your code interactively in the MATLAB Editor while you work.

To verify that continuous code checking is enabled:

- 1** In MATLAB, select the **Home** tab and then click **Preferences**.
- 2** In the **Preferences** dialog box, select **Code Analyzer**.
- 3** In the **Code Analyzer Preferences** pane, verify that **Enable integrated warning and error messages** is selected.

Fix Errors Detected at Code Generation Time

When the code generator detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “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 14-23.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generator determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls for Code Generation” on page 16-2. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

See Also

- “Use Fixed-Point Code Generation Reports” on page 14-36
- “Why Test MEX Functions in MATLAB?” (MATLAB Coder)
- “When to Generate Code from MATLAB Algorithms” on page 21-2
- “Debugging Strategies” on page 14-23
- “Declaring MATLAB Functions as Extrinsic Functions” on page 16-10

Avoid Multiword Operations in Generated Code

This example shows how to avoid multiword operations in generated code by using the `accumpos` function instead of simple addition in your MATLAB algorithm. Similarly, you can use `accumneg` for subtraction.

This example requires a MATLAB Coder license.

Write a simple MATLAB algorithm that adds two numbers and returns the result.

```
function y = my_add1(a, b)
y = a+b;
```

Write a second MATLAB algorithm that adds two numbers using `accumpos` and returns the result.

```
function y = my_add2(a, b)
y = accumpos(a, b); % floor, wrap
```

`accumpos` adds `a` and `b` using the data type of `a`. `b` is cast into the data type of `a`. If `a` is a `fi` object, by default, `accumpos` sets the rounding mode to 'Floor' and the overflow action to 'Wrap'. It ignores the `fimath` properties of `a` and `b`.

Compare the outputs of the two functions in MATLAB.

```
a = fi(1.25, 1, 32,5);
b = fi(0.125, 0, 32);
%%
y1 = my_add1(a, b)
y2 = my_add2(a, b)

y1 =

    1.3750

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 62
        FractionLength: 34

y2 =

    1.3750
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 5
```

For the simple addition, the word length grows but using `accumpos`, the word length of the result is the same as that of `a`.

Generate C code for the function `my_add1`. First, disable use of the `long long` data type because it is not usually supported by the target hardware.

```
hw = coder.HardwareImplementation;
hw.ProdHWDeviceType = 'Generic->32-bit Embedded Processor';
hw.ProdLongLongMode = false;
hw.ProdBitPerLong = 32;
cfg = coder.config('lib');
cfg.HardwareImplementation = hw;
codegen my_add1 -args {a,b} -report -config cfg
```

MATLAB Coder generates a C static library and provides a link to the code generation report.

View the generated code for the simple addition. Click the `View report` link to open the code generation report and then scroll to the code for the `my_add1` function.

```
/* Function Declarations */
static void MultiWordAdd(const unsigned long u1[], const unsigned long u2[],
    unsigned long y[], int n);
static void MultiWordSignedWrap(const unsigned long u1[], int n1, unsigned int
    n2, unsigned long y[]);
static void sLong2MultiWord(long u, unsigned long y[], int n);
static void sMultiWord2MultiWord(const unsigned long u1[], int n1, unsigned long
    y[], int n);
static void sMultiWord2sMultiWordSat(const unsigned long u1[], int n1, unsigned
    long y[], int n);
static void sMultiWordShl(const unsigned long u1[], int n1, unsigned int n2,
    unsigned long y[], int n);
static void sMultiWordShr(const unsigned long u1[], int n1, unsigned int n2,
    unsigned long y[], int n);
static void uLong2MultiWord(unsigned long u, unsigned long y[], int n);
```

The generated C code contains multiple multiword operations.

Generate C code for the function `my_add2`.

```
codegen my_add2 -args {a,b} -report -config cfg
```

View the generated code for the addition using `accumpos`. Click the `View report` link to open the code generation report and then scroll to the code for the `my_add2` function.

```
int my_add2(int a, unsigned int b)
{
    int y;
    y = a + (int)(b >> 29);
    /* floor, wrap */
    return y;
}
```

For this function, the generated code contains no multiword operations.

Find Potential Data Type Issues in Generated Code

In this section...

“Data Type Issues Overview” on page 14-108

“Enable Highlighting of Potential Data Type Issues” on page 14-108

“Find and Address Cumbersome Operations” on page 14-109

“Find and Address Expensive Rounding” on page 14-110

“Find and Address Expensive Comparison Operations” on page 14-112

“Find and Address Multiword Operations” on page 14-113

Data Type Issues Overview

When you convert MATLAB code to fixed point, you can highlight potential data type issues in the generated HTML report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations.

- The double-precision check highlights expressions that result in a double-precision operation. When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone.

For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to **C99 (ISO)**.

- The single-precision check highlights expressions that result in a single operation.
- The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient” on page 46-11.

Enable Highlighting of Potential Data Type Issues

Procedure 14.9. Enable the highlight option using the Fixed-Point Converter app

- 1 On the **Convert to Fixed Point** page, click the **Settings** arrow .

- 2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to **Yes**.

When conversion is complete, open the fixed-point conversion HTML report to view the highlighting. Click **View report** in the **Type Validation Output** tab.

Procedure 14.10. Enable the highlight option using the command-line interface

- 1 Create a fixed-point code configuration object:


```
fixptcfg = coder.config('fixpt');
```
- 2 Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.


```
fixptcfg.HighlightPotentialDataTypeIssues = true;
```

Find and Address Cumbersome Operations

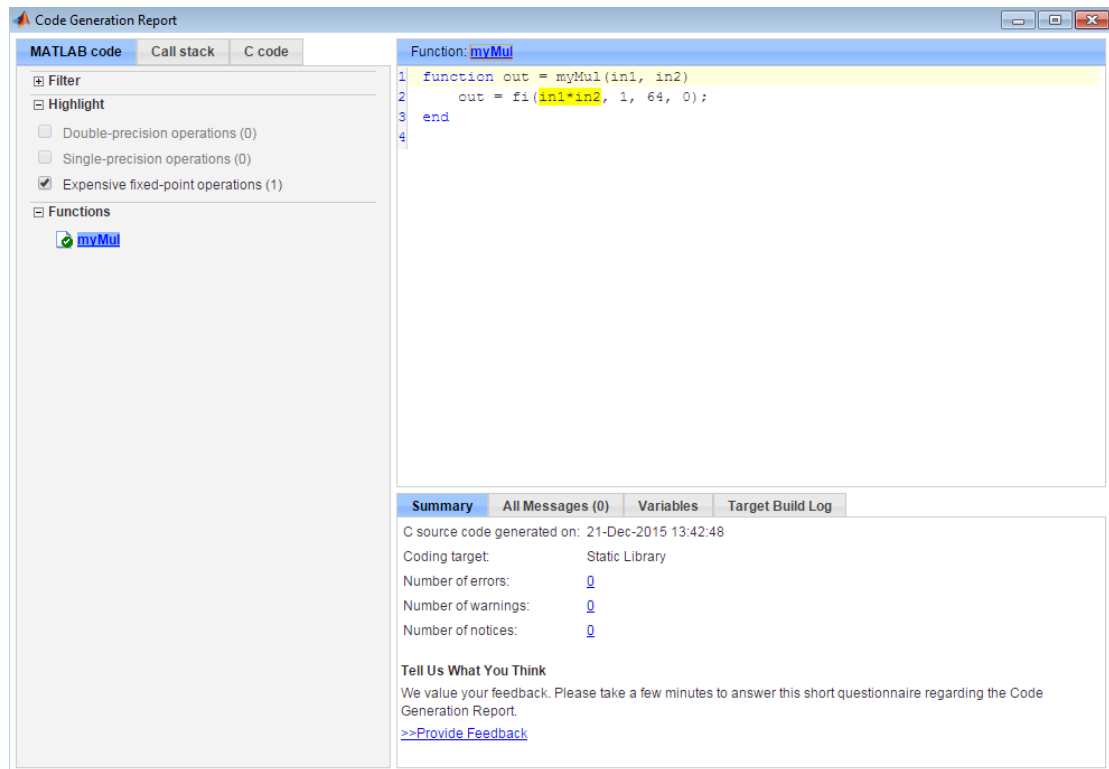
Cumbersome operations usually occur due to an insufficient range of output. Avoid inputs to a multiply or divide operation that have word lengths larger than the base integer type of your processor. Software can process operations with larger word lengths, but this approach requires more code and runs slower.

This example requires Embedded Coder[®] and Fixed-Point Designer licenses to run. The target word length for the processor in this example is 64.

- 1 Create the function `myMul`.


```
function out = myMul(in1, in2)
    out = fi(in1*in2, 1, 64, 0);
end
```
- 2 Generate code for `myMul`.


```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
fm = fimath('ProductMode', 'SpecifyPrecision', 'ProductWordLength', 64);
codegen -config cfg myMul -args {fi(1, 1, 64, 4, fm), fi(1, 1, 64, 4, fm)}
```
- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



To resolve this issue, modify the data types of `in1` and `in2` so that the word length of the product does not exceed the target word length of 64.

Find and Address Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

- 1 Create the function `myRounding`.


```
function [quot] = myRounding(in1, in2)
    quot = in1 / in2;
end
```

- 2 Generate code for myRounding.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRounding -args {fi(1, 1, 16, 2), fi(1, 1, 16, 4)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.

The screenshot shows the 'Code Generation Report' window with the 'MATLAB code' tab selected. The left pane shows the 'Highlight' section expanded, with 'Expensive fixed-point operations (1)' checked. The main pane displays the MATLAB code for the 'myRounding' function, with the line 'quot = in1 / in2;' highlighted in yellow. The bottom pane shows a 'Summary' section with the following information:

Summary	All Messages (0)	Variables	Target Build Log
C source code generated on: 21-Dec-2015 13:53:18			
Coding target:	Static Library		
Number of errors:	0		
Number of warnings:	0		
Number of notices:	0		

Tell Us What You Think
 We value your feedback. Please take a few minutes to answer this short questionnaire regarding the Code Generation Report.
[>>Provide Feedback](#)

This division operation uses the default rounding method, `nearest`. Changing the rounding method to `FLOOR` provides a more efficient implementation.

Find and Address Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, before comparing an unsigned integer to a signed integer, one of the inputs must be cast to the signedness of the other. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

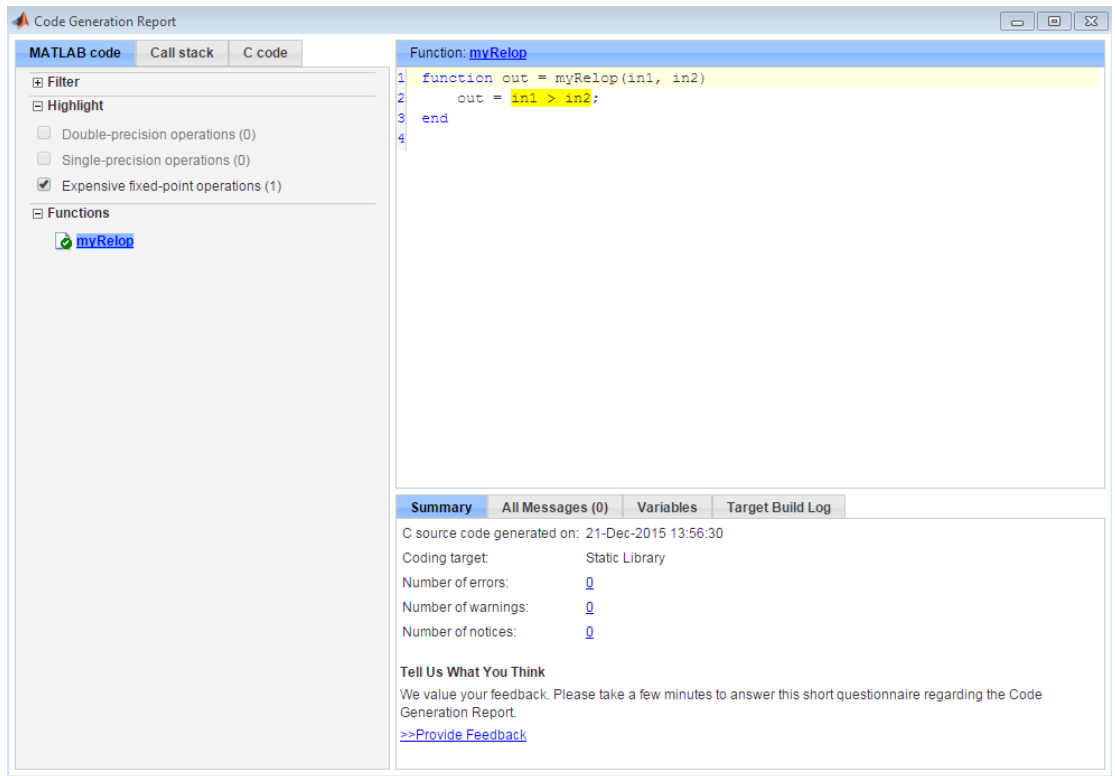
- 1 Create the function `myRelop`.

```
function out = myRelop(in1, in2)
    out = in1 > in2;
end
```

- 2 Generate code for `myRelop`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRelop -args {fi(1, 1, 14, 3, 1), fi(1, 0, 14, 3, 1)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The first input argument, `in1`, is signed, while `in2` is unsigned. Extra code is generated because a cast must occur before the two inputs can be compared.

Change the signedness and scaling of one of the inputs to generate more efficient code.

Find and Address Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

This example requires Embedded Coder and Fixed-Point Designer licenses to run. The target word length is 64 in this example.

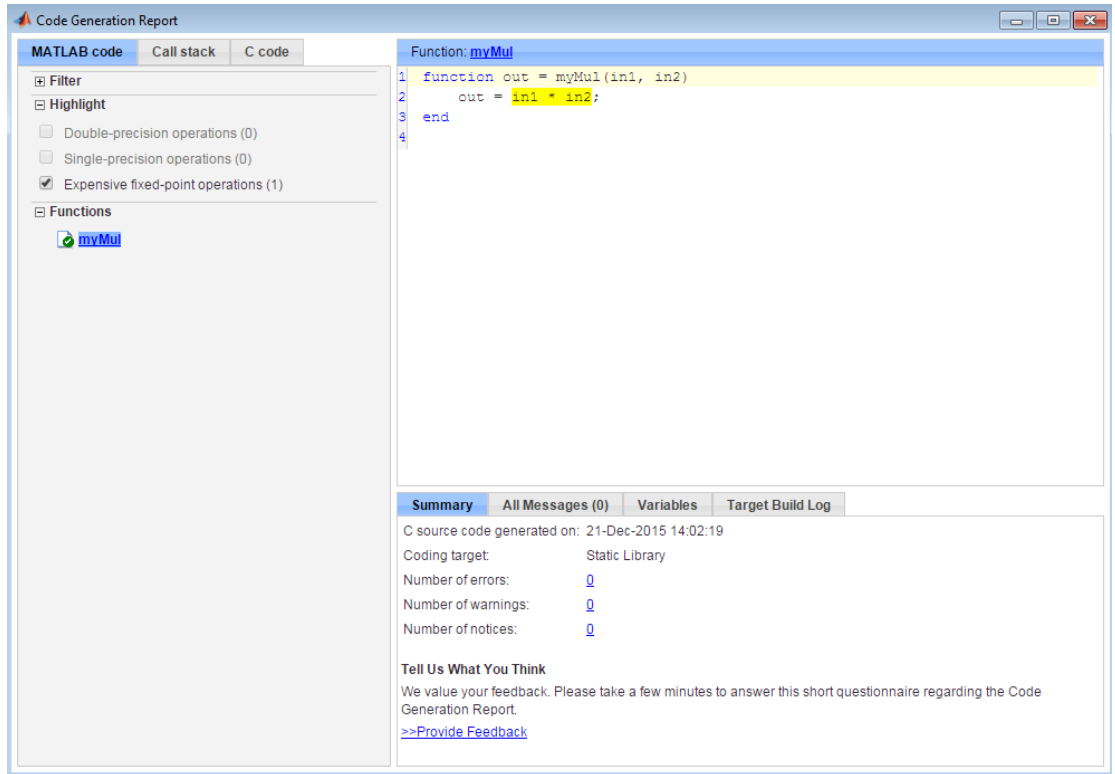
- 1 Create the function `myMul`.

```
function out = myMul(in1, in2)
    out = in1 * in2;
end
```

- 2 Generate code for `myMul`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myMul -args {fi(1, 1, 33, 4), fi(1, 1, 32, 4)}
```

- 3 Click **View report**.
- 4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
- 5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The `in1 * in2` operation is highlighted in the HTML report. On the bottom pane, click the **Variables** tab. The word length of `in1` is 33 bits, and the word length of `in2` is 32 bits. Hovering over the highlighted expression reveals that the product has a word length of 65, which is larger than the target word length of 64. Therefore, the software detects a multiword operation.

To resolve this issue, modify the data types of `in1` and `in2` so the word length of the product does not exceed the target word length, or specify the `ProductMode` property of the local `fimath` object.

Interoperability with Other Products

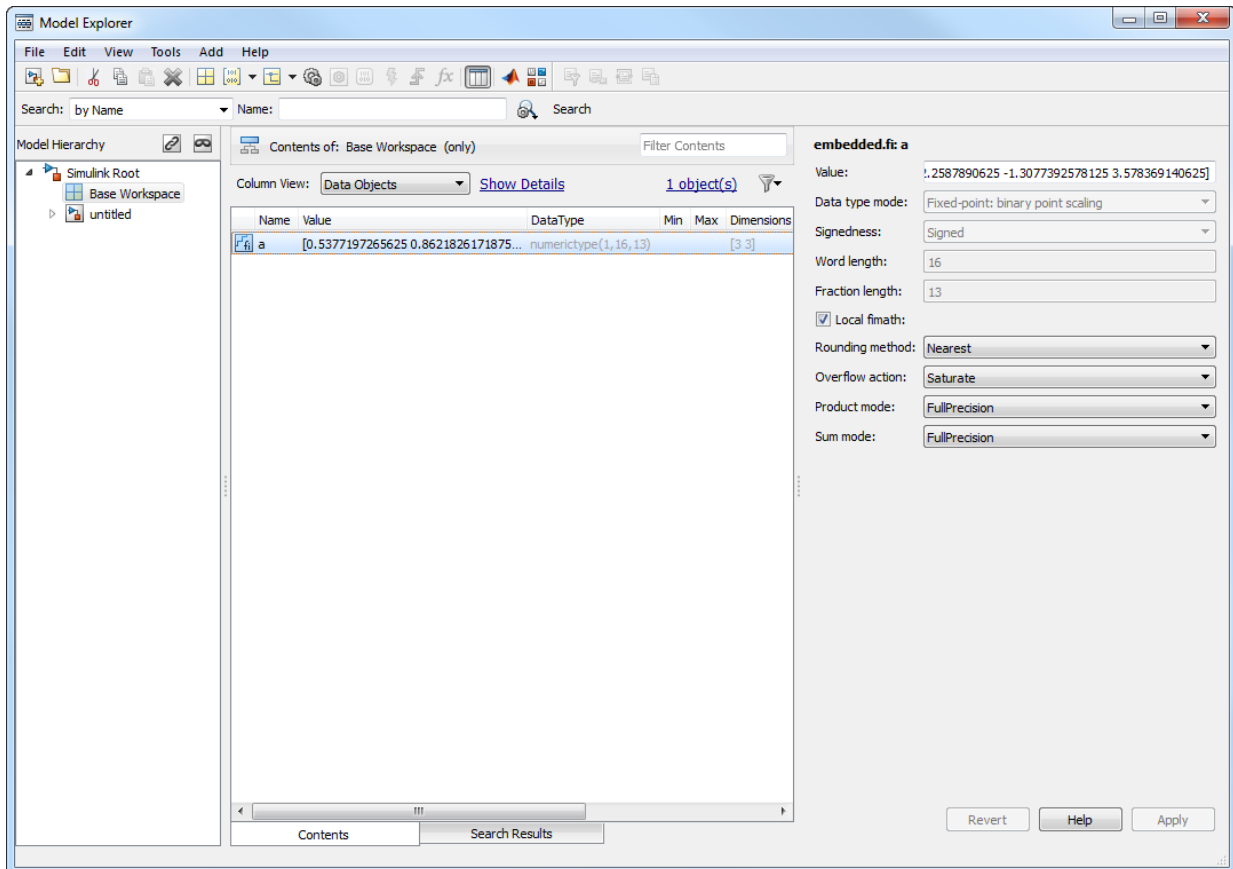
- “fi Objects with Simulink” on page 15-2
- “fi Objects with DSP System Toolbox” on page 15-10
- “Ways to Generate Code” on page 15-15

fi Objects with Simulink

In this section...
“View and Edit fi objects in Model Explorer” on page 15-2
“Reading Fixed-Point Data from the Workspace” on page 15-3
“Writing Fixed-Point Data to the Workspace” on page 15-4
“Setting the Value and Data Type of Block Parameters” on page 15-8
“Logging Fixed-Point Signals” on page 15-8
“Accessing Fixed-Point Block Data During Simulation” on page 15-8

View and Edit fi objects in Model Explorer

You can view and edit fi objects and their local fimath properties using Model Explorer in Simulink. You can change the writable properties of fi objects from the Model Explorer, but you cannot change the numeric type properties of fi objects after creation.



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 =

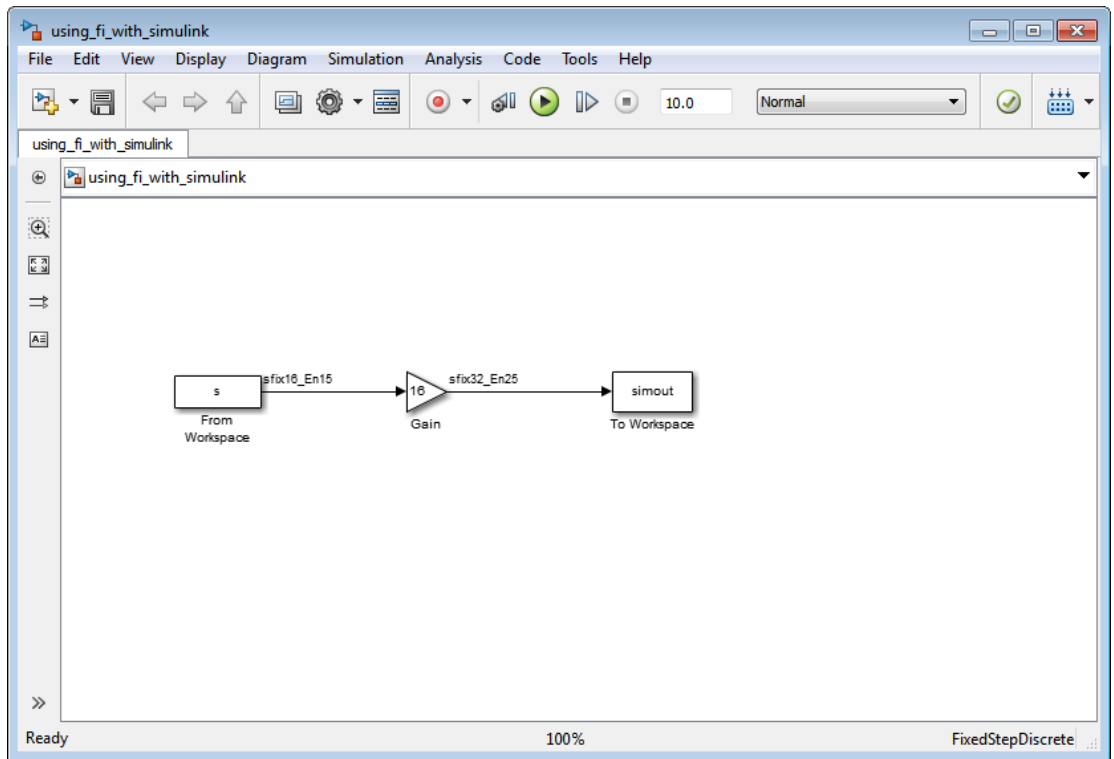
    signals: [1x1 struct]
    time: [11x1 double]
```

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 Designer expressions to specify the value and data type of block parameters in Simulink. For more information, see “Specify Fixed-Point Data Types” (Simulink) in the Simulink documentation.

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:

- 1 Select the signal.
- 2 Open the **Record** dropdown.
- 3 Select **Log/Unlog Selected Signals**.

For more information, refer to “Export Signal Data Using Signal Logging” (Simulink) 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 15-10
“Writing Fixed-Point Signals to the Workspace” on page 15-10

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 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:

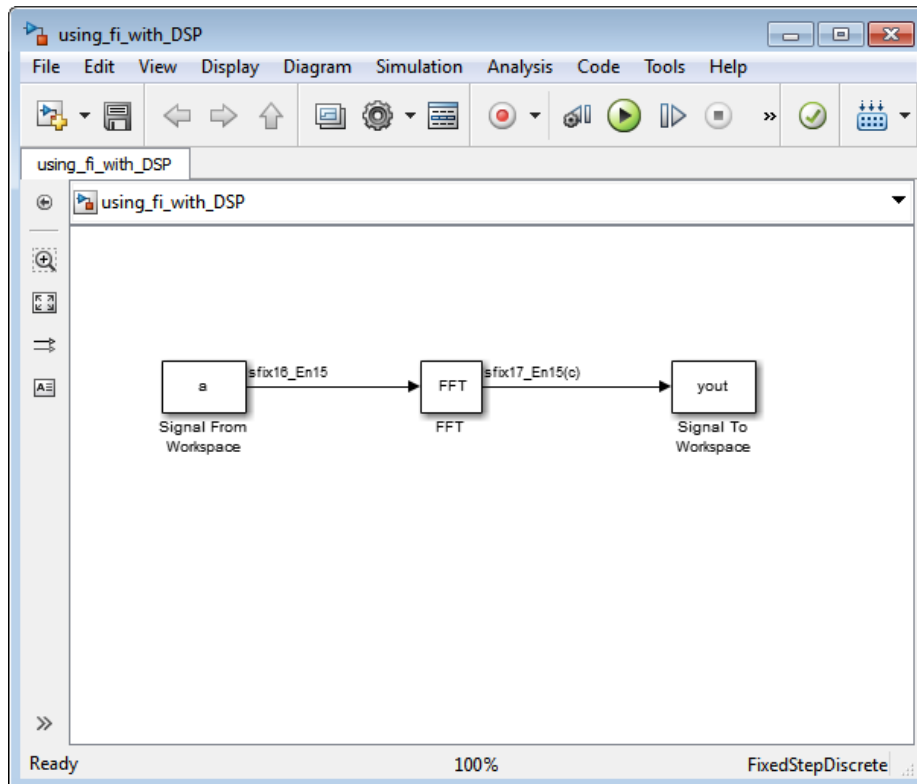
- **Signal** — a
- **Sample time** — 1
- **Samples per frame** — 2
- **Form output after final data value by** — Setting to zero

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

Ways to Generate Code

There are several ways to use Fixed-Point Designer software to generate code:

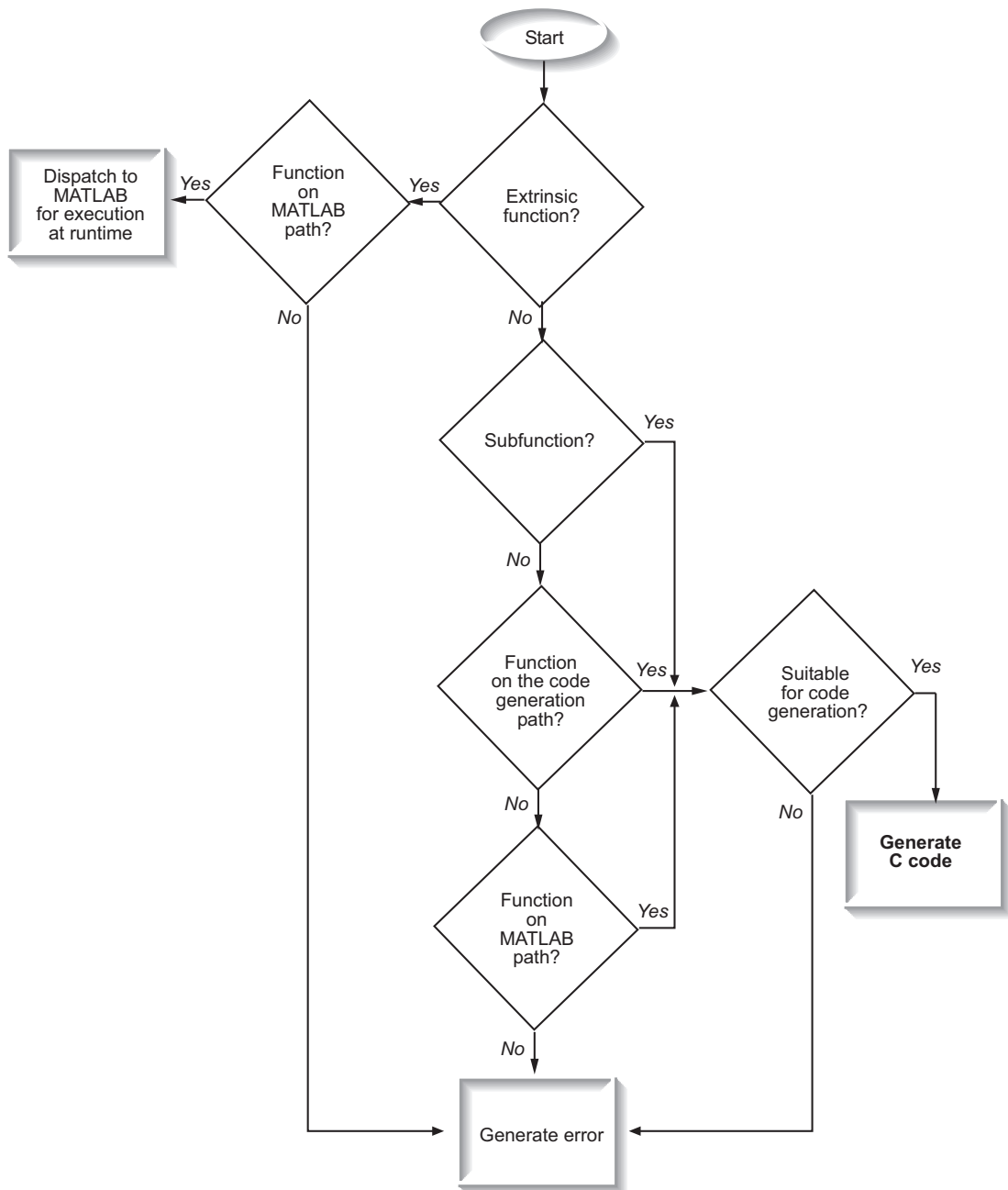
- The Fixed-Point Designer `fiaccel` function converts your fixed-point MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.
- The MATLAB Coder `codegen` function automatically converts MATLAB code to C/C++ code. Using the MATLAB Coder software allows you to accelerate your MATLAB code that uses Fixed-Point Designer software. To use the `codegen` function with Fixed-Point Designer software, you also need to have a MATLAB Coder license. For more information, see “C Code Generation at the Command Line” (MATLAB Coder) 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 Designer 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 for Code Generation” on page 16-2
- “Resolution of File Types on Code Generation Path” on page 16-6
- “Compilation Directive `%#codegen`” on page 16-8
- “Extrinsic Functions” on page 16-9
- “Code Generation for Recursive Functions” on page 16-18
- “Force Code Generator to Use Run-Time Recursion” on page 16-21

Resolution of Function Calls for Code Generation

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:



Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path

See “Compile Path Search Order” on page 16-4.

- Attempts to compile functions unless the code generator determines that it should not compile them or you explicitly declare them to be extrinsic.

If a MATLAB function is not supported for code generation, you can declare it to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions” on page 16-10. During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, the code generator attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, compilation errors occur.

The code generator detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 16-6

Compile Path Search Order

During code generation, function calls are resolved on two paths:

- 1 Code generation path

MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.

- 2 MATLAB path

If the function is not on the code generation path, MATLAB searches this path.

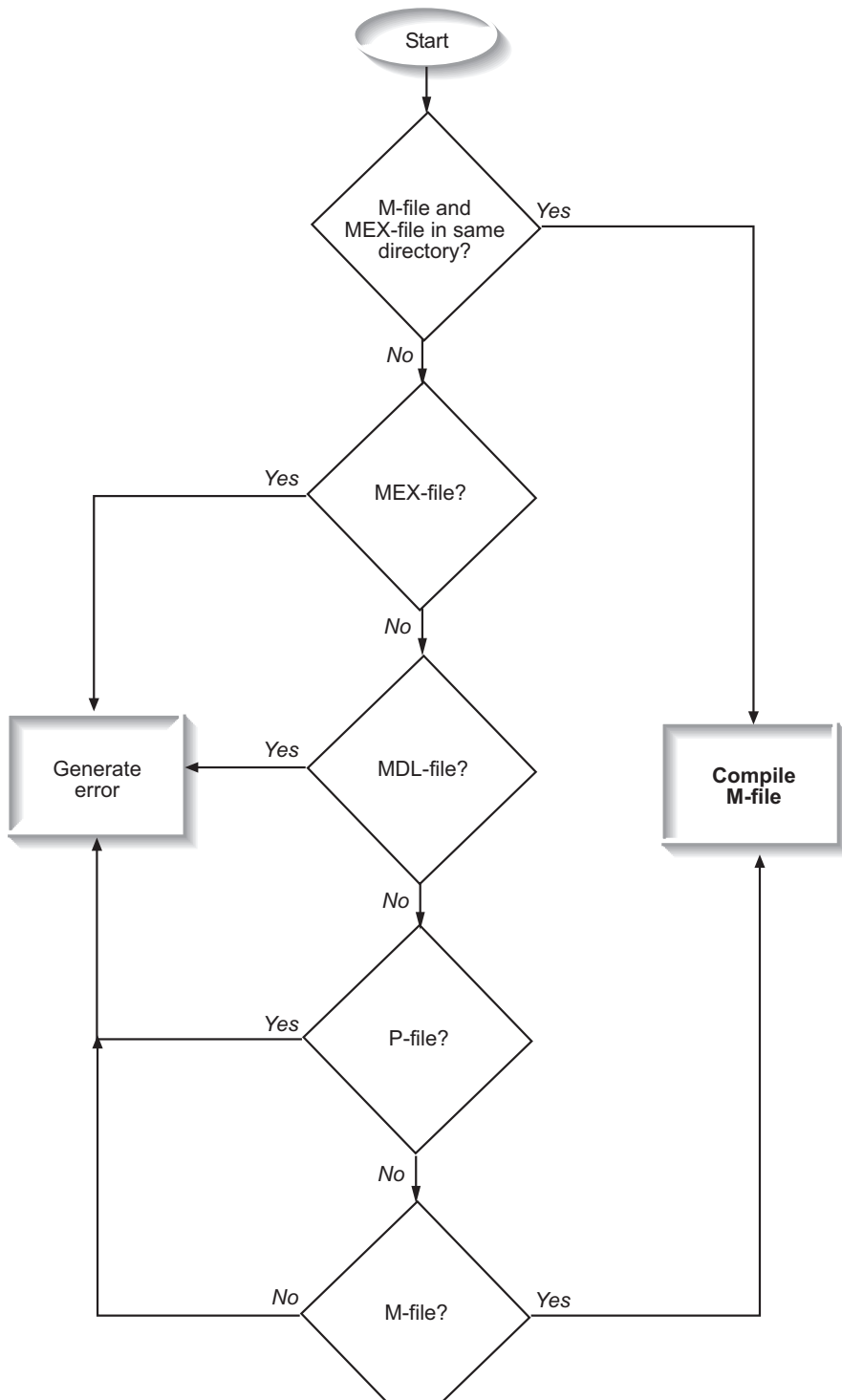
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order” (MATLAB)).

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:



Compilation Directive `%#codegen`

Add the `%#codegen` directive (or pragma) to your function after the function signature to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during code generation.

```
function y = my_fcn(x) %#codegen  
  
.....
```

Note: The `%#codegen` directive is not necessary for MATLAB Function blocks. Code inside a MATLAB Function block is always intended for code generation. The `%#codegen` directive, or the absence of it, does not change the error checking behavior.

Extrinsic Functions

The code generator attempts to generate code for functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using `coder.extrinsic`. During simulation, the code generator produces code for these functions, but does not generate their internal code. During standalone code generation, the code generator attempts to determine whether the visualization function affects the output of the function in which it is called. Provided that the output does not change, the code generator proceeds with code generation, but excludes the visualization function from the generated code. Otherwise, compilation errors occur.

For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot` and then run the generated MEX function, the code generator dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.

The screenshot shows the MATLAB code editor interface. On the left, there is a 'Filter' pane with 'Functions' expanded, showing 'stats' and 'stats > avg'. The main editor displays the following code:

```

Function: stats
Calls: Select a function call:

1 function [mean, stdev] = stats(vals)
2 %#codegen
3
4 % Calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 len = length(vals);
8 mean = avg(vals, len);
9 stdev = sqrt(sum((vals-avg(vals,len)).^2)/len);
10 plot(vals, '-+');
11

```

A tooltip points to the `plot` function call on line 10, stating: "Only supported within the MATLAB environment."

For unsupported functions other than common visualization functions, you must declare the functions to be extrinsic (see “Resolution of Function Calls for Code Generation” on page 16-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “Resolution of Extrinsic Functions During Simulation” on page 16-14).

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

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 16-10).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 16-14).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

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

Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`. You do not have to declare `axis` as extrinsic because `axis` is one of the common visualization functions that the code generator automatically treats as extrinsic.

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
% and displays the triangle.
```

```
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
```

```
function create_plot(a, b, color)
%Declare patch as extrinsic
```

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

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

The code generator does not produce code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

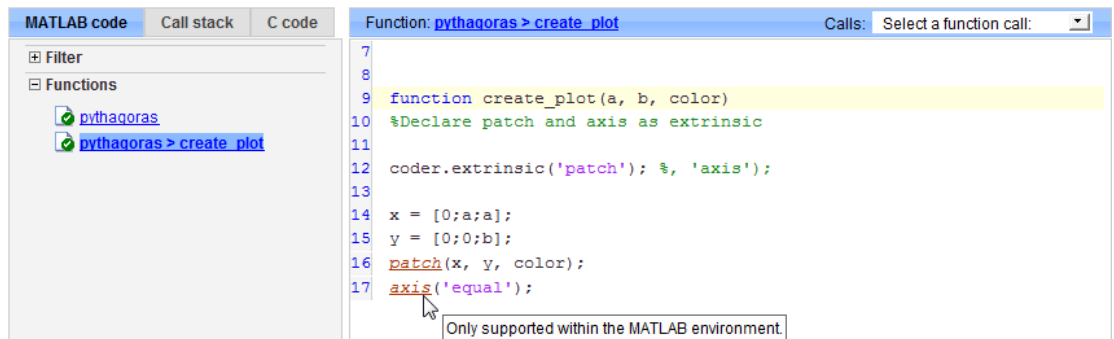
To test the function, follow these steps:

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


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

- Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.



The screenshot shows the MATLAB code generation report for the `pythagoras` function. The code is displayed in a window titled "Function: pythagoras > create_plot". The code includes the following lines:

```

7
8
9 function create_plot(a, b, color)
10 %Declare patch and axis as extrinsic
11
12 coder.extrinsic('patch'); %,'axis');
13
14 x = [0;a;a];
15 y = [0;0;b];
16 patch(x, y, color);
17 axis('equal');

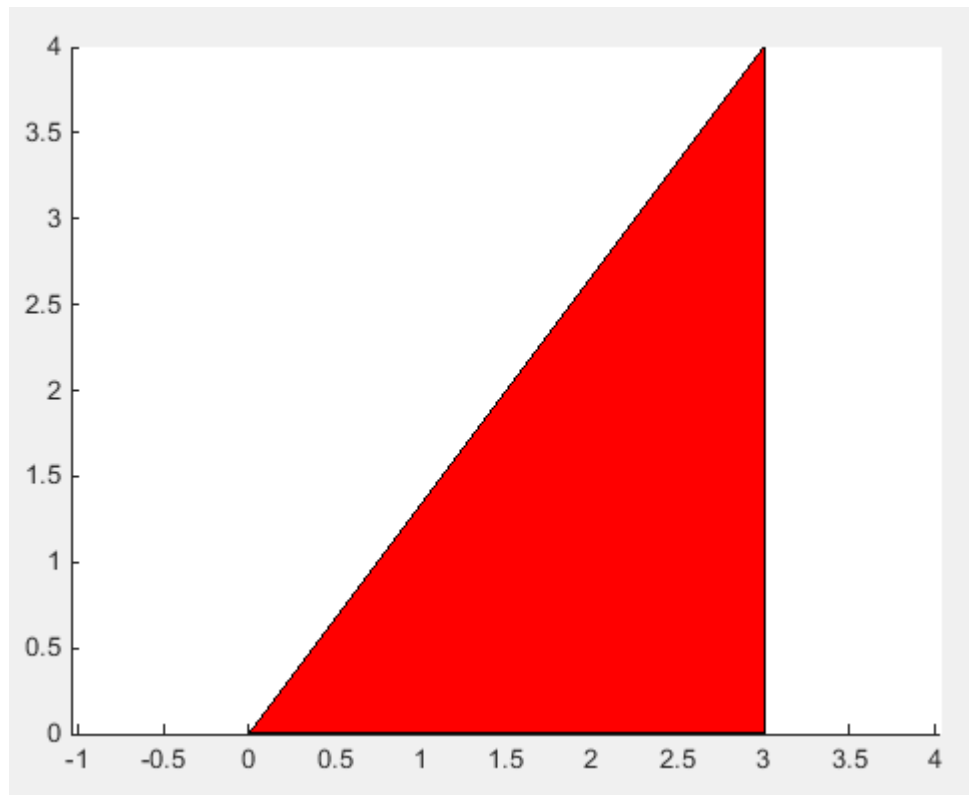
```

The `patch` and `axis` function calls on lines 16 and 17 are highlighted in red. A tooltip points to the `axis` call, indicating "Only supported within the MATLAB environment".

- Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the `coder.extrinsic` Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output during simulation, without generating unnecessary code (see “Resolution of Extrinsic Functions During Simulation” on page 16-14).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with `mxArrays`” on page 16-15).

- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 16-13).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 16-13). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 16-14).

Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` 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 16-14.

Calling MATLAB Functions Using `feval`

The function `feval` is automatically interpreted as an extrinsic function during code generation. Therefore, you can use `feval` to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

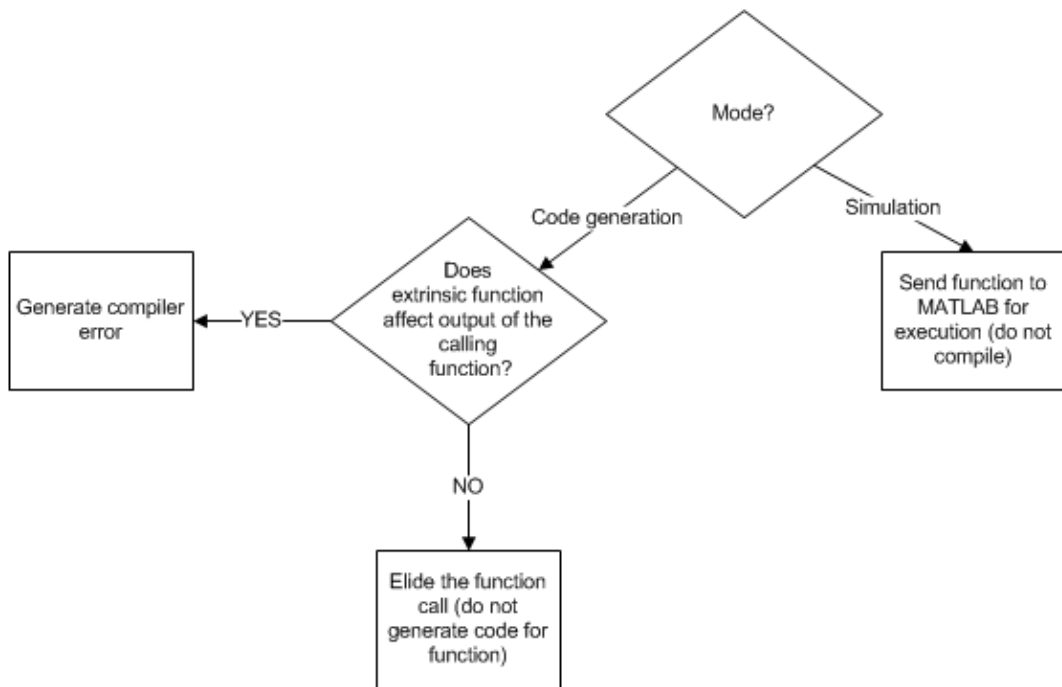
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not compiled — which has the same result as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

Resolution of Extrinsic Functions During Simulation

The code generator resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, the code generator attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with `mxArrays`” on page 16-15). Provided that the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, the code generator issues a compiler error.

Working with `mxArrays`

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables

- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting `mxArrays` to Known Types” on page 16-16.

Converting `mxArrays` to Known Types

To convert an `mxArray` to a known type, assign the `mxArray` to a variable whose type is defined. At run time, the `mxArray` is converted to the type of the variable assigned to it. However, if the data in the `mxArray` is not consistent with the type of the variable, you get a run-time error.

For example, consider this code:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level function `foo` calls the extrinsic MATLAB function `rat`, which returns two `mxArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxArrays` to another MATLAB function — in this case, `min` — you cannot assign the `mxArray` returned by `min` to the output `y`.

If you run this function `foo` in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

```
Function output 'y' cannot be of MATLAB type.
```

To fix this problem, define `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller, or read or write to the caller workspace do not work during code generation. Such functions include:
 - `dbstack`
 - `evalin`
 - `assignin`
 - `save`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code can produce unpredictable results if your extrinsic function performs the following actions at run time:
 - Change folders
 - Change the MATLAB path
 - Delete or add MATLAB files
 - Change warning states
 - Change MATLAB preferences
 - Change Simulink parameters

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Code Generation for Recursive Functions

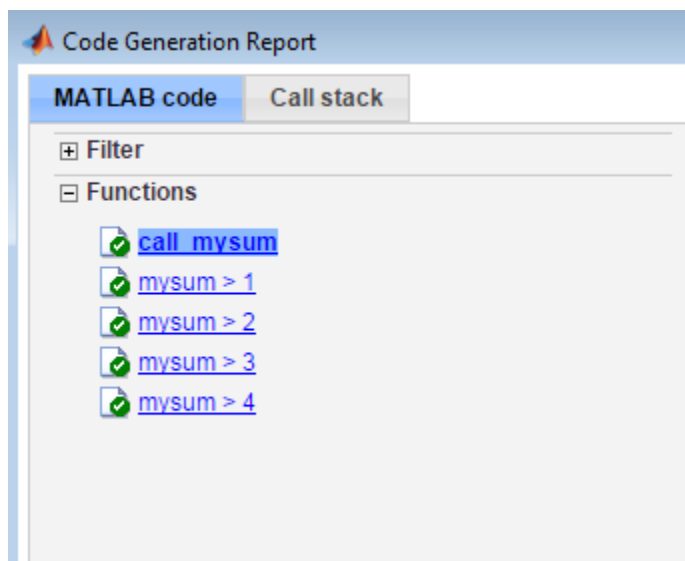
To generate code for recursive MATLAB functions, the code generator uses compile-time recursion or run-time recursion. You can influence whether the code generator uses compile-time or run-time recursion by modifying your MATLAB code. See “Force Code Generator to Use Run-Time Recursion” on page 16-21.

You can disallow recursion or disable run-time recursion by modifying configuration parameters.

When you use recursive functions in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Recursive Function Limitations for Code Generation” on page 16-20.

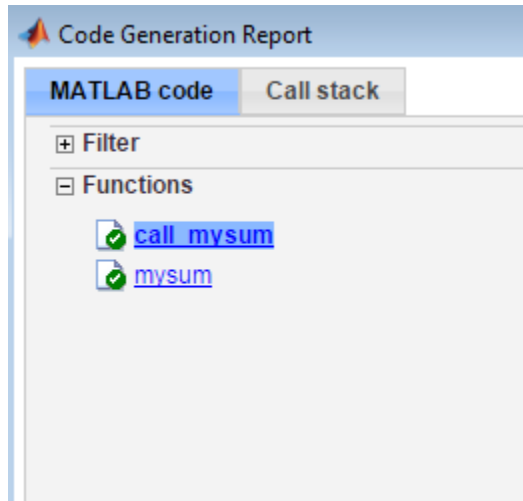
Compile-Time Recursion

With compile-time recursion, the code generator creates multiple versions of a recursive function in the generated code. The inputs to each version have values or sizes that are customized for that version. These versions are known as *function specializations*. You can see if the code generator used compile-time recursion by looking at the code generation report. Here is an example of compile-time recursion in the report.



Run-Time Recursion

With run-time recursion, the code generator produces a recursive function in the generated code. You can see if the code generator used run-time recursion by looking at the code generation report. Here is an example of run-time recursion in the report.



Disallow Recursion

In a code acceleration configuration object, set the value of the `CompileTimeRecursionLimit` configuration parameter to 0.

Disable Run-Time Recursion

Some coding standards, such as MISRA[®], do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C[®], disable run-time recursion.

In a code acceleration configuration object, set `EnableRuntimeRecursion` to `false`.

If your code requires run-time recursion and run-time recursion is disabled, you must rewrite your code so that it uses compile-time recursion or does not use recursion.

Recursive Function Limitations for Code Generation

When you use recursion in MATLAB code that is intended for code generation, follow these restrictions:

- Assign all outputs of a run-time recursive function before the first recursive call in the function.
- Assign all elements of cell array outputs of a run-time recursive function.
- Inputs and outputs of run-time recursive functions cannot be classes.
- The `StackUsageMax` code acceleration configuration parameter is ignored for run-time recursion.

More About

- “Force Code Generator to Use Run-Time Recursion” on page 16-21
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 46-38
- “Compile-Time Recursion Limit Reached” on page 46-33
- “Set Up C Code Compilation Options” on page 14-25
- “Use Fixed-Point Code Generation Reports” on page 14-36

Force Code Generator to Use Run-Time Recursion

When your MATLAB code includes recursive function calls, the code generator uses compile-time or run-time recursion. With compile-time recursion, the code generator creates multiple versions of the recursive function in the generated code. These versions are known as function specializations. With run-time recursion, the code generator produces a recursive function. If compile-time recursion results in too many function specializations or if you prefer run-time recursion, you can try to force the code generator to use run-time recursion. Try one of these approaches:

- “Treat the Input to the Recursive Function as a Nonconstant” on page 16-21
- “Make the Input to the Recursive Function Variable-Size” on page 16-22
- “Assign Output Variable Before the Recursive Call” on page 16-23

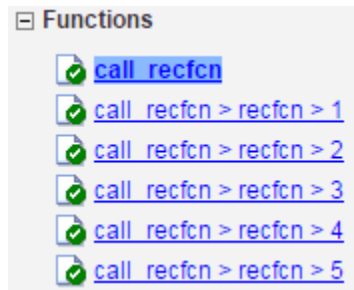
Treat the Input to the Recursive Function as a Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 5;
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

`call_recfcn` calls `recfcn` with the value 5 for the second argument. `recfcn` calls itself recursively until `x` is 1. For each `recfcn` call, the input argument `x` has a different value. The code generator produces five specializations of `recfcn`, one for each call.

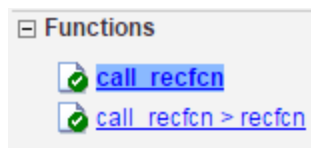


To force run-time recursion, in `call_recfcn`, in the call to `recfcn`, instruct the code generator to treat the value of the input argument `x` as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
A = ones(1,n);
x = coder.ignoreConst(5);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

, you see only one specialization.



Make the Input to the Recursive Function Variable-Size

Consider this code:

```
function z = call_mysum(A)
%#codegen
```

```

z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end

```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. To force the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```

function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end

```

Assign Output Variable Before the Recursive Call

The code generator uses compile-time recursion for this code:

```

function y = callrecursive(n)
x = 10;
y = myrecursive(x,n);
end

function y = myrecursive(x,n)
coder.inline('never')
if x > 1

```

```
        y = n + myrecursive(x-1,n-1);  
  
else  
    y = n;  
end  
end
```

To force the code generator to use run-time recursion, modify `myrecursive` so that the output `y` is assigned before the recursive call. Place the assignment `y = n` in the `if` block and the recursive call in the `else` block.

```
function y = callrecursive(n)  
x = 10;  
y = myrecursive(x,n);  
end  
  
function y = myrecursive(x,n)  
coder.inline('never')  
if x == 1  
    y = n;  
else  
    y = n + myrecursive(x-1,n-1);  
end  
end
```

More About

- “Code Generation for Recursive Functions” on page 16-18
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 46-38
- “Compile-Time Recursion Limit Reached” on page 46-33

Code Generation for MATLAB Classes

- “MATLAB Classes Definition for Code Generation” on page 17-2
- “Classes That Support Code Generation” on page 17-8
- “Generate Code for MATLAB Value Classes” on page 17-9
- “Generate Code for MATLAB Handle Classes and System Objects” on page 17-14
- “MATLAB Classes in Code Generation Reports” on page 17-17
- “Troubleshooting Code Generation Issues with MATLAB Classes” on page 17-20
- “Handle Object Limitations for Code Generation” on page 17-22
- “System Objects in MATLAB Code Generation” on page 17-25
- “Specify Objects as Inputs” on page 17-29

MATLAB Classes Definition for Code Generation

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

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an @-folder.	"Create a Single, Self-Contained Class Definition File" (MATLAB)
Restricted set of language features.	"Language Limitations" on page 17-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 17-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 17-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 17-14 "Handle Object Limitations for Code Generation" on page 17-22
Calls to base class constructor.	"Calls to Base Class Constructor" on page 17-6
Global variables containing MATLAB objects are not supported for code generation.	N/A
Inheritance from built-in MATLAB classes is not supported.	"Inheritance from Built-In MATLAB Classes Not Supported" on page 17-7

Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events

- Listeners
- Arrays of objects
- Recursive data structures
 - Linked lists
 - Trees
 - Graphs
- Overloadable operators `subsref`, `subsassign`, and `subsindex`

In MATLAB, classes can define their own versions of the `subsref`, `subsassign`, and `subsindex` methods. Code generation does not support classes that have their own definitions of these methods.

- The `empty` method

In MATLAB, classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.

- The following MATLAB handle class methods:
 - `addlistener`
 - `delete`
 - `eq`
 - `findobj`
 - `findpro`
- The `AbortSet` property attribute

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

```
codegen ClassNameA
```

- A handle class object cannot be an entry-point function input or output.
- A value class object can be an entry-point function input or output. However, if a value class object contains a handle class object, then the value class object cannot be

an entry-point function input or output. A handle class object cannot be an entry-point function input or output.

- Code generation does not support classes in matrices or structures. As a workaround, consider using cell arrays because code generation supports classes in cell arrays.
- Code generation does not support assigning an object of a value class into a nontunable property. For example, `obj.prop=v`; is invalid when `prop` is a nontunable property and `v` is an object based on a value class.
- You cannot use `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the `coder.ceval` function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- The `coder.nullcopy` function does not support MATLAB classes as inputs.
- The `coder.Constant` function does not support MATLAB classes as inputs.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you do when running your code in the MATLAB environment:

- Code generation does not support the property restriction syntax. For example, the following class definition is not allowed because it uses the property restriction syntax to restrict the types of the `Number` and `Today` properties.

```
classdef Myclass
    properties
        Number double
        Today char = date;
    end
end
```

- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of varying class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generator requires a complete assignment to each variable. Similarly, before using properties, you must explicitly define their class, size, and complexity.

- Initial values:
 - If the property does not have an explicit initial value, the code generator assumes that it is undefined at the beginning of the constructor. The code generator does not assign an empty matrix as the default.
 - If the property does not have an initial value and the code generator cannot determine that the property is assigned prior to first use, the software generates a compilation error.
 - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

For example, for a nontunable property, you can use the following assignment:

```
mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
```

You cannot use the following partial assignments:

```
mySystemObject.nonTunableProperty.fieldA = 'a';
mySystemObject.nonTunableProperty.fieldB = 'b';
```

- `coder. varsize` is not supported for class properties.
- If the initial value of a property is an object, then the property must be constant. To make a property constant, declare the `Constant` attribute in the property block. For example:


```
classdef MyClass
    properties (Constant)
        p1 = MyClass2;
    end
end
```
- MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns `true (1)`.
- If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.
- If a property is constant and its value is an object, you cannot change the value of a property of that object. For example, suppose that:

- `obj` is an object of `myClass1`.
- `myClass1` has a constant property `p1` that is an object of `myClass2`.
- `myClass2` has a property `p2`.

Code generation does not support the following code:

```
obj.p1.p2 = 1;
```

Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must come before `for`, `if`, `return`, `switch` or `while` statements.

For example, if you define a class `B` based on class `A`:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
            obj = obj@A(a,b);
        end
    end
end
```

Because the class definition for `B` uses an `if` statement before calling the base class constructor for `A`, you cannot generate code for function `callB`:

```
function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
end
```

However, you can generate code for `callB` if you define class `B` as:

```
classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end
    end
end

function [a,b] = getaandb(varargin)
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end
```

Inheritance from Built-In MATLAB Classes Not Supported

You cannot generate code for classes that inherit from built-in MATLAB classes. For example, you cannot generate code for the following class:

```
classdef myclass < double
```

Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 17-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 17-14

For more information, see:

- “Role of Classes in MATLAB” (MATLAB)
- “MATLAB Classes Definition for Code Generation” on page 17-2

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

- 1 In a writable folder, create a MATLAB value class, `Shape`. Save the code as `Shape.m`.

```

classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out = obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
end
methods(Abstract = true)
    getarea(obj);
end
methods(Static)
    function d = distanceBetweenShapes(shape1,shape2)
        xDist = abs(shape1.centerX - shape2.centerX);
        yDist = abs(shape1.centerY - shape2.centerY);
        d = sqrt(xDist^2 + yDist^2);
    end
end
end
end

```

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

```

classdef Square < Shape
% Create a Square at coordinates center X and center Y

```

```
% with sides of length of side
properties
    side;
end
methods
    function obj = Square(side,centerX,centerY)
        obj@Shape(centerX,centerY);
        obj.side = side;
    end
    function Area = getarea(obj)
        Area = obj.side^2;
    end
end
end
```

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

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

- 4 Write a function that uses this class.

```
function [TotalArea, Distance] = use_shape
    %#codegen
    s = Square(2,1,2);
    r = Rhombus(3,4,7,10);
    TotalArea = s.area + r.area;
    Distance = Shape.distanceBetweenShapes(s,r);
```

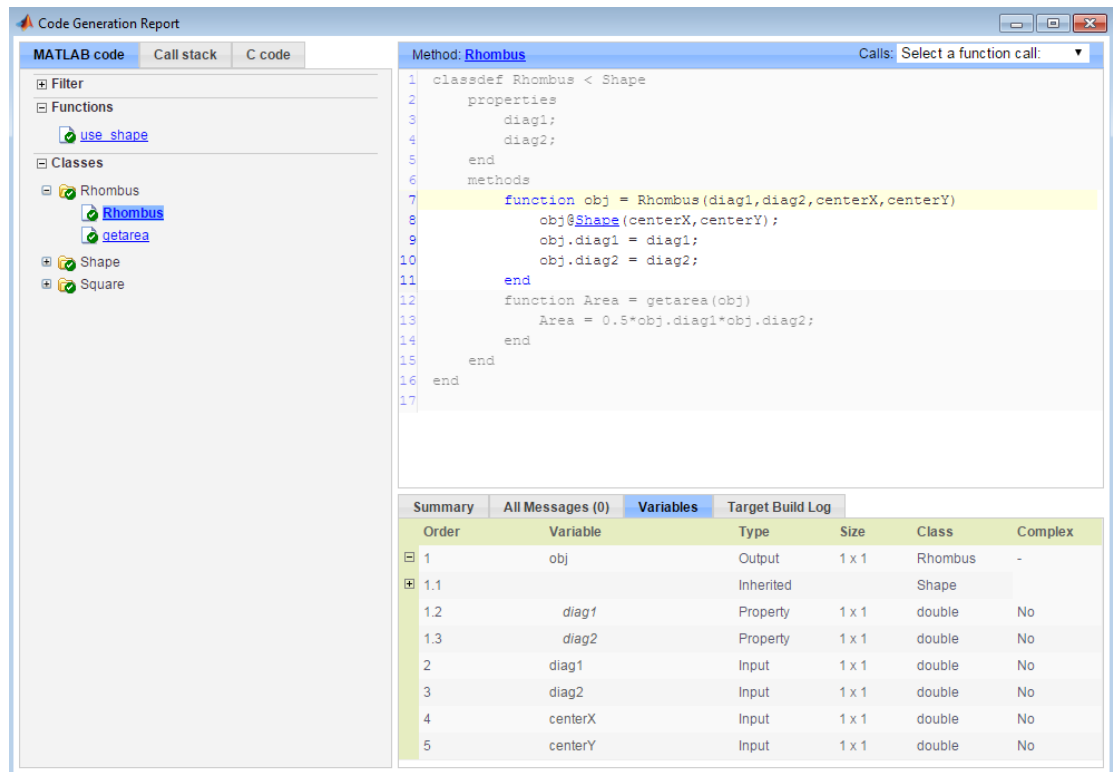
- 5 Generate a static library for use_shape and generate a code generation report.

codegen -config:lib -report use_shape

codegen generates a C static library with the default name, use_shape, and supporting files in the default folder, codegen/lib/use_shape.

- 6 Click the **View report** link.
- 7 In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the list of properties, expand the list. Within the list of properties, the list of inherited properties is collapsed. In the following report, the lists of properties and inherited properties are expanded.



The screenshot shows the Code Generation Report window with the MATLAB code tab selected. The code defines the Rhombus class, including its properties (diag1, diag2), methods (constructor, getarea), and a class constructor function. The Variables tab is also visible, showing a table of variables used in the code.

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1		Inherited		Shape	
1.2	diag1	Property	1 x 1	double	No
1.3	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

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

The **Calls** list shows that there is a call to the Rhombus constructor from use_shape and that this constructor calls the Shape constructor.

The screenshot shows the 'Code Generation Report' window with the 'MATLAB code' tab selected. On the left, a tree view shows the project structure with 'Rhombus' selected under 'Classes'. The main pane displays the MATLAB code for the Rhombus class, which inherits from Shape. A call stack is visible in the top right corner, showing a call from 'use_shape' to the 'Rhombus' constructor, which in turn calls the 'Shape' constructor. A table at the bottom provides a summary of variables and their properties.

```

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

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1		Inherited		Shape	
1.2	diag1	Property	1 x 1	double	No
1.3	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

- 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.

The screenshot shows the 'Code Generation Report' window with the 'MATLAB code' tab selected. On the left, a tree view shows the project structure with 'use_shape' under Functions and 'Rhombus', 'Shape', and 'Square' under Classes. The main area displays the MATLAB code for the 'Shape' method, including a constructor and a distance calculation function. Below the code is a table with columns for Summary, All Messages (0), Variables, and Target Build Log. The Variables table lists three variables: 'obj' (Output, 1 x 1, Rhombus, -), 'centerX' (Input, 1 x 1, double, No), and 'centerY' (Input, 1 x 1, double, No).

```
11 methods
12     function out = get.area(obj)
13         out = obj.getarea();
14     end
15     function obj = Shape(centerX,centerY)
16         obj.centerX = centerX;
17         obj.centerY = centerY;
18     end
19 end
20 methods(Abstract = true)
21     getarea(obj);
22 end
23 methods(Static)
24     function d = distanceBetweenShapes(shape1,shape2)
25         xDist = abs(shape1.centerX - shape2.centerX);
26         yDist = abs(shape1.centerY - shape2.centerY);
27         d = sqrt(xDist^2 + yDist^2);
28     end
29 end
30 end
31
```

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
2	centerX	Input	1 x 1	double	No
3	centerY	Input	1 x 1	double	No

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

- 1 In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access=protected)
        % stepImpl method is called by the step method
        function y = stepImpl(~,x)
            y = x+1;
        end
    end
end
```

- 2 Write a function that uses this System object.

```
function y = testAddOne(x)
%#codegen
    p = AddOne();
    y = p.step(x);
end
```

- 3 Generate a MEX function for this code.

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

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

- 4 Click the **View report** link.
- 5 In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.

The screenshot shows the 'Code Generation Report' window with the 'MATLAB code' tab selected. On the left, a tree view shows the project structure: 'Functions' containing 'testAddOne' and 'Classes' containing 'AddOne' with sub-items 'AddOne' and 'stepImp'. The main area displays the MATLAB code for the 'testAddOne' function:

```

1 function y = testAddOne(x)
2 %#codegen
3 p = AddOne();
4 y
5 end
    
```

A popup window titled 'Information for the selected variable:' is overlaid on the code, showing details for the variable 'p':

Size	1 x 1
Class	AddOne

Below the code, there is a 'Variables' tab in a table with the following data:

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	x	Input	1 x 1	double	No
3	p	Local	1 x 1	AddOne	-

- To view the class definition, on the **Classes** panel, click AddOne.

The screenshot shows the 'Code Generation Report' window with the following structure:

- Navigation:** MATLAB code (selected), Call stack, C code.
- Filter:** A search box for filtering content.
- Functions:** A list containing 'testAddOne'.
- Classes:** A tree view showing 'AddOne' with sub-items 'AddOne' and 'stepImpl'.
- Method View:** 'Method: AddOne' is selected, showing a dropdown for 'Calls: Select a function call:'. The code is displayed in a text area with line numbers 1 through 11.
- Summary Panel:** Includes tabs for 'Summary', 'All Messages (0)', 'Variables', and 'Target Build Log'. Below these is a table with columns: Order, Variable, Type, Size, Class, and Complex.

```

1 classdef AddOne < matlab.System
2 % ADDONE Compute an output value that increments the input by one
3
4     methods (Access=protected)
5         % stepImpl method is called by the step method
6         function y = stepImpl(~,x)
7             y = x+1;
8         end
9     end
10 end
11

```

Order	Variable	Type	Size	Class	Complex

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

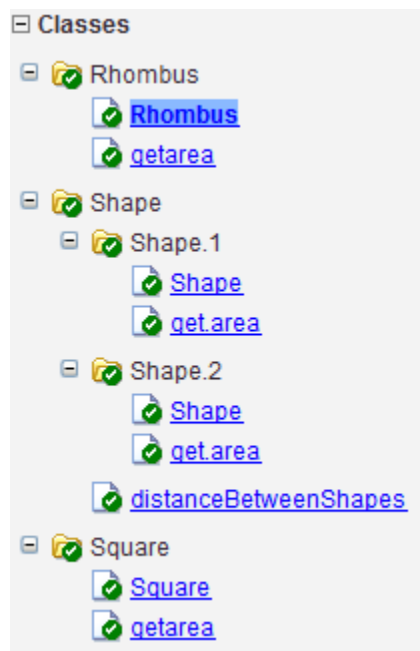
If a class has a default constructor, the report displays the constructor in italics.

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” (MATLAB).

In the Variables Tab

The report displays the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. To expand the list, click the + symbol next to the object name. Within the list of properties, the list of inherited properties is collapsed. To expand the list of inherited properties, click the + symbol next to `Inherited`.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a MATLAB object

with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

Troubleshooting Code Generation Issues with MATLAB Classes

Class *cClass* does not have a property with name *name*

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
    properties
        myprop
    end
    methods
        function this = MyClass
            this.myprop = MyClass2;
        end
        function y = mymethod(this)
            y = this.myprop;
        end
    end
end
```

```
classdef MyClass2 < handle
    properties
        aa
    end
end
```

You cannot generate code for function `foo`.

```
function foo
```

```
h = MyClass;
```

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

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.

Solution

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo  
h = MyClass;  
  
b=h.mymethod();  
b.aa=12;
```

Handle Object Limitations for Code Generation

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

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

- “A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop” on page 17-22
- “A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object” on page 17-22

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

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

```
classdef mycls < handle
    properties
        prop
    end
end

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

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

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

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

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

```
classdef mycls < handle
    properties
        prop
    end
end

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

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

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

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

```
function usehandle3(x)
```

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

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

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

doinit(myobj);

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

System Objects in MATLAB Code Generation

In this section...

“Usage Rules and Limitations for System Objects for Generating Code” on page 17-25

“System Objects in codegen” on page 17-28

“System Objects in the MATLAB Function Block” on page 17-28

“System Objects in the MATLAB System Block” on page 17-28

“System Objects and MATLAB Compiler Software” on page 17-28

You can generate C/C++ code in MATLAB from your system that contains System objects by using MATLAB Coder. You can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms.

Usage Rules and Limitations for System Objects for Generating Code

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

Object Construction and Initialization

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

Inputs and Outputs

- System objects accept a maximum of 1024 inputs. A maximum of eight dimensions per input is supported.
- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that support for variable-size is enabled. Code generation support for variable-size data also requires that variable-

size support is enabled. By default in MATLAB, support for variable-size data is enabled.

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

Properties

- In MATLAB System blocks, you cannot use variable-size for discrete state properties of System objects. Private properties can be variable-size.
- Objects cannot be used as default values for properties.
- You can only assign values to nontunable properties once, including the assignment in the constructor.
- Nontunable property values must be constant.
- If a tunable property has dependant data type properties, you can set tunable properties only at construction time or after the object is locked.
- For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

Global Variables

- Global variables are allowed in a System object, unless you are using that System object in Simulink via the MATLAB System block. To avoid syncing global variables between a MEX file and the workspace, use a `coder` configuration object. For example:

```
f = coder.MEXConfig;  
f.GlobalSyncMethod = 'NoSync'
```


Then, include `'-config f'` in your `codegen` command.

Methods

- Code generation support is available only for these System object methods:
 - `get`
 - `getNumInputs`
 - `getNumOutputs`
 - `isDone` (for sources only)
 - `isLocked`
 - `release`
 - `reset`
 - `set` (for tunable properties)
 - `step`
- For System objects that you define, code generation support is available only for these methods:
 - `getDiscreteStateImpl`
 - `getNumInputsImpl`
 - `getNumOutputsImpl`
 - `infoImpl`
 - `isDoneImpl`
 - `isInputDirectFeedthroughImpl`
 - `outputImpl`
 - `processTunedPropertiesImpl`
 - `releaseImpl` — Code is not generated automatically for this method. To release an object, you must explicitly call the `release` method in your code.
 - `resetImpl`
 - `setupImpl`
 - `stepImpl`
 - `updateImpl`
 - `validateInputsImpl`

- `validatePropertiesImpl`

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” (MATLAB Coder) and “MATLAB Classes” (MATLAB Coder) for more information.

Note: Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “What Is a MATLAB Function Block?” (Simulink).

System Objects in the MATLAB System Block

Using the MATLAB System block, you can include in a Simulink model individual System objects that you create with a class definition file. The model can then generate embeddable code. For more information, see “MATLAB System Block” (Simulink).

System Objects and MATLAB Compiler Software

MATLAB Compiler™ software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

Specify Objects as Inputs

When you accelerate code by using `fiaccl`, to specify the type of an input that is a value class object, you can provide an example object with the `-args` option.

- 1 Define the value class. For example, define a class `myRectangle`.

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

- 2 Define a function that takes an object of the value class as an input. For example:

```
function z = getarea(r)
    %#codegen
    z = calcarea(r);
end
```

- 3 Define an object of the class.

```
rect_obj = myRectangle(fi(4),fi(5))

rect_obj =

    myRectangle with properties:

        length: [1×1 embedded.fi]
        width: [1×1 embedded.fi]
```

- 4 Pass the example object to `fiaccl` by using the `-args` option.

```
fiaccl getarea -args {rect_obj} -report
```

In the report, you see that `r` has the same properties, `length` and `width`, as the example object `rect_obj`.

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
⊕ 1	<code>z</code>	Output	1 x 1	<code>embedded.fi</code>	No	Signed	32	24
⊖ 2	<code>r</code>	Input	1 x 1	<code>myRectangle</code>	-	-	-	-
⊕ 2.1	<code>length</code>	Property	1 x 1	<code>embedded.fi</code>	No	Signed	16	12
⊕ 2.2	<code>width</code>	Property	1 x 1	<code>embedded.fi</code>	No	Signed	16	12

Instead of providing an example object, you can create a type for an object of the value class and provide the type with the `-args` option.

- 1 Define an object of the class:

```
rect_obj = myRectangle(fi(4),fi(5))
```

```
rect_obj =
```

```
myRectangle with properties:
```

```
length: [1x1 embedded.fi]
```

```
width: [1x1 embedded.fi]
```

- 2 To create a type for an object of `myRectangle` that has the same property types as `rect_obj`, use `coder.typeof`. `coder.typeof` creates a `coder.ClassType` object that defines a type for a class.

```
t= coder.typeof(rect_obj)
```

```
t =
```

```
coder.ClassType
```

```
1x1 myRectangle
```

```
length: 1x1 embedded.fi
```

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

```
Signedness: Signed
```

```
WordLength: 16
```

```
FractionLength: 12
```

```
width : 1x1 embedded.fi
```

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

```
Signedness: Signed
WordLength: 16
FractionLength: 12
```

- 3 Pass the type to `fiaccel` by using the `-args` option.

```
fiaccel getarea -args {t} -report
```

After you create the type, you can change the types of the properties.

```
t.Properties.length = coder.typeof(fi(0,1,32,29))
t.Properties.width = coder.typeof(fi(0,1,32,29))
```

You can also add or delete properties. For example, to add a property `newprop`:

```
t.Properties.newprop = coder.typeof(int16(1))
```

Consistency Between `coder.ClassType` Object and Class Definition File

When you accelerate code, the properties of the `coder.ClassType` object that you pass to `fiaccel` must be consistent with the properties in the class definition file. If the class definition file has properties that your code does not use, the `coder.ClassType` object does not have to include those properties. `fiaccel` removes properties that you do not use.

Limitations for Using Objects as Entry-Point Function Inputs

Entry-point function inputs that are objects have these limitations:

- An object that is an entry-point function input must be an object of a value class. Objects of handle classes cannot be entry-point function inputs. Therefore, a value class that contains a handle class cannot be an entry-point function input.
- `coder.Constant` does not support objects.
- An object cannot be an input or output argument for an extrinsic function.
- An object cannot be a global variable.

See Also

`coder.typeof`

More About

- “Specify Properties of Entry-Point Function Inputs” on page 29-2
- “MATLAB Classes Definition for Code Generation” on page 17-2

Defining Data for Code Generation

- “Data Definition for Code Generation” on page 18-2
- “Code Generation for Complex Data” on page 18-4
- “Code Generation for Character Arrays” on page 18-9
- “Array Size Restrictions for Code Generation” on page 18-10
- “Code Generation for Constants in Structures and Arrays” on page 18-11

Data Definition for Code Generation

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

Data	What's Different	More Information
Arrays	Maximum number of elements is restricted	“Array Size Restrictions for Code Generation” on page 18-10
Complex numbers	<ul style="list-style-type: none"> • Complexity of variables must be set at time of assignment and before first use • Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero <hr/> <p>Note: Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</p>	“Code Generation for Complex Data” on page 18-4
Characters	Restricted to 8 bits of precision	“Code Generation for Character Arrays” on page 18-9
Enumerated data	<ul style="list-style-type: none"> • Supports integer-based enumerated types only • Restricted use in <code>switch</code> statements and <code>for</code>-loops 	“Enumerations”
Function handles	<ul style="list-style-type: none"> • Using the same bound variable to reference different function 	“Function Handles”

Data	What's Different	More Information
	<p data-bbox="647 302 891 361">handles can cause a compile-time error.</p> <ul data-bbox="609 374 911 604" style="list-style-type: none"><li data-bbox="609 374 911 496">• Cannot pass function handles to or from primary or extrinsic functions<li data-bbox="609 513 911 604">• Cannot view function handles from the debugger	

Code Generation for Complex Data

In this section...

“Restrictions When Defining Complex Variables” on page 18-4

“Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 18-4

“Results of Expressions That Have Complex Operands” on page 18-8

Restrictions When Defining Complex Variables

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

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

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

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

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

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

Code Generation for Complex Data with Zero-Valued Imaginary Parts

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

- In some cases, results from functions that sort complex data by absolute value can differ from the MATLAB results. See “Functions That Sort Complex Values by Absolute Value” on page 18-5.
- For functions that require that complex inputs are sorted by absolute value, complex inputs with zero-valued imaginary parts must be sorted by absolute value. These functions include `ismember`, `union`, `intersect`, `setdiff`, and `setxor`.

Functions That Sort Complex Values by Absolute Value

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

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

- 1 Write this function:

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

- 2 Call `mysort` in MATLAB.

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

ans =

    -2    -1     0     1     2
```

- 3 Generate a MEX function for complex inputs.

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

- 4 Call the MEX Function with real inputs.

```
mysort_mex(A)

ans =
```

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

You generated the MEX function for complex inputs, therefore, it treats the real inputs as complex numbers with zero-valued imaginary parts. It sorts the numbers by the absolute values of the complex numbers. Because the imaginary parts are zero, the MEX function returns the results to the MATLAB workspace as real numbers. See “Inputs and Outputs for MEX Functions Generated for Complex Arguments” on page 18-7.

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

- 1 Write this function:

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

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

- 2 Call `myfun` in MATLAB.

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

```
ans =
```

```
12.5777
 2.0000
-3.5777
```

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

- 3 Generate a MEX function for complex inputs.

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

- 4 Call the MEX function.

```
myfun_mex(A)
```

```
ans =
```

```
12.5777
-3.5777
 2.0000
```

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

Inputs and Outputs for MEX Functions Generated for Complex Arguments

For MEX functions created by `fiaccl`:

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

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

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

```
codegen foo -report
```

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

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

```
z = foo_mex

ans =

    1
```

Results of Expressions That Have Complex Operands

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

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

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

Exceptions to this behavior are:

- When the imaginary parts of complex results are zero, MEX functions return the results to the MATLAB workspace as real values. See “Inputs and Outputs for MEX Functions Generated for Complex Arguments” on page 18-7.
- When the imaginary part of the argument is zero, complex arguments to extrinsic functions are real.

```
function y = foo()  
    coder.extrinsic('sqrt')  
    x = 1 + 0i; % x is complex  
    y = sqrt(x); % x is real, y is real  
end
```

- Functions that take complex arguments but produce real results return real values.

```
y = real(x); % y is the real part of the complex number x.  
y = imag(x); % y is the real-valued imaginary part of x.  
y = isreal(x); % y is false (0) for a complex number x.
```

- Functions that take real arguments but produce complex results return complex values.

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

Code Generation for Character Arrays

The code generator translates the 16-bit Unicode encoding of a character in MATLAB to an 8-bit encoding that the locale setting determines. The code generator does not support characters that require more than 1 byte in MATLAB. This restriction applies to character arrays that are passed between MATLAB and the code generator. For example, the restriction applies to entry-point function inputs and to outputs from extrinsic calls. For code generation, some MATLAB functions accept only 7-bit ASCII characters. See “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2.

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

More About

- “Locale Settings for MATLAB Process” (MATLAB)

Array Size Restrictions for Code Generation

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

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

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

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

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

These restrictions apply even on a 64-bit platform.

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

See Also

- “Variable-Size Data”

Code Generation for Constants in Structures and Arrays

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

Fields or elements are assigned inside control constructs

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

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

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

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

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

Constants are assigned to array elements using non-scalar indexing

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

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

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

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

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

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

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

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

Defining Functions for Code Generation

- “Code Generation for Variable Length Argument Lists” on page 19-2
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3
- “Code Generation for Anonymous Functions” on page 19-6
- “Code Generation for Nested Functions” on page 19-7

Code Generation for Variable Length Argument Lists

When you use `varargin` and `varargout` for code generation, there are these restrictions:

- In the Fixed-Point Converter app, you cannot use an entry-point function that uses `varargout`.
- You cannot use `varargin` or `varargout` in the function definition for a top-level function in a MATLAB Function block or in a Stateflow chart that uses MATLAB as the action language.
- If you use `varargin` to define an argument to an entry-point function, the code generator produces the function with a fixed number of arguments. This fixed number of arguments is based on the number of arguments that you specify when you generate code.
- You cannot write to `varargin`. If you want to write to input arguments, copy the values into a local variable.
- To index into `varargin` and `varargout`, use curly braces `{}`, not parentheses `()`.
- The code generator must be able to determine the value of the index into `varargin` or `varargout`.

More About

- “Nonconstant Index into `varargin` or `varargout` in a for-Loop” on page 46-45
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3

Specify Number of Entry-Point Function Input or Output Arguments to Generate

You can control the number of input or output arguments in a generated entry-point function. From one MATLAB function, you can generate entry-point functions that have different signatures.

Control Number of Input Arguments

If your entry-point function uses `varargin`, specify the properties for the arguments that you want in the generated function.

Consider this function:

```
function [x, y] = myops(varargin)
%#codegen
if (nargin > 1)
    x = varargin{1} + varargin{2};
    y = varargin{1} * varargin{2};
else
    x = varargin{1};
    y = -varargin{1};
end
```

To generate a function that takes only one argument, provide one argument with `-args`.

```
fiaccel myops -args {fi(3)} -report
```

You can also control the number of input arguments when the MATLAB function does not use `varargin`.

Consider this function:

```
function [x, y] = myops(a,b)
%#codegen
if (nargin > 1)
    x = a + b;
    y = a * b;
else
    x = a;
    y = -a;
end
```

To generate a function that takes only one argument, provide one argument with `-args`.

```
fiaccel myops -args {fi(3)} -report
```

Control the Number of Output Arguments

When you use `fiaccel`, you can specify the number of output arguments by using the `-nargout` option.

Consider this function:

```
function [x, y] = myops(a,b)
%#codegen
if (nargin > 1)
    x = a + b;
    y = a * b;
else
    x = a;
    y = -a;
end
```

Generate a function that has one output argument.

```
fiaccel myops -args {fi(2) fi(3)} -nargout 1 -report
```

You can also use `-nargout` to specify the number of arguments for an entry-point function that uses `varargout`.

Rewrite `myops` to use `varargout`.

```
function varargout = myops(a,b)
%#codegen
if (nargin > 1)
    varargout{1} = a + b;
    varargout{2} = a * b;
else
    varargout{1} = a;
    varargout{2} = -a;
end
```

Generate code for one output argument.

```
fiaccel myops -args {fi(2) fi(3)} -nargout 1 -report
```

More About

- “Code Generation for Variable Length Argument Lists” on page 19-2
- “Specify Properties of Entry-Point Function Inputs” on page 29-2

Code Generation for Anonymous Functions

You can use anonymous functions in MATLAB code intended for code generation. For example, you can generate code for the following MATLAB code that defines an anonymous function that finds the square of a number.

```
sqr = @(x) x.^2;  
a = sqr(5);
```

Anonymous functions are useful for creating a function handle to pass to a MATLAB function that evaluates an expression over a range of values. For example, this MATLAB code uses an anonymous function to create the input to the `fzero` function:

```
b = 2;  
c = 3.5;  
x = fzero(@(x) x^3 + b*x + c,0);
```

Anonymous Function Limitations for Code Generation

Anonymous functions have the code generation limitations of value classes and cell arrays.

More About

- “MATLAB Classes Definition for Code Generation” on page 17-2
- “Cell Array Limitations for Code Generation” on page 28-9
- “Parameterizing Functions” (MATLAB)

Code Generation for Nested Functions

You can generate code for MATLAB functions that contain nested functions. For example, you can generate code for the function `parent_fun`, which contains the nested function `child_fun`.

```
function parent_fun
x = 5;
child_fun

    function child_fun
        x = x + 1;
    end

end
```

Nested Function Limitations for Code Generation

When you generate code for nested functions, you must adhere to the code generation restrictions for value classes, cell arrays, and handle classes. You must also adhere to these restrictions:

- If the parent function declares a persistent variable, it must assign the persistent variable before it calls a nested function that uses the persistent variable.
- A nested recursive function cannot refer to a variable that the parent function uses.
- If a nested function refers to a structure variable, you must define the structure by using `struct`.
- If a nested function uses a variable defined by the parent function, you cannot use `coder.ysize` with the variable in either the parent or the nested function.

More About

- “MATLAB Classes Definition for Code Generation” on page 17-2
- “Handle Object Limitations for Code Generation” on page 17-22
- “Cell Array Limitations for Code Generation” on page 28-9
- “Code Generation for Recursive Functions” on page 16-18

Defining MATLAB Variables for C/C++ Code Generation

- “Variables Definition for Code Generation” on page 20-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 20-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 20-7
- “Reassignment of Variable Properties” on page 20-9
- “Reuse the Same Variable with Different Properties” on page 20-10
- “Avoid Overflows in for-Loops” on page 20-14
- “Supported Variable Types” on page 20-16

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 20-3.

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

In this section...

“Define Variables By Assignment Before Using Them” on page 20-3

“Use Caution When Reassigning Variables” on page 20-5

“Use Type Cast Operators in Variable Definitions” on page 20-5

“Define Matrices Before Assigning Indexed Variables” on page 20-6

Define Variables By Assignment Before Using Them

For C/C++ code generation, you should explicitly and unambiguously define the class, size, and complexity of variables before using them in operations or returning them as outputs. Define variables by assignment, but note that the assignment copies not only the value, but also the size, class, and complexity represented by that value to the new variable. For example:

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

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

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

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in “Eliminate Redundant Copies of Variables in Generated Code” on page 20-7.

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

Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

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

Here, x is assigned only if $c > 0$ and used only when $c > 0$. This code works in MATLAB, but generates a compilation error during code generation because it detects that x is undefined on some execution paths (when $c \leq 0$),

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

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

Defining Fields in a Structure

Consider the following MATLAB code:

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

```

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

```

Here, the first part of the `if` statement uses only the field `a`, and the `else` clause uses fields `a` and `b`. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation” on page 25-2.

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

```

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

```

Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 20-9.

Use Type Cast Operators in Variable Definitions

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

```

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

```

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

Define Matrices Before Assigning Indexed Variables

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

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

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

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 27-30.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

“When Redundant Copies Occur” on page 20-7

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 20-7

“Defining Uninitialized Variables” on page 20-8

When Redundant Copies Occur

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

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 20-7.

How to Eliminate Redundant Copies by Defining Uninitialized Variables

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

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

What happens if you access uninitialized data?

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

Defining Uninitialized Variables

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

```
function X = fcn %#codegen

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

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

```
function X = fcn2 %#codegen

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

Reassignment of Variable Properties

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

Dynamically sized variables

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

Variables reused in the code for different purposes

You can reassign the type (class, size, and complexity) of a variable after the initial assignment if each occurrence of the variable can have only one type. In this case, the variable is renamed in the generated code to create multiple independent variables. For more information, see “Reuse the Same Variable with Different Properties” on page 20-10.

Reuse the Same Variable with Different Properties

In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 20-10

“When You Cannot Reuse Variables” on page 20-10

“Limitations of Variable Reuse” on page 20-13

When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if the code generator can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report (see “Use Fixed-Point Code Generation Reports” on page 14-36).

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

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

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

When You Cannot Reuse Variables

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

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

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

Variable Reuse in an if Statement

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

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

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

- 2 Compile `example1`.

For example, to generate a MEX function, enter:

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

Note: `codegen` requires a MATLAB Coder license.

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

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

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

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

Information for the selected variable:	
Size	1 x 1
Complex	No
Class	double

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

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

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

Information for the selected variable:	
Size	:25
Complex	No
Class	double

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

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

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

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

- 8 In the list of variables, place your pointer over `t>2`

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

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

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

Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generator detects that a loop variable might overflow on the last iteration of the `for`-loop, it reports an error.

To avoid this error, use the workarounds provided in the following table.

Loop conditions causing the error	Workaround
<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 	Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace: <pre>N=intmax('int16') for k=N-10:N</pre> with: <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 	Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace: <pre>N=intmin('int32') for k=N+10:-1:N</pre> with: <pre>for k=10:-1:1</pre>
<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>	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: <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N % Loop body end to M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N) % Loop body</pre>

Loop conditions causing the error	Workaround
<ul style="list-style-type: none">• The loop counter increments or decrements by a value not equal to 1• On last loop iteration, the loop variable value is not equal to the end value	end Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.
Note: The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.	

Supported Variable Types

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

Type	Description
char	Character array
complex	Complex data. Cast function takes real and imaginary components
double	Double-precision floating point
int8, int16, int32, int64	Signed integer
logical	Boolean true or false
single	Single-precision floating point
struct	Structure
uint8, uint16, uint32, uint64	Unsigned integer
Fixed-point	See “Fixed-Point Data Types” on page 1-2.

Design Considerations for C/C++ Code Generation

- “When to Generate Code from MATLAB Algorithms” on page 21-2
- “Which Code Generation Feature to Use” on page 21-4
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 21-6
- “MATLAB Code Design Considerations for Code Generation” on page 21-7
- “Differences Between Generated Code and MATLAB Code” on page 21-9
- “Potential Differences Reporting” on page 21-14
- “Potential Differences Messages” on page 21-16
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 21-21

When to Generate Code from MATLAB Algorithms

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

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

When Not to Generate Code from MATLAB Algorithms

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

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

To:	Use:
Interface C code with MATLAB	MATLAB <code>mex</code> function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	<code>codegen</code> function	MATLAB Coder	Try this in “MEX Function Generation at the Command Line” (MATLAB Coder).
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder app	MATLAB Coder	Try this in “C Code Generation Using the MATLAB Coder App” (MATLAB Coder).
	<code>codegen</code> function	MATLAB Coder	Try this in “C Code Generation at the Command Line” (MATLAB Coder).
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder app	MATLAB Coder	See “Accelerate MATLAB Algorithms” (MATLAB Coder).
	<code>codegen</code> function	MATLAB Coder	
Integrate MATLAB code into Simulink	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code” (Simulink).
Speed up fixed-point MATLAB code	<code>fiaccel</code> function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB” on page 14-3.
Integrate custom C code into MATLAB and generate efficient, readable code	<code>codegen</code> function	MATLAB Coder	Learn more in “Specify External File Locations” (MATLAB Coder).
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/

To...	Use...	Required Product	To Explore Further...
			products/ slhdlcoder.

Prerequisites for C/C++ Code Generation from MATLAB

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

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

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

- Data types

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

- Array sizing

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

- Memory

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

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

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

- Speed

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

To improve the speed of the generated code:

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

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

See Also

- “Data Definition Basics”
- “Code Generation for Variable-Size Arrays” on page 27-2
- “Control Run-Time Checks” on page 14-64

Differences Between Generated Code and MATLAB Code

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

Here are some of the differences:

- “Character Size” on page 21-9
- “Order of Evaluation in Expressions” on page 21-9
- “Termination Behavior” on page 21-11
- “Size of Variable-Size N-D Arrays” on page 21-11
- “Size of Empty Arrays” on page 21-11
- “Size of Empty Array That Results from Deleting Elements of an Array” on page 21-11
- “Floating-Point Numerical Results” on page 21-12
- “NaN and Infinity Patterns” on page 21-13
- “Code Generation Target” on page 21-13
- “MATLAB Class Initial Values” on page 21-13
- “Variable-Size Data” on page 21-13
- “Complex Numbers” on page 21-13

When you run your program, run-time error checks can detect some of these differences. To help you identify and address differences before you deploy code, the code generator reports a subset of the differences as potential differences.

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Character Arrays” on page 18-9.

Order of Evaluation in Expressions

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

generated code may produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that:

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

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

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

- Rewrite

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

as

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

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

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

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

as

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

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

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

as

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

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

Termination Behavior

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

Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code, but always returns `[4 2]` in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 27-26.

Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. See “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 27-27.

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

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

Case	Example Code	Size of Empty Array in MATLAB	Size of Empty Array in Generated Code
Delete all elements of an m-by-n array by using the colon operator (:).	<pre>coder.varsize('X',[4,4],[1,1]); X = zeros(2); X(:) = [];</pre>	0-by-0	1-by-0
Delete all elements of a row vector by	<pre>coder.varsize('X',[1,4],[0,1]); X = zeros(1,4); X(:) = [];</pre>	0-by-0	1-by-0

Case	Example Code	Size of Empty Array in MATLAB	Size of Empty Array in Generated Code
using the colon operator (:).			
Delete all elements of a column vector by using the colon operator (:).	<pre>coder.varsize('X',[4,1],[1,0]); X = zeros(4,1); X(:) = [];</pre>	0-by-0	0-by-1
Delete all elements of a column vector by deleting one element at a time.	<pre>coder.varsize('X',[4,1],[1,0]); X = zeros(4,1); for i = 1:4 X(i) = []; end</pre>	1-by-0	0-by-1

Floating-Point Numerical Results

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

When computer hardware uses extended precision registers

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

For certain advanced library functions

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

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

For implementation of BLAS library functions

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

NaN and Infinity Patterns

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

Code Generation Target

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

MATLAB Class Initial Values

Before code generation, at class loading time, MATLAB computes class initial values. The code generator uses the value that MATLAB computes. It does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generator does not execute this function. If the function modifies a global state, for example, a persistent variable, code generator might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation” on page 17-4.

Variable-Size Data

See “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 27-24.

Complex Numbers

See “Code Generation for Complex Data” on page 18-4.

More About

- “Potential Differences Reporting” on page 21-14
- “Potential Differences Messages” on page 21-16

Potential Differences Reporting

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

Addressing Potential Differences Messages

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

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

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

If your analysis or testing confirms the reported difference, consider modifying your code. Some potential differences messages provide a workaround. For additional information about some of the potential differences messages, see “Potential Differences Messages” on page 21-16. Even if you modify your code to prevent a difference from occurring at run time, the code generator might still report the potential difference.

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

Disabling and Enabling Potential Differences Reporting

By default, potential differences reporting is enabled for code acceleration with `fiaccel`. To disable it, in a code acceleration configuration object, set `ReportPotentialDifferences` to `false`.

More About

- “Potential Differences Messages” on page 21-16
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 27-24
- “Differences Between Generated Code and MATLAB Code” on page 21-9

Potential Differences Messages

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

Here are some of the potential differences messages:

- “Automatic Dimension Incompatibility” on page 21-16
- “mtimes No Dynamic Scalar Expansion” on page 21-17
- “Matrix-Matrix Indexing” on page 21-17
- “Vector-Vector Indexing” on page 21-18
- “Size Mismatch” on page 21-18

Automatic Dimension Incompatibility

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

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

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

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

mtimes No Dynamic Scalar Expansion

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

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

Matrix-Matrix Indexing

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

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

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

To avoid this issue, force your data to be a vector by using the colon operator for indexing. For example, suppose that your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing.

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
```

```
D = A(B);  
end  
...
```

The indexing in the first branch specifies that `C` and `B(:)` are compile-time vectors. Therefore, the code generator applies the indexing rule for indexing one vector with another vector. The orientation of the result is the orientation of the data vector, `C`.

Vector-Vector Indexing

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

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

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

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

Size Mismatch

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

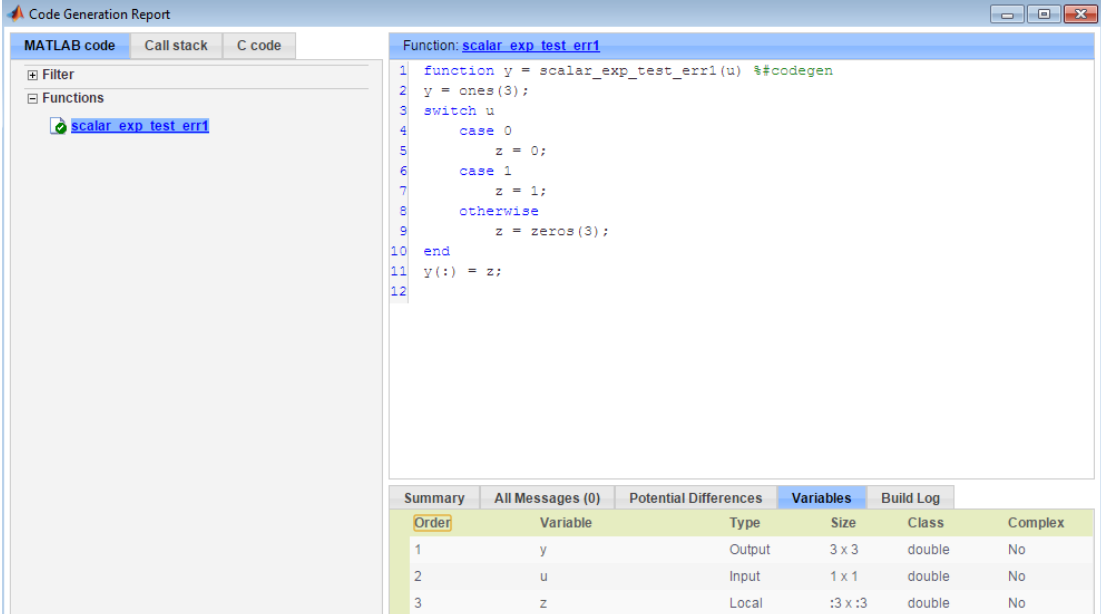
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. If one operand is a scalar and the other is not, scalar expansion applies the scalar to every element of the other operand.

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

Consider this function:

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

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



The screenshot shows the Code Generation Report window for the function `scalar_exp_test_err1`. The MATLAB code is displayed as follows:

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

Below the code, a table summarizes the variables used in the function:

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

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

```
scalar_exp_test_err1_mex(0)
```

```
Sizes mismatch: 9 ~= 1.
```

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

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

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

More About

- “Potential Differences Reporting” on page 21-14
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 27-24
- “Differences Between Generated Code and MATLAB Code” on page 21-9

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

MATLAB Features That Code Generation Supports

Code generation from MATLAB code supports the following language features:

- n-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 18-10)
- matrix operations, including deletion of rows and columns
- variable-sized data (see “Code Generation for Variable-Size Arrays” on page 27-2)
- subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 27-30)
- complex numbers (see “Code Generation for Complex Data” on page 18-4)
- numeric classes (see “Supported Variable Types” on page 20-16)
- double-precision, single-precision, and integer math
- fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB” on page 14-3)
- program control statements `if`, `switch`, `for`, `while`, and `break`
- arithmetic, relational, and logical operators
- local functions
- persistent variables
- global variables
- structures (see “Structure Definition for Code Generation” on page 25-2)
- cell arrays (see “Cell Arrays”)
- characters (see “Code Generation for Character Arrays” on page 18-9)
- function handles (see “Function Handle Limitations for Code Generation” on page 23-2)
- anonymous functions (see “Code Generation for Anonymous Functions” on page 19-6)
- recursive functions
- nested functions
- variable length input and output argument lists

- subset of MATLAB toolbox functions (see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2)
- subset of functions and System objects in several toolboxes (see “Functions and Objects Supported for C/C++ Code Generation — Category List” on page 26-67)
- MATLAB classes (see “MATLAB Classes Definition for Code Generation” on page 17-2)
- function calls (see “Resolution of Function Calls for Code Generation” on page 16-2)

MATLAB Language Features That Code Generation Does Not Support

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

- implicit expansion

Code generation does not support implicit expansion of arrays with compatible sizes during execution of element-wise operations or functions. If your MATLAB code relies on implicit expansion, code generation results in a size-mismatch error. For fixed-size arrays, the error occurs at compile time. For variable-size arrays, the error occurs at run time. For more information about implicit expansion, see “Compatible Array Sizes for Basic Operations” (MATLAB).

- string arrays
- categorical arrays
- date and time arrays
- Java
- Map containers
- sparse matrices
- tables
- time series objects
- try/catch statements

This list is not exhaustive. To see if a feature is supported for code generation, see “MATLAB Features That Code Generation Supports” on page 21-21.

Code Generation for Enumerated Data

- “Code Generation for Enumerations” on page 22-2
- “Customize Enumerated Types in Generated Code” on page 22-6

Code Generation for Enumerations

Enumerations represent a fixed set of named values. Enumerations help make your MATLAB code more readable.

For code generation, when you use enumerations, adhere to these restrictions:

- “Define Enumerations for Code Generation” on page 22-2
- “Use Allowed Operations on Enumerations” on page 22-4
- “Use MATLAB Toolbox Functions That Support Enumerations” on page 22-4

Define Enumerations for Code Generation

For code generation, the enumeration class must derive from one of these base types: `int8`, `uint8`, `int16`, `uint16`, or `int32`. For example:

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

If you use MATLAB Coder to generate C/C++ code, you can use the base type to control the size of an enumerated type in the generated code. You can:

- Represent an enumerated type as a fixed-size integer that is portable to different targets.
- Reduce memory usage.
- Interface with legacy code.
- Match company standards.

The base type determines the representation of the enumerated type in generated C/C++ code.

If the base type is `int32`, the code generator produces a C enumerated type. Consider the following MATLAB enumerated type definition:

```

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

```

This enumerated type definition results in the following C code:

```

enum LEDcolor
{
    GREEN = 1,
    RED
};

typedef enum LEDcolor LEDcolor;

```

For built-in integer base types other than `int32`, the code generator produces a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. Consider the following MATLAB enumerated type definition:

```

classdef LEDcolor < int16
    enumeration
        GREEN(1),
        RED(2)
    end
end

```

This enumerated type definition results in the following C code:

```

typedef short LEDcolor;
#define GREEN ((LEDcolor)1)
#define RED ((LEDcolor)2)

```

The C type in the `typedef` statement depends on:

- The integer sizes defined for the production hardware in the hardware implementation object or the project settings. See `coder.HardwareImplementation` in the MATLAB Coder documentation.
- The setting that determines the use of built-in C types or MathWorks typedefs in the generated code. See “Specify Data Types Used in Generated Code” (MATLAB Coder) and “How MATLAB Coder Infers C/C++ Data Types” (MATLAB Coder) in the MATLAB Coder documentation.

Use Allowed Operations on Enumerations

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

Operation	Example
assignment operator, =	<code>xon = LEDcolor.GREEN</code> <code>xoff = LEDcolor.RED</code>
relational operators, < > <= >= == ~= You cannot use == or ~= to test equality between an enumeration member and a character array or cell array of character arrays.	<code>xon == xoff</code>
cast operation	<code>double(LEDcolor.RED)</code>
indexing operation	<code>m = [1 2]</code> <code>n = LEDcolor(m)</code> <code>p = n(LEDcolor.GREEN)</code>
control flow statements: if, switch, while	<code>if state == sysMode.ON</code> <code>led = LEDcolor.GREEN;</code> <code>else</code> <code>led = LEDcolor.RED;</code> <code>end</code>

Use MATLAB Toolbox Functions That Support Enumerations

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

- `cast`
- `cat`
- `circshift`
- `fliplr`
- `flipud`
- `histc`
- `intersect`
- `ipermute`

- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismember`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `setdiff`
- `setxor`
- `shiftdim`
- `sort`
- `sortrows`
- `squeeze`
- `union`
- `unique`

More About

- “Customize Enumerated Types in Generated Code” on page 22-6

Customize Enumerated Types in Generated Code

For code generation, to customize an enumeration, in the static methods section of the class definition, include customized versions of the methods listed in this table.

Method	Description	Default Value Returned or Specified	When to Use
<code>getDefaultValue</code>	Returns the default enumerated value.	First value in the enumeration class definition.	For a default value that is different than the first enumeration value, provide a <code>getDefaultValue</code> method that returns the default value that you want. See “Specify a Default Enumeration Value” on page 22-7.
<code>getHeaderFile</code>	Specifies the file that defines an externally defined enumerated type.	' '	To use an externally defined enumerated type, provide a <code>getHeaderFile</code> method that returns the path to the header file that defines the type. In this case, the code generator does not produce the class definition. See “Specify a Header File” on page 22-8
<code>addClassNameToEnumNames</code>	Specifies whether the class name becomes a prefix in the generated code.	<code>false</code> — prefix is not used.	If you want the class name to become a prefix in the generated code, set the

Method	Description	Default Value Returned or Specified	When to Use
			return value of the <code>addClassNameToEnumNames</code> method to <code>true</code> . See “Include Class Name Prefix in Generated Enumerated Type Value Names” on page 22-8.

Specify a Default Enumeration Value

If the value of a variable that is cast to an enumerated type does not match one of the enumerated type values:

- Generated MEX reports an error.
- Generated C/C++ code replaces the value of the variable with the enumerated type default value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumeration class definition. To specify a different default value, add your own `getDefaultValue` method to the methods section. In this example, the first enumeration member value is `LEDcolor.GREEN`, but the `getDefaultValue` method returns `LEDcolor.RED`:

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

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

Specify a Header File

To specify that an enumerated type is defined in an external file, provide a customized `getHeaderFile` method. This example specifies that `LEDcolor` is defined in the external file `my_LEDcolor.h`.

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

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

You must provide `my_LEDcolor.h`. For example:

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

Include Class Name Prefix in Generated Enumerated Type Value Names

By default, the generated enumerated type value name does not include the class name prefix. For example:

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

typedef enum LEDcolor LEDcolor;
```

To include the class name prefix, provide an `addClassNameToEnumNames` method that returns `true`. For example:


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

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

In the generated type definition, the enumerated value names include the class prefix LEDcolor.

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

typedef enum LEDcolor LEDcolor;
```

More About

- [Modifying Superclass Methods and Properties \(MATLAB\)](#)
- [“Code Generation for Enumerations” on page 22-2](#)

Code Generation for Function Handles

Function Handle Limitations for Code Generation

When you use function handles in MATLAB code intended for code generation, adhere to the following restrictions:

Do not use the same bound variable to reference different function handles

In some cases, using the same bound variable to reference different function handles causes a compile-time error. For example, this code does not compile:

```
function y = foo(p)
x = @plus;
if p
    x = @minus;
end
y = x(1, 2);
```

Do not pass function handles to or from `coder.ceval`

You cannot pass function handles as inputs to or outputs from `coder.ceval`. For example, suppose that `f` and `str.f` are function handles:

```
f = @sin;
str.x = pi;
str.f = f;
```

The following statements result in compilation errors:

```
coder.ceval('foo', @sin);
coder.ceval('foo', f);
coder.ceval('foo', str);
```

Do not associate a function handle with an extrinsic function

You cannot create a function handle that references an extrinsic MATLAB function.

Do not pass function handles to or from extrinsic functions

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions.

Do not pass function handles to or from entry-point functions

You cannot pass function handles as inputs to or outputs from entry-point functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle, 'function_handle') && isa(data, 'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as inputs. `plotFcn` attempts to call the function referenced by the `fhandle` with the input `data` and plot the results. However, this code generates a compilation error. The error indicates that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of inputs.

More About

- “Declaring MATLAB Functions as Extrinsic Functions” on page 16-10

Generate Efficient and Reusable Code

- “Modularize MATLAB Code” on page 24-2
- “Eliminate Redundant Copies of Function Inputs” on page 24-3
- “Inline Code” on page 24-6
- “Control Inlining” on page 24-8
- “Fold Function Calls into Constants” on page 24-11
- “Control Stack Space Usage” on page 24-13
- “Stack Allocation and Performance” on page 24-14
- “Dynamic Memory Allocation and Performance” on page 24-15
- “Minimize Dynamic Memory Allocation” on page 24-16
- “Provide Maximum Size for Variable-Size Arrays” on page 24-17
- “Disable Dynamic Memory Allocation During Code Generation” on page 24-23
- “Set Dynamic Memory Allocation Threshold” on page 24-24
- “Excluding Unused Paths from Generated Code” on page 24-26
- “Prevent Code Generation for Unused Execution Paths” on page 24-27
- “Generate Code with Parallel for-Loops (parfor)” on page 24-29
- “Minimize Redundant Operations in Loops” on page 24-31
- “Unroll for-Loops” on page 24-33
- “Disable Support for Integer Overflow or Non-Finites” on page 24-35
- “Integrate External/Custom Code” on page 24-37
- “MATLAB Coder Optimizations in Generated Code” on page 24-43
- “Generate Reusable Code” on page 24-46

Modularize MATLAB Code

For large MATLAB code, streamline code generation by modularizing the code:

- 1** Break up your MATLAB code into smaller, self-contained sections.
- 2** Save each section in a MATLAB function.
- 3** Generate C/C++ code for each function.
- 4** Call the generated C/C++ functions in sequence from a wrapper MATLAB function using `coder.ceval`.
- 5** Generate C/C++ code for the wrapper function.

Besides streamlining code generation for the original MATLAB code, this approach also supplies you with C/C++ code for the individual sections. You can reuse the code for the individual sections later by integrating them with other generated C/C++ code using `coder.ceval`.

Eliminate Redundant Copies of Function Inputs

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

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

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, the generated code passes the variable by reference instead of redundantly copying the input to a temporary variable. In the preceding example, input `A` is passed by reference in the generated code because it also acts as an output for function `foo`:

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

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

By contrast, suppose that you rewrite function `foo` without the optimization:

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

The generated code passes the inputs by value and returns the value of the output:

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

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
    x1=u1+1;
    y1=bar(x1);
end

function y2=bar(u2)
    % Since foo does not use x1 later in the function,
    % it would be optimal to do this operation in place
    x2=u2.*2;
    % The change in dimensions in the following code
    % means that it cannot be done in place
    y2=[x2,x2];
end
```

You can modify this code to eliminate redundant copies.

```
function y1=foo(u1) %#codegen
    u1=u1+1;
    [y1, u1]=bar(u1);
end

function [y2, u2]=bar(u2)
    u2=u2.*2;
    % The change in dimensions in the following code
    % still means that it cannot be done in place
    y2=[u2,u2];
end
```

The reference parameter optimization does not apply to constant inputs. If the same variable is an input and an output, and the input is constant, the code generator treats the output as a separate variable. For example, consider the function `foo`:

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

Generate code in which `A` has a constant value 2.

```
codegen -config:lib foo -args {coder.Constant(2) 3} -report
```

The generated code defines the constant A and returns the value of the output.

```
...  
#define A                               (2.0)  
...  
double foo(double B)  
{  
    return A * B;  
}  
...
```

Inline Code

Inlining is a technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code. The code generator uses internal heuristics to determine whether to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

In this section...

“Prevent Function Inlining” on page 24-6

“Use Inlining in Control Flow Statements” on page 24-6

Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
    coder.inline('never');
    y = x;
end
```

Use Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
```

```
% Vector division produces a for-loop.  
% Prohibit inlining to reduce code size.  
    coder.inline('never');  
end  
  
if any(divisor == 0)  
    error('Can not divide by 0');  
end  
  
y = dividend / divisor;
```

Related Examples

- “Control Inlining” (MATLAB Coder)

Control Inlining

Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose that you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at many different sites in your code. The generated code can be large due to the function being inlined every time it is called.

The call sites must be different. For instance, inlining does not lead to large code if the function to be inlined is called several times inside a loop.

- You have limited RAM or stack space.

In this section...
“Control Size of Functions Inlined” on page 24-8
“Control Size of Functions After Inlining” on page 24-9
“Control Stack Size Limit on Inlined Functions” on page 24-9

Control Size of Functions Inlined

You can use the MATLAB Coder app or the command-line interface to control the maximum size of functions that can be inlined. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThreshold`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineThreshold = 100;
```

Generate code using this configuration object.

Control Size of Functions After Inlining

You can use the MATLAB Coder app or the command-line interface to control the maximum size of functions after inlining. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold max**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineThresholdMax = 100;
```

Generate code using this configuration object.

Control Stack Size Limit on Inlined Functions

Specifying a limit on the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even after the function is executed. The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that a certain value of `InlineStackLimit` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can use the MATLAB Coder app or the command-line interface to control the stack size limit on inlined functions.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline stack limit**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineStackLimit`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineStackLimit = 2000;
```

Generate code using this configuration object.

Related Examples

- “Inline Code” (MATLAB Coder)

Fold Function Calls into Constants

This example shows how to specify constants in generated code using `coder.const`. The code generator folds an expression or a function call in a `coder.const` statement into a constant in generated code. Because the generated code does not have to evaluate the expression or call the function every time, this optimization reduces the execution time of the generated code.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int i0;
    static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                         49, 64, 81, 100 };

    for (i0 = 0; i0 < 10; i0++) {
        y[i0] = (double)iv0[i0] + Shift;
    }
}
```

See Also

`coder.const`


Control Stack Space Usage

This example shows how to set the maximum stack space that the generated code uses. Set the maximum stack usage when:

- You have limited stack space, for instance, in embedded targets.
- Your C compiler reports a run-time stack overflow.

The value of the property, `StackUsageMax`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that a certain value of `StackUsageMax` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Control Stack Space Usage Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
- 3 Click **More Settings**.
- 4 On the **Memory** tab, set **Stack usage max** to the value that you want.

Control Stack Space Usage at the Command Line

- 1 Create a configuration object for code generation.

Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

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

- 2 Set the property, `StackUsageMax`, to the value that you want.

```
cfg.StackUsageMax=400000;
```

More About

- “Stack Allocation and Performance” (MATLAB Coder)

Stack Allocation and Performance

By default, local variables are allocated on the stack. Large variables that do not fit on the stack are statically allocated in memory.

Stack allocation typically uses memory more efficiently than static allocation. However, stack space is sometimes limited, typically in embedded processors. MATLAB Coder allows you to manually set a limit on the stack space usage to make your generated code suitable for your target hardware. You can choose this limit based on the target hardware configurations. For more information, see “Control Stack Space Usage” (MATLAB Coder).

For limited stack space, you can choose to allocate large variables on the heap instead of using static allocation. Heap allocation is slower but more memory-efficient than static allocation. To allocate large variables on the heap, do one of the following:

Allocate Heap Space from Command Line

- 1 Create a configuration object. Set the property, `MultiInstanceCode`, to `true`.

```
cfg = coder.config('exe');  
cfg.MultiInstanceCode = true;
```

- 2 Generate code using this configuration object.

Allocate Heap Space Using the MATLAB Coder App

- 1 Using the MATLAB Coder app, in the project settings dialog box, on the **Memory** tab, select the **Generate re-entrant code** check box.
 - Generate code.

Dynamic Memory Allocation and Performance

To achieve faster execution of generated code, minimize dynamic (or run-time) memory allocation of arrays.

MATLAB Coder does not provide a size for unbounded arrays in generated code. Instead, such arrays are referenced indirectly through pointers. For such arrays, memory cannot be allocated during compilation of generated code. Based on storage requirements for the arrays, memory is allocated and freed at run time as required. This run-time allocation and freeing of memory leads to slower execution of the generated code.

When Dynamic Memory Allocation Occurs

Dynamic memory allocation occurs when the code generator cannot find upper bounds for variable-size arrays. The software cannot find upper bounds when you specify the size of an array using a variable that is not a compile-time constant. An example of such a variable is an input variable (or a variable computed from an input variable).

Instances in the MATLAB code that can lead to dynamic memory allocation are:

- Array initialization: You specify array size using a variable whose value is known only at run time.
- After initialization of an array:
 - You declare the array as variable-size using `coder.varsize` without explicit upper bounds. After this declaration, you expand the array by concatenation inside a loop. The number of loop runs is known only at run time.
 - You use a `reshape` function on the array. At least one of the size arguments to the `reshape` function is known only at run time.

If you know the maximum size of the array, you can avoid dynamic memory allocation. You can then provide an upper bound for the array and prevent dynamic memory allocation in generated code. For more information, see “Minimize Dynamic Memory Allocation” on page 24-16.

Minimize Dynamic Memory Allocation

When possible, minimize dynamic memory allocation because it leads to slower execution of generated code. Dynamic memory allocation occurs when the code generator cannot find upper bounds for variable-size arrays.

If you know the maximum size of a variable-size array, you can avoid dynamic memory allocation. Follow these steps:

- 1 “Provide Maximum Size for Variable-Size Arrays” on page 24-17.
- 2 Depending on your requirements, do one of the following:
 - “Disable Dynamic Memory Allocation During Code Generation” on page 24-23.
 - “Set Dynamic Memory Allocation Threshold” (MATLAB Coder)

Caution: If a variable-size array in the MATLAB code does not have a maximum size, disabling dynamic memory allocation leads to a code generation error. Before disabling dynamic memory allocation, you must provide a maximum size for variable-size arrays in your MATLAB code.

More About

- “Dynamic Memory Allocation and Performance” (MATLAB Coder)

Provide Maximum Size for Variable-Size Arrays

To constrain array size for variable-size arrays, do one of the following:

- **Constrain Array Size Using `assert` Statements**

If the variable specifying array size is not a compile-time constant, use an `assert` statement with relational operators to constrain the variable. Doing so helps the code generator to determine a maximum size for the array.

The following examples constrain array size using `assert` statements:

- **When Array Size Is Specified by Input Variables**

Define a function `array_init` which initializes an array `y` with input variable `N`:

```
function y = array_init (N)
    assert(N <= 25); % Generates exception if N > 25
    y = zeros(1,N);
```

The `assert` statement constrains input `N` to a maximum size of 25. In the absence of the `assert` statement, `y` is assigned a pointer to an array in the generated code, thus allowing dynamic memory allocation.

- **When Array Size Is Obtained from Computation Using Input Variables**

Define a function, `array_init_from_prod`, which takes two input variables, `M` and `N`, and uses their product to specify the maximum size of an array, `y`.

```
function y = array_init_from_prod (M,N)
    size=M*N;
    assert(size <= 25); % Generates exception if size > 25
    y=zeros(1,size);
```

The `assert` statement constrains the product of `M` and `N` to a maximum of 25.

Alternatively, if you restrict `M` and `N` individually, it leads to dynamic memory allocation:

```
function y = array_init_from_prod (M,N)
    assert(M <= 5);
    assert(N <= 5);
    size=M*N;
    y=zeros(1,size);
```

This code causes dynamic memory allocation because M and N can both have unbounded negative values. Therefore, their product can be unbounded and positive even though, individually, their positive values are bounded.

Tip: Place the `assert` statement on a variable immediately before it is used to specify array size.

Tip: You can use `assert` statements to restrict array sizes in most cases. When expanding an array inside a loop, this strategy does not work if the number of loop runs is known only at run time.

- **Restrict Concatenations in a Loop Using `coder.ysize` with Upper Bounds**

You can expand arrays beyond their initial size by concatenation. When you concatenate additional elements inside a loop, there are two syntax rules for expanding arrays.

- 1 Array size during initialization is not a compile-time constant**

If the size of an array during initialization is not a compile-time constant, you can expand it by concatenating additional elements:

```
function out=ExpandArray(in) % Expand an array by five elements
    out = zeros(1,in);
    for i=1:5
        out = [out 0];
    end
```

- 2 Array size during initialization is a compile-time constant**

Before concatenating elements, you have to declare the array as variable-size using `coder.ysize`:

```
function out=ExpandArray() % Expand an array by five elements
    out = zeros(1,5);
    coder.ysize('out');
    for i=1:5
        out = [out 0];
    end
```


Either case leads to dynamic memory allocation. To prevent dynamic memory allocation in such cases, use `coder. varsize` with explicit upper bounds. This example shows how to use `coder. varsize` with explicit upper bounds:

Restrict Concatenations Using `coder. varsize` with Upper Bounds

- 1 Define a function, `RunningAverage`, that calculates the running average of an `N`-element subset of an array:

```
function avg=RunningAverage(N)
% Array whose elements are to be averaged
NumArray=[1 6 8 2 5 3];
% Initialize average:
% These will also be the first two elements of the function output
avg=[0 0];
% Place a bound on the argument
coder. varsize('avg',[1 8]);
% Loop to calculate running average
for i=1:N
    s=0;
    s=s+sum(NumArray(1:i));
    avg=[avg s/i];
% Increase the size of avg as required by concatenation
end
```

The output, `avg`, is an array that you can expand as required to accommodate the running averages. As a new running average is calculated, it is added to the array `avg` through concatenation, thereby expanding the array.

Because the maximum number of running averages is equal to the number of elements in `NumArray`, you can supply an explicit upper bound for `avg` in the `coder. varsize` statement. In this example, the upper bound is 8 (the two initial elements plus the six elements of `NumArray`).

- 2 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned an array of size 8 (static memory allocation). The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void RunningAverage (double N, double avg_data[8], int avg_size[2])
```

- 3 By contrast, if you remove the explicit upper bound, the generated code dynamically allocates `avg`.

Replace the statement

```
coder.varsized('avg',[1 8]);
```

with:

```
coder.varsized('avg');
```

- 4 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned a pointer to an array, thereby allowing dynamic memory allocation. The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void Test(double N, mxArray_real_T *avg)
```

Note: Dynamic memory allocation also occurs if you precede `coder.varsized('avg')` with the following `assert` statement:

```
assert(N < 6);
```

The `assert` statement does not restrict the number of concatenations within the loop.

- **Constrain Array Size When Rearranging a Matrix**

The statement `out = reshape(in,m,n,...)` takes an array, `in`, as an argument and returns array, `out`, having the same elements as `in`, but reshaped as an `m`-by-`n`-by-`...` matrix. If one of the size variables `m`, `n`, `...` is not a compile-time constant, then dynamic memory allocation of `out` takes place.

To avoid dynamic memory allocation, use an `assert` statement before the `reshape` statement to restrict the size variables `m`, `n`, `...` to `numel(in)`. This example shows how to use an `assert` statement before a `reshape` statement:

Rearrange a Matrix into Given Number of Rows

- 1 Define a function, `ReshapeMatrix`, which takes an input variable, `N`, and reshapes a matrix, `mat`, to have `N` rows:

```
function [out1,out2] = ReshapeMatrix(N)

    mat = [1 2 3 4 5; 4 5 6 7 8]
    % Since mat has 10 elements, N must be a factor of 10
    % to pass as argument to reshape

    out1 = reshape(mat,N,[]);
    % N is not restricted

    assert(N < numel(mat));
    % N is restricted to number of elements in mat
    out2 = reshape(mat,N,[]);
```

- 2 Generate code for `ReshapeArray` using the `codegen` command (the input argument does not have to be a factor of 10):

```
codegen -config:lib -report ReshapeArray -args 3
```

While `out1` is dynamically allocated, `out2` is assigned an array with size 100 (=10 X 10) in the generated code.

Tip: If your system has limited memory, do not use the `assert` statement in this way. For an `n`-element matrix, the `assert` statement creates an `n`-by-`n` matrix, which might be large.

Related Examples


- “Minimize Dynamic Memory Allocation” (MATLAB Coder)
- “Disable Dynamic Memory Allocation During Code Generation” (MATLAB Coder)
- “Set Dynamic Memory Allocation Threshold” (MATLAB Coder)

More About

- “Dynamic Memory Allocation and Performance” (MATLAB Coder)

Disable Dynamic Memory Allocation During Code Generation

To disable dynamic memory allocation using the MATLAB Coder app:

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Memory** tab, under **Variable Sizing Support**, set **Dynamic memory allocation** to **Never**.

To disable dynamic memory allocation at the command line:

- 1 In the MATLAB workspace, define the configuration object:

```
cfg=coder.config('lib');
```
- 2 Set the `DynamicMemoryAllocation` property of the configuration object to `Off`:

```
cfg.DynamicMemoryAllocation = 'Off';
```

If a variable-size array in the MATLAB code does not have a maximum upper bound, disabling dynamic memory allocation leads to a code generation error. Therefore, you can identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the arrays that are dynamically allocated in the generated code.

Related Examples

- “Minimize Dynamic Memory Allocation” (MATLAB Coder)
- “Provide Maximum Size for Variable-Size Arrays” (MATLAB Coder)
- “Set Dynamic Memory Allocation Threshold” (MATLAB Coder)

More About

- “Dynamic Memory Allocation and Performance” (MATLAB Coder)


Set Dynamic Memory Allocation Threshold

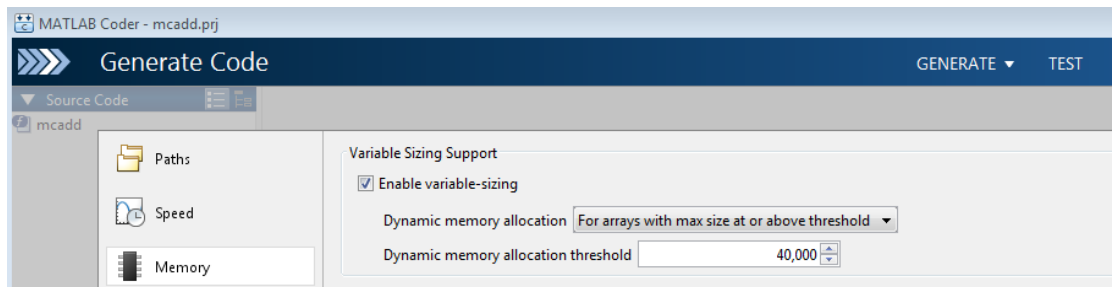
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays, but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-size arrays, you can disable dynamic memory allocation for arrays less than a certain size.

Specify this threshold when you want to:

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

Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Memory** tab, select the **Enable variable-sizing** check box.
- 4 Set **Dynamic memory allocation** to **For arrays with max size at or above threshold**.
- 5 Set **Dynamic memory allocation threshold** to the value that you want.



The **Dynamic memory allocation threshold** value is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Set Dynamic Memory Allocation Threshold at the Command Line

- 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

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

- 2 Set `DynamicMemoryAllocation` to `'Threshold'`.

```
cfg.DynamicMemoryAllocation='Threshold';
```

- 3 Set the property, `DynamicMemoryAllocationThreshold`, to the value that you want.

```
cfg.DynamicMemoryAllocationThreshold = 40000;
```

The value stored in `DynamicMemoryAllocationThreshold` is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Related Examples

- “Minimize Dynamic Memory Allocation” (MATLAB Coder)
- “Provide Maximum Size for Variable-Size Arrays” (MATLAB Coder)
- “Disable Dynamic Memory Allocation During Code Generation” (MATLAB Coder)

More About

- “Dynamic Memory Allocation and Performance” (MATLAB Coder)

Excluding Unused Paths from Generated Code

In certain situations, you do not need some branches of an: `if`, `elseif`, `else` statement, or a `switch`, `case`, `otherwise` statement in your generated code. For instance:

- You have a MATLAB function that performs multiple tasks determined by a control-flow variable. You might not need some of the tasks in the code generated from this function.
- You have an `if/elseif/if` statement in a MATLAB function performing different tasks based on the nature (type/value) of the input. In some cases, you know the nature of the input beforehand. If so, you do not need some branches of the `if` statement.

You can prevent code generation for the unused branches of an `if/elseif/else` statement or a `switch/case/otherwise` statement. Declare the control-flow variable as a constant. The code generator produces code only for the branch that the control-flow variable chooses.

Related Examples

- “Prevent Code Generation for Unused Execution Paths” (MATLAB Coder)

Prevent Code Generation for Unused Execution Paths

In this section...

“Prevent Code Generation When Local Variable Controls Flow” on page 24-27

“Prevent Code Generation When Input Variable Controls Flow” on page 24-28

If a variable controls the flow of an: `if`, `elseif`, `else` statement, or a `switch`, `case`, `otherwise` statement, declare it as constant so that code generation takes place for one branch of the statement only.

Depending on the nature of the control-flow variable, you can declare it as constant in two ways:

- If the variable is local to the MATLAB function, assign it to a constant value in the MATLAB code. For an example, see “Prevent Code Generation When Local Variable Controls Flow” on page 24-27.
- If the variable is an input to the MATLAB function, you can declare it as constant using `coder.Constant`. For an example, see “Prevent Code Generation When Input Variable Controls Flow” on page 24-28.

Prevent Code Generation When Local Variable Controls Flow

- 1 Define a function `SquareOrCube` which takes an input variable, `in`, and squares or cubes its elements based on whether the choice variable, `ch`, is set to `s` or `c`:

```
function out = SquareOrCube(ch,in) %#codegen
    if ch=='s'
        out = in.^2;
    elseif ch=='c'
        out = in.^3;
    else
        out = 0;
    end
```

- 2 Generate code for `SquareOrCube` using the `codegen` command:

```
codegen -config:lib SquareOrCube -args {'s',zeros(2,2)}
```

The generated C code squares or cubes the elements of a 2-by-2 matrix based on the input for `ch`.

- 3 Add the following line to the definition of `SquareOrCube`:

```
ch = 's';
```

The generated C code squares the elements of a 2-by-2 matrix. The choice variable, `ch`, and the other branches of the `if/elseif/if` statement do not appear in the generated code.

Prevent Code Generation When Input Variable Controls Flow

- 1 Define a function `MathFunc`, which performs different mathematical operations on an input, `in`, depending on the value of the input, `flag`:

```
function out = MathFunc(flag,in) %#codegen
    %# codegen
    switch flag
        case 1
            out=sin(in);
        case 2
            out=cos(in);
        otherwise
            out=sqrt(in);
    end
```

- 2 Generate code for `MathFunc` using the `codegen` command:

```
codegen -config:lib MathFunc -args {1,zeros(2,2)}
```

The generated C code performs different math operations on the elements of a 2-by-2 matrix based on the input for `flag`.

- 3 Generate code for `MathFunc`, declaring the argument, `flag`, as a constant using `coder.Constant`:

```
codegen -config:lib MathFunc -args {coder.Constant(1),zeros(2,2)}
```

The generated C code finds the sine of the elements of a 2-by-2 matrix. The variable, `flag`, and the `switch/case/otherwise` statement do not appear in the generated code.

More About

- “Excluding Unused Paths from Generated Code” (MATLAB Coder)

Generate Code with Parallel for-Loops (parfor)

This example shows how to generate C code for a MATLAB algorithm that contains a `parfor`-loop.

- 1 Write a MATLAB function that contains a `parfor`-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 Generate C code for `test_parfor`. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor
```

Because you did not specify the maximum number of threads to use, the generated C code executes the loop iterations in parallel on the available number of cores.

- 3 To specify a maximum number of threads, rewrite the function `test_parfor` as follows:

```
function a = test_parfor(u) %#codegen
a=ones(10,256);
r=rand(10,256);
parfor (i=1:10,u)
    a(i,:)=real(fft(r(i,:)));
end
```

- 4 Generate C code for `test_parfor`. Use `-args 0` to specify that the input, `u`, is a scalar double. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor -args 0
```

In the generated code, the iterations of the `parfor`-loop run on at most the number of cores specified by the input, `u`. If less than `u` cores are available, the iterations run on the cores available at the time of the call.

More About

- “Algorithm Acceleration Using Parallel for-Loops (`parfor`)” (MATLAB Coder)
- “Classification of Variables in `parfor`-Loops” (MATLAB Coder)

- “Reduction Assignments in parfor-Loops” (MATLAB Coder)

Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix **B** is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
    C=C + inv(B)*A^i*B;
end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

- 1 Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of n terms in the following power series expansion:

$$C = 1 + B^{-1}AB + B^{-1}A^2B + \dots$$

```
function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));

% Perform the series sum
for i=1:n
    C=C+inv(B)*A^i*B;
end
```

- 2 Generate code for `SeriesFunc` with 4-by-4 matrices passed as input arguments for **A** and **B**:

```
X = coder.typeof(zeros(4));
codegen -config:lib -launchreport SeriesFunc -args {X,X,10}
```

In the generated code, the inversion of **B** is performed n times inside the loop. It is more economical to perform the inversion operation once outside the loop because it does not depend on the loop index.

- 3 Modify `SeriesFunc` as follows:

```
function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));

% Perform the inversion outside the loop
inv_B=inv(B);

% Perform the series sum
for i=1:n
    C=C+inv_B*A^i*B;
end
```

This procedure performs the inversion of **B** only once, leading to faster execution of the generated code.

Unroll for-Loops

When the code generator unrolls a `for`-loop, instead of producing a `for`-loop in the generated code, it produces a copy of the loop body for each iteration. For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can significantly increase code generation time and generate inefficient code.

The code generator uses heuristics to determine when to unroll a `for`-loop. To force loop unrolling, use `coder.unroll`. For example:

```
function z = call_myloop()
    %#codegen
    z = myloop(5);
end

function b = myloop(n)
    b = zeros(1,n);
    coder.unroll();
    for i = 1:n
        b(i)=i+n;
    end
end
```

Here is the generated code for the `for`-loop:

```
z[0] = 6.0;
z[1] = 7.0;
z[2] = 8.0;
z[3] = 9.0;
z[4] = 10.0;
```

To control when a `for`-loop is unrolled, use the `coder.unroll flag` argument. For example, unroll the loop only when the number of iterations is less than 10.

```
function z = call_myloop()
    %#codegen
    z = myloop(5);
end

function b = myloop(n)
    unroll_flag = n < 10;
    b = zeros(1,n);
    coder.unroll(unroll_flag);
    for i = 1:n
```

```
        b(i)=i+n;
end
end
```

To unroll a `for`-loop, the code generator must be able to determine the bounds of the `for`-loop. For example, code generation fails for the following code because the value of `n` is not known at code generation time.

```
function b = myloop(n)
b = zeros(1,n);
coder.unroll();
for i = 1:n
    b(i)=i+n;
end
end
```

See Also

`coder.unroll`

More About

- “Nonconstant Index into `vargin` or `vargout` in a `for`-Loop” on page 46-45

Disable Support for Integer Overflow or Non-Finites


The code generator produces supporting code for the following situations:

- The result of an integer operation falls outside the range that a data type can represent. This situation is known as integer overflow.
- An operation generates non-finite values (`inf` and `NaN`). The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c`, and `rtGetNaN.c` (with corresponding header files).

If you know that these situations do not occur, you can suppress generation of the supporting code. You therefore reduce the size of the generated code and increase its speed. However, if one of these situations occurs, it is possible that the generated code does not match the behavior of the original MATLAB code.

Disable Support for Integer Overflow

You can use the MATLAB Coder app or the command-line interface to disable support for integer overflow. When you disable this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.


- Using the app:
 - 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
 - 2 Click **More Settings**.
 - 3 On the **Speed** tab, clear **Saturate** on integer overflow.
- At the command line:
 - 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```
 - 2 Set the `SaturateOnIntegerOverflow` property to `false`.

```
cfg.SaturateOnIntegerOverflow = false;
```

Disable Support for Non-Finite Numbers

You can use the MATLAB Coder app or the command-line interface to disable support for non-finite numbers(`inf` and `NaN`).

- Using the app:
 - 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
 - 2 Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
 - 3 Click **More Settings**.
 - 4 On the **Speed** tab, clear the **Support non-finite numbers** check box.
- At the command line:
 - 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```
 - 2 Set the `SupportNonFinite` property to `false`.

```
cfg.SupportNonFinite = false;
```

Integrate External/Custom Code

This example shows how to integrate external or custom code to enhance performance of generated code. Although MATLAB Coder generates optimized code for most applications, you might have custom code optimized for your specific requirements. For example:

- You have custom libraries optimized for your target environment.
- You have custom libraries for functions not supported by MATLAB Coder.
- You have custom libraries that meet standards set by your company.

In such cases, you can integrate your custom code with the code generated by MATLAB Coder.

This example illustrates how to integrate the function `cublasSgemm` from the NVIDIA® CUDA® Basic Linear Algebra Subroutines (CUBLAS) library in generated code. This function performs matrix multiplication on a Graphics Processing Unit (GPU).

- 1 Define a class `ExternalLib_API` that derives from the class `coder.ExternalDependency`. `ExternalLib_API` defines an interface to the CUBLAS library through the following methods:
 - `getDescriptiveName`: Returns a descriptive name for `ExternalLib_API` to be used for error messages.
 - `isSupportedContext`: Determines if the build context supports the CUBLAS library.
 - `updateBuildInfo`: Adds header file paths and link files to the build information.
 - `GPU_MatrixMultiply`: Defines the interface to the CUBLAS library function `cublasSgemm`.

ExternalLib_API.m

```
classdef ExternalLib_API < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(-)
            bName = 'ExternalLib_API';
```

```
end

function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('CUBLAS library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, ~, ~] = ctx.getStdLibInfo();

    % Include header file path
    % Include header files later using coder.cinclude
    hdrFilePath = 'C:\My_Includes';
    buildInfo.addIncludePaths(hdrFilePath);

    % Include link files
    linkFiles = strcat('libcublas', linkLibExt);
    linkPath = 'C:\My_Libs';
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    linkFiles = strcat('libcudart', linkLibExt);
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

end

%API for library function 'cuda_MatrixMultiply'
function C = GPU_MatrixMultiply(A, B)
    assert(isa(A, 'single'), 'A must be single. ');
    assert(isa(B, 'single'), 'B must be single. ');

    if(coder.target('MATLAB'))
        C=A*B;
    else

        % Include header files
```

```
% for external functions and typedefs
% Header path included earlier using updateBuildInfo
coder.cinclude('"cuda_runtime.h"');
coder.cinclude('"cublas_v2.h"');

% Compute dimensions of input matrices
m = int32(size(A, 1));
k = int32(size(A, 2));
n = int32(size(B, 2));

% Declare pointers to matrices on destination GPU
d_A = coder.opaque('float*');
d_B = coder.opaque('float*');
d_C = coder.opaque('float*');

% Compute memory to be allocated for matrices
% Single = 4 bytes
size_A = m*k*4;
size_B = k*n*4;
size_C = m*n*4;

% Define error variables
error = coder.opaque('cudaError_t');
cudaSuccessV = coder.opaque('cudaError_t', ...
    'cudaSuccess');

% Assign memory on destination GPU
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_A), size_A);
assert(error == cudaSuccessV, ...
    'cudaMalloc(A) failed');
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_B), size_B);
assert(error == cudaSuccessV, ...
    'cudaMalloc(B) failed');
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_C), size_C);
assert(error == cudaSuccessV, ...
    'cudaMalloc(C) failed');

% Define direction of copying
hostToDevice = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyHostToDevice');
```

```
% Copy matrices to destination GPU
error = coder.ceval('cudaMemcpy', ...
    d_A, coder.rref(A), size_A, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(A) failed');

error = coder.ceval('cudaMemcpy', ...
    d_B, coder.rref(B), size_B, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(B) failed');

% Define type and size for result
C = zeros(m, n, 'single');

error = coder.ceval('cudaMemcpy', ...
    d_C, coder.rref(C), size_C, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

% Define handle variables for external library
handle = coder.opaque('cublasHandle_t');
blasSuccess = coder.opaque('cublasStatus_t', ...
    'CUBLAS_STATUS_SUCCESS');

% Initialize external library
ret = coder.opaque('cublasStatus_t');
ret = coder.ceval('cublasCreate', coder.wref(handle));
assert(ret == blasSuccess, 'cublasCreate failed');

TRANSA = coder.opaque('cublasOperation_t', ...
    'CUBLAS_OP_N');
alpha = single(1);
beta = single(0);

% Multiply matrices on GPU
ret = coder.ceval('cublasSgemm', handle, ...
    TRANSA, TRANSA, m, n, k, ...
    coder.rref(alpha), d_A, m, ...
    d_B, k, ...
    coder.rref(beta), d_C, k);

assert(ret == blasSuccess, 'cublasSgemm failed');

% Copy result back to local host
deviceToHost = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyDeviceToHost');
```

```

        error = coder.ceval('cudaMemcpy', coder.wref(C), ...
            d_C, size_C, deviceToHost);
        assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');
    end
end
end
end
end

```

- 2 To perform the matrix multiplication using the interface defined in method `GPU_MatrixMultiply` and the build information in `ExternalLib_API`, include the following line in your MATLAB code:

```
C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

For instance, you can define a MATLAB function `Matrix_Multiply` that solely performs this matrix multiplication.

```
function C = Matrix_Multiply(A, B) %#codegen
    C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

- 3 Define a MEX configuration object using `coder.config`. For using the CUBLAS libraries, set the target language for code generation to C++.

```
cfg=coder.config('mex');
cfg.TargetLang='C++';
```

- 4 Generate code for `Matrix_Multiply` using `cfg` as the configuration object and two 2 X 2 matrices of type `single` as arguments. Since `cublasSgemv` supports matrix multiplication for data type `float`, the corresponding MATLAB matrices must have type `single`.

```
codegen -config cfg Matrix_Multiply ...
        -args {ones(2,'single'),ones(2,'single')}
```

- 5 Test the generated MEX function `Matrix_Multiply_mex` using two 2 X 2 identity matrices of type `single`.

```
Matrix_Multiply_mex(eye(2,'single'),eye(2,'single'))
```

The output is also a 2 X 2 identity matrix.

See Also

`coder.BuildConfig` | `assert` | `coder.ceval` | `coder.ExternalDependency` | `coder.opaque` | `coder.rref` | `coder.wref`

Related Examples

- “Encapsulate Interface to an External C Library” (MATLAB Coder)

More About

- “Encapsulating the Interface to External Code” (MATLAB Coder)

MATLAB Coder Optimizations in Generated Code

In this section...

“Constant Folding” on page 24-43

“Loop Fusion” on page 24-44

“Successive Matrix Operations Combined” on page 24-44

“Unreachable Code Elimination” on page 24-45

In order to improve the execution speed and memory usage of generated code, MATLAB Coder introduces the following optimizations:

Constant Folding

When possible, the code generator evaluates expressions in your MATLAB code that involve compile-time constants only. In the generated code, it replaces these expressions with the result of the evaluations. This behavior is known as constant folding. Because of constant folding, the generated code does not have to evaluate the constants during execution.

The following example shows MATLAB code that is constant-folded during code generation. The function `MultiplyConstant` multiplies every element in a matrix by a scalar constant. The function evaluates this constant using the product of three compile-time constants, `a`, `b`, and `c`.

```
function out=MultiplyConstant(in) %#codegen
    a=pi^4;
    b=1/factorial(4);
    c=exp(-1);
    out=in.*(a*b*c);
end
```

The code generator evaluates the expressions involving compile-time constants, `a`, `b`, and `c`. It replaces these expressions with the result of the evaluation in generated code.

Constant folding can occur when the expressions involve scalars only. To explicitly enforce constant folding of expressions in other cases, use the `coder.const` function. For more information, see “Fold Function Calls into Constants” (MATLAB Coder).

Control Constant Folding

You can control the maximum number of instructions that can be constant-folded from the command line or the project settings dialog box.

- At the command line, create a configuration object for code generation. Set the property `ConstantFoldingTimeout` to the value that you want.

```
cfg=coder.config('lib');  
cfg.ConstantFoldingTimeout = 200;
```

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the field **Constant folding timeout** to the value that you want.

Loop Fusion

When possible, the code generator fuses successive loops with the same number of runs into a single loop in the generated code. This optimization reduces loop overhead.

The following code contains successive loops, which are fused during code generation. The function `SumAndProduct` evaluates the sum and product of the elements in an array `Arr`. The function uses two separate loops to evaluate the sum `y_f_sum` and product `y_f_prod`.

```
function [y_f_sum,y_f_prod] = SumAndProduct(Arr) %#codegen  
    y_f_sum = 0;  
    y_f_prod = 1;  
    for i = 1:length(Arr)  
        y_f_sum = y_f_sum+Arr(i);  
    end  
    for i = 1:length(Arr)  
        y_f_prod = y_f_prod*Arr(i);  
    end
```

The code generated from this MATLAB code evaluates the sum and product in a single loop.

Successive Matrix Operations Combined

When possible, the code generator converts successive matrix operations in your MATLAB code into a single loop operation in generated code. This optimization reduces excess loop overhead involved in performing the matrix operations in separate loops.

The following example contains code where successive matrix operations take place. The function `ManipulateMatrix` multiplies every element of a matrix `Mat` with a `factor`. To every element in the result, the function then adds a `shift`:

```
function Res=ManipulateMatrix(Mat, factor, shift)
    Res=Mat*factor;
    Res=Res+shift;
end
```

The generated code combines the multiplication and addition into a single loop operation.

Unreachable Code Elimination

When possible, the code generator suppresses code generation from unreachable procedures in your MATLAB code. For instance, if a branch of an `if`, `elseif`, `else` statement is unreachable, then code is not generated for that branch.

The following example contains unreachable code, which is eliminated during code generation. The function `SaturateValue` returns a value based on the range of its input `x`.

```
function y_b = SaturateValue(x) %#codegen
    if x>0
        y_b = x;
    elseif x>10 %This is redundant
        y_b = 10;
    else
        y_b = -x;
    end
```

The second branch of the `if`, `elseif`, `else` statement is unreachable. If the variable `x` is greater than 10, it is also greater than 0. Therefore, the first branch is executed in preference to the second branch.

MATLAB Coder does not generate code for the unreachable second branch.

Generate Reusable Code

With MATLAB, you can generate reusable code in the following ways:

- Write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or MATLAB function library.
- Compile external functions on the MATLAB path and integrate them into generated C code for embedded targets.

See “Resolution of Function Calls for Code Generation” (MATLAB Coder).

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 25-2
- “Structure Operations Allowed for Code Generation” on page 25-3
- “Define Scalar Structures for Code Generation” on page 25-4
- “Define Arrays of Structures for Code Generation” on page 25-6
- “Index Substructures and Fields” on page 25-8
- “Assign Values to Structures and Fields” on page 25-10
- “Pass Large Structures as Input Parameters” on page 25-12

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 25-3
Observe restrictions on properties and values of scalar structures.	“Define Scalar Structures for Code Generation” on page 25-4
Make structures uniform in arrays.	“Define Arrays of Structures for Code Generation” on page 25-6
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 25-8
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 25-10

Structure Operations Allowed for Code Generation

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

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

Define Scalar Structures for Code Generation

In this section...

“Restrictions When Defining Scalar Structures by Assignment” on page 25-4
 “Adding Fields in Consistent Order on Each Control Flow Path” on page 25-4
 “Restriction on Adding New Fields After First Use” on page 25-5

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

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

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

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


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

Restriction on Adding New Fields After First Use

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

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

For example, consider this code:

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

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

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

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

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

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

Define Arrays of Structures for Code Generation

In this section...

“Ensuring Consistency of Fields” on page 25-6

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 25-6

“Defining an Array of Structures by Using struct” on page 25-7

“Defining an Array of Structures Using Concatenation” on page 25-7

Ensuring Consistency of Fields

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

Once you have created the array of structures, you can make the structure fields variable-size using `coder. varsizes`. For more information, see “Declare a Variable-Size Structure Field.” (MATLAB Coder).

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 25-4.
- 2 Call `repmat`, passing the scalar structure and the dimensions of the array.
- 3 Assign values to each structure using standard array indexing and structure dot notation.

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

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

```

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

```

Defining an Array of Structures by Using `struct`

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

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

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

Defining an Array of Structures Using Concatenation

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

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

```

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

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

```

Index Substructures and Fields

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

Reference substructure field values individually using dot notation

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

```

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

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

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

```

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

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

Reference field values individually in structure arrays

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

```

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

```

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

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

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 25-6 for more information.

Do not reference fields dynamically

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

Assign Values to Structures and Fields

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

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

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

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

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

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

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

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

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

Do not assign 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 16-15).

Do not assign classes to structure fields

You cannot assign classes to structure fields.

Do not assign cell arrays to structure fields

You cannot assign cell arrays to structure fields.

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 generator 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, Classes, and System Objects Supported for Code Generation

- “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2
- “Functions and Objects Supported for C/C++ Code Generation — Category List” on page 26-67

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

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

To find supported functions, classes, and System objects by MATLAB category or toolbox, see “Functions and Objects Supported for C/C++ Code Generation — Category List” on page 26-67.

Note: For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB” on page 14-3.

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

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

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

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

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

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

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

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

Name	Product
cheb2ap*	Signal Processing Toolbox
cheb2ord*	Signal Processing Toolbox
chebwin*	Signal Processing Toolbox
cheby1*	Signal Processing Toolbox
cheby2*	Signal Processing Toolbox
chi2cdf	Statistics and Machine Learning Toolbox
chi2inv	Statistics and Machine Learning Toolbox
chi2pdf	Statistics and Machine Learning Toolbox
chi2rnd*	Statistics and Machine Learning Toolbox
chi2stat	Statistics and Machine Learning Toolbox
chol	MATLAB
cholupdate	MATLAB
circpol2pol*	Phased Array System Toolbox
circshift	MATLAB
c12tf*	DSP System Toolbox
class	MATLAB
ClassificationECOC* or CompactClassificationECOC*	Statistics and Machine Learning Toolbox
ClassificationEnsemble*, ClassificationBaggedEnsemble*, or CompactClassificationEnsemble*	Statistics and Machine Learning Toolbox
ClassificationLinear*	Statistics and Machine Learning Toolbox
ClassificationSVM* or CompactClassificationSVM*	Statistics and Machine Learning Toolbox

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

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

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

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

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

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

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

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

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

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

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

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

Name	Product
dsp.SignalSource*	DSP System Toolbox
dsp.SineWave*	DSP System Toolbox
dsp.SpectrumAnalyzer*	DSP System Toolbox
dsp.SpectrumEstimator*	DSP System Toolbox
dsp.StandardDeviation*	DSP System Toolbox
dsp.StateLevels*	DSP System Toolbox
dsp.SubbandAnalysisFilter*	DSP System Toolbox
dsp.SubbandSynthesisFilter*	DSP System Toolbox
dsp.TimeScope*	DSP System Toolbox
dsp.TransferFunctionEstimator*	DSP System Toolbox
dsp.UDPReceiver*	DSP System Toolbox
dsp.UDPSender*	DSP System Toolbox
dsp.UpperTriangularSolver*	DSP System Toolbox
dsp.VariableBandwidthFIRFilter*	DSP System Toolbox
dsp.VariableBandwidthIIRFilter*	DSP System Toolbox
dsp.VariableFractionDelay*	DSP System Toolbox
dsp.VariableIntegerDelay*	DSP System Toolbox
dsp.Variance*	DSP System Toolbox
dsp.VectorQuantizerDecoder*	DSP System Toolbox
dsp.VectorQuantizerEncoder*	DSP System Toolbox
dsp.Window*	DSP System Toolbox
dsp.ZeroCrossingDetector*	DSP System Toolbox
dvbs21dpc*	Communications System Toolbox
dwt*	Wavelet Toolbox
dwt2*	Wavelet Toolbox
dyadup*	Wavelet Toolbox
edge*	Image Processing Toolbox
effearthradius*	Phased Array System Toolbox

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For an alphabetical list of supported functions, classes, and System objects, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2.

Note: For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB” on page 14-3.

In this section...

“Aerospace Toolbox” on page 26-69
“Arithmetic Operations in MATLAB” on page 26-69
“Audio System Toolbox” on page 26-70
“Automated Driving System Toolbox” on page 26-71
“Bit-Wise Operations MATLAB” on page 26-72
“Casting in MATLAB” on page 26-73
“Character Functions in MATLAB” on page 26-73
“Communications System Toolbox” on page 26-74
“Complex Numbers in MATLAB” on page 26-80
“Computer Vision System Toolbox” on page 26-81
“Control Flow in MATLAB” on page 26-85
“Control System Toolbox” on page 26-85
“Data and File Management in MATLAB” on page 26-85
“Data Types in MATLAB” on page 26-86
“Desktop Environment in MATLAB” on page 26-87
“Discrete Math in MATLAB” on page 26-87

In this section...

- “DSP System Toolbox” on page 26-87
- “Error Handling in MATLAB” on page 26-93
- “Exponents in MATLAB” on page 26-93
- “Filtering and Convolution in MATLAB” on page 26-94
- “Fixed-Point Designer” on page 26-94
- “Histograms in MATLAB” on page 26-100
- “Image Acquisition Toolbox” on page 26-100
- “Image Processing in MATLAB” on page 26-101
- “Image Processing Toolbox” on page 26-101
- “Input and Output Arguments in MATLAB” on page 26-105
- “Interpolation and Computational Geometry in MATLAB” on page 26-105
- “Linear Algebra in MATLAB” on page 26-106
- “Logical and Bit-Wise Operations in MATLAB” on page 26-107
- “MATLAB Compiler” on page 26-107
- “Matrices and Arrays in MATLAB” on page 26-108
- “Neural Network Toolbox” on page 26-111
- “Numerical Integration and Differentiation in MATLAB” on page 26-111
- “Optimization Functions in MATLAB” on page 26-112
- “Optimization Toolbox” on page 26-112
- “Phased Array System Toolbox” on page 26-113
- “Polynomials in MATLAB” on page 26-119
- “Programming Utilities in MATLAB” on page 26-120
- “Relational Operators in MATLAB” on page 26-120
- “Robotics System Toolbox” on page 26-120
- “Rounding and Remainder Functions in MATLAB” on page 26-122
- “Set Operations in MATLAB” on page 26-122
- “Signal Processing in MATLAB” on page 26-123
- “Signal Processing Toolbox” on page 26-123
- “Special Values in MATLAB” on page 26-126

In this section...

“Specialized Math in MATLAB” on page 26-127

“Statistics in MATLAB” on page 26-127

“Statistics and Machine Learning Toolbox” on page 26-128

“System Identification Toolbox” on page 26-134

“System object Methods” on page 26-135

“Trigonometry in MATLAB” on page 26-135

“Wavelet Toolbox” on page 26-137

“WLAN System Toolbox” on page 26-138

Aerospace Toolbox

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

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

quatconj*

quatdivide*

quatinv*

quatmod*

quatmultiply*

quatnorm*

quatnormalize*

Arithmetic Operations in MATLAB

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

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

ctranspose

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

Audio System Toolbox

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

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

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

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

Automated Driving System Toolbox

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

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

cameas
cameasjac
constacc

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

Bit-Wise Operations MATLAB

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

flintmax

swapbytes*

Casting in MATLAB

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

cast

char*

class

double

int8, int16, int32, int64

logical

single

typecast*

uint8, uint16, uint32, uint64

Character Functions in MATLAB

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

bin2dec*

blanks

char

deblank*

dec2bin*

dec2hex*

hex2dec*

hex2num*

iscellstr

ischar

isletter*

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

Communications System Toolbox

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

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

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

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

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

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

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

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

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

Complex Numbers in MATLAB

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

complex

conj
cplxpair
imag
isnumeric
isreal
isscalar
real
unwrap*

Computer Vision System Toolbox

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

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

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

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

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

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

Name
vision.MarkerInserter*
vision.ShapeInserter*

Control Flow in MATLAB

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

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

Control System Toolbox

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

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

extendedKalmanFilter*
unscentedKalmanFilter*

Data and File Management in MATLAB

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

computer*

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

Data Types in MATLAB

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

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

Desktop Environment in MATLAB

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

ismac*
ispc*
isunix*

Discrete Math in MATLAB

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

factor*
gcd
isprime*
lcm
nchoosek*
primes*

DSP System Toolbox

C code generation for the following functions and System objects requires the DSP System Toolbox license. Many DSP System Toolbox functions require constant inputs for code generation. To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see “Specify Constant Inputs at the Command Line” on page 29-7.

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

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

Name
dsp.CrossSpectrumEstimator*
dsp.LevinsonSolver*
dsp.LPCToAutocorrelation*
dsp.LPCToCepstral*
dsp.LPCToLSF*
dsp.LPCToLSP*
dsp.LPCToRC*
dsp.LSFToLPC*
dsp.LSPToLPC*
dsp.RCToAutocorrelation*
dsp.RCToLPC*
dsp.SpectrumEstimator*
dsp.TransferFunctionEstimator*
Filters
ca2tf*
c12tf*
dsp.AdaptiveLatticeFilter*
dsp.AffineProjectionFilter*
dsp.AllpassFilter*
dsp.AllpoleFilter*
dsp.BiquadFilter*
dsp.Channelizer*
dsp.ChannelSynthesizer*
dsp.CICCompensationDecimator*
dsp.CICCompensationInterpolator*
dsp.CICDecimator*
dsp.CICInterpolator*
dsp.Differentiator*

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

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

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

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

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

Error Handling in MATLAB

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

assert*
error*

Exponents in MATLAB

exp
expm
expm1
factorial
log*
log2
log10

log1p
nextpow2
nthroot
reallog
realpow
realsqrt
sqrt*

Filtering and Convolution in MATLAB

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

conv*
conv2
convn
deconv*
detrend*
filter*
filter2

Fixed-Point Designer

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

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

- You can use parallel for (`parfor`) loops in code compiled with `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.

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

<code>abs</code>
<code>accumneg</code>
<code>accumpos</code>
<code>add*</code>
<code>all</code>
<code>any</code>
<code>atan2</code>
<code>bitand*</code>
<code>bitandreduce</code>
<code>bitcmp</code>
<code>bitconcat</code>
<code>bitget</code>
<code>bitor*</code>
<code>bitorreduce</code>
<code>bitreplicate</code>
<code>bitrol</code>
<code>bitror</code>
<code>bitset</code>
<code>bitshift</code>
<code>bitsliceget</code>
<code>bitsll*</code>
<code>bitsra*</code>

bitsrl*
bitxor*
bitxorreduce
ceil
complex
conj
conv*
convergent
cordicabs*
cordicangle*
cordicatan2*
cordiccart2pol*
cordicceexp*
cordicccos*
cordicpol2cart*
cordicrotate*
cordicsin*
cordicsincos*
cordicsqrt*
cos
ctranspose
diag*
divide*
double*
end
eps*
eq*
fi*

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

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

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

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

Histograms in MATLAB

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

hist*
histc*
histcounts*

Image Acquisition Toolbox

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

Image Processing in MATLAB

Function	Remarks and Limitations
im2double	—
rgb2gray	—

Image Processing Toolbox

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

Image Processing Toolbox provides three types of code generation support:

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

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

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

adapthresh*

affine2d*

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

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

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

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

Input and Output Arguments in MATLAB

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

Interpolation and Computational Geometry in MATLAB

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

cart2pol
cart2sph
inpolygon*
interp1*

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

Linear Algebra in MATLAB

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

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

null*
orth*
rsf2csf
schur*
sqrtm

Logical and Bit-Wise Operations in MATLAB

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

MATLAB Compiler

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

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

isdeployed*
ismcc*

Matrices and Arrays in MATLAB

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

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

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

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

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

Neural Network Toolbox

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

Numerical Integration and Differentiation in MATLAB

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

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

Optimization Functions in MATLAB

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

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

Optimization Toolbox

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

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

fminbnd*
fminsearch*

fzero*
lsqnonneg*
optimget*
optimset*

Phased Array System Toolbox

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

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

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

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

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

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

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

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

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

Polynomials in MATLAB

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

poly*
polyder*
polyeig*

polyfit*
polyint
polyval
polyvalm
roots*

Programming Utilities in MATLAB

Function	Remarks and Limitations
mfilename	—

Relational Operators in MATLAB

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

eq*
ge
gt
le
lt
ne*

Robotics System Toolbox

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

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

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

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

quat2tform
rotm2axang
rotm2eul
rotm2quat
rotm2tform
tform2axang
tform2eul
tform2quat
tform2rotm
tform2trvec
trvec2tform

Rounding and Remainder Functions in MATLAB

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

ceil
fix
floor
mod*
rem*
round*

Set Operations in MATLAB

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

intersect*
ismember*
issorted*
setdiff*

setxor*
union*
unique*

Signal Processing in MATLAB

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

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

Signal Processing Toolbox

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

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

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

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

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

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

Special Values in MATLAB

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

eps
inf*
intmax
intmin
NaN or nan*

pi
realmax
realmin

Specialized Math in MATLAB

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

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

Statistics in MATLAB

corrcoef*

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

Statistics and Machine Learning Toolbox

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

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

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

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

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

gpstat
hygecdf
hygeinv
hygepdf
hygernd*
hygestat
icdf
logncdf
logninv
lognpdf
lognrnd*
lognstat
mnpdf
nbincdf
nbinv
nbpdf
nbprnd*
nbprstat
ncfcdf
ncfinv
ncfpdf
ncfrnd*
ncfstat
nctcdf
nctinv
nctpdf
nctrnd*
nctstat

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

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

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

System Identification Toolbox

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

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

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

System object Methods

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

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

Trigonometry in MATLAB

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

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

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

Wavelet Toolbox

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

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

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

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

WLAN System Toolbox

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

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

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

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

wlanVHTSIGBRecover*
Propagation Channel
wlanTGacChannel*
wlanTGahChannel*
wlanTGnChannel*

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

Code Generation for Variable-Size Data

- “Code Generation for Variable-Size Arrays” on page 27-2
- “Control Memory Allocation for Variable-Size Arrays” on page 27-5
- “Specify Upper Bounds for Variable-Size Arrays” on page 27-7
- “Define Variable-Size Data for Code Generation” on page 27-9
- “C Code Interface for Arrays” on page 27-15
- “Diagnose and Fix Variable-Size Data Errors” on page 27-20
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 27-24
- “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 27-33

Code Generation for Variable-Size Arrays

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

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

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

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

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

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

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

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

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

You can define variable-size arrays by:

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

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

For more information, see “Define Variable-Size Data for Code Generation” on page 27-9.

You can control whether variable-size arrays are allowed for code generation. See “Enabling and Disabling Support for Variable-Size Arrays” on page 27-3.

Memory Allocation for Variable-Size Arrays

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

You can control whether dynamic memory allocation is allowed or when it is used for code generation. See “Control Memory Allocation for Variable-Size Arrays” on page 27-5.

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

Enabling and Disabling Support for Variable-Size Arrays

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

- In a code configuration object, set the `EnableVariableSizing` parameter to `true` or `false`.

Variable-Size Arrays in a Code Generation Report

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

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

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

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

More About

- “Control Memory Allocation for Variable-Size Arrays” on page 27-5
- “Specify Upper Bounds for Variable-Size Arrays” on page 27-7
- “Define Variable-Size Data for Code Generation” on page 27-9

Control Memory Allocation for Variable-Size Arrays

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

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

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

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

Provide Upper Bounds for Variable-Size Arrays

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

- 1 Specify upper bounds for a variable-size array. See “Specify Upper Bounds for Variable-Size Arrays” on page 27-7.
- 2 Make sure that the size of the array, in bytes, is less than the dynamic memory allocation threshold. See “Configure Code Generator to Use Dynamic Memory Allocation for Arrays Bigger Than a Threshold” on page 27-6.

Disable Dynamic Memory Allocation

By default, dynamic memory allocation is enabled. To disable it, in a configuration object for fixed-point acceleration, set the `DynamicMemoryAllocation` parameter to `'Off'`.

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

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

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

Use the dynamic memory allocation threshold to:

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

The default dynamic memory allocation threshold is 64 kilobytes. To change the threshold, in a configuration object for fixed-point acceleration, set the `DynamicMemoryAllocationThreshold`.

To instruct the code generator to use dynamic memory allocation for variable-size arrays whose size is greater than or equal to the threshold, in the configuration object, set the `DynamicMemoryAllocationThreshold` to `'Threshold'`.

More About

- “Code Generation for Variable-Size Arrays” on page 27-2

Specify Upper Bounds for Variable-Size Arrays

Specify upper bounds for an array when:

- Dynamic memory allocation is disabled.

If dynamic memory allocation is disabled, you must specify upper bounds for all arrays.

- You do not want the code generator to use dynamic memory allocation for the array.

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

Specify Upper Bounds for Variable-Size Inputs

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.

To specify upper bounds for variable-size inputs, use the `coder.typeof` construct with the `fiaccel -args` option. For example:

```
fiaccel foo -args {coder.typeof(fi(0),[3 100],1)}
```

This command specifies that the input to function `foo` is a matrix of `fi` types with two variable dimensions. The upper bound for the first dimension is 3. The upper bound for the second dimension is 100.

Specify Upper Bounds for Local Variables

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

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

To constrain the value of variables that specify the dimensions of variable-size arrays, use the `assert` function with relational operators. For example:

```
function y = dim_need_bound(n) %#codegen
```

```
assert (n <= 5);  
L = ones(n,n);  
M = zeros(n,n);  
M = [L; M];  
y = M;
```

This `assert` statement constrains input `n` to a maximum size of 5. `L` is variable-size with upper bounds of 5 in each dimension. `M` is variable-size with an upper bound of 10 in the first dimension and 5 in the second dimension.

Specify the Upper Bounds for All Instances of a Local Variable

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

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

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

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

See Also

`coder. typeof` | `coder. varsize`

More About

- “Code Generation for Variable-Size Arrays” on page 27-2
- “Define Variable-Size Data for Code Generation” on page 27-9

Define Variable-Size Data for Code Generation

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

Method	See
Assign the data from a variable-size matrix constructor such as: <ul style="list-style-type: none"> • ones • zeros • repmat 	“Use a Matrix Constructor with Nonconstant Dimensions” on page 27-9
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Assign Multiple Sizes to the Same Variable” on page 27-10
Define all instances of a variable to be variable-size.	“Define Variable-Size Data Explicitly by Using <code>coder.varsize</code> ” on page 27-10

Use a Matrix Constructor with Nonconstant Dimensions

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

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

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

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

Assign Multiple Sizes to the Same Variable

Before you use (read) a variable in your code, you can make it variable-size by assigning multiple, constant sizes to it. When the code generator uses static allocation on the stack, it infers the upper bounds from the largest size specified for each dimension. When you assign the same size to a given dimension across all assignments, the code generator assumes that the dimension is fixed at that size. The assignments can specify different shapes and sizes.

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

Inferring Upper Bounds from Multiple Definitions with Different Shapes

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

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

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

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

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

Define Variable-Size Data Explicitly by Using `coder.varsize`

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

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

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

- Define B as a variable-size array:

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

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

Specify Which Dimensions Vary

You can use the function `coder.varsize` to specify which dimensions vary. For example, the following statement defines B as an array whose first dimension is fixed at 2, but whose second dimension can grow to a size of 16:

```
coder.varsize('B',[2, 16],[0 1])
```

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size. Dimensions that correspond to ones or `true` vary in size. `coder.varsize` usually treats dimensions of size 1 as fixed. See “Define Variable-Size Matrices with Singleton Dimensions” on page 27-12.

Allow a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix Y with fixed 2-by-2 dimensions before the first use (where the statement `Y = Y + u` reads from Y). However, `coder.varsize` defines Y as a variable-size matrix, allowing it to change size based on decision logic in the `else` clause:

```
function Y = var_by_if(u) %#codegen
if (u > 0)
    Y = zeros(2,2);
    coder.varsize('Y');
    if (u < 10)
        Y = Y + u;
    end
else
    Y = zeros(5,5);
end
```

Without `coder.varsize`, the code generator infers `Y` to be a fixed-size, 2-by-2 matrix. It generates a size mismatch error.

Define Variable-Size Matrices with Singleton Dimensions

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

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

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

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

- You initialize variable-size data with singleton dimensions by using matrix constructor expressions or matrix functions.

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

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

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

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



```

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

```

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

Define Variable-Size Structure Fields

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

```

function y=struct_example() %#codegen

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

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

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

```

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

Here are other examples:

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

In this example, `data` is a scalar variable that contains matrix `A`. Each element of matrix `A` contains a variable-size field `B`.

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

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

See Also

`coder.typeof` | `coder.varsizes`

More About

- “Code Generation for Variable-Size Arrays” on page 27-2
- “Specify Upper Bounds for Variable-Size Arrays” on page 27-7

C Code Interface for Arrays

In this section...

“C Code Interface for Statically Allocated Arrays” on page 27-15

“C Code Interface for Dynamically Allocated Arrays” on page 27-16

“Utility Functions for Creating emxArray Data Structures” on page 27-18

C Code Interface for Statically Allocated Arrays

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

For example, consider the MATLAB function `myunique_tol`.

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

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

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

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

In the generated code, the function declaration is:

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

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

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

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

C Code Interface for Dynamically Allocated Arrays

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

emxArray Structure Definition

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

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

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

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

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

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

C Code Interface for Structure Fields

Field	Description
<code>*data</code>	Pointer to data of type <code><baseType></code> .
<code>*size</code>	Pointer to first element of size vector. Length of the vector equals the number of dimensions.
<code>allocatedSize</code>	Number of elements currently allocated for the array. If the size changes, MATLAB reallocates memory based on the new size.
<code>numDimensions</code>	Number of dimensions of the size vector, that is, the number of dimensions you can access without crossing into unallocated or unused memory.
<code>canFreeData</code>	Boolean flag indicating how to deallocate memory: <ul style="list-style-type: none"> <code>true</code> – MATLAB deallocates memory automatically <code>false</code> – Calling program determines when to deallocate memory

Utility Functions for Creating emxArray Data Structures

When you generate code that uses variable-size data, the code generator exports a set of utility functions that you can use to create and interact with `emxArrays` in your generated code. To call these functions in your main C function, include the generated header file. For example, when you generate code for function `foo`, include `foo_emxAPI.h` in your main C function. .

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

Function	Arguments	Description
<code>emxArray_<baseType></code> <code>*emxCreateWrapper_<baseType></code> <code>(...)</code>	<code>*data</code> <code>num_rows</code> <code>num_cols</code>	Creates a new two-dimensional <code>emxArray</code> , but does not allocate it on the heap. Instead uses memory provided by the user and sets <code>canFreeData</code> to <code>false</code> so it does not inadvertently free user memory, such as the stack.
<code>emxArray_<baseType></code> <code>*emxCreateWrapperND_<baseType></code> <code>(...)</code>	<code>*data</code> <code>numDimensions</code> <code>*size</code>	Same as <code>emxCreateWrapper_<baseType></code> , except it creates a new N-dimensional <code>emxArray</code> .
<code>emxArray_<baseType></code> <code>*emxCreate_<baseType> (...)</code>	<code>num_rows</code> <code>num_cols</code>	Creates a new two-dimensional <code>emxArray</code> on the heap, initialized to zero. All data elements have the data type specified by <code><baseType></code> .

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

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

Diagnose and Fix Variable-Size Data Errors

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 27-20

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 27-22

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

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

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

Compiling this function produces this error:

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

There are several ways to fix this error:

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

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



```

    A = B;
end
Y = A;

```

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

```

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

```

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

```

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

```

Empty Matrix Reshaped to Match Variable-Size Specification

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

```

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

```

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

reshapes the empty matrix $Y = []$ in generated code to $Y = \text{zeros}(1,0)$ so it matches the `coder.ysize` specification.

Performing Binary Operations on Fixed and Variable-Size Operands

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

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

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

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

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

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

There are several ways to fix the problem:

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

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

Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

“Incompatibility with MATLAB for Scalar Expansion” on page 27-24

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 27-26

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 27-27

“Incompatibility with MATLAB in Determining Class of Empty Arrays” on page 27-28

“Incompatibility with MATLAB in Matrix-Matrix Indexing” on page 27-29

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 27-29

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 27-30

“Incompatibility with MATLAB in Concatenating Variable-Size Matrices” on page 27-31

“Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements” on page 27-31

Incompatibility with MATLAB for Scalar Expansion

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

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

Consider this function:

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

```

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

```

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



The screenshot shows the Code Generation Report window for the function `scalar_exp_test_err1`. The MATLAB code is displayed on the left, and the generated C code is shown on the right. Below the code, there is a table summarizing the variables used in the code.

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

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

```

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

```

```

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

```

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

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

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

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

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

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

Workarounds

If your application requires generated code to return the same size of variable-size N-D arrays as MATLAB code, consider one of these workarounds:

- Use the two-argument form of `size`.

For example, `size(A,n)` returns the same answer in generated code and MATLAB code.

- Rewrite `size(A)`:

```
B = size(A);
X = B(1:ndims(A));
```

This version returns X with a variable-length output. However, you cannot pass a variable-size X to matrix constructors such as `zeros` that require a fixed-size argument.

Incompatibility with MATLAB in Determining Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. The size might be 1×0 or 0×1 in generated code, but 0×0 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 1×1 and x has size 0×0 . To support this use case, the code generator determines the size for x as $[1 \times :?]$. Because there is another assignment $x = []$ after the concatenation, the size of x in the generated code is 1×0 instead of 0×0 .

For incompatibilities with MATLAB in determining the size of an empty array that results from deleting elements of an array, see “Size of Empty Array That Results from Deleting Elements of an Array” on page 21-11.

Workaround

If your application checks whether a matrix is empty, use one of these workarounds:

- Rewrite your code to use the `isempty` function instead of the `size` function.
- Instead of using `x=[]` to create empty arrays, create empty arrays of a specific size using `zeros`. For example:

```
function y = test_empty(n) %#codegen
x = zeros(1,0);
i=0;
while (i < 10)
    x = [5 x];
    i = i + 1;
end
if n > 0
    x = zeros(1,0);
end
y=size(x);
end
```

Incompatibility with MATLAB in Determining Class of Empty Arrays

The class of an empty array in generated code can be different from its class in MATLAB source code. Therefore, do not write code that relies on the class of empty matrices.

For example, consider the following code:

```
function y = fun(n)
x = [];
if n > 1
    x = ['a' x];
end
y=class(x);
end
```

`fun(0)` returns `double` in MATLAB, but `char` in the generated code. When the statement `n > 1` is false, MATLAB does not execute `x = ['a' x]`. The class of `x` is `double`, the class of the empty array. However, the code generator considers all execution paths. It determines that based on the statement `x = ['a' x]`, the class of `x` is `char`.

Workaround

Instead of using `x=[]` to create an empty array, create an empty array of a specific class. For example, use `blanks(0)` to create an empty array of characters.

```
function y = fun(n)
x = blanks(0);
if n > 1
    x = ['a' x];
end
```



```
end
y=class(x);
end
```

Incompatibility with MATLAB in Matrix-Matrix Indexing

In matrix-matrix indexing, you use one matrix to index into another matrix. In MATLAB, the general rule for matrix-matrix indexing is that the size and orientation of the result match the size and orientation of the index matrix. For example, if **A** and **B** are matrices, `size(A(B))` equals `size(B)`. When **A** and **B** are vectors, MATLAB applies a special rule. The special vector-vector indexing rule is that the orientation of the result is the orientation of the data matrix. For example, if **A** is 1-by-5 and **B** is 3-by-1, then `A(B)` is 1-by-3.

The code generator applies the same matrix-matrix indexing rules as MATLAB. If **A** and **B** are variable-size matrices, to apply the matrix-matrix indexing rules, the code generator assumes that the `size(A(B))` equals `size(B)`. If, at run time, **A** and **B** become vectors and have different orientations, then the assumption is incorrect. Therefore, when run-time error checks are enabled, an error can occur.

To avoid this issue, force your data to be a vector by using the colon operator for indexing. For example, suppose that your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing.

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

The indexing in the first branch specifies that **C** and `B(:)` are compile-time vectors. Therefore, the code generator applies the indexing rule for indexing one vector with another vector. The orientation of the result is the orientation of the data vector, **C**.

Incompatibility with MATLAB in Vector-Vector Indexing

In MATLAB, the special rule for vector-vector indexing is that the orientation of the result is the orientation of the data vector. For example, if **A** is 1-by-5 and **B** is 3-by-1,

then $A(B)$ is 1-by-3. If, however, the data vector A is a scalar, then the orientation of $A(B)$ is the orientation of the index vector B .

The code generator applies the same vector-vector indexing rules as MATLAB. If A and B are variable-size vectors, to apply the indexing rules, the code generator assumes that the orientation of B matches the orientation of A . At run time, if A is scalar and the orientation of A and B do not match, then the assumption is incorrect. Therefore, when run-time error checks are enabled, a run-time error can occur.

To avoid this issue, make the orientations of the vectors match. Alternatively, index single elements by specifying the row and column. For example, $A(\text{row}, \text{column})$.

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

The following limitation applies to matrix indexing operations for code generation:

- Initialization of the following style:

```
for i = 1:10
    M(i) = 5;
end
```

In this case, the size of M changes as the loop is executed. Code generation does not support increasing the size of an array over time.

For code generation, preallocate M .

```
M = zeros(1,10);
for i = 1:10
    M(i) = 5;
end
```

The following limitation applies to matrix indexing operations for code generation when dynamic memory allocation is disabled:

- $M(i:j)$ where i and j change in a loop

During code generation, memory is not dynamically allocated for the size of the expressions that change as the program executes. To implement this behavior, use `for`-loops as shown:

```
...
```

```
M = ones(10,10);  
for i=1:10  
    for j = i:10  
        M(i,j) = 2*M(i,j);  
    end  
end  
...
```

Note: The matrix *M* must be defined before entering the loop.

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

For code generation, when you concatenate variable-size arrays, the dimensions that are not being concatenated must match exactly.

Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements

Suppose that:

- *c* is a variable-size cell array.
- You access the contents of *c* by using curly braces. For example, *c*{2:4}.
- You include the results in concatenation. For example, [*a* *c*{2:4} *b*].
- *c*{*I*} returns no elements. Either *c* is empty or the indexing inside the curly braces produces an empty result.

For these conditions, MATLAB omits *c*{*I*} from the concatenation. For example, [*a* *c*{*I*} *b*] becomes [*a* *b*]. The code generator treats *c*{*I*} as the empty array [*c*{*I*}]. The concatenation becomes [... [*c*{*i*}] ...]. This concatenation then omits the array [*c*{*I*}]. So that the properties of [*c*{*I*}] are compatible with the concatenation [... [*c*{*i*}] ...], the code generator assigns the class, size, and complexity of [*c*{*I*}] according to these rules:

- The class and complexity are the same as the base type of the cell array.
- The size of the second dimension is always 0.
- For the rest of the dimensions, the size of *N_i* depends on whether the corresponding dimension in the base type is fixed or variable size.

- If the corresponding dimension in the base type is variable size, the dimension has size 0 in the result.
- If the corresponding dimension in the base type is fixed size, the dimension has that size in the result.

Suppose that `c` has a base type with class `int8` and size `10x7x8x?`. In the generated code, the class of `[c{I}]` is `int8`. The size of `[c{I}]` is `0x0x8x0`. The second dimension is 0. The first and last dimensions are 0 because those dimensions are variable size in the base type. The third dimension is 8 because the size of the third dimension of the base type is a fixed size 8.

Inside concatenation, if curly-brace indexing of a variable-size cell array returns no elements, the generated code can have the following differences from MATLAB:

- The class of `[...c{i}...]` in the generated code can differ from the class in MATLAB.

When `c{I}` returns no elements, MATLAB removes `c{I}` from the concatenation. Therefore, `c{I}` does not affect the class of the result. MATLAB determines the class of the result based on the classes of the remaining arrays, according to a precedence of classes. See “Valid Combinations of Unlike Classes” (MATLAB). In the generated code, the class of `[c{I}]` affects the class of the result of the overall concatenation `[...[c{I}]...]` because the code generator treats `c{I}` as `[c{I}]`. The previously described rules determine the class of `[c{I}]`.

- In the generated code, the size of `[c{I}]` can differ from the size in MATLAB.

In MATLAB, the concatenation `[c{I}]` is a `0x0 double`. In the generated code, the previously described rules determine the size of `[c{I}]`.

Variable-Sizing Restrictions for Code Generation of Toolbox Functions

In this section...

“Common Restrictions” on page 27-33

“Toolbox Functions with Restrictions for Variable-Size Data” on page 27-34

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Restrictions for Variable-Size Data” on page 27-34.

Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape $1 \times n$ 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

This restriction applies to functions that take the working dimension (the dimension along which to operate) as input. In MATLAB and in code generation, if you do not supply the working dimension, the function selects it. In MATLAB, the function selects the first dimension whose size does not equal 1. For code generation, the function selects the first dimension that has a variable size or that has a fixed size that does not equal 1. If the working dimension has a variable size and it becomes 1 at run time, then the working dimension is different from the working dimension in MATLAB. Therefore, when run-time error checks are enabled, an error can occur.

For example, suppose that X is a variable-size matrix with dimensions $1 \times 3 \times 5$. In the generated code, `sum(X)` behaves like `sum(X,2)`. In MATLAB, `sum(X)` behaves like `sum(X,2)` unless `size(X,2)` is 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`.

To avoid this issue, specify the intended working dimension explicitly as a constant value. For example, `sum(X,2)`.

Array-to-vector restriction

The function issues an error when a variable-size array that is not a variable-length vector assumes the shape of a vector at run time. To avoid the issue, specify the input explicitly as a variable-length vector instead of a variable-size array.

Array-to-scalar restriction

The function issues an error if a variable-size array assumes a scalar value at run time. To avoid this issue, specify scalars as fixed size.

Toolbox Functions with Restrictions for Variable-Size Data

The following table lists functions that have code generation restrictions for variable-size data. For additional restrictions for these functions, and restrictions for all functions and objects supported for code generation, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” (MATLAB Coder).

Function	Restrictions for Variable-Size Data
all	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33. • 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 27-33. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
bsxfun	<ul style="list-style-type: none"> • Dimensions expand only where one input array or the other has a fixed length of 1.
cat	<ul style="list-style-type: none"> • Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 27-33. • Input vectors must have the same orientation, either both row vectors or both column vectors.

Function	Restrictions for Variable-Size Data
cov	<ul style="list-style-type: none"> For <code>cov(X)</code>, see “Array-to-vector restriction” on page 27-34.
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 27-33.
detrend	<ul style="list-style-type: none"> For first argument for row vectors only, see “Array-to-vector restriction” on page 27-34 .
diag	<ul style="list-style-type: none"> See “Array-to-vector restriction” on page 27-34 .
diff	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 27-33. Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, <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 27-33.
filter	<ul style="list-style-type: none"> For first and second arguments, see “Variable-length vector restriction” on page 27-33. See “Automatic dimension restriction” on page 27-33.
hist	<ul style="list-style-type: none"> For second argument, see “Variable-length vector restriction” on page 27-33. For second input argument, see “Array-to-scalar restriction” on page 27-34.
histc	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 27-33.
ifft	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 27-33.
ind2sub	<ul style="list-style-type: none"> First input (the size vector input) must be fixed size.

Function	Restrictions for Variable-Size Data
interp1	<ul style="list-style-type: none"> • For the xq input, see “Array-to-vector restriction” on page 27-34. • If v becomes a row vector at run time, the array to vector restriction applies. If v becomes a column vector at run time, this restriction does not apply.
ipermute	<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 27-33.
magic	<ul style="list-style-type: none"> • Argument must be a constant. • Output can be fixed-size matrices only.
max	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33.
mean	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
median	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
min	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33.
mode	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions for Variable-Size Data
mtimes	<p>Consider the multiplication $A*B$. If the code generator is aware that A is scalar and B is a matrix, the code generator produces code for scalar-matrix multiplication. However, if the code generator is aware that A and B are variable-size matrices, it produces code for a general matrix multiplication. At run time, if A turns out to be scalar, the generated code does not change its behavior. Therefore, when run-time error checks are enabled, a size mismatch error can occur.</p>
nchoosek	<ul style="list-style-type: none"> • The second input, k, must be a fixed-size scalar. • The second input, k, must be a constant for static allocation. If you enable dynamic allocation, the second input can be a variable. • You cannot create a variable-size array by passing in a variable, k, unless you enable dynamic allocation.
permute	<ul style="list-style-type: none"> • Order input must be fixed-size.
planerot	<ul style="list-style-type: none"> • Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.
poly	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 27-33.
polyfit	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 27-33.
prod	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 27-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
rand	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>rand(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>rand([1 N])</code> may produce a variable-length vector of $1 \times M$ where M is the upper bound on N.

Function	Restrictions for Variable-Size Data
randi	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randi(imax, 1, N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randi(imax, [1 N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
randn	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randn(1, N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randn([1 N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
reshape	<ul style="list-style-type: none"> • If the input is a variable-size array and the output array has at least one fixed-length dimension, do not specify the output dimension sizes in a size vector <code>SZ</code>. Instead, specify the output dimension sizes as scalar values, <code>SZ1, . . . , SZN</code>. Specify fixed-size dimensions as constants. • When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.
roots	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 27-33.
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. Therefore, at run time the number of shifts is constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must have the same number of dimensions when you supply a positive number of shifts.

Function	Restrictions for Variable-Size Data
std	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 27-33.• 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 27-33.• 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 27-33.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
typecast	<ul style="list-style-type: none">• See “Variable-length vector restriction” on page 27-33 on first argument.
var	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 27-33.• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Code Generation for Cell Arrays

- “Code Generation for Cell Arrays” on page 28-2
- “Control Whether a Cell Array Is Variable-Size” on page 28-5
- “Define Cell Array Inputs” on page 28-8
- “Cell Array Limitations for Code Generation” on page 28-9

Code Generation for Cell Arrays

When you generate code from MATLAB code that contains cell arrays, the code generator classifies the cell arrays as *homogeneous* or *heterogeneous*. This classification determines how a cell array is represented in the generated code. It also determines how you can use the cell array in MATLAB code from which you generate code.

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Cell Array Limitations for Code Generation” on page 28-9.

Homogeneous vs. Heterogeneous Cell Arrays

A homogeneous cell array has these characteristics:

- The cell array is represented as an array in the generated code.
- All elements have the same properties. The type associated with the cell array specifies the properties of all elements rather than the properties of individual elements.
- The cell array can be variable-size.
- You can index into the cell array with an index whose value is determined at run time.

A heterogeneous cell array has these characteristics:

- The cell array is represented as a structure in the generated code. Each element is represented as a field of the structure.
- The elements can have different properties. The type associated with the cell array specifies the properties of each element individually.
- The cell array cannot be variable-size.
- You must index into the cell array with a constant index or with `for`-loops that have constant bounds.

The code generator uses heuristics to determine the classification of a cell array as homogeneous or heterogeneous. It considers the properties (class, size, complexity) of the elements and other factors, such as how you use the cell array in your program. Depending on how you use a cell array, the code generator can classify a cell array as homogeneous in one case and heterogeneous in another case. For example, consider the

cell array `{1 [2 3]}`. The code generator can classify this cell array as a heterogeneous 1-by-2 cell array. The first element is double scalar. The second element is a 1-by-2 array of doubles. However, if you index into this cell array with an index whose value is determined at run time, the code generator classifies it as a homogeneous cell array. The elements are variable-size arrays of doubles with an upper bound of 2.

Controlling Whether a Cell Array Is Homogeneous or Heterogeneous

For cell arrays with certain characteristics, you cannot control the classification as homogeneous or heterogeneous:

- If the elements have different classes, the cell array must be heterogeneous.
- If the cell array is variable-size, it must be homogeneous.
- If you index into the cell array with an index whose value is determined at run time, the cell array must be homogeneous.

For other cell arrays, you can control the classification as homogeneous or heterogeneous.

To control the classification of cell arrays that are entry-point function inputs, use the `coder.CellType` methods `makeHomogeneous` or `makeHeterogeneous`.

To control the classification of cell arrays that are not entry-point function inputs:

- If the cell array is fixed-size, you can force an otherwise homogeneous cell array to be heterogeneous by using `coder.cstructname`. For example:

```
function y = mycell()
%#codegen
c = {1 2 3};
coder.cstructname(c, 'myname');
y = c;
end
```

- If the cell array elements have the same class, you can force a cell array to be homogeneous by using `coder.varsizes`. See “Control Whether a Cell Array Is Variable-Size” on page 28-5.

Cell Arrays in Reports

To see whether a cell array is homogeneous or heterogeneous, view the variable in the code generation report.

For a homogeneous cell array, the report has one entry that specifies the properties of all elements. The notation `{:}` indicates that all elements of the cell array have the same properties.

Summary		All Messages (0)		Variables				
Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
1	z	Output	1 x 1	double	No	-	-	-
⊕ 2	n	Input	1 x 1	embedded.fi	No	Signed	16	13
⊖ 3	c	Local	1 x 2	cell	-	-	-	-
3.1	<i>c{:}</i>	Element	1 x 1	double	No	-	-	-

For a heterogeneous cell array, the report has an entry for each element. For example, for a heterogeneous cell array `c` with two elements, the entry for `c{1}` shows the properties for the first element. The entry for `c{2}` shows the properties for the second element.

Summary		All Messages (0)		Variables				
Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
1	z	Output	1 x 1	double	No	-	-	-
⊕ 2	n	I/O	1 x 1	embedded.fi	No	Signed	16	13
⊖ 3	c	Local	1 x 2	cell	-	-	-	-
3.1	<i>c{1}</i>	Element	1 x 1	double	No	-	-	-
3.2	<i>c{2}</i>	Element	1 x 1	char	-	-	-	-

See Also

`coder.CellType` | `coder.cstructname` | `coder.varsizes`

More About

- “Control Whether a Cell Array Is Variable-Size” on page 28-5
- “Cell Array Limitations for Code Generation” on page 28-9
- “Use Fixed-Point Code Generation Reports” on page 14-36

Control Whether a Cell Array Is Variable-Size

The code generator classifies a variable-size cell array as homogeneous. The cell array elements must have the same class. In the generated code, the cell array is represented as an array.

If a cell array is an entry-point function input, to make it variable-size, use `coder.typeof` or `coder.newtype` to create a type for a variable-size cell array. For example, to create a type for a cell array whose first dimension is fixed and whose second dimension has an upper bound of 10, use this code:

```
t = coder.typeof({1 2 3}, [1 10], [0 1])
```

See “Specify Variable-Size Cell Array Inputs” on page 14-58.

If a cell array is not an entry-point function input, to make it variable-size:

- Create the cell array by using the `cell` function. For example:

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

For code generation, when you create a variable-size cell array by using `cell`, you must adhere to certain restrictions. See “Definition of Variable-Size Cell Array by Using `cell`” on page 28-10.

- Grow the cell array. For example:

```
function z = mycell(n)
%#codegen
c = {1 2 3};
for i = 1:n
    c{end + 1} = 1;
end
z = c{n};
end
```

- Force the cell array to be variable-size by using `coder.varsizes`. Consider this code:

```
function y = mycellfun()
%#codegen
c = {1 2 3};
coder.varsize('c', [1 10]);
y = c;
end
```

Without `coder.varsize`, `c` is fixed-size with dimensions 1-by-3. With `coder.varsize`, `c` is variable-size with an upper bound of 10.

Sometimes, using `coder.varsize` changes the classification of a cell array from heterogeneous to homogeneous. Consider this code:

```
function y = mycell()
%#codegen
c = {1 [2 3]};
y = c{2};
end
```

The code generator classifies `c` as heterogeneous because the elements have different sizes. `c` is fixed-size with dimensions 1-by-2. If you use `coder.varsize` with `c`, it becomes homogeneous. For example:

```
function y = mycell()
%#codegen
c = {1 [2 3]};
coder.varsize('c', [1 10], [0 1]);
y = c{2};
end
```

`c` becomes a variable-size homogeneous cell array with dimensions 1-by-:10.

To force `c` to be homogeneous, but not variable-size, specify that none of the dimensions vary. For example:

```
function y = mycell()
%#codegen
c = {1 [2 3]};
coder.varsize('c', [1 2], [0 0]);
y = c{2};
end
```

See Also

`coder.CellType` | `coder.varsize`

More About

- “Code Generation for Cell Arrays” on page 28-2
- “Cell Array Limitations for Code Generation” on page 28-9
- “Code Generation for Variable-Size Arrays” on page 27-2

Define Cell Array Inputs

To define types for cell arrays that are inputs to entry-point functions, use one of these approaches:

Define Types	See
At the command line	“Specify Cell Array Inputs at the Command Line” on page 14-55
Programmatically in the MATLAB file	“Define Input Properties Programmatically in MATLAB File” on page 14-46

See Also

`coder.CellType`

More About

- “Code Generation for Cell Arrays” on page 28-2

Cell Array Limitations for Code Generation

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to these restrictions:

- “Cell Array Element Assignment” on page 28-9
- “Definition of Variable-Size Cell Array by Using `cell`” on page 28-10
- “Cell Array Indexing” on page 28-13
- “Growing a Cell Array by Using `{end + 1}`” on page 28-14
- “Variable-Size Cell Arrays” on page 28-15
- “Cell Array Contents” on page 28-16
- “Cell Arrays in Structures” on page 28-16
- “Passing Cell Arrays to External C/C++ Functions” on page 28-16

Cell Array Element Assignment

You must assign a cell array element on all execution paths before you use it. For example:

```
function z = foo(n)
%#codegen
c = cell(1,3);
if n < 1
    c{2} = 1;

else
    c{2} = n;
end
z = c{2};
end
```

The code generator considers passing a cell array to a function or returning it from a function as a use of all elements of the cell array. Therefore, before you pass a cell array to a function or return it from a function, you must assign all of its elements. For example, the following code is not allowed because it does not assign a value to `c{2}` and `c` is a function output.

```
function c = foo()
```

```
 %#codegen
c = cell(1,3);
c{1} = 1;
c{3} = 3;
end
```

The assignment of values to elements must be consistent on all execution paths. The following code is not allowed because `y{2}` is double on one execution path and char on the other execution path.

```
function y = foo(n)
y = cell(1,3)
if n > 1;
    y{1} = 1;
    y{2} = 2;
    y{3} = 3;
else
    y{1} = 10;
    y{2} = 'a';
    y{3} = 30;
end
```

Definition of Variable-Size Cell Array by Using `cell`

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array. For example:

```
function z = mycell(n, j)
 %#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. If the code generator detects that some elements are not assigned, code generation fails with a message like this message:

Unable to determine that every element of 'y' is assigned before this line.

Sometimes, even though your code assigns all elements of the cell array, the code generator reports this message because the analysis does not detect that all elements are assigned. See “Unable to Determine That Every Element of Cell Array Is Assigned” on page 46-41.

To avoid this error, follow these guidelines:

- When you use `cell` to define a variable-size cell array, write code that follows this pattern:

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

Here is the pattern for a multidimensional cell array:

```
function z = mycell(m,n,p)
%#codegen
x = cell(m,n,p);
for i = 1:m
    for j = 1:n
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

- Increment or decrement the loop counter by 1.
- Define the cell array within one loop or one set of nested loops. For example, this code is not allowed:

```
function z = mycell(n, j)
x = cell(1,n);
for i = 1:5
    x{i} = 5;
```

```
end
for i = 6:n
    x{i} = 5;
end
z = x{j};
end
```

- Use the same variables for the cell dimensions and loop initial and end values. For example, code generation fails for the following code because the cell creation uses `n` and the loop end value uses `m`:

```
function z = mycell(n, j)
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
z = x{j};
end
```

Rewrite the code to use `n` for the cell creation and the loop end value:

```
function z = mycell(n, j)
x = cell(1,n);
for i = 1:n
    x{i} = 2;
end
z = x{j};
end
```

- Create the cell array with this pattern:

```
x = cell(1,n)
```

Do not assign the cell array to a field of a structure or a property of an object. For example, this code is not allowed:

```
myobj.prop = cell(1,n)
for i = 1:n
    ...
end
```

Do not use the `cell` function inside the cell array constructor `{}`. For example, this code is not allowed:

```
x = {cell(1,n)};
```


- The cell array creation and the loop that assigns values to the cell array elements must be together in a unique execution path. For example, the following code is not allowed.

```
function z = mycell(n)
if n > 3
    c = cell(1,n);
else
    c = cell(n,1);
end
for i = 1:n
    c{i} = i;
end
z = c{n};
end
```

To fix this code, move the assignment loop inside the code block that creates the cell array.

```
function z = cellerr(n)
if n > 3
    c = cell( 1,n);
    for i = 1:n
        c{i} = i;
    end
else
    c = cell(n,1);
    for i = 1:n
        c{i} = i;
    end
end
z = c{n};
end
```

Cell Array Indexing

- You cannot index cell arrays by using smooth parentheses (). Consider indexing cell arrays by using curly braces {} to access the contents of the cell.
- You must index into heterogeneous cell arrays by using constant indices or by using for-loops with constant bounds.

For example, the following code is not allowed.

```
x = {1, 'mytext'};  
disp(x{randi});
```

You can index into a heterogeneous cell array in a `for`-loop with constant bounds because the code generator unrolls the loop. Unrolling creates a separate copy of the loop body for each loop iteration, which makes the index in each loop iteration constant. However, if the `for`-loop has a large body or it has many iterations, the unrolling can increase compile time and generate inefficient code.

If `A` and `B` are constant, the following code shows indexing into a heterogeneous cell array in a `for`-loop with constant bounds.

```
x = {1, 'mytext'};  
for i = A:B  
    disp(x{i});  
end
```

Growing a Cell Array by Using `{end + 1}`

To grow a cell array `X`, you can use `X{end + 1}`. For example:

```
...  
X = {1 2};  
X{end + 1} = 'a';  
...
```

When you use `{end + 1}` to grow a cell array, follow these restrictions:

- Use only `{end + 1}`. Do not use `{end + 2}`, `{end + 3}`, and so on.
- Use `{end + 1}` with vectors only. For example, the following code is not allowed because `X` is a matrix, not a vector:

```
...  
X = {1 2; 3 4};  
X{end + 1} = 5;
```

- Use `{end + 1}` only with a variable. In the following code, `{end + 1}` does not cause `{1 2 3}` to grow. In this case, the code generator treats `{end + 1}` as an out-of-bounds index into `X{2}`.

```
...
```

```

X = {'a' { 1 2 3 }};
X{2}{end + 1} = 4;
...

```

- When `{end + 1}` grows a cell array in a loop, the cell array must be variable-size. Therefore, the cell array must be homogeneous.

This code is allowed because X is homogeneous.

```

...
X = {1 2};
for i=1:n
    X{end + 1} = 3;
end
...

```

This code is not allowed because X is heterogeneous.

```

...
X = {1 'a' 2 'b'};
for i=1:n
    X{end + 1} = 3;
end
...

```

Variable-Size Cell Arrays

- Heterogeneous cell arrays cannot be variable-size.
- If you use `coder. varsizes` to make a variable-size cell array, define the cell array with curly braces. For example:

```

...
c = {1 [2 3]};
coder. varsizes('c')
...

```

Do not use the `cell` function. For example, this code is not allowed:

```

...
c = cell(1,3);
coder. varsizes('c')
...

```

Cell Array Contents

Cell arrays cannot contain `mxarrays`. In a cell array, you cannot store a value that an extrinsic function returns.

Cell Arrays in Structures

Structures cannot contain cell arrays.

Passing Cell Arrays to External C/C++ Functions

You cannot pass a cell array to `coder.ceval`. If a variable is an input argument to `coder.ceval`, define the variable as an array or structure instead of as a cell array.

More About

- “Code Generation for Cell Arrays” on page 28-2
- “Differences Between Generated Code and MATLAB Code” on page 21-9

Primary Functions

Specify Properties of Entry-Point Function Inputs

In this section...

“Why You Must Specify Input Properties” on page 29-2

“Properties to Specify” on page 29-2

“Rules for Specifying Properties of Primary Inputs” on page 29-4

“Methods for Defining Properties of Primary Inputs” on page 29-4

“Define Input Properties by Example at the Command Line” on page 29-5

“Specify Constant Inputs at the Command Line” on page 29-7

“Specify Variable-Size Inputs at the Command Line” on page 29-9

Why You Must Specify Input Properties

Fixed-Point Designer must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, Fixed-Point Designer 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 Fixed-Point Designer. If your primary function has no input parameters, Fixed-Point Designer 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.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For	Specify properties				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	Specify properties for each field according to its class When a primary input is a structure, the code generator treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:				

For	Specify properties				
	<ul style="list-style-type: none"> For each field of input structures, specify class, size, and complexity. For each field that is fixed-point class, also specify <code>numerictype</code>, and <code>fimath</code>. 				
Other inputs	✓	✓	✓		

Default Property Values

Fixed-Point Designer assigns the following default values for properties of primary function inputs.

Property	Default
<code>class</code>	<code>double</code>
<code>size</code>	<code>scalar</code>
<code>complexity</code>	<code>real</code>
<code>numerictype</code>	No default
<code>fimath</code>	MATLAB default <code>fimath</code> object

Supported Classes

The following table presents the class names supported by Fixed-Point Designer.

Class Name	Description
<code>logical</code>	Logical array of true and false values
<code>char</code>	Character array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>int64</code>	64-bit signed integer array
<code>uint64</code>	64-bit unsigned integer array

Class Name	Description
<code>single</code>	Single-precision floating-point or fixed-point number array
<code>double</code>	Double-precision floating-point or fixed-point number array
<code>struct</code>	Structure array
<code>embedded.fi</code>	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules:

- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature. For example, the first element in the cell array defines the properties of the first primary function input.
- To generate fewer arguments than those arguments that occur in the MATLAB function, specify properties for only the number of arguments that you want in the generated function.
- If the MATLAB function has input arguments, to generate a function that has no input arguments, pass an empty cell array to `-args`.
- For each primary function input whose class is fixed point (`fi`), specify the input `numericity` and `fimath` properties.
- For each primary function input whose class is `struct`, specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
	<ul style="list-style-type: none"> • If you are working in a project, easy to use • Does not alter original MATLAB code • 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

Method	Advantages	Disadvantages
<p>“Define Input Properties by Example at the Command Line” on page 29-5</p> <hr/> <p>Note: If you define input properties programmatically in the MATLAB file, you cannot use this method</p>	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke <code>fiaccel</code> (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
<p>“Define Input Properties Programmatically in the MATLAB File” (MATLAB Coder)</p>	<ul style="list-style-type: none"> • Integrated with MATLAB code; no need to redefine properties each time you invoke • 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 • 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 29-5
- “Rules for Using the `-args` Option” on page 29-6
- “Specifying Properties of Primary Inputs by Example” on page 29-6
- “Specifying Properties of Primary Fixed-Point Inputs by Example” on page 29-7

Command-Line Option `-args`

The `fiaccel` 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 `fiaccel`.

For information about specifying cell array inputs, see “Specify Cell Array Inputs at the Command Line” (MATLAB Coder).

Rules for Using the `-args` Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature. For example, the first element in the cell array defines the properties of the first primary function input.
- To generate fewer arguments than those arguments that occur in the MATLAB function, specify properties for only the number of arguments that you want in the generated function.
- If the MATLAB function has input arguments, to generate a function that has no input arguments, pass an empty cell array to `-args`.
- For each primary function input whose class is fixed point (`fi`), specify the input `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, specify the properties of each of its fields in the order that they appear in the structure definition.

Specifying Properties of Primary Inputs by Example

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
```

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 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;
```

Specify Constant Inputs at the Command Line

If you know that your primary inputs do not change at run time, you can reduce overhead in the generated code by specifying that the primary inputs are constant values.

Constant inputs are commonly used for flags that control how an algorithm executes and values that specify the sizes or types of data.

To specify that inputs are constants, use the `-args` command-line option with a `coder.Constant` object. To specify that an input is a constant with the size, class, complexity, and value of `constant_input`, use the following syntax:

```
-args {coder.Constant(constant_input)}
```

Calling Functions with Constant Inputs

`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.

Specifying a Structure as a Constant Input

Suppose that you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix, 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)
```

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.

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 = numericitype(1,32,24);
if ((mod(numel(A),n) == 0) && ...
    (n>=1 && n<=numel(A)))
    B = fi(zeros(1,numel(A)/n),Tb);
    k = 1;
    for i = 1 : numel(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,'SumMode','KeepLSB'),[1 100],1),0}...
-report
```

- 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,'SumMode','KeepLSB'),[1 100],1)
```

```
fiaccel nway -args {vareg, double(0)} -report
```

More About

- “Specify Objects as Inputs” on page 17-29
- “Specify Cell Array Inputs at the Command Line” on page 14-55
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3

Define Input Properties Programmatically in the MATLAB File

For code generation, you can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

How to Use `assert` with MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

When specifying input properties using the `assert` function, use one of the following methods. Use the exact syntax that is provided; do not modify it.

- “Specify Any Class” on page 29-12
- “Specify `fi` Class” on page 29-13
- “Specify Structure Class” on page 29-13
- “Specify Cell Array Class” on page 29-14
- “Specify Fixed Size” on page 29-14
- “Specify Scalar Size” on page 29-14
- “Specify Upper Bounds for Variable-Size Inputs” on page 29-15
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 29-15
- “Specify Size of Individual Dimensions” on page 29-16
- “Specify Real Input” on page 29-16
- “Specify Complex Input” on page 29-16
- “Specify `numerictype` of Fixed-Point Input” on page 29-17
- “Specify `fimath` of Fixed-Point Input” on page 29-17
- “Specify Multiple Properties of Input” on page 29-17

Specify Any Class

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter `param` to the MATLAB class `class_name`. For example, to set the class of input `U` to a 32-bit signed integer, call:


```
...
assert(isa(U, 'int32'));
...
```

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U, 'embedded.fi'));
...
```

You must specify both the `fi` class and the `numerictype`. See “Specify `numerictype` of Fixed-Point Input” on page 29-17. You can also set the `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 29-17. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.

Specify Structure Class

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U, 'struct'));
```

...

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify Cell Array Class

```
assert(iscell( param ))
assert(isa( param, 'cell' ))
```

Sets the input parameter *param* to the MATLAB class `cell` (cell array). For example, to set the class of input `C` to a `cell`, call:

```
...
assert(iscell(C));
...
```

or

```
...
assert(isa(C, 'cell'));
...
```

To specify the properties of cell array elements, see “Specifying Properties of Cell Arrays” on page 29-20.

Specify Fixed Size

```
assert ( all ( size ( param ) == [ dims ] ) )
```

Sets the input parameter *param* to the size that dimensions *dims* specifies. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar ( param ) )
assert ( all ( size ( param ) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. To set the size of input `U` to scalar, call:

```

...
assert(isscalar(U));
...
or
...
assert(all(size(U)== [1]));
...

```

Specify Upper Bounds for Variable-Size Inputs

```

assert ( all(size(param)<=[N0 N1 ...]));
assert ( all(size(param)<[N0 N1 ...]));

```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input *U* to be less than or equal to a 3-by-2 matrix, call:

```

assert(all(size(U)<=[3 2]));

```

Note: You can also specify upper bounds for variable-size inputs using `coder. varsize`.

Specify Inputs with Fixed- and Variable-Size Dimensions

```

assert ( all(size(param)>=[M0 M1 ...]));
assert ( all(size(param)<=[N0 N1 ...]));

```

When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, *k*, the lower-bound *M_k* must be less than or equal to the upper-bound *N_k*.
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be nonnegative.

To fix the size of the first dimension of input *U* to 3 and set the second dimension as variable size with upper bound of 2, call:

```

assert(all(size(U)>=[3 0]));

```

```
assert(all(size(U)<=[3 2]));
```

Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);  
assert (size(param, k)<=Nk);  
assert (size(param, k)<Nk);
```

You can specify individual dimensions and all dimensions simultaneously. You can also specify individual dimensions instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension *k* of input parameter *param*. To set the upper-bound size of the first dimension of input *U* to 3, call:

```
assert(size(U,1)<=3)
```

To fix the size of the second dimension of input *U* to 2, call:

```
assert(size(U,2)==2)
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *param* is real. To specify that input *U* is real, call:

```
...  
assert(isreal(U));  
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *param* is complex. To specify that input *U* is complex, call:

```
...  
assert(~isreal(U));  
...
```

Specify numericity of Fixed-Point Input

```
assert ( isequal ( numericity ( fiparam ), T ) )
```

Sets the `numericity` properties of `fi` input parameter *fiparam* to the `numericity` object *T*. For example, to specify the `numericity` property of fixed-point input *U* as a signed `numericity` object *T* with 32-bit word length and 30-bit fraction length, use the following code:

```
%#codegen
...
% Define the numericity object.
T = numericity(1, 32, 30);

% Set the numericity property of input U to T.
assert(isequal(numericity(U),T));
...
```

Specifying the `numericity` for a variable does not automatically specify that the variable is fixed point. You must specify both the `fi` class and the `numericity`.

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *fiparam* to the `fimath` object *F*. For example, to specify the `fimath` property of fixed-point input *U* so that it saturates on integer overflow, use the following code:

```
%#codegen
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

If you do not specify the `fimath` properties using `assert`, `codegen` uses the MATLAB default `fimath` value.

Specify Multiple Properties of Input

```
assert ( function1 ( params ) &&
         function2 ( params ) &&
```

```
function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
 %#codegen
 ...
 assert(isa(U,'double') &&
        ~isreal(U) &&
        all(size(U) == [3 3]) &&
        isa(V,'uint16'));
 ...
```

Rules for Using `assert` Function

When using the `assert` function to specify the properties of primary function inputs, follow these rules:

- Call `assert` functions at the beginning of the primary function, before control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- For a fixed-point input, you must specify both the `fi` class and the `numericity`. See “Specify `numericity` of Fixed-Point Input” on page 29-17. You can also set the `fimath` properties. See “Specify `fimath` of Fixed-Point Input” on page 29-17. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
 - You must also specify an upper-bound size for each dimension of the input parameter.
 - For each dimension, k , the lower-bound M_k must be less than or equal to the upper-bound N_k .
 - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
 - Bounds must be nonnegative.

- If you specify individual dimensions, the following rules apply:
 - You must specify the size of each dimension at least once.
 - The last dimension specification takes precedence over earlier specifications.

Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs.

Input	Property	Value
pennywhistle	class	int16
	size	220500-by-1 vector
	complexity	real (by default)
win	class	double
	size	1024-by-1 vector
	complexity	real (by default)

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...

```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...

```

Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Designer software.

In the following example, the primary MATLAB function `mcsqrtfi` takes one fixed-point input `x`. The code specifies the following properties for this input.

Property	Value
<code>class</code>	<code>fi</code>
<code>numerictype</code>	numerictype object T, as specified in the primary function
<code>fimath</code>	fimath object F, 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);
```

You must specify both the `fi` class and the `numerictype`.

Specifying Properties of Cell Arrays

To specify the MATLAB class `cell` (cell array), use one of the following syntaxes:

```
assert(iscell(param))
assert(isa(param,'cell'))
```

For example, to set the class of input `C` to `cell`, use:

```
...
assert(iscell(C));
```



```
...
```

```
or
```

```
...  
assert(isa(C, 'cell'));  
...
```

You can also specify the size of the cell array and the properties of the cell array elements. The number of elements that you specify determines whether the cell array is homogeneous or heterogeneous. See “Code Generation for Cell Arrays” (MATLAB Coder).

If you specify the properties of the first element only, the cell array is homogeneous. For example, the following code specifies that **C** is a 1x3 homogeneous cell array whose elements are 1x1 double.

```
...  
assert(isa(C, 'cell'));  
assert(all(size(C) == [1 3]));  
assert(isa(C{1}, 'double'));  
...
```

If you specify the properties of the first element only, but also assign a structure type name to the cell array, the cell array is heterogeneous. Each element has the properties of the first element. For example, the following code specifies that **C** is a 1x3 heterogeneous cell array. Each element is a 1x1 double.

```
...  
assert(isa(C, 'cell'));  
assert(all(size(C) == [1 3]));  
assert(isa(C{1}, 'double'));  
coder.cstructname(C, 'myname');  
...
```

If you specify the properties of each element, the cell array is heterogeneous. For example, the following code specifies a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x3 double.

```
...  
assert(isa(C, 'cell'));  
assert(all(size(C) == [1 2]));  
assert(isa(C{1}, 'char'));  
assert(all(size(C{2}) == [1 3]));  
assert(isa(C{2}, 'double'));  
...
```

```
...
```

If you specify more than one element, you cannot specify that the cell array is variable size, even if all elements have the same properties. For example, the following code specifies a variable-size cell array. Because the code specifies the properties of the first and second elements, code generation fails.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
assert(isa(C{2}, 'double'));
...
```

In the previous example, if you specify the first element only, you can specify that the cell array is variable-size. For example:

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
...
```

Specifying Class and Size of Scalar Structure

Suppose that you define `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

The following code specifies the properties of the function input `S` and its fields:

```
function y = fcn(S) %#codegen

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));
...
```

In most cases, when you do not explicitly specify values for properties, MATLAB Coder uses defaults—except for structure fields. The only way to name a field in a structure

is to set at least one of its properties. At a minimum, you must specify the class of a structure field.

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume that you have defined **S** as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input **S** by using the first element of the array:

```
 %#codegen  
function y = fcn(S)  
  
% Specify the class of the input S as struct.  
assert(isstruct(S));  
  
% Specify the size of the fields r and i  
% based on the first element of the array.  
assert(all(size(S) == [1 2]));  
assert(isa(S(1).r,'double'));  
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. At a minimum, you must specify the class of all fields.

System Objects Supported for Code Generation

Code Generation for System Objects

You can generate C and C++ code for a subset of System objects provided by the following toolboxes.

Toolbox Name	See
Communications System Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)
Computer Vision System Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)
DSP System Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)
Image Acquisition Toolbox	<ul style="list-style-type: none"> • <code>imaq.VideoDevice</code> • “Code Generation with VideoDevice System Object” (Image Acquisition Toolbox)
Phased Array System Toolbox	“Code Generation” (Phased Array System Toolbox)
System Identification Toolbox	“Generate Code for Online Parameter Estimation in MATLAB” (System Identification Toolbox)
WLAN System Toolbox	“System Objects in MATLAB Code Generation” (MATLAB Coder)

To use these System objects, you need to install the requisite toolbox. For a list of System objects supported for C and C++ code generation, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 26-2 and “Functions and Objects Supported for C/C++ Code Generation — Category List” on page 26-67.

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative

computations or stream data processing. This enables efficient processing of long data sets. For general information about MATLAB objects, see “Object-Oriented Programming” (MATLAB).

Fixed-Point Designer for Simulink Models

Getting Started

- “Installation” on page 31-2
- “Sharing Fixed-Point Models” on page 31-3
- “Physical Quantities and Measurement Scales” on page 31-4
- “Why Use Fixed-Point Hardware?” on page 31-12
- “Why Use the Fixed-Point Designer Software?” on page 31-14
- “Developing and Testing Fixed-Point Systems” on page 31-15
- “Supported Data Types” on page 31-18
- “Configure Blocks with Fixed-Point Output” on page 31-19
- “Configure Blocks with Fixed-Point Parameters” on page 31-29
- “Pass Fixed-Point Data Between Simulink Models and MATLAB” on page 31-32
- “Cast from Doubles to Fixed Point” on page 31-36

Installation

To determine if the Fixed-Point Designer software is installed on your system, type
`ver`

at the MATLAB command line. When you enter this command, the MATLAB Command Window displays information about the version of MATLAB software you are running, including a list of installed add-on products and their version numbers. Check the list to see if the Fixed-Point Designer software appears.

For information about installing this product, refer to the installation documentation.

If you experience installation difficulties and have Web access, look for the installation and license information at the MathWorks Web site (<http://www.mathworks.com/support>).

Sharing Fixed-Point Models

You can edit a model containing fixed-point blocks without the Fixed-Point Designer software. However, you must have a Fixed-Point Designer software license to

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types
- Run a model containing fixed-point data types
- Generate code from a model containing fixed-point data types
- Log the minimum and maximum values produced by a simulation
- Automatically scale the output of a model

If you do not have the Fixed-Point Designer software, you can work with a model containing Simulink blocks with fixed-point settings as follows:

- 1 From the Simulink model **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.

In the Fixed-Point Tool:

- In the **Model Hierarchy** pane, select the root model.
 - Set the **Fixed-point instrumentation mode** parameter to **Force Off**.
 - Set the **Data type override** parameter to **Double** or **Single**.
 - Set the **Data type override applies to** parameter to **All numeric types**.
- 2 If you use **fi** objects or embedded numeric data types in your model, set the **fipref** **DataTypeOverride** property to **TrueDoubles** and the **DataTypeOverrideAppliesTo** property to **All numeric types**.

At the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ...  
          'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

Note: If you use **fi** objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set **fipref** to prevent the checkout of a Fixed-Point Designer license.

Physical Quantities and Measurement Scales

In this section...
“Introduction” on page 31-4
“Selecting a Measurement Scale” on page 31-5
“Select a Measurement Scale for Temperature” on page 31-6

Introduction

The decision to use fixed-point hardware is simply a choice to represent numbers in a particular form. This representation often offers advantages in terms of the power consumption, size, memory usage, speed, and cost of the final product.

A measurement of a physical quantity can take many numerical forms. For example, the boiling point of water is 100 degrees Celsius, 212 degrees Fahrenheit, 373 kelvin, or 671.4 degrees Rankine. No matter what number is given, the physical quantity is exactly the same. The numbers are different because four different scales are used.

Well known standard scales like Celsius are very convenient for the exchange of information. However, there are situations where it makes sense to create and use unique nonstandard scales. These situations usually involve making the most of a limited resource.

For example, nonstandard scales allow map makers to get the maximum detail on a fixed size sheet of paper. A typical road atlas of the USA will show each state on a two-page display. The scale of inches to miles will be unique for most states. By using a large ratio of miles to inches, all of Texas can fit on two pages. Using the same scale for Rhode Island would make poor use of the page. Using a much smaller ratio of miles to inches would allow Rhode Island to be shown with the maximum possible detail.

Fitting measurements of a variable inside an embedded processor is similar to fitting a state map on a piece of paper. The map scale should allow all the boundaries of the state to fit on the page. Similarly, the binary scale for a measurement should allow the maximum and minimum possible values to fit. The map scale should also make the most of the paper in order to get maximum detail. Similarly, the binary scale for a measurement should make the most of the processor in order to get maximum precision.

Use of standard scales for measurements has definite compatibility advantages. However, there are times when it is worthwhile to break convention and use a unique

nonstandard scale. There are also occasions when a mix of uniqueness and compatibility makes sense. See the sections that follow for more information.

Selecting a Measurement Scale

Suppose that you want to make measurements of the temperature of liquid water, and that you want to represent these measurements using 8-bit unsigned integers. Fortunately, the temperature range of liquid water is limited. No matter what scale you use, liquid water can only go from the freezing point to the boiling point. Therefore, this is the range of temperatures that you must capture using just the 256 possible 8-bit values: 0,1,2,...,255.

One approach to representing the temperatures is to use a standard scale. For example, the units for the integers could be Celsius. Hence, the integers 0 and 100 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives a trivial conversion from the integers to degrees Celsius. On the downside, the numbers 101 to 255 are unused. By using this standard scale, more than 60% of the number range has been wasted.

A second approach is to use a nonstandard scale. In this scale, the integers 0 and 255 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives maximum precision since there are 254 values between freezing and boiling instead of just 99. On the downside, the units are roughly 0.3921568 degree Celsius per bit so the conversion to Celsius requires division by 2.55, which is a relatively expensive operation on most fixed-point processors.

A third approach is to use a “semistandard” scale. For example, the integers 0 and 200 could represent water at the freezing point and at the boiling point, respectively. The units for this scale are 0.5 degrees Celsius per bit. On the downside, this scale doesn't use the numbers from 201 to 255, which represents a waste of more than 21%. On the upside, this scale permits relatively easy conversion to a standard scale. The conversion to Celsius involves division by 2, which is a very easy shift operation on most processors.

Measurement Scales: Beyond Multiplication

One of the key operations in converting from one scale to another is multiplication. The preceding case study gave three examples of conversions from a quantized integer value Q to a real-world Celsius value V that involved only multiplication:

$$V = \begin{cases} \frac{100^\circ\text{C}}{100} Q_1 & \text{Conversion 1} \\ \frac{100^\circ\text{C}}{255} Q_2 & \text{Conversion 2} \\ \frac{100^\circ\text{C}}{200} Q_3 & \text{Conversion 3} \end{cases}$$

Graphically, the conversion is a line with slope S , which must pass through the origin. A line through the origin is called a purely linear conversion. Restricting yourself to a purely linear conversion can be very wasteful and it is often better to use the general equation of a line:

$$V = SQ + B.$$

By adding a bias term B , you can obtain greater precision when quantizing to a limited number of bits.

The general equation of a line gives a very useful conversion to a quantized scale. However, like all quantization methods, the precision is limited and errors can be introduced by the conversion. The general equation of a line with quantization error is given by

$$V = SQ + B \pm \text{Error}.$$

If the quantized value Q is rounded to the nearest representable number, then

$$-\frac{S}{2} \leq \text{Error} \leq \frac{S}{2}.$$

That is, the amount of quantization error is determined by both the number of bits and by the scale. This scenario represents the best-case error. For other rounding schemes, the error can be twice as large.

Select a Measurement Scale for Temperature

On typical electronically controlled internal combustion engines, the flow of fuel is regulated to obtain the desired ratio of air to fuel in the cylinders just prior to

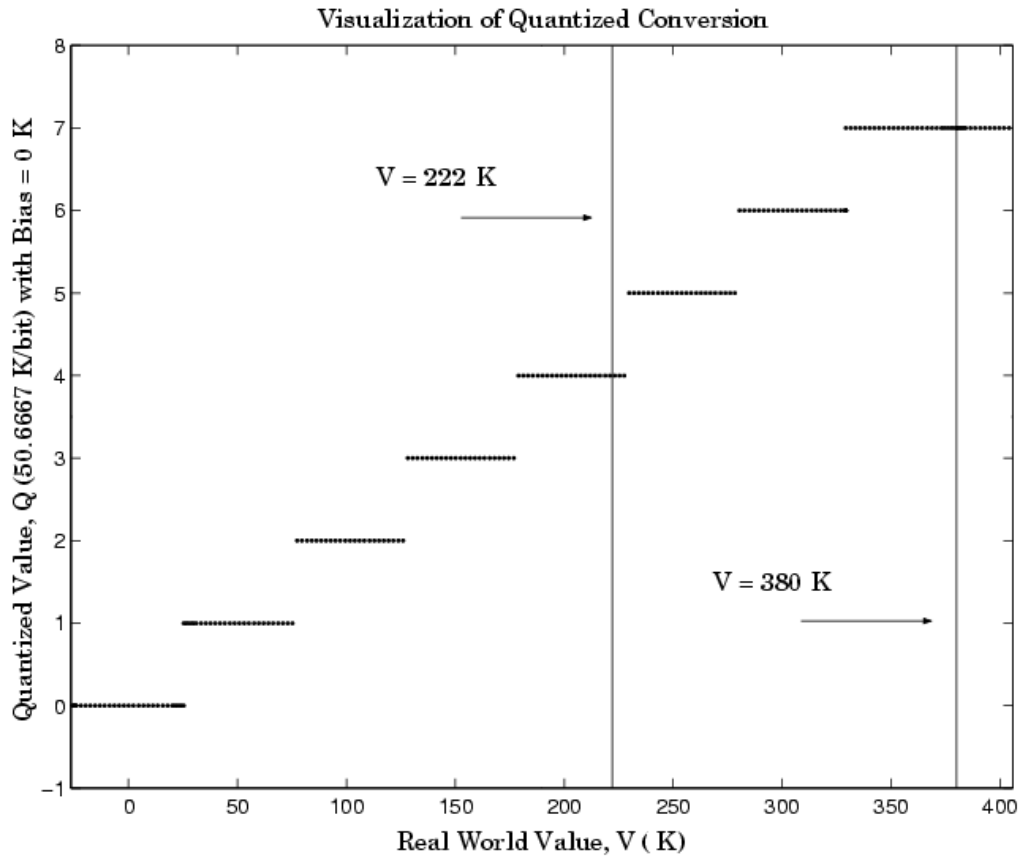
combustion. Therefore, knowledge of the current air flow rate is required. Some manufacturers use sensors that directly measure air flow, while other manufacturers calculate air flow from measurements of related signals. The relationship of these variables is derived from the ideal gas equation. The ideal gas equation involves division by air temperature. For proper results, an absolute temperature scale such as kelvin or Rankine must be used in the equation. However, quantization directly to an absolute temperature scale would cause needlessly large quantization errors.

The temperature of the air flowing into the engine has a limited range. On a typical engine, the radiator is designed to keep the block below the boiling point of the cooling fluid. Assume a maximum of 225°F (380 K). As the air flows through the intake manifold, it can be heated to this maximum temperature. For a cold start in an extreme climate, the temperature can be as low as -60°F (222 K). Therefore, using the absolute kelvin scale, the range of interest is 222 K to 380 K.

The air temperature needs to be quantized for processing by the embedded control system. Assuming an unrealistic quantization to 3-bit unsigned numbers: 0,1,2,...,7, the purely linear conversion with maximum precision is

$$V = \frac{380 \text{ K}}{7.5 \text{ bit}} Q.$$

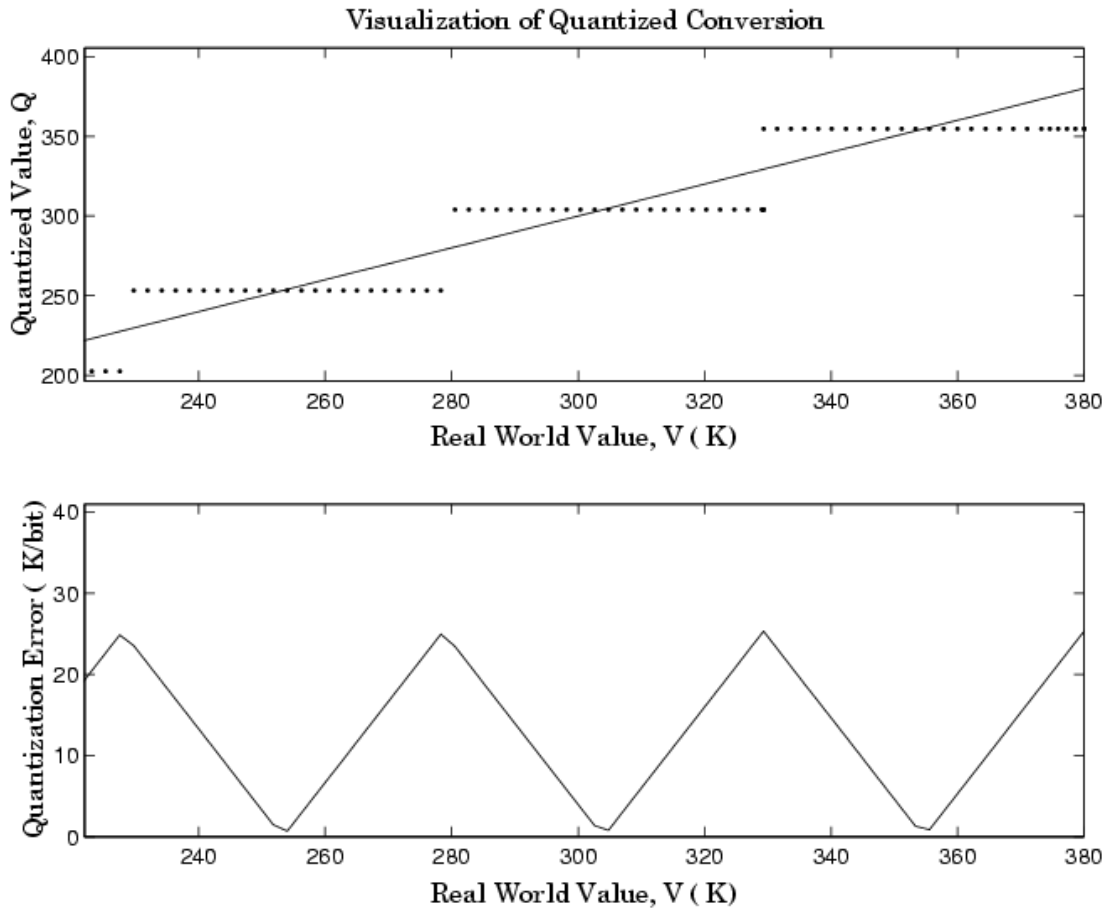
The quantized conversion and range of interest are shown in the following figure.



Notice that there are 7.5 possible quantization values. This is because only half of the first bit corresponds to temperatures (real-world values) greater than zero.

The quantization error is $-25.33 \text{ K/bit} \leq \text{Error} \leq 25.33 \text{ K/bit}$.

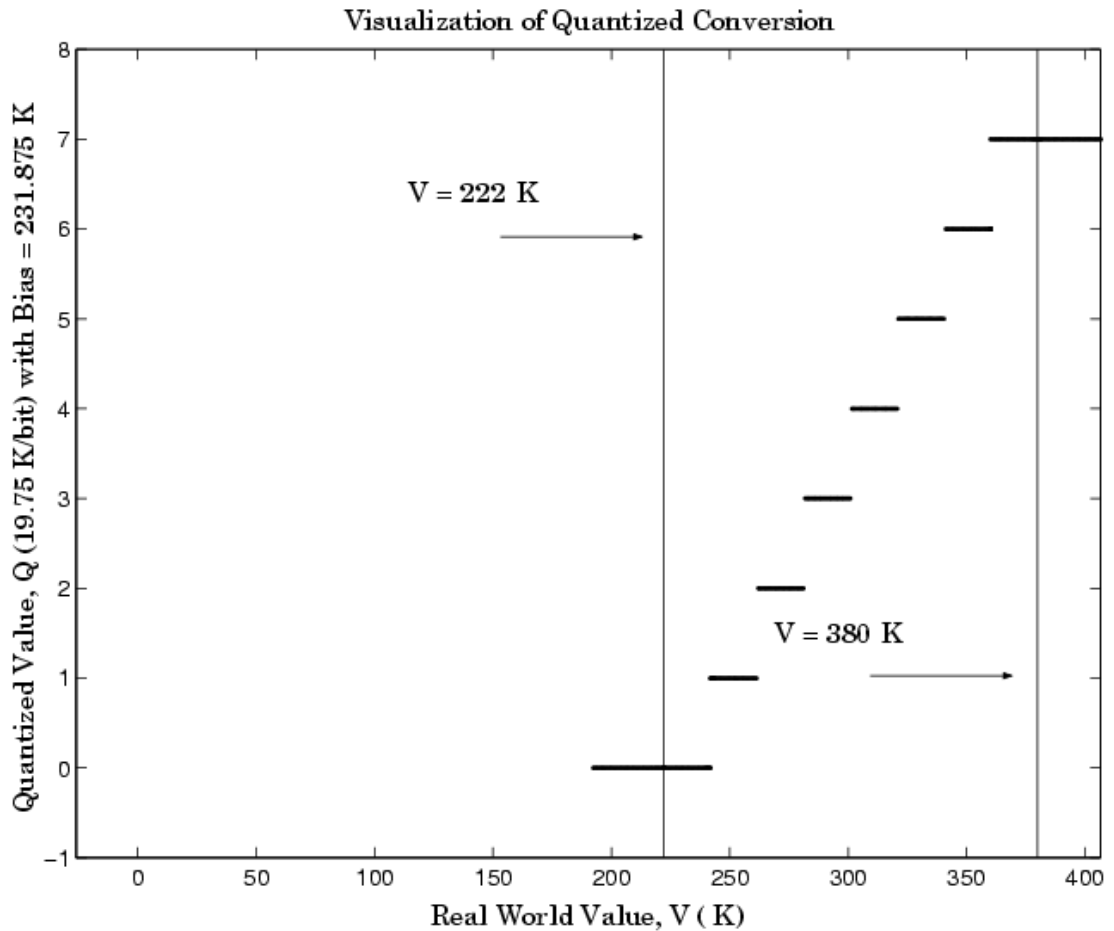
The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



As an alternative to the purely linear conversion, consider the general linear conversion with maximum precision:

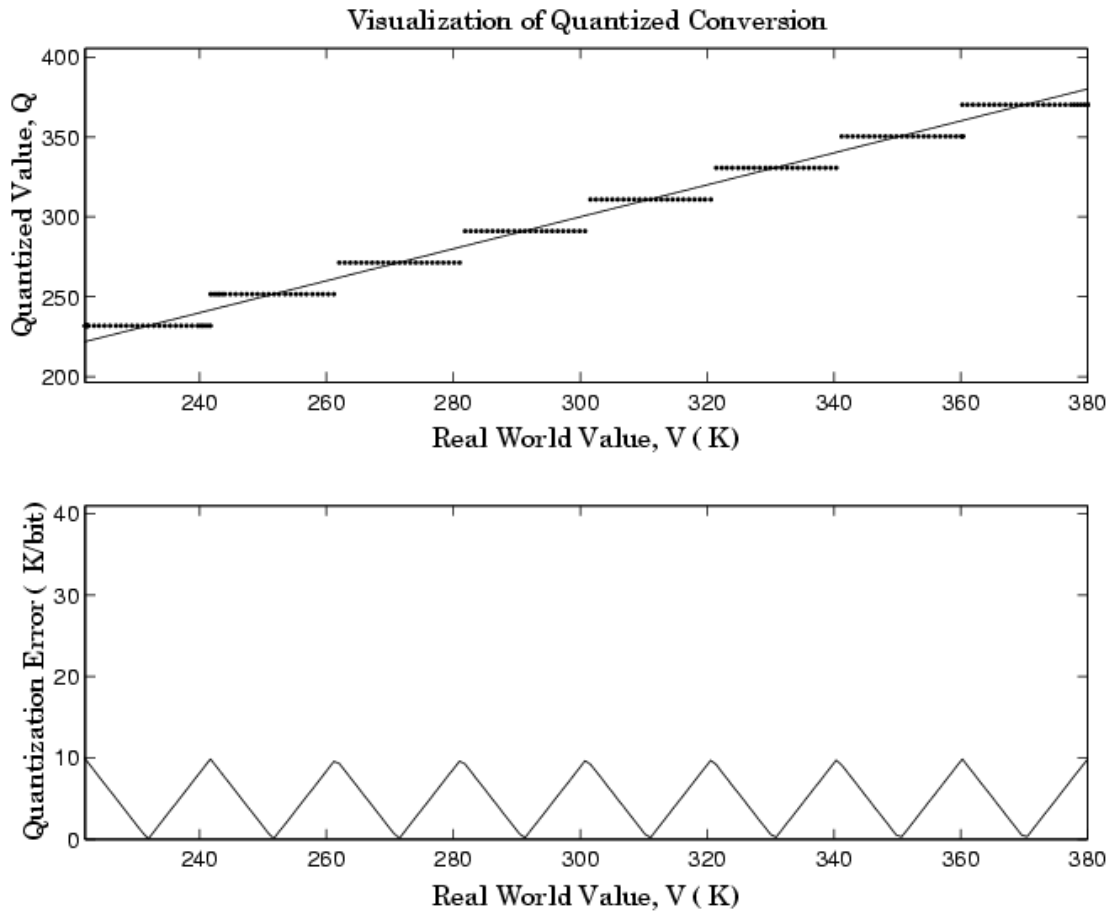
$$V = \left(\frac{380 \text{ K} - 222 \text{ K}}{8} \right) Q + 222 \text{ K} + 0.5 \left(\frac{380 \text{ K} - 222 \text{ K}}{8} \right)$$

The quantized conversion and range of interest are shown in the following figure.



The quantization error is $-9.875 \text{ K/bit} \leq \text{Error} \leq 9.875 \text{ K/bit}$, which is approximately 2.5 times smaller than the error associated with the purely linear conversion.

The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



Clearly, the general linear scale gives much better precision than the purely linear scale over the range of interest.

More About

- "Fixed-Point Data Types" on page 1-2
- "Scaling" on page 1-4

Why Use Fixed-Point Hardware?

Digital hardware is becoming the primary means by which control systems and signal processing filters are implemented. Digital hardware can be classified as either off-the-shelf hardware (for example, microcontrollers, microprocessors, general-purpose processors, and digital signal processors) or custom hardware. Within these two types of hardware, there are many architecture designs. These designs range from systems with a single instruction, single data stream processing unit to systems with multiple instruction, multiple data stream processing units.

Within digital hardware, numbers are represented as either fixed-point or floating-point data types. For both these data types, word sizes are fixed at a set number of bits. However, the dynamic range of fixed-point values is much less than floating-point values with equivalent word sizes. Therefore, in order to avoid overflow or unreasonable quantization errors, fixed-point values must be scaled. Since floating-point processors can greatly simplify the real-time implementation of a control law or digital filter, and floating-point numbers can effectively approximate real-world numbers, then why use a microcontroller or processor with fixed-point hardware support?

- **Size and Power Consumption** — The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. For example, consider a portable telephone where one of the product design goals is to make it as portable (small and light) as possible. If one of today's high-end floating-point, general-purpose processors is used, a large heat sink and battery would also be needed, resulting in a costly, large, and heavy portable phone.
- **Memory Usage and Speed** — In general fixed-point calculations require less memory and less processor time to perform.
- **Cost** — Fixed-point hardware is more cost effective where price/cost is an important consideration. When digital hardware is used in a product, especially mass-produced products, fixed-point hardware costs much less than floating-point hardware and can result in significant savings.

After making the decision to use fixed-point hardware, the next step is to choose a method for implementing the dynamic system (for example, control system or digital filter). Floating-point software emulation libraries are generally ruled out because of timing or memory size constraints. Therefore, you are left with fixed-point math where binary integer values are scaled.

More About

- “Fixed-Point Data Types” on page 1-2

Why Use the Fixed-Point Designer Software?

The Fixed-Point Designer software allows you to efficiently design control systems and digital filters that you will implement using fixed-point arithmetic. With the Fixed-Point Designer software, you can construct Simulink and Stateflow models that contain detailed fixed-point information about your systems. You can then perform bit-true simulations with the models to observe the effects of limited range and precision on your designs.

You can configure the Fixed-Point Tool to automatically log the overflows, saturations, and signal extremes of your simulations. You can also use it to automate data typing and scaling decisions and to compare your fixed-point implementations against idealized, floating-point benchmarks.

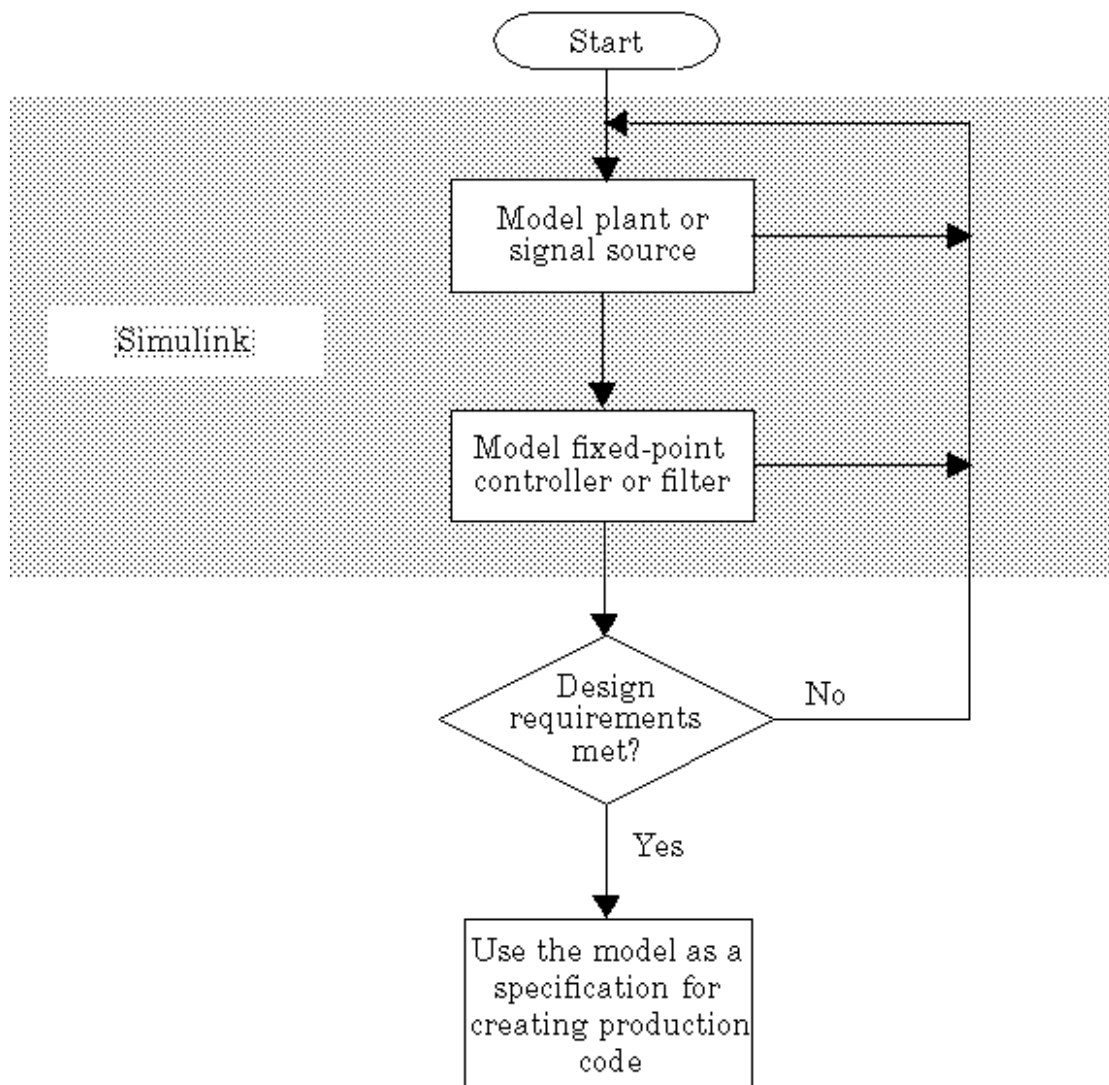
You can use the Fixed-Point Designer software with the Simulink Coder product to automatically generate efficient, integer-only C code representations of your designs. You can use this C code in a production target or for rapid prototyping. In addition, you can use the Fixed-Point Designer software with the Embedded Coder product to generate real-time C code for use on an integer production, embedded target. You can also use Fixed-Point Designer with HDL Coder to generate portable, synthesizable VHDL and Verilog code from Simulink models and Stateflow charts.

More About

- “Why Use Fixed-Point Hardware?” on page 31-12

Developing and Testing Fixed-Point Systems

The Fixed-Point Designer software provides tools that aid in the development and testing of fixed-point dynamic systems. You directly design dynamic system models in the Simulink software that are ready for implementation on fixed-point hardware. The development cycle is illustrated below.



Using the MATLAB, Simulink, and Fixed-Point Designer software, you follow these steps of the development cycle:

- 1** Model the system (plant or signal source) within the Simulink software using double-precision numbers. Typically, the model will contain nonlinear elements.
- 2** Design and simulate a fixed-point dynamic system (for example, a control system or digital filter) with fixed-point Simulink blocks that meets the design, performance, and other constraints.
- 3** Analyze the results and go back to step 1 if needed.

When you have met the design requirements, you can use the model as a specification for creating production code using the Simulink Coder product or generating HDL code using the HDL Coder product.

The above steps interact strongly. In steps 1 and 2, there is a significant amount of freedom to select different solutions. Generally, you fine-tune the model based upon feedback from the results of the current implementation (step 3). There is no specific modeling approach. For example, you may obtain models from first principles such as equations of motion, or from a frequency response such as a sine sweep. There are many controllers that meet the same frequency-domain or time-domain specifications. Additionally, for each controller there are an infinite number of realizations.

The Fixed-Point Designer software helps expedite the design cycle by allowing you to simulate the effects of various fixed-point controller and digital filter structures.

More About

- “Why Use Fixed-Point Hardware?” on page 31-12

Supported Data Types

The Fixed-Point Designer software supports the following integer and fixed-point data types for simulation and code generation:

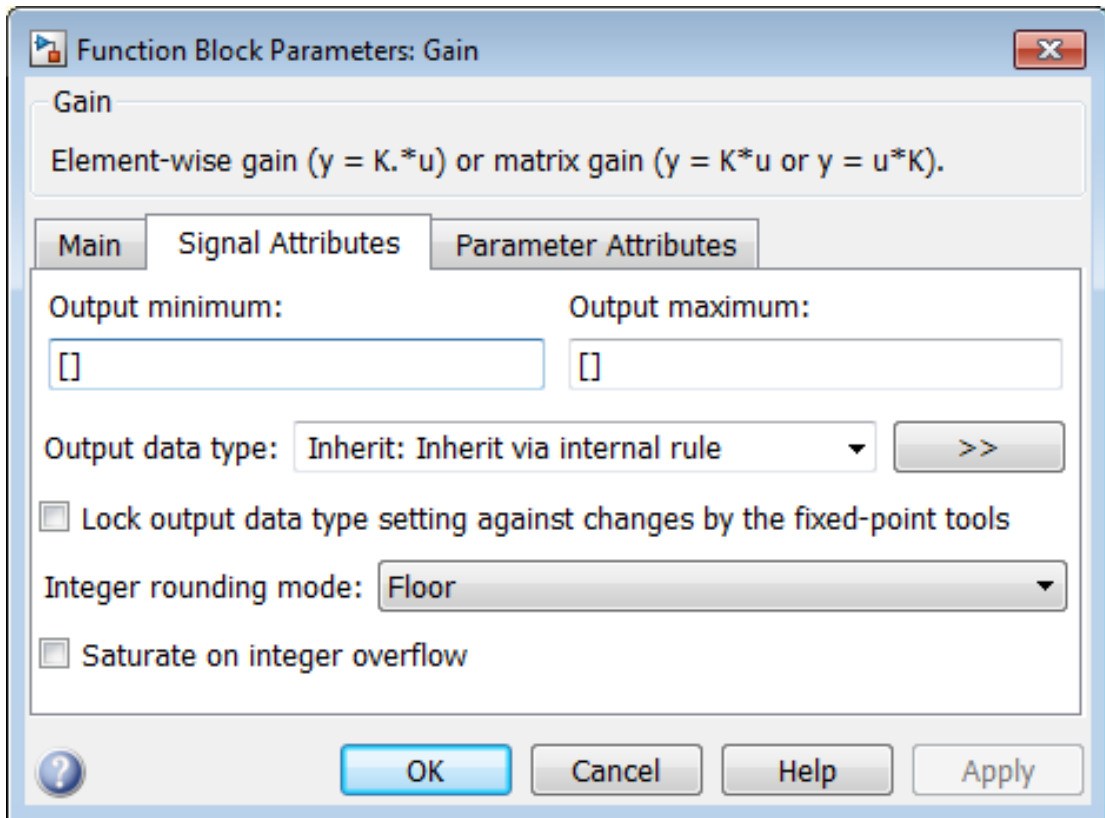
- Unsigned data types from 1 to 128 bits
- Signed data types from 2 to 128 bits
- Boolean, double, and single
- Scaled doubles

The software supports all scaling choices including pure integer, binary point, and slope bias. For slope bias scaling, it does not support complex fixed-point types that have non-zero bias or non-trivial slope.

The save data type support extends to signals, parameters, and states.

Configure Blocks with Fixed-Point Output

To create a fixed-point model, configure Simulink blocks to output fixed-point signals. Simulink blocks that support fixed-point output provide parameters that allow you to specify whether a block should output fixed-point signals and, if so, the size, scaling, and other attributes of the fixed-point output. These parameters typically appear on the **Signal Attributes** pane of the block's parameter dialog box.



The following sections explain how to use these parameters to configure a block for fixed-point output.

In this section...

“Specify the Output Data Type and Scaling” on page 31-20

In this section...

“Specify Fixed-Point Data Types with the Data Type Assistant” on page 31-22

“Rounding” on page 31-25

“Overflow Handling” on page 31-25

“Lock the Output Data Type Setting” on page 31-26

“Real-World Values Versus Stored Integer Values” on page 31-26

Specify the Output Data Type and Scaling

Many Simulink blocks allow you to specify an output data type and scaling using a parameter that appears on the block dialog box. This parameter (typically named **Output data type**) provides a pull-down menu that lists the data types a particular block supports. In general, you can specify the output data type as a rule that inherits a data type, a built-in data type, an expression that evaluates to a data type, or a Simulink data type object. For more information, see “Control Signal Data Types” (Simulink).

The Fixed-Point Designer software enables you to configure Simulink blocks with:

- **Fixed-point data types**

Fixed-point data types are characterized by their word size in bits and by their binary point—the means by which fixed-point values are scaled. See “Fixed-Point Data Types” on page 1-2 for more information.

- **Floating-point data types**

Floating-point data types are characterized by their sign bit, fraction (mantissa) field, and exponent field. See “Floating-Point Numbers” on page 32-26 for more information.

To configure blocks with Fixed-Point Designer data types, specify the data type parameter on a block dialog box as an expression that evaluates to a data type. Alternatively, you can use an assistant that simplifies the task of entering data type expressions (see “Specify Fixed-Point Data Types with the Data Type Assistant” on page 31-22). The sections that follow describe varieties of fixed-point and floating-point data types, and the corresponding functions that you use to specify them.

Integers

To specify unsigned and signed integers, use the `uint` and `sint` functions, respectively.

For example, to configure a 16-bit unsigned integer via the block dialog box, specify the **Output data type** parameter as `uint(16)`. To configure a 16-bit signed integer, specify the **Output data type** parameter as `sint(16)`.

For integer data types, the default binary point is assumed to lie to the right of all bits.

Fractional Numbers

To specify unsigned and signed fractional numbers, use the `ufrac` and `sfrac` functions, respectively.

For example, to configure the output as a 16-bit unsigned fractional number via the block dialog box, specify the **Output data type** parameter to be `ufrac(16)`. To configure a 16-bit signed fractional number, specify **Output data type** to be `sfrac(16)`.

Fractional numbers are distinguished from integers by their default scaling. Whereas signed and unsigned integer data types have a default binary point to the right of all bits, unsigned fractional data types have a default binary point to the left of all bits, while signed fractional data types have a default binary point to the right of the sign bit.

Both unsigned and signed fractional data types support *guard bits*, which act to guard against overflow. For example, `sfrac(16,4)` specifies a 16-bit signed fractional number with 4 guard bits. The guard bits lie to the left of the default binary point.

Generalized Fixed-Point Numbers

You can specify unsigned and signed generalized fixed-point numbers with the `ufix` and `sfix` functions, respectively.

For example, to configure the output as a 16-bit unsigned generalized fixed-point number via the block dialog box, specify the **Output data type** parameter to be `ufix(16)`. To configure a 16-bit signed generalized fixed-point number, specify **Output data type** to be `sfix(16)`.

Generalized fixed-point numbers are distinguished from integers and fractionals by the absence of a default scaling. For these data types, a block typically inherits its scaling from another block.

Note: Alternatively, you can use the `fixdt` function to create integer, fractional, and generalized fixed-point objects. The `fixdt` function also allows you to specify scaling for fixed-point data types.

Floating-Point Numbers

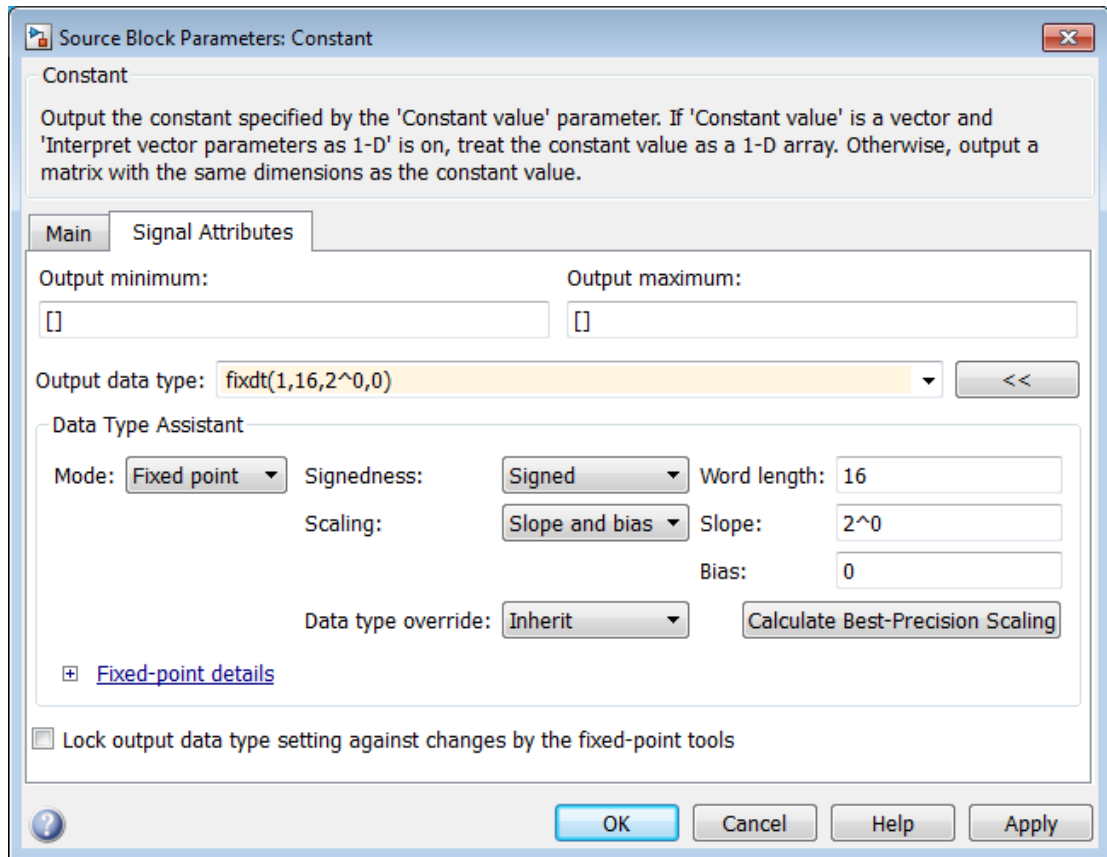
The Fixed-Point Designer software supports single-precision and double-precision floating-point numbers as defined by the IEEE[®] Standard 754-1985 for Binary Floating-Point Arithmetic. You can specify floating-point numbers with the Simulink `float` function.

For example, to configure the output as a single-precision floating-point number via the block dialog box, specify the **Output data type** parameter as `float('single')`. To configure a double-precision floating-point number, specify **Output data type** as `float('double')`.

Specify Fixed-Point Data Types with the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool that simplifies the task of specifying data types for Simulink blocks and data objects. The assistant appears on block and object dialog boxes, adjacent to parameters that provide data type control, such as the **Output data type** parameter. For more information about accessing and interacting with the assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

You can use the **Data Type Assistant** to specify a fixed-point data type. When you select **Fixed point** in the **Mode** field, the assistant displays fields for describing additional attributes of a fixed-point data type, as shown in this example:



You can set the following fixed-point attributes:

Signedness

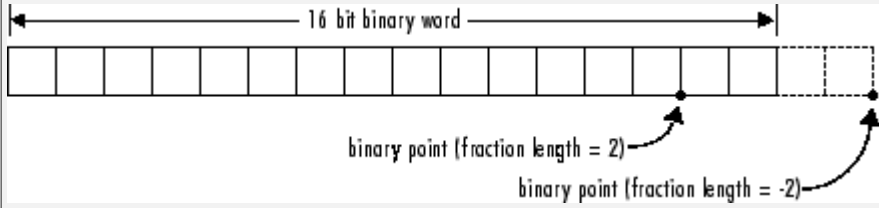
Select whether you want the fixed-point data to be **Signed** or **Unsigned**. Signed data can represent positive and negative quantities. Unsigned data represents positive values only.

Word length

Specify the size (in bits) of the word that will hold the quantized integer. Large word sizes represent large quantities with greater precision than small word sizes. Fixed-point word sizes up to 128 bits are supported for simulation.

Scaling

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. You can select the following scaling modes:

Scaling Mode	Description
Binary point	<p>If you select this mode, the assistant displays the Fraction length field, specifying the binary point location.</p> <p>Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:</p>  <p>The diagram shows a horizontal row of 16 boxes representing bits. Above the row, a double-headed arrow spans the entire width and is labeled "16 bit binary word". Below the row, two arrows point to specific bit positions. The first arrow points to the second bit from the right and is labeled "binary point (fraction length = 2)". The second arrow points to the right edge of the row and is labeled "binary point (fraction length = -2)".</p> <p>See “Binary-Point-Only Scaling” on page 32-7 for more information.</p>
Slope and bias	<p>If you select this mode, the assistant displays fields for entering the Slope and Bias.</p> <ul style="list-style-type: none"> • Slope can be any <i>positive</i> real number. • Bias can be any real number. <p>See “Slope and Bias Scaling” on page 32-7 for more information.</p>
Best precision	<p>If you select this mode, the block scales a constant vector or matrix such that the precision of its elements is maximized. This mode is available only for particular blocks.</p> <p>See “Constant Scaling for Best Precision” on page 32-15 for more information.</p>

Calculate Best-Precision Scaling

The Fixed-Point Designer software can automatically calculate “best-precision” values for both **Binary point** and **Slope and bias** scaling, based on the values that you

specify for other parameters on the dialog box. To calculate best-precision-scaling values automatically, enter values for the block's **Output minimum** and **Output maximum** parameters. Afterward, click the **Calculate Best-Precision Scaling** button in the assistant.

Rounding

You specify how fixed-point numbers are rounded with the **Integer rounding mode** parameter. The following rounding modes are supported:

- **Ceiling** — This mode rounds toward positive infinity and is equivalent to the MATLAB `ceil` function.
- **Convergent** — This mode rounds toward the nearest representable number, with ties rounding to the nearest even integer. Convergent rounding is equivalent to the Fixed-Point Designer `convergent` function.
- **Floor** — This mode rounds toward negative infinity and is equivalent to the MATLAB `floor` function.
- **Nearest** — This mode rounds toward the nearest representable number, with the exact midpoint rounded toward positive infinity. Rounding toward nearest is equivalent to the Fixed-Point Designer `nearest` function.
- **Round** — This mode rounds to the nearest representable number, with ties for positive numbers rounding in the direction of positive infinity and ties for negative numbers rounding in the direction of negative infinity. This mode is equivalent to the Fixed-Point Designer `round` function.
- **Simplest** — This mode automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible.
- **Zero** — This mode rounds toward zero and is equivalent to the MATLAB `fix` function.

For more information about each of these rounding modes, see “Rounding” on page 33-5.

Overflow Handling

To control how overflow conditions are handled for fixed-point operations, use the **Saturate on integer overflow** check box.

If this box is selected, overflows saturate to either the maximum or minimum value represented by the data type. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

If this box is not selected, overflows wrap to the appropriate value that is representable by the data type. For example, the number 130 does not fit in a signed 8-bit integer, and would wrap to -126.

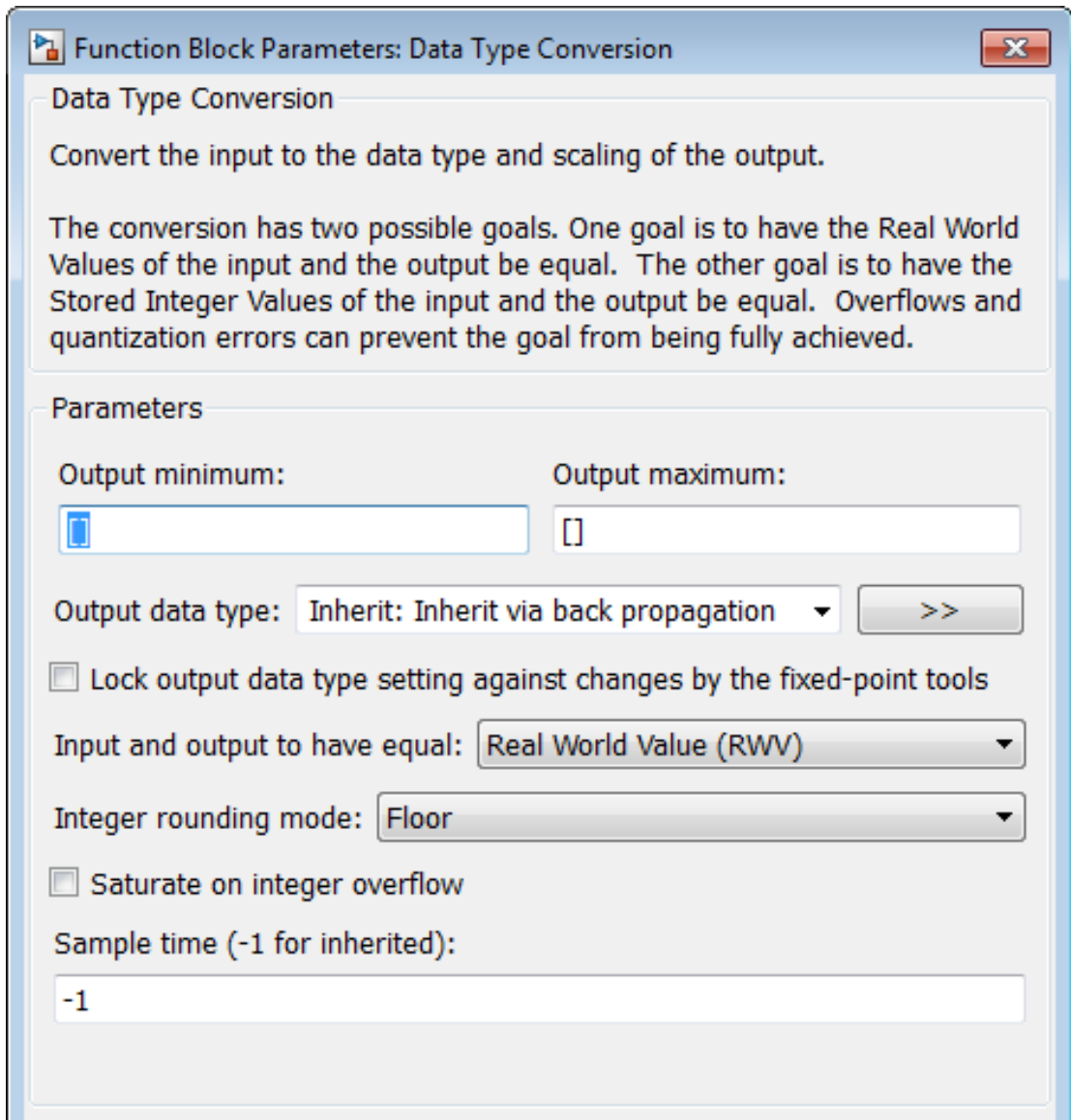
Lock the Output Data Type Setting

If the output data type is a generalized fixed-point number, you have the option of locking its output data type setting by selecting the **Lock output data type setting against changes by the fixed-point tools** check box.

When locked, the Fixed-Point Tool and automatic scaling script `autofixexp` do not change the output data type setting. Otherwise, the Fixed-Point Tool and `autofixexp` script are free to adjust the output data type setting.

Real-World Values Versus Stored Integer Values

You can configure Data Type Conversion blocks to treat signals as real-world values or as stored integers with the **Input and output to have equal** parameter.



The possible values are Real World Value (RWV) and Stored Integer (SI).

In terms of the variables defined in “Scaling” on page 32-6, the real-world value is given by V and the stored integer value is given by Q . You may want to treat numbers as stored integer values if you are modeling hardware that produces integers as output.

More About

- “Configure Blocks with Fixed-Point Parameters” on page 31-29

Configure Blocks with Fixed-Point Parameters

Certain Simulink blocks allow you to specify fixed-point numbers as the values of parameters used to compute the block's output, e.g., the **Gain** parameter of a Gain block.

Note: S-functions and the Stateflow Chart block do not support fixed-point parameters.

You can specify a fixed-point parameter value either directly by setting the value of the parameter to an expression that evaluates to a `fi` object, or indirectly by setting the value of the parameter to an expression that refers to a fixed-point Simulink.Parameter object.

In this section...
“Specify Fixed-Point Values Directly” on page 31-29
“Specify Fixed-Point Values Via Parameter Objects” on page 31-30

Note: Simulating or performing data type override on a model with `fi` objects requires a Fixed-Point Designer software license. See “Sharing Fixed-Point Models” on page 31-3 for more information.

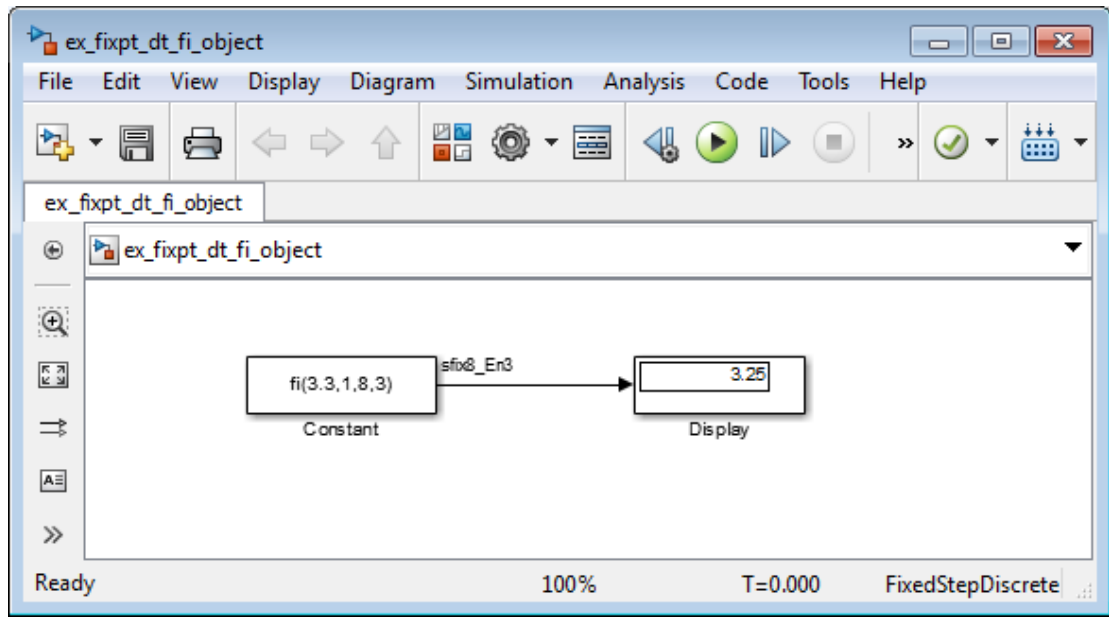
Specify Fixed-Point Values Directly

You can specify fixed-point values for block parameters using `fi` objects. In the block dialog's parameter field, simply enter the name of a `fi` object or an expression that includes the `fi` constructor function.

For example, entering the expression

```
fi(3.3,1,8,3)
```

as the **Constant value** parameter for the Constant block specifies a signed fixed-point value of 3.3, with a word length of 8 bits and a fraction length of 3 bits.



Specify Fixed-Point Values Via Parameter Objects

You can specify fixed-point parameter objects for block parameters using instances of the `Simulink.Parameter` class. To create a fixed-point parameter object, either specify a `fi` object as the parameter object's `Value` property, or specify the relevant fixed-point data type for the parameter object's `DataType` property.

For example, suppose you want to create a fixed-point constant in your model. You could do this using a fixed-point parameter object and a Constant block as follows:

- 1 Enter the following command at the MATLAB prompt to create an instance of the `Simulink.Parameter` class:

```
my_fixpt_param = Simulink.Parameter
```

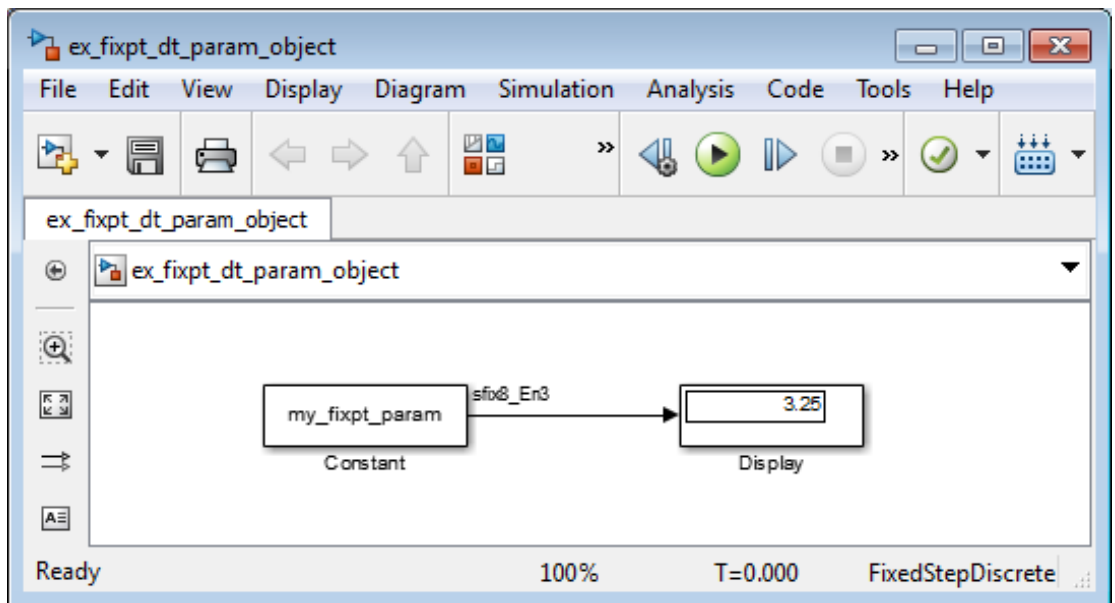
- 2 Specify either the name of a `fi` object or an expression that includes the `fi` constructor function as the parameter object's `Value` property:

```
my_fixpt_param.Value = fi(3.3,1,8,3)
```


Alternatively, you can set the parameter object's **Value** and **DataType** properties separately. In this case, specify the relevant fixed-point data type using a `Simulink.AliasType` object, a `Simulink.NumericType` object, or a `fixdt` expression. For example, the following commands independently set the parameter object's value and data type, using a `fixdt` expression as the **DataType**:

```
my_fixpt_param.Value = 3.3;
my_fixpt_param.DataType = 'fixdt(1,8,2^-3,0)'
```

- 3 Specify the parameter object as the value of a block's parameter. For example, `my_fixpt_param` specifies the **Constant value** parameter for the Constant block in the following model:



Consequently, the Constant block outputs a signed fixed-point value of 3.3, with a word length of 8 bits and a fraction length of 3 bits.

More About

- “Configure Blocks with Fixed-Point Output” on page 31-19

Pass Fixed-Point Data Between Simulink Models and MATLAB

You can read fixed-point data from the MATLAB software into your Simulink models, and there are a number of ways in which you can log fixed-point information from your models and simulations to the workspace.

Read Fixed-Point Data from the Workspace

Use the From Workspace block to read fixed-point data from the MATLAB workspace into a Simulink model. To do this, the data must be in structure format with a Fixed-Point Designer `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`.

Write 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 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]'

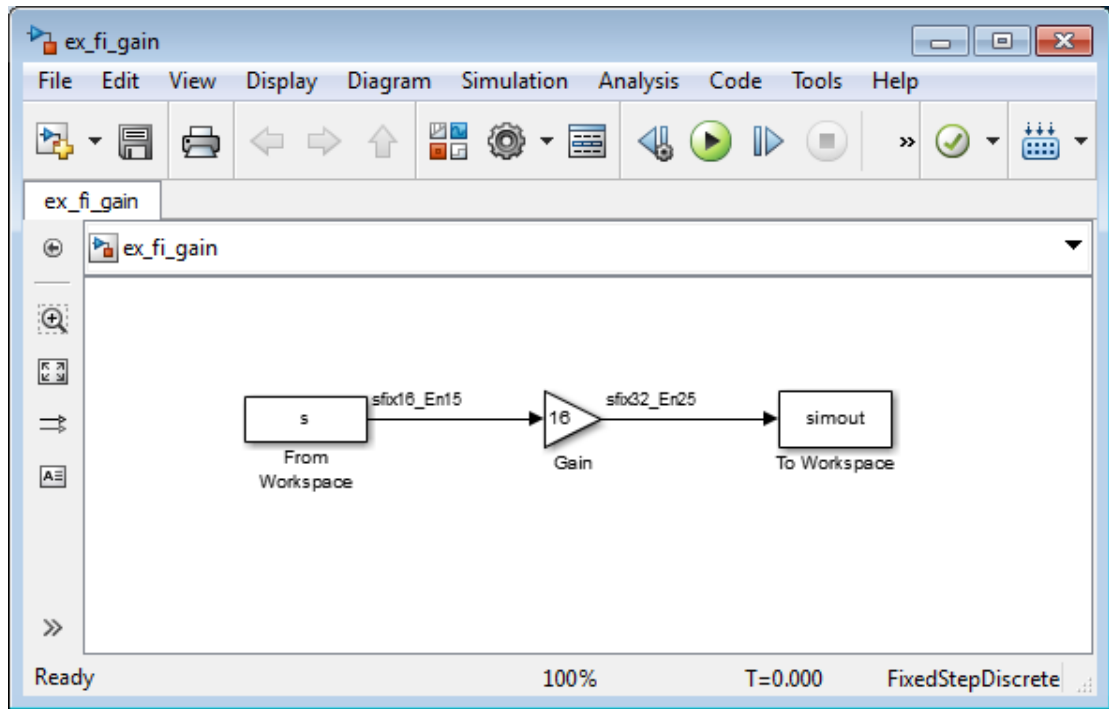
s =

    signals: [1x1 struct]
      time: [11x1 double]

```

The From Workspace block in the following model has the `fi` structure `s` in the **Data** parameter. In the model, the following parameters in the **Solver** pane of the Configuration Parameters dialog box 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

```

Log Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as Fixed-Point Designer `fi` objects.

To enable signal logging for a signal:

- 1 Select the signal.
- 2 Click the **Simulation Data Inspector** button arrow.
- 3 Select **Log Selected Signals**.

For more information, refer to “Signal Logging” (Simulink).

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 larger data storage container size.

Access Fixed-Point Block Data During Simulation

Simulink provides an application programming 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 about the API, refer to “Access Block Data During Simulation” (Simulink).

See Also

From Workspace | To Workspace

More About

- “Signal Logging” (Simulink)

Cast from Doubles to Fixed Point

In this section...
“About This Example” on page 31-36
“Block Descriptions” on page 31-37
“Simulations” on page 31-38

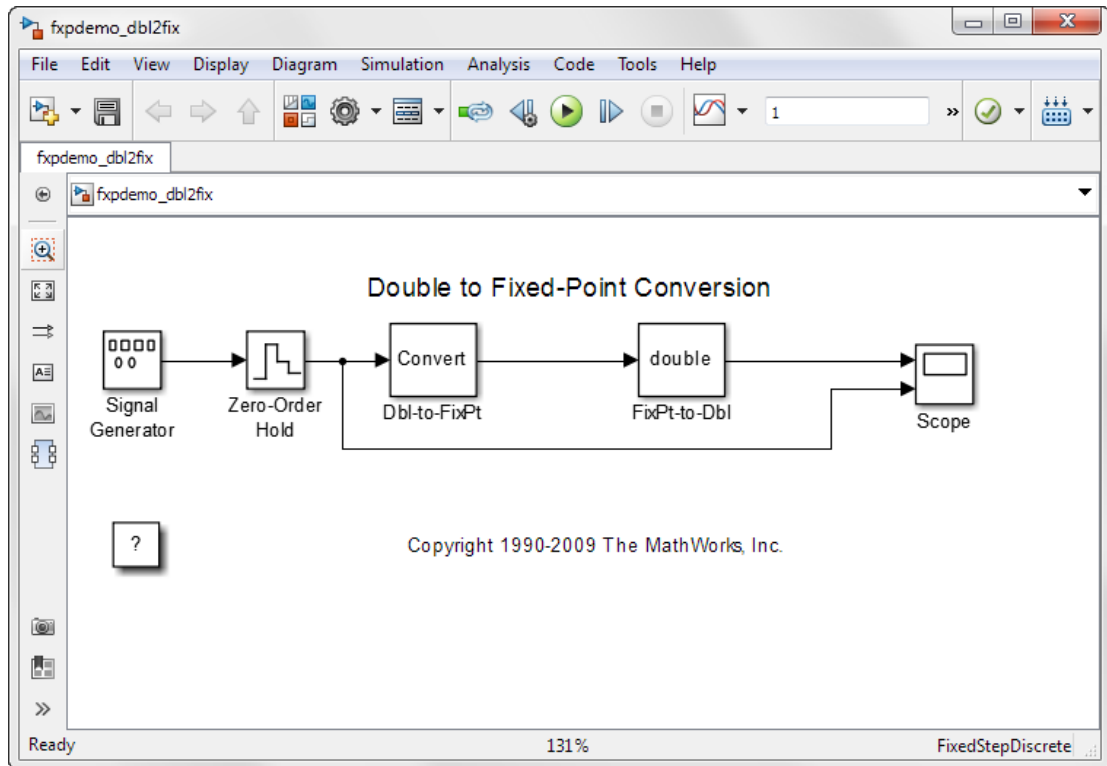
About This Example

The purpose of this example is to show you how to simulate a continuous real-world doubles signal using a generalized fixed-point data type. Although simple in design, the model gives you an opportunity to explore many of the important features of the Fixed-Point Designer software, including

- Data types
- Scaling
- Rounding
- Logging minimum and maximum simulation values to the workspace
- Overflow handling

This example uses the `fxpdemo_dbl2fix` model. Open the model:

```
fxpdemo_dbl2fix
```



The sections that follow describe the model and its simulation results.

Block Descriptions

In this example, you configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval $[-5 \ 5]$. The Signal Generator block always outputs double-precision numbers.

The Data Type Conversion (Dbl-to-FixPt) block converts the double-precision numbers from the Signal Generator block into one of the Fixed-Point Designer data types. For simplicity, the size of the output signal is 5 bits in this example.

The Data Type Conversion (FixPt-to-Dbl) block converts one of the Fixed-Point Designer data types into a Simulink data type. In this example, it outputs double-precision numbers.

Simulations

The following sections describe how to simulate the model using binary-point-only scaling and [Slope Bias] scaling.

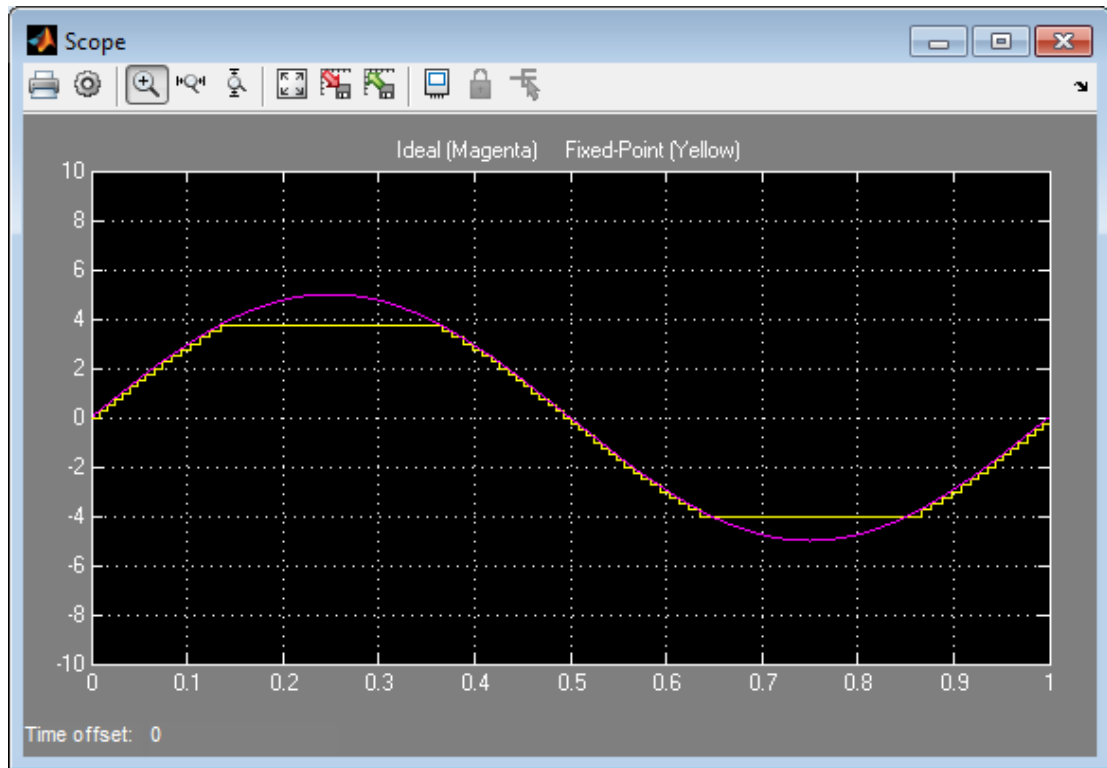
Binary-Point-Only Scaling

When using binary-point-only scaling, your goal is to find the optimal power-of-two exponent E , as defined in “Scaling” on page 32-6. For this scaling mode, the fractional slope F is 1 and there is no bias.

To run the simulation:

- 1 Configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval [-5 5].
 - a Double-click the Signal Generator block to open the **Block Parameters** dialog.
 - b Set the **Wave form** parameter to **sine**.
 - c Set the **Amplitude** parameter to 5.
 - d Click **OK**.
- 2 Configure the Data Type Conversion (Dbl-to-FixPt) block.
 - a Double-click the **Dbl-to-FixPt** block to open the **Block Parameters** dialog.
 - b Verify that the **Output data type** parameter is `fixdt(1,5,2)`.
`fixdt(1,5,2)` specifies a 5-bit, signed, fixed-point number with scaling 2^{-2} , which puts the binary point two places to the left of the rightmost bit. Hence the maximum value is $011.11 = 3.75$, a minimum value of $100.00 = -4.00$, and the precision is $(1/2)^2 = 0.25$.
 - c Verify that the **Integer rounding mode** parameter is **Floor**. **Floor** rounds the fixed-point result toward negative infinity.
 - d Select the **Saturate on integer overflow** checkbox to prevent the block from wrapping on overflow.
 - e Click **OK**.
- 3 Select **Simulation > Run** in your Simulink model window.

The Scope displays the real-world and fixed-point simulation results.



The simulation shows the quantization effects of fixed-point arithmetic. Using a 5-bit word with a precision of $(1/2)^2 = 0.25$ produces a discretized output that does not span the full range of the input signal.

If you want to span the complete range of the input signal with 5 bits using binary-point-only scaling, then your only option is to sacrifice precision. Hence, the output scaling is 2^{-1} , which puts the binary point one place to the left of the rightmost bit. This scaling gives a maximum value of $0111.1 = 7.5$, a minimum value of $1000.0 = -8.0$, and a precision of $(1/2)^1 = 0.5$.

To simulate using a precision of 0.5, set the **Output data type** parameter of the Data Type Conversion (Dbl-to-FixPt) block to `fixdt(1, 5, 1)` and rerun the simulation.

[Slope Bias] Scaling

When using [Slope Bias] scaling, your goal is to find the optimal fractional slope F and fixed power-of-two exponent E , as defined in “Scaling” on page 32-6. There is no bias for this example because the sine wave is on the interval $[-5 \ 5]$.

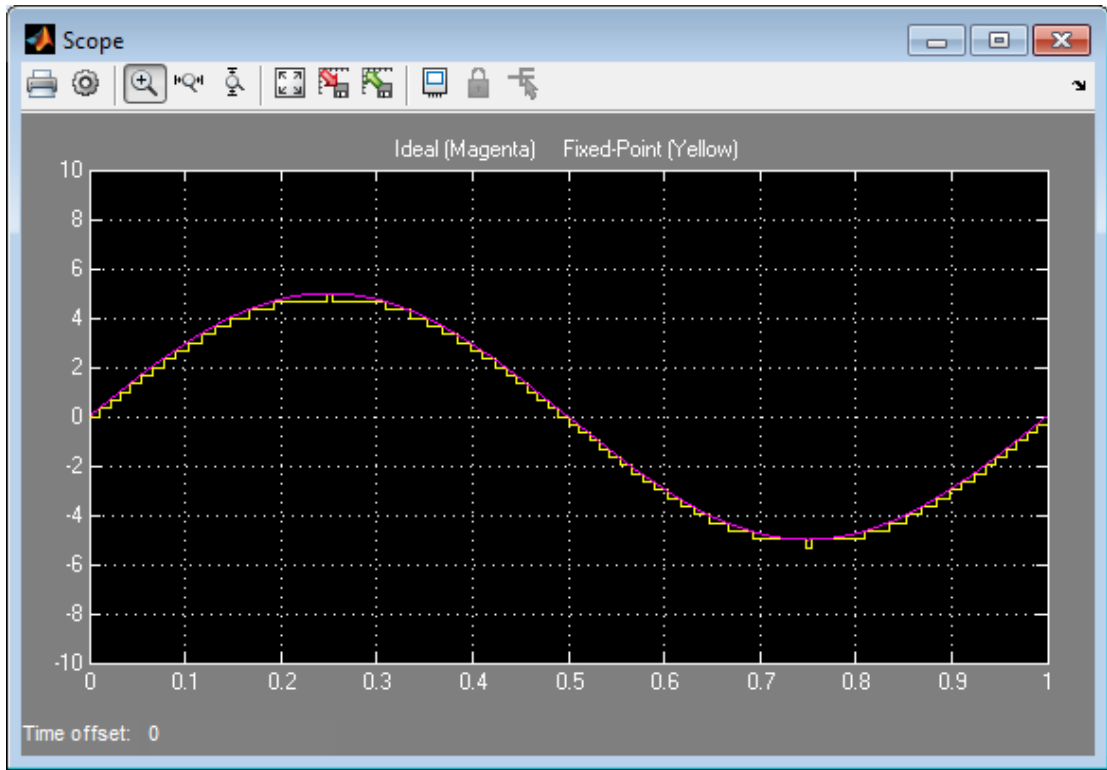
To arrive at a value for the slope, you begin by assuming a fixed power-of-two exponent of -2. To find the fractional slope, you divide the maximum value of the sine wave by the maximum value of the scaled 5-bit number. The result is $5.00/3.75 = 1.3333$. The slope (and precision) is $1.3333 \cdot (0.25) = 0.3333$. You specify the [Slope Bias] scaling as `[0.3333 0]` by entering the expression `fixdt(1,5,0.3333,0)` as the value of the **Output data type** parameter.

You could also specify a fixed power-of-two exponent of -1 and a corresponding fractional slope of 0.6667. The resulting slope is the same since E is reduced by 1 bit but F is increased by 1 bit. The Fixed-Point Designer software would automatically store F as 1.3332 and E as -2 because of the normalization condition of $1 \leq F < 2$.

To run the simulation:

- 1 Configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval $[-5 \ 5]$.
 - a Double-click the Signal Generator block to open the **Block Parameters** dialog.
 - b Set the **Wave form** parameter to `sine`.
 - c Set the **Amplitude** parameter to 5.
 - d Click **OK**.
- 2 Configure the Data Type Conversion (Dbl-to-FixPt) block.
 - a Double-click the **Dbl-to-FixPt** block to open the **Block Parameters** dialog.
 - b Set the **Output data type** parameter to `fixdt(1,5,0.3333,0)` to specify [Slope Bias] scaling as `[0.3333 0]`.
 - c Verify that the **Integer rounding mode** parameter is `Floor`. `Floor` rounds the fixed-point result toward negative infinity.
 - d Select the **Saturate on integer overflow** checkbox to prevent the block from wrapping on overflow.
 - e Click **OK**.
- 3 Select **Simulation > Run** in your Simulink model window.

The Scope displays the real-world and fixed-point simulation results.



You do not need to find the slope using this method. You need only the range of the data you are simulating and the size of the fixed-point word used in the simulation. You can achieve reasonable simulation results by selecting your scaling based on the formula

$$\frac{(max_value - min_value)}{2^{ws} - 1},$$

where

- *max_value* is the maximum value to be simulated.
- *min_value* is the minimum value to be simulated.
- *ws* is the word size in bits.

- $2^{ws} - 1$ is the largest value of a word with size ws .

For this example, the formula produces a slope of 0.32258.

More About

- “Physical Quantities and Measurement Scales” on page 31-4

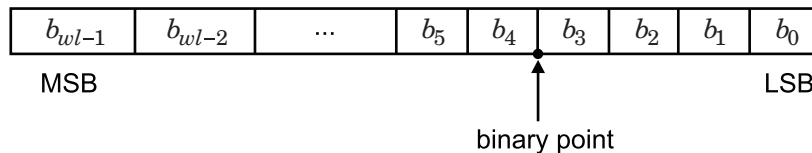
Data Types and Scaling

- “Data Types and Scaling in Digital Hardware” on page 32-2
- “Scaling” on page 32-6
- “Quantization” on page 32-9
- “Range and Precision” on page 32-11
- “Fixed-Point Numbers in Simulink” on page 32-15
- “Display Port Data Types” on page 32-20
- “Scaled Doubles” on page 32-22
- “Use Scaled Doubles to Avoid Precision Loss” on page 32-24
- “Floating-Point Numbers” on page 32-26

Data Types and Scaling in Digital Hardware

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, 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. With the Fixed-Point Designer software, fixed-point data types can be integers, fractionals, or generalized fixed-point numbers. The main difference between these data types is their default binary point. 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.

Binary Point Interpretation

The binary point is the means by which fixed-point numbers are scaled. It is usually the software that determines the binary point. When performing basic math functions such as addition or subtraction, the hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a scale factor. They are performing signed or unsigned fixed-point binary algebra as if the binary point is to the right of b_0 .

Fixed-Point Designer supports the general binary point scaling $V = Q * 2^E$. V is the real-world value, Q is the stored integer value, and E is equal to $-FractionLength$. In other words, $RealWorldValue = StoredInteger * 2^{-FractionLength}$.

$FractionLength$ defines the scaling of the stored integer value. The word length limits the values that the stored integer can take, but it does not limit the values $FractionLength$ can take. The software does not restrict the value of exponent E based on the word length of the stored integer Q . Because E is equal to $-FractionLength$, restricting the binary point to being contiguous with the fraction is unnecessary; the fraction length can be negative or greater than the word length.

For example, a word consisting of three unsigned bits is usually represented in scientific notation in one of the following ways.

$$bbb. = bbb. \times 2^0$$

$$bbb = bbb. \times 2^{-1}$$

$$b.bb = bbb. \times 2^{-2}$$

$$bbb = bbb. \times 2^{-3}$$

If the exponent were greater than 0 or less than -3, then the representation would involve lots of zeros.

$$bbb00000. = bbb. \times 2^5$$

$$bbb00. = bbb. \times 2^2$$

$$.00bbb = bbb. \times 2^{-5}$$

$$.00000bbb = bbb. \times 2^{-8}$$

These extra zeros never change to ones, however, so they don't show up in the hardware. Furthermore, unlike floating-point exponents, a fixed-point exponent never shows up in the hardware, so fixed-point exponents are not limited by a finite number of bits.

Consider a signed value with a word length of 8, a fraction length of 10, and a stored integer value of 5 (binary value 00000101). The real-world value is calculated using the formula

$RealWorldValue = StoredInteger * 2^{-FractionLength}$. In this case, $RealWorldValue = 5 * 2^{-10} = 0.0048828125$. Because the fraction length is 2

bits longer than the word length, the binary value of the stored integer is $x.xx00000101$, where x is a placeholder for implicit zeros. 0.0000000101 (binary) is equivalent to 0.0048828125 (decimal). For an example using a `fi` object, see “Create a `fi` Object With Fraction Length Greater Than Word Length”.

Signed Fixed-Point Numbers

Computer hardware typically represents the negation of a binary fixed-point number in three different ways: sign/magnitude, one's complement, and two's complement. Two's complement is the preferred representation of signed fixed-point numbers and is the only representation used by Fixed-Point Designer software.

Negation using two's complement consists of a bit inversion (translation into one's complement) followed by the addition of a one. For example, the two's complement of 000101 is 111011 .

Whether a fixed-point value is signed or unsigned is usually not encoded explicitly within the binary word; that is, there is no sign bit. Instead, the sign information is implicitly defined within the computer architecture.

Floating-Point Data Types

Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field. Fixed-Point Designer adheres to the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (referred to simply as the IEEE Standard 754 throughout this guide) and supports singles and doubles.

When choosing a data type, you must consider these factors:

- The numerical range of the result
- The precision required of the result
- The associated quantization error (i.e., the rounding mode)
- The method for dealing with exceptional arithmetic conditions

These choices depend on your specific application, the computer architecture used, and the cost of development, among others.

With the Fixed-Point Designer software, you can explore the relationship between data types, range, precision, and quantization error in the modeling of dynamic digital

systems. With the Simulink Coder product, you can generate production code based on that model. With HDL Coder, you can generate portable, synthesizable VHDL and Verilog code from Simulink models and Stateflow charts.

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

More About

- “Scaling” on page 32-6
- “Quantization” on page 32-9
- “Range and Precision” on page 32-11

Scaling

The dynamic range of fixed-point numbers is much less than floating-point numbers with equivalent word sizes. To avoid overflow conditions and minimize quantization errors, fixed-point numbers must be scaled.

With the Fixed-Point Designer software, you can select a fixed-point data type whose scaling is defined by its binary point, or you can select an arbitrary linear scaling that suits your needs. This section presents the scaling choices available for fixed-point data types.

You can represent a fixed-point number by a general slope and bias encoding 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 is assumed to be at the far right of the word. In Fixed-Point Designer 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

$$\text{real-world value} = 2^{-\text{fraction length}} \times \text{integer}$$

The scaling modes available to you within this encoding scheme are described in the sections that follow. For detailed information about how the supported scaling modes effect fixed-point operations, refer to “Recommendations for Arithmetic and Scaling” on page 33-40.

Binary-Point-Only Scaling

Binary-point-only or power-of-two scaling involves moving the binary point within the fixed-point word. The advantage of this scaling mode is to minimize the number of processor arithmetic operations.

With binary-point-only scaling, the components of the general slope and bias formula have the following values:

- $F = 1$
- $S = F 2^E = 2^E$
- $B = 0$

The scaling of a quantized real-world number is defined by the slope S , which is restricted to a power of two. The negative of the power-of-two exponent is called the fraction length. The fraction length is the number of bits to the right of the binary point. For Binary-Point-Only scaling, specify fixed-point data types as

- signed types — `fixdt(1, WordLength, FractionLength)`
- unsigned types — `fixdt(0, WordLength, FractionLength)`

Integers are a special case of fixed-point data types. Integers have a trivial scaling with slope 1 and bias 0, or equivalently with fraction length 0. Specify integers as

- signed integer — `fixdt(1, WordLength, 0)`
- unsigned integer — `fixdt(0, WordLength, 0)`

Slope and Bias Scaling

When you scale by slope and bias, the slope S and bias B of the quantized real-world number can take on any value. The slope must be a positive number. Using slope and bias, specify fixed-point data types as

- `fixdt(Signed, WordLength, Slope, Bias)`

Unspecified Scaling

Specify fixed-point data types with an unspecified scaling as

- `fixdt(Signed, WordLength)`

Simulink signals, parameters, and states must never have unspecified scaling. When scaling is unspecified, you must use some other mechanism such as automatic best precision scaling to determine the scaling that the Simulink software uses.

Quantization

The quantization Q of a real-world value V is represented by a weighted sum of bits. Within the context of the general slope and bias encoding scheme, the value of an unsigned fixed-point quantity is given by

$$\tilde{V} = S \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right] + B,$$

while the value of a signed fixed-point quantity is given by

$$\tilde{V} = S \cdot \left[-b_{ws-1} 2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i \right] + B,$$

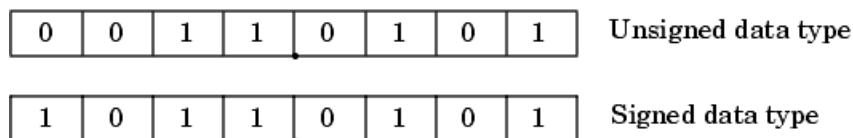
where

- b_i are binary digits, with $b_i = 1, 0$, for $i = 0, 1, \dots, ws-1$
- The word size in bits is given by ws , with $ws = 1, 2, 3, \dots, 128$.
- S is given by $F = 2^E$, where the scaling is unrestricted because the binary point does not have to be contiguous with the word.

b_i are called *bit multipliers* and 2^i are called the *weights*.

Fixed-Point Format

Formats for 8-bit signed and unsigned fixed-point values are shown in the following figure.



Note that you cannot discern whether these numbers are signed or unsigned data types merely by inspection since this information is not explicitly encoded within the word.

The binary number **0011.0101** yields the same value for the unsigned and two's complement representation because the MSB = 0. Setting $B = 0$ and using the appropriate weights, bit multipliers, and scaling, the value is

$$\begin{aligned}\tilde{V} &= (F 2^E)Q = 2^E \left[\sum_{i=0}^{ws-1} b_i 2^i \right] \\ &= 2^{-4} (0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= 3.3125.\end{aligned}$$

Conversely, the binary number **1011.0101** yields different values for the unsigned and two's complement representation since the MSB = 1.

Setting $B = 0$ and using the appropriate weights, bit multipliers, and scaling, the unsigned value is

$$\begin{aligned}\tilde{V} &= (F 2^E)Q = 2^E \left[\sum_{i=0}^{ws-1} b_i 2^i \right] \\ &= 2^{-4} (1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= 11.3125,\end{aligned}$$

while the two's complement value is

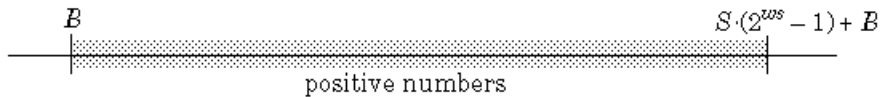
$$\begin{aligned}\tilde{V} &= (F 2^E)Q = 2^E \left[-b_{ws-1} 2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i \right] \\ &= 2^{-4} (-1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= -4.6875.\end{aligned}$$

Range and Precision

The *range* of a number gives the limits of the representation, while the *precision* gives the distance between successive numbers in the representation. The range and precision of a fixed-point number depend on the length of the word and the scaling.

Range

The following figure illustrates the range of representable numbers for an unsigned fixed-point number of size ws , scaling S , and bias B .



The following figure illustrates the range of representable numbers for a two's complement fixed-point number of size ws , scaling S , and bias B where the values of ws , scaling S , and bias B allow for both negative and positive numbers.



For both the signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{ws} .

For example, if the fixed-point data type is an integer with scaling defined as $S = 1$ and $B = 0$, then the maximum unsigned value is 2^{ws-1} , because zero must be represented. In two's complement, negative numbers must be represented as well as zero, so the maximum value is $2^{ws-1} - 1$. Additionally, since there is only one representation for zero, there must be an unequal number of positive and negative numbers. This means there is a representation for -2^{ws-1} but not for 2^{ws-1} .

Precision

The precision of a data type is given by the slope. In this usage, precision means the difference between neighboring representable values.

Fixed-Point Data Type Parameters

The low limit, high limit, and default binary-point-only scaling for the supported fixed-point data types discussed in “Binary-Point-Only Scaling” on page 32-7 are given in the following table. See “Limitations on Precision” on page 33-4 and “Limitations on Range” on page 33-31 for more information.

Fixed-Point Data Type Range and Default Scaling

Name	Data Type	Low Limit	High Limit	Default Scaling (~Precision)
Unsigned Integer	<code>fixdt(0,ws,0)</code>	0	$2^{ws} - 1$	1
Signed Integer	<code>fixdt(1,ws,0)</code>	-2^{ws-1}	$2^{ws-1} - 1$	1
Unsigned Binary Point	<code>fixdt(0,ws,fl)</code>	0	$(2^{ws} - 1)2^{-fl}$	2^{-fl}
Signed Binary Point	<code>fixdt(1,ws,fl)</code>	$-2^{ws-1-1}2^{-fl}$	$(2^{ws-1} - 1)2^{-fl}$	2^{-fl}
Unsigned Slope Bias	<code>fixdt(0,ws,s,b)</code>	b	$s(2^{ws} - 1) + b$	s
Signed Slope Bias	<code>fixdt(1,ws,s,b)</code>	$-s(2^{ws-1}) + b$	$s(2^{ws-1} - 1) + b$	s

s = Slope, b = Bias, ws = WordLength, fl = FractionLength

Range of an 8-Bit Fixed-Point Data Type — Binary-Point-Only Scaling

The precisions, range of signed values, and range of unsigned values for an 8-bit generalized fixed-point data type with binary-point-only scaling are listed in the

follow table. Note that the first scaling value (2^1) represents a binary point that is not contiguous with the word.

Scaling	Precision	Range of Signed Values (Low, High)	Range of Unsigned Values (Low, High)
2^1	2.0	-256, 254	0, 510
2^0	1.0	-128, 127	0, 255
2^{-1}	0.5	-64, 63.5	0, 127.5
2^{-2}	0.25	-32, 31.75	0, 63.75
2^{-3}	0.125	-16, 15.875	0, 31.875
2^{-4}	0.0625	-8, 7.9375	0, 15.9375
2^{-5}	0.03125	-4, 3.96875	0, 7.96875
2^{-6}	0.015625	-2, 1.984375	0, 3.984375
2^{-7}	0.0078125	-1, 0.9921875	0, 1.9921875
2^{-8}	0.00390625	-0.5, 0.49609375	0, 0.99609375

Range of an 8-Bit Fixed-Point Data Type — Slope and Bias Scaling

The precision and ranges of signed and unsigned values for an 8-bit fixed-point data type using slope and bias scaling are listed in the following table. The slope starts at a value of 1.25 with a bias of 1.0 for all slopes. Note that the slope is the same as the precision.

Bias	Slope/Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
1	1.25	-159, 159.75	1, 319.75
1	0.625	-79, 80.375	1, 160.375
1	0.3125	-39, 40.6875	1, 80.6875
1	0.15625	-19, 20.84375	1, 40.84375
1	0.078125	-9, 10.921875	1, 20.921875
1	0.0390625	-4, 5.9609375	1, 10.9609375
1	0.01953125	-1.5, 3.48046875	1, 5.98046875

Bias	Slope/Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
1	0.009765625	-0.25, 2.240234375	1, 3.490234375
1	0.0048828125	0.375, 1.6201171875	1, 2.2451171875

Fixed-Point Numbers in Simulink

In this section...

“Constant Scaling for Best Precision” on page 32-15

“Fixed-Point Data Type and Scaling Notation” on page 32-17

Constant Scaling for Best Precision

The following fixed-point Simulink blocks provide a mode for scaling parameters whose values are constant vectors or matrices:

- Constant
- Discrete FIR Filter
- Gain
- Relay
- Repeating Sequence Stair

This scaling mode is based on binary-point-only scaling. Using this mode, you can scale a constant vector or matrix such that a common binary point is found based on the best precision for the largest value in the vector or matrix.

Constant scaling for best precision is available only for fixed-point data types with unspecified scaling. All other fixed-point data types use their specified scaling. You can use the **Data Type Assistant** (see “Specify Data Types Using Data Type Assistant” (Simulink)) on a block dialog box to enable the best precision scaling mode.

1

On a block dialog box, click the **Show data type assistant** button  .

The **Data Type Assistant** appears.

2 In the **Data Type Assistant**, and from the **Mode** list, select **Fixed point**.

The **Data Type Assistant** displays additional options associated with fixed-point data types.

3 From the **Scaling** list, select **Best precision**.

Data Type Assistant

Mode: Fixed point Signedness: Signed Word length: 16

Scaling: Best precision

Data type override: Inherit

[+ Fixed-point details](#)

To understand how you might use this scaling mode, consider a 3-by-3 matrix of doubles, M , defined as

```

3.3333e-003  3.3333e-004  3.3333e-005
3.3333e-002  3.3333e-003  3.3333e-004
3.3333e-001  3.3333e-002  3.3333e-003

```

Now suppose you specify M as the value of the **Gain** parameter for a Gain block. The results of specifying your own scaling versus using the constant scaling mode are described here:

- **Specified Scaling**

Suppose the matrix elements are converted to a signed, 10-bit generalized fixed-point data type with binary-point-only scaling of 2^{-7} (that is, the binary point is located seven places to the left of the right most bit). With this data format, M becomes

```

0          0          0
3.1250e-002  0          0
3.3594e-001  3.1250e-002  0

```

Note that many of the matrix elements are zero, and for the nonzero entries, the scaled values differ from the original values. This is because a double is converted to a binary word of fixed size and limited precision for each element. The larger and more precise the conversion data type, the more closely the scaled values match the original values.

- **Constant Scaling for Best Precision**

If M is scaled based on its largest matrix value, you obtain

```

2.9297e-003  0          0
3.3203e-002  2.9297e-003  0
3.3301e-001  3.3203e-002  2.9297e-003

```

Best precision would automatically select the fraction length that minimizes the quantization error. Even though precision was maximized for the given word length, quantization errors can still occur. In this example, a few elements still quantize to zero.

Fixed-Point Data Type and Scaling Notation

Simulink data type names must be valid MATLAB identifiers with less than 128 characters. The data type name provides information about container type, number encoding, and scaling.

You can represent a fixed-point number using the fixed-point scaling equation

$$V \approx \tilde{V} = SQ + B,$$

where

- V is the real-world value.
- \tilde{V} is the approximate real-world value.
- $S = F2^E$ is the slope.
- F is the slope adjustment factor.
- E is the fixed power-of-two exponent.
- Q is the stored integer.
- B is the bias.

For more information, see “Scaling” on page 32-6.

The following table provides a key for various symbols that appear in Simulink products to indicate the data type and scaling of a fixed-point value.

Symbol	Description	Example
Container Type		
<code>ufix</code>	Unsigned fixed-point data type	<code>ufix8</code> is an 8-bit unsigned fixed-point data type
<code>sfix</code>	Signed fixed-point data type	<code>sfix128</code> is a 128-bit signed fixed-point data type

Symbol	Description	Example
<code>flt_u</code>	Scaled Doubles override of an unsigned fixed-point data type (<code>ufix</code>)	<code>flt_u32</code> is a scaled doubles override of <code>ufix32</code>
<code>flt_s</code>	Scaled Doubles override of a signed fixed-point data type (<code>sfix</code>)	<code>flt_s64</code> is a scaled doubles override of <code>sfix64</code>
Number Encoding		
<code>e</code>	10^{\wedge}	<code>125e8</code> equals $125 * (10^{\wedge}(8))$
<code>n</code>	Negative	<code>n31</code> equals <code>-31</code>
<code>p</code>	Decimal point	<code>1p5</code> equals <code>1.5</code> <code>p2</code> equals <code>0.2</code>
Scaling Encoding		
<code>S</code>	Slope	<code>ufix16_S5_B7</code> is a 16-bit unsigned fixed-point data type with Slope of 5 and Bias of 7
<code>B</code>	Bias	<code>ufix16_S5_B7</code> is a 16-bit unsigned fixed-point data type with Slope of 5 and Bias of 7
<code>E</code>	Fixed exponent (2^{\wedge}) A negative fixed exponent describes the fraction length	<code>sfix32_En31</code> is a 32-bit signed fixed-point data type with a fraction length of 31
<code>F</code>	Slope adjustment factor	<code>ufix16_F1p5_En50</code> is a 16-bit unsigned fixed-point data type with a SlopeAdjustmentFactor of 1.5 and a FixedExponent of -50

Symbol	Description	Example
C,c,D, or d	Compressed encoding for Bias	No example available. For backwards compatibility only.
	Note: If you pass this symbol to the <code>slDataTypeAndScale</code> function, it returns a valid <code>fixdt</code> data type.	To identify and replace calls to <code>slDataTypeAndScale</code> , use the “Check for calls to <code>slDataTypeAndScale</code> ” (Simulink) Model Advisor check.
T or t	Compressed encoding for Slope	No example available. For backwards compatibility only.
	Note: If you pass this symbol to the <code>slDataTypeAndScale</code> , it returns a valid <code>fixdt</code> data type.	To identify and replace calls to <code>slDataTypeAndScale</code> , use the “Check for calls to <code>slDataTypeAndScale</code> ” (Simulink) Model Advisor check.

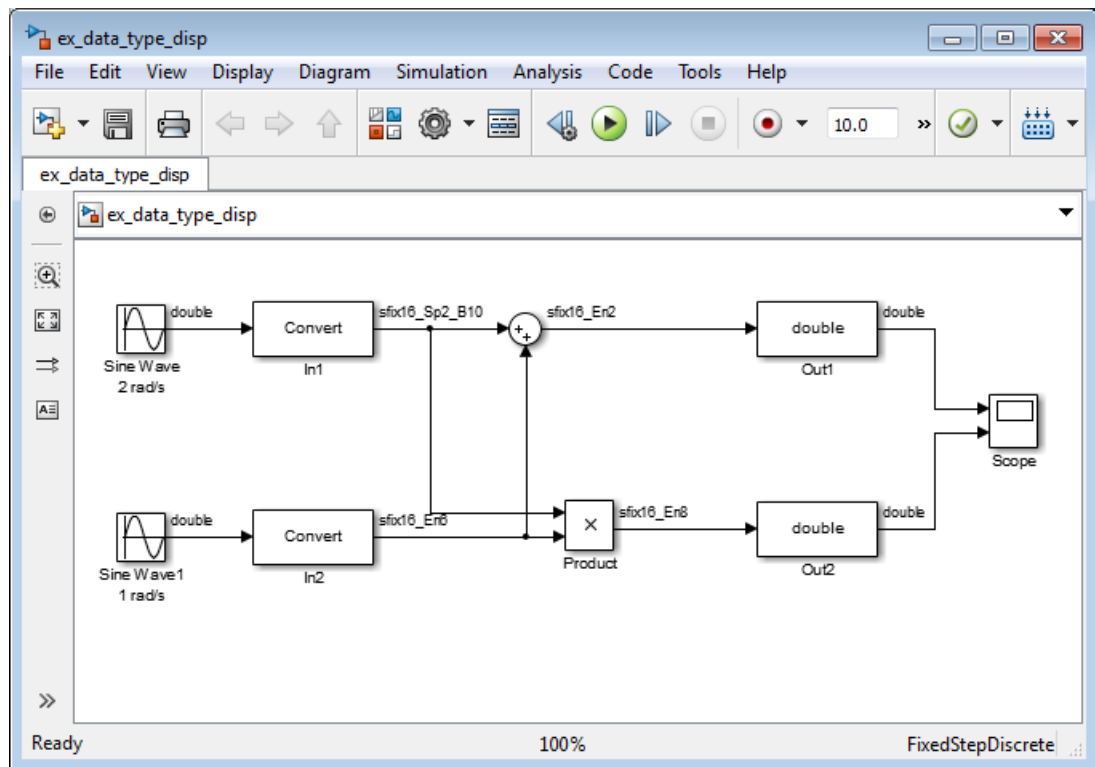
Display Port Data Types

To display the data types for the ports in your model.

- 1 From the Simulink **Display** menu, select **Signals and Ports**, and then select **Port Data Types**.

The port display for fixed-point signals consists of three parts: the data type, the number of bits, and the scaling. These three parts reflect the block **Output data type** parameter value or the data type and scaling that is inherited from the driving block or through back propagation.

The following model displays its port data types.



In the model, the data type displayed with the In1 block indicates that the output data type name is `sfix16_Sp2_B10`. This corresponds to `fixdt(1, 16, 0.2, 10)` which

is a signed 16 bit fixed-point number with slope 0.2 and bias 10.0. The data type displayed with the In2 block indicates that the output data type name is `sfix16_En6`. This corresponds to `fixdt(1, 16, 6)` which is a signed 16 bit fixed-point number with fraction length of 6.

Scaled Doubles

What Are Scaled Doubles?

Scaled doubles are a hybrid between floating-point and fixed-point numbers. The Fixed-Point Designer software stores them as doubles with the scaling, sign, and word length information retained. For example, the storage container for a fixed-point data type `sfixed16_En14` is `int16`. The storage container of the equivalent scaled doubles data type, `flts16_En14` is floating-point double. For details of the fixed-point scaling notation, see “Fixed-Point Data Type and Scaling Notation” on page 32-17. The Fixed-Point Designer software applies the scaling information to the stored floating-point double to obtain the real-world value. Storing the value in a double almost always eliminates overflow and precision issues.

What is the Difference between Scaled Double and Double Data Types?

The storage container for both the scaled double and double data types is floating-point double. Therefore both data type override settings, `Double` and `Scaled double`, provide the range and precision advantages of floating-point doubles. Scaled doubles retain the information about the specified data type and scaling, but doubles do not retain this information. Because scaled doubles retain the information about the specified scaling, they can also be used for overflow detection.

Consider an example where you are storing 0.75001 degrees Celsius in a data type `sfixed16_En13`. For this data type:

- The slope, $S = 2^{-13}$.
- The bias, $B = 0$.

Using the scaling equation $V \approx \tilde{V} = SQ + B$, where V is the real-world value and Q is the stored value.

- $B = 0$.
- $\tilde{V} = SQ = 2^{-13}Q = 0.75001$.

Because the storage container of the data type `sfixed16_En13` is 16 bits, the stored integer Q can only be represented as an integer within these 16 bits, so the ideal value of Q is quantized to 6144 causing precision loss.

If you override the data type `sfix16_En13` with `Double`, the data type changes to `Double` and you lose the information about the scaling. The stored-value equals the real-world value `0.75001`.

If you override the data type `sfix16_En13` with `Scaled Double`, the data type changes to `flts16_En13`. The scaling is still given by `_En13` and is identical to that of the original data type. The only difference is the storage container used to hold the stored value which is now `double` so the stored-value is `6144.08192`. This example shows one advantage of using scaled doubles: the virtual elimination of quantization errors.

When to Use Scaled Doubles

The Fixed-Point Tool enables you to perform various data type overrides on fixed-point signals in your simulations. Use scaled doubles to override the fixed-point data types and scaling using double-precision numbers to avoid quantization effects. Overriding the fixed-point data types provides a floating-point benchmark that represents the ideal output.

Scaled doubles are useful for:

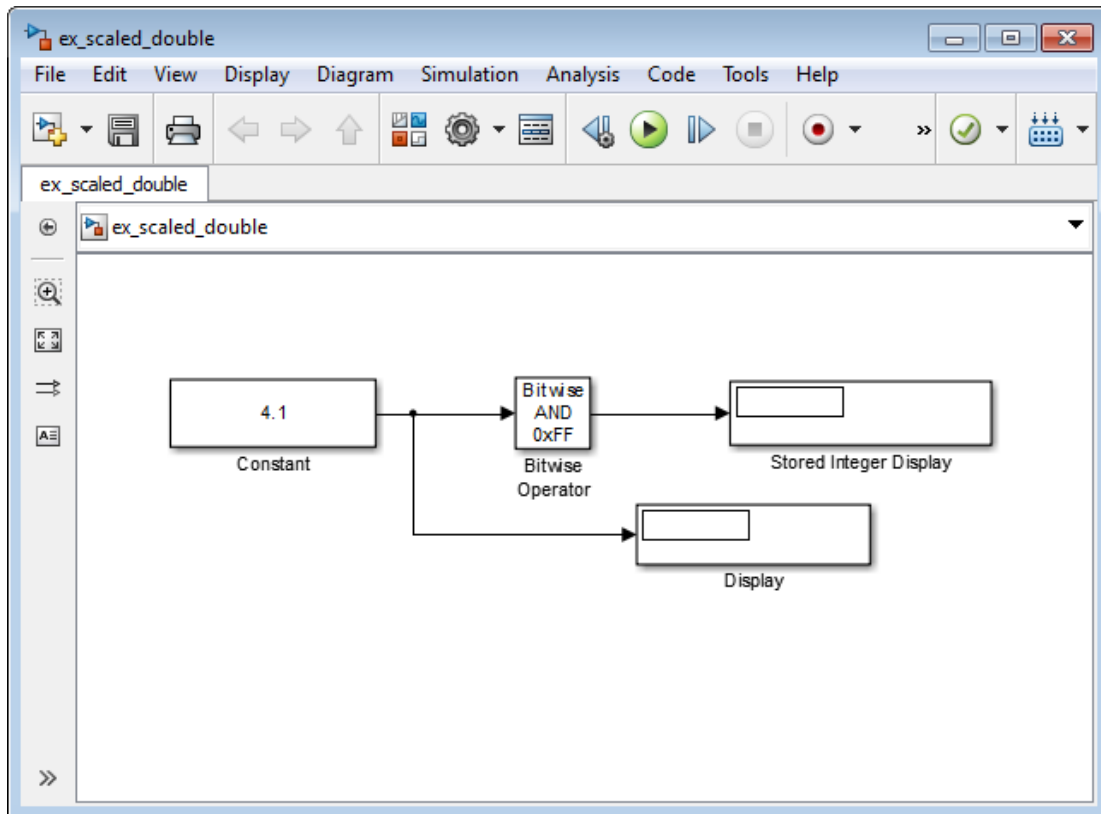
- Testing and debugging
- Detecting overflows
- Applying data type overrides to individual subsystems

If you apply a data type override to subsystems in your model rather than to the whole model, Scaled doubles provide the information that the fixed-point portions of the model need for consistent data type propagation.

Use Scaled Doubles to Avoid Precision Loss

This example uses the `ex_scaled_double` model to show how you can avoid precision loss by overriding the data types in your model with scaled doubles. For more information about scaled doubles, see “Scaled Doubles” on page 32-22.

About the Model



In this model:

- The Constant block output data type is `fixdt(1,8,4)`.
- The Bitwise Operator block uses the AND operator and the bit mask `0xFF` to pass the input value to the output. Because the **Treat mask as** parameter is set to **Stored**

Integer, the block outputs the stored integer value, S , of its input. The encoding scheme is $V = SQ + B$, where V is the real-world value and Q is the stored integer value. For more information, see “Scaling” on page 32-6.

Running the Example

- 1 Open the `ex_scaled_double` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_scaled_double
```

- 2 From the model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.

The Fixed-Point Tool opens.

- 3 In the Fixed-Point Tool, set the **Data type override** parameter to `Use local settings` and click **Apply**.

- 4 From the model menu, select **Simulation > Run**.

The simulation runs and the **Display** block displays `4.125` as the output value of the **Constant** block. The **Stored Integer Display** block displays `0100 0010`, which is the binary equivalent of the stored integer value. Precision loss occurs because the output data type, `fixdt(1,8,4)`, cannot represent the output value `4.1` exactly.

- 5 In the Fixed-Point Tool, set the **Data type override** parameter to `Scaled double` and the **Data type override applies to** parameter to `All numeric types`. Then click **Apply** and rerun the simulation.

Note: You cannot use a **Data type override** setting of `Double` because the **Bitwise Operator** block does not support floating-point data types.

The simulation runs and this time the **Display** block correctly displays `4.1` as the output value of the **Constant** block. The **Stored Integer Display** block displays `65`, which is the binary equivalent of the stored integer value. Because the model uses scaled doubles to override the data type `fixdt(1,8,4)`, the compiled output data type changes to `flts8_En4`, which is the scaled doubles equivalent of `fixdt(1,8,4)`. No precision loss occurs because the scaled doubles retain the information about the specified data type and scaling, and they use a double to hold the stored value.

Floating-Point Numbers

In this section...

“Floating-Point Numbers” on page 32-26

“Scientific Notation” on page 32-26

“The IEEE Format” on page 32-27

“Range and Precision” on page 32-29

“Exceptional Arithmetic” on page 32-31

Floating-Point Numbers

Fixed-point numbers are limited in that they cannot simultaneously represent very large or very small numbers using a reasonable word size. This limitation can be overcome by using scientific notation. With scientific notation, you can dynamically place the binary point at a convenient location and use powers of the binary to keep track of that location. Thus, you can represent a range of very large and very small numbers with only a few digits.

You can represent any binary floating-point number in scientific notation form as $f2^e$, where f is the fraction (or mantissa), 2 is the radix or base (binary in this case), and e is the exponent of the radix. The radix is always a positive number, while f and e can be positive or negative.

When performing arithmetic operations, floating-point hardware must take into account that the sign, exponent, and fraction are all encoded within the same binary word. This results in complex logic circuits when compared with the circuits for binary fixed-point operations.

The Fixed-Point Designer software supports single-precision and double-precision floating-point numbers as defined by the IEEE Standard 754.

Scientific Notation

A direct analogy exists between scientific notation and radix point notation. For example, scientific notation using five decimal digits for the fraction would take the form

$$\pm d.ddd \times 10^p = \pm dddd.0 \times 10^{p-4} = \pm 0.ddd \times 10^{p+1},$$

where $d = 0, \dots, 9$ and p is an integer of unrestricted range.

Radix point notation using five bits for the fraction is the same except for the number base

$$\pm b.bbbb \times 2^q = \pm bbbb.b \times 2^{q-4} = \pm 0.bbbb \times 2^{q+1},$$

where $b = 0, 1$ and q is an integer of unrestricted range.

For fixed-point numbers, the exponent is fixed but there is no reason why the binary point must be contiguous with the fraction. For more information, see “Binary-Point-Only Scaling” on page 32-7.

The IEEE Format

The IEEE Standard 754 has been widely adopted, and is used with virtually all floating-point processors and arithmetic coprocessors—with the notable exception of many DSP floating-point processors.

Among other things, this standard specifies four floating-point number formats, of which singles and doubles are the most widely used. Each format contains three components: a sign bit, a fraction field, and an exponent field. These components, as well as the specific formats for singles and doubles, are discussed in the sections that follow.

The Sign Bit

While two's complement is the preferred representation for signed fixed-point numbers, IEEE floating-point numbers use a sign/magnitude representation, where the sign bit is explicitly included in the word. Using this representation, a sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number.

The Fraction Field

In general, floating-point numbers can be represented in many different ways by shifting the number to the left or right of the binary point and decreasing or increasing the exponent of the binary by a corresponding amount.

To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction of the form $1.f$ where f has a fixed size for a given data type. Since the leftmost fraction bit is always a 1, it is unnecessary to store

this bit and is therefore implicit (or hidden). Thus, an n -bit fraction stores an $n+1$ -bit number. The IEEE format also supports denormalized numbers, which have a fraction of the form $0.f$. Normalized and denormalized formats are discussed in more detail in the next section.

The Exponent Field

In the IEEE format, exponent representations are biased. This means a fixed value (the bias) is subtracted from the field to get the true exponent value. For example, if the exponent field is 8 bits, then the numbers 0 through 255 are represented, and there is a bias of 127. Note that some values of the exponent are reserved for flagging Inf (infinity), NaN (not-a-number), and denormalized numbers, so the true exponent values range from -126 to 127. See the sections “Inf” on page 32-32 and “NaN” on page 32-32.

Single-Precision Format

The IEEE single-precision floating-point format is a 32-bit word divided into a 1-bit sign indicator s , an 8-bit biased exponent e , and a 23-bit fraction f . For more information, see “The Sign Bit” on page 32-27, “The Exponent Field” on page 32-28, and “The Fraction Field” on page 32-27. A representation of this format is given below.



The relationship between this format and the representation of real numbers is given by

$$value = \begin{cases} (-1)^s (2^{e-127})(1.f) & \text{normalized, } 0 < e < 255, \\ (-1)^s (2^{e-126})(0.f) & \text{denormalized, } e = 0, f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

“Exceptional Arithmetic” on page 32-31 discusses denormalized values.

Double-Precision Format

The IEEE double-precision floating-point format is a 64-bit word divided into a 1-bit sign indicator s , an 11-bit biased exponent e , and a 52-bit fraction f . For more information,

see “The Sign Bit” on page 32-27, “The Exponent Field” on page 32-28, and “The Fraction Field” on page 32-27. A representation of this format is shown in the following figure.



The relationship between this format and the representation of real numbers is given by

$$value = \begin{cases} (-1)^s (2^{e-1023})(1.f) & \text{normalized, } 0 < e < 2047, \\ (-1)^s (2^{e-1022})(0.f) & \text{denormalized, } e = 0, f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

“Exceptional Arithmetic” on page 32-31 discusses denormalized values.

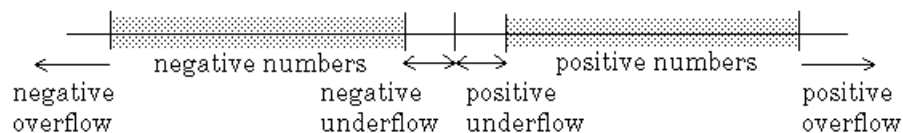
Range and Precision

The range of a number gives the limits of the representation while the precision gives the distance between successive numbers in the representation. The range and precision of an IEEE floating-point number depend on the specific format.

Range

The range of representable numbers for an IEEE floating-point number with f bits allocated for the fraction, e bits allocated for the exponent, and the bias of e given by

$bias = 2^{(e-1)} - 1$ is given below.



where

- Normalized positive numbers are defined within the range $2^{(1-bias)}$ to $(2-2^{-f})2^{bias}$.
- Normalized negative numbers are defined within the range $-2^{(1-bias)}$ to $-(2-2^{-f})2^{bias}$.
- Positive numbers greater than $(2-2^{-f})2^{bias}$ and negative numbers greater than $-(2-2^{-f})2^{bias}$ are overflows.
- Positive numbers less than $2^{(1-bias)}$ and negative numbers less than $-2^{(1-bias)}$ are either underflows or denormalized numbers.
- Zero is given by a special bit pattern, where $e = 0$ and $f = 0$.

Overflows and underflows result from exceptional arithmetic conditions. Floating-point numbers outside the defined range are always mapped to $\pm\text{Inf}$.

Note: You can use the MATLAB commands `realmin` and `realmax` to determine the dynamic range of double-precision floating-point values for your computer.

Precision

Because of a finite word size, a floating-point number is only an approximation of the “true” value. Therefore, it is important to have an understanding of the precision (or accuracy) of a floating-point result. In general, a value v with an accuracy q is specified by $v \pm q$. For IEEE floating-point numbers,

$$v = (-1)^s(2^{e-bias})(1.f)$$

and

$$q = 2^{-f} \times 2^{e-bias}$$

Thus, the precision is associated with the number of bits in the fraction field.

Note: In the MATLAB software, floating-point relative accuracy is given by the command `eps`, which returns the distance from 1.0 to the next larger floating-point number. For a computer that supports the IEEE Standard 754, `eps` = 2^{-52} or $2.22045 \cdot 10^{-16}$.

Floating-Point Data Type Parameters

The high and low limits, exponent bias, and precision for the supported floating-point data types are given in the following table.

Data Type	Low Limit	High Limit	Exponent Bias	Precision
Single	$2^{-126} \approx 10^{-38}$	$2^{128} \approx 3 \cdot 10^{38}$	127	$2^{-23} \approx 10^{-7}$
Double	$2^{-1022} \approx 2 \cdot 10^{-308}$	$2^{1024} \approx 2 \cdot 10^{308}$	1023	$2^{-52} \approx 10^{-16}$

Because of the sign/magnitude representation of floating-point numbers, there are two representations of zero, one positive and one negative. For both representations $e = 0$ and $f.0 = 0.0$.

Exceptional Arithmetic

In addition to specifying a floating-point format, the IEEE Standard 754 specifies practices and procedures so that predictable results are produced independently of the hardware platform. Specifically, denormalized numbers, `Inf`, and `NaN` are defined to deal with exceptional arithmetic (underflow and overflow).

If an underflow or overflow is handled as `Inf` or `NaN`, then significant processor overhead is required to deal with this exception. Although the IEEE Standard 754 specifies practices and procedures to deal with exceptional arithmetic conditions in a consistent manner, microprocessor manufacturers might handle these conditions in ways that depart from the standard.

Denormalized Numbers

Denormalized numbers are used to handle cases of exponent underflow. When the exponent of the result is too small (i.e., a negative exponent with too large a magnitude), the result is denormalized by right-shifting the fraction and leaving the exponent at its minimum value. The use of denormalized numbers is also referred to as gradual underflow. Without denormalized numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normalized numbers. Thus, denormalized numbers provide extended range for small numbers at the expense of precision.

Inf

Arithmetic involving **Inf** (infinity) is treated as the limiting case of real arithmetic, with infinite values defined as those outside the range of representable numbers, or $-\infty \leq (\text{representable numbers}) < \infty$. With the exception of the special cases discussed below (**NaN**), any arithmetic operation involving **Inf** yields **Inf**. **Inf** is represented by the largest biased exponent allowed by the format and a fraction of zero.

NaN

A **NaN** (not-a-number) is a symbolic entity encoded in floating-point format. There are two types of **NaN**: signaling and quiet. A signaling **NaN** signals an invalid operation exception. A quiet **NaN** propagates through almost every arithmetic operation without signaling an exception. The following operations result in a **NaN**: $\infty - \infty$, $-\infty + \infty$, $0 \times \infty$, $0/0$, and ∞/∞ .

Both types of **NaN** are represented by the largest biased exponent allowed by the format and a fraction that is nonzero. The bit pattern for a quiet **NaN** is given by $0.f$ where the most significant number in f must be a one, while the bit pattern for a signaling **NaN** is given by $0.f$ where the most significant number in f must be zero and at least one of the remaining numbers must be nonzero.

Arithmetic Operations

- “Fixed-Point Arithmetic Operations” on page 33-3
- “Limitations on Precision” on page 33-4
- “Rounding” on page 33-5
- “Rounding Modes for Fixed-Point Simulink Blocks” on page 33-6
- “Rounding Mode: Ceiling” on page 33-8
- “Rounding Mode: Convergent” on page 33-9
- “Rounding Mode: Floor” on page 33-11
- “Rounding Mode: Nearest” on page 33-12
- “Rounding Mode: Round” on page 33-14
- “Rounding Mode: Simplest” on page 33-16
- “Rounding Mode: Zero” on page 33-20
- “Pad with Trailing Zeros” on page 33-23
- “Limitations on Precision and Errors” on page 33-24
- “Maximize Precision” on page 33-25
- “Net Slope and Net Bias Precision” on page 33-26
- “Detect Net Slope and Bias Precision Issues” on page 33-29
- “Detect Fixed-Point Constant Precision Loss” on page 33-30
- “Limitations on Range” on page 33-31
- “Saturation and Wrapping” on page 33-33
- “Guard Bits” on page 33-36
- “Determine the Range of Fixed-Point Numbers” on page 33-37
- “Handle Overflows in Simulink Models” on page 33-38
- “Recommendations for Arithmetic and Scaling” on page 33-40
- “Parameter and Signal Conversions” on page 33-52
- “Rules for Arithmetic Operations” on page 33-56

- “The Summation Process” on page 33-66
- “The Multiplication Process” on page 33-69
- “The Division Process” on page 33-71
- “Shifts” on page 33-72
- “Conversions and Arithmetic Operations” on page 33-74

Fixed-Point Arithmetic Operations

When developing a dynamic system using floating-point arithmetic, you generally don't have to worry about numerical limitations since floating-point data types have high precision and range. Conversely, when working with fixed-point arithmetic, you must consider these factors when developing dynamic systems:

- **Overflow**

Adding two sufficiently large negative or positive values can produce a result that does not fit into the representation. This will have an adverse effect on the control system.

- **Quantization**

Fixed-point values are rounded. Therefore, the output signal to the plant and the input signal to the control system do not have the same characteristics as the ideal discrete-time signal.

- **Computational noise**

The accumulated errors that result from the rounding of individual terms within the realization introduce noise into the control signal.

- **Limit cycles**

In the ideal system, the output of a stable transfer function (digital filter) approaches some constant for a constant input. With quantization, limit cycles occur where the output oscillates between two values in steady state.

This chapter describes the limitations involved when arithmetic operations are performed using encoded fixed-point variables. It also provides recommendations for encoding fixed-point variables such that simulations and generated code are reasonably efficient.

Limitations on Precision

Computer words consist of a finite numbers of bits. This means that the binary encoding of variables is only an approximation of an arbitrarily precise real-world value. Therefore, the limitations of the binary representation automatically introduce limitations on the precision of the value. For a general discussion of range and precision, refer to “Range and Precision” on page 32-11.

The precision of a fixed-point word depends on the word size and binary point location. Extending the precision of a word can always be accomplished with more bits, but you face practical limitations with this approach. Instead, you must carefully select the data type, word size, and scaling such that numbers are accurately represented. Rounding and padding with trailing zeros are typical methods implemented on processors to deal with the precision of binary words.

Rounding

The result of any operation on a fixed-point number is typically stored in a register that is longer than the number's original format. When the result is put back into the original format, the extra bits must be disposed of. That is, the result must be *rounded*. Rounding involves going from high precision to lower precision and produces quantization errors and computational noise.

Choose a Rounding Mode

To choose the most suitable rounding mode for your application, you need to consider your system requirements and the properties of each rounding mode. The most important properties to consider are:

- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?
- Bias — What is the expected value of the rounded values minus the original values?
- Possibility of overflow — Does the rounding method introduce the possibility of overflow?

For more information on when to use each rounding mode, see “Rounding Methods” on page 1-9.

Choosing a Rounding Mode for Diagnostic Purposes

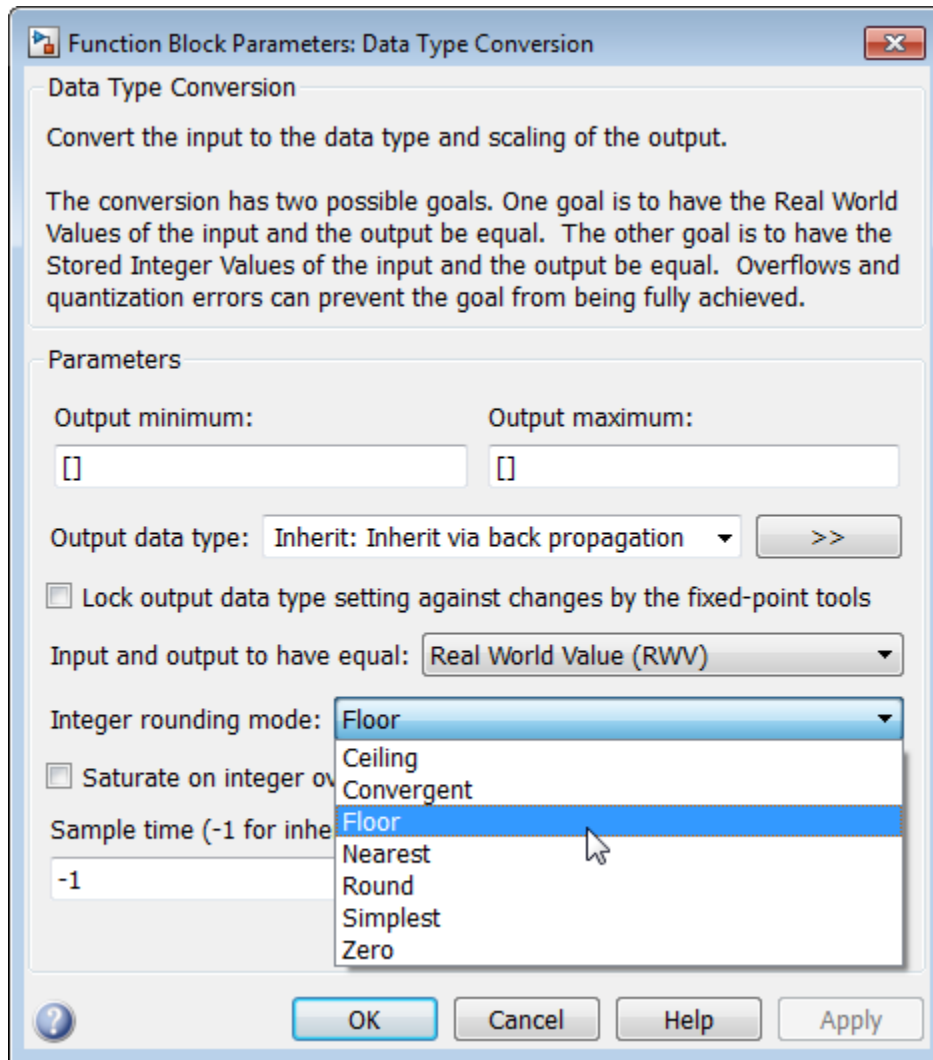
Rounding toward ceiling and rounding toward floor are sometimes useful for diagnostic purposes. For example, after a series of arithmetic operations, you may not know the exact answer because of word-size limitations, which introduce rounding. If every operation in the series is performed twice, once rounding to positive infinity and once rounding to negative infinity, you obtain an upper limit and a lower limit on the correct answer. You can then decide if the result is sufficiently accurate or if additional analysis is necessary.

More About

- “Rounding Modes for Fixed-Point Simulink Blocks” on page 33-6

Rounding Modes for Fixed-Point Simulink Blocks

Fixed-point Simulink blocks support the rounding modes shown in the expanded drop-down menu of the following dialog box.



The following table illustrates the differences between these rounding modes:

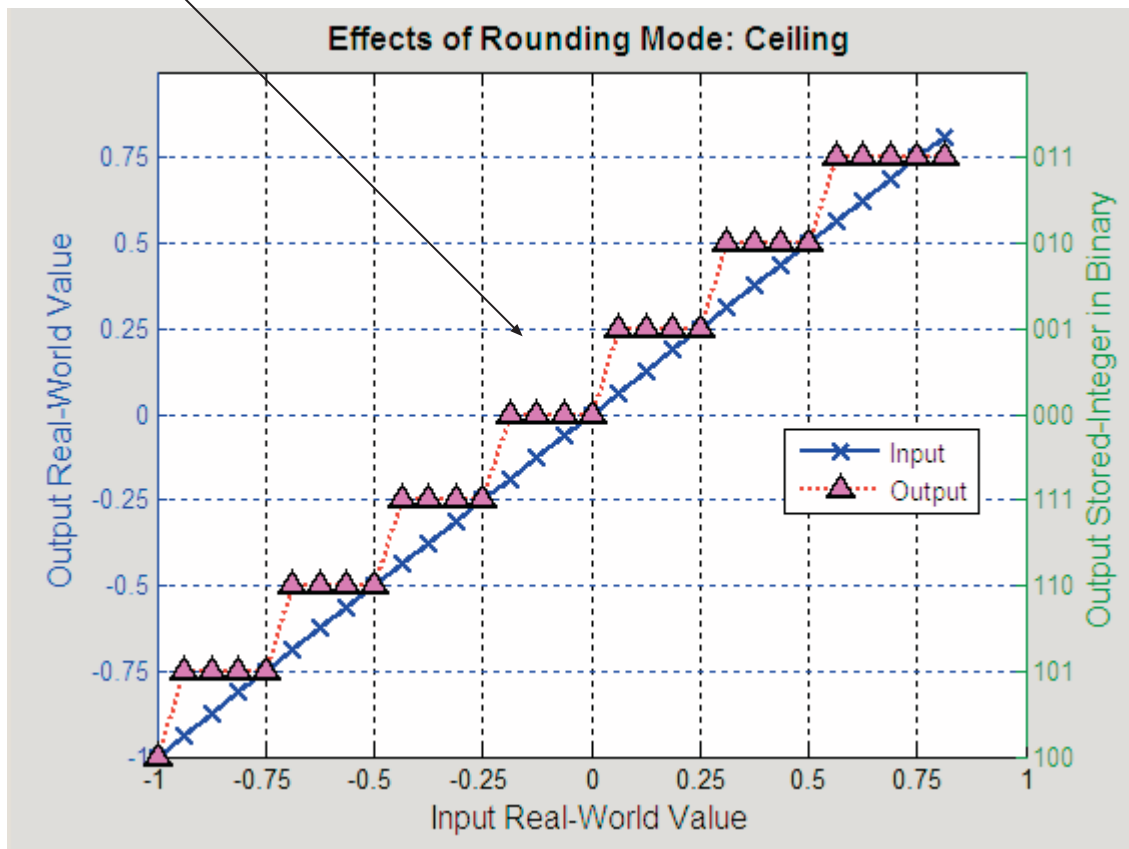
Rounding Mode	Description	Tie Handling
Ceiling	Rounds to the nearest representable number in the direction of positive infinity.	N/A
Floor	Rounds to the nearest representable number in the direction of negative infinity.	N/A
Zero	Rounds to the nearest representable number in the direction of zero.	N/A
Convergent	Rounds to the nearest representable number.	Ties are rounded toward the nearest even integer.
Nearest	Rounds to the nearest representable number.	Ties are rounded to the closest representable number in the direction of positive infinity.
Round	Rounds to the nearest representable number.	<p>For positive numbers, ties are rounded toward the closest representable number in the direction of positive infinity.</p> <p>For negative numbers, ties are rounded toward the closest representable number in the direction of negative infinity.</p>
Simplest	Automatically chooses between Floor and Zero to produce generated code that is as efficient as possible.	N/A

Rounding Mode: Ceiling

When you round toward ceiling, both positive and negative numbers are rounded toward positive infinity. As a result, a positive cumulative bias is introduced in the number.

In the MATLAB software, you can round to ceiling using the `ceil` function. Rounding toward ceiling is shown in the following figure.

All numbers are rounded toward positive infinity

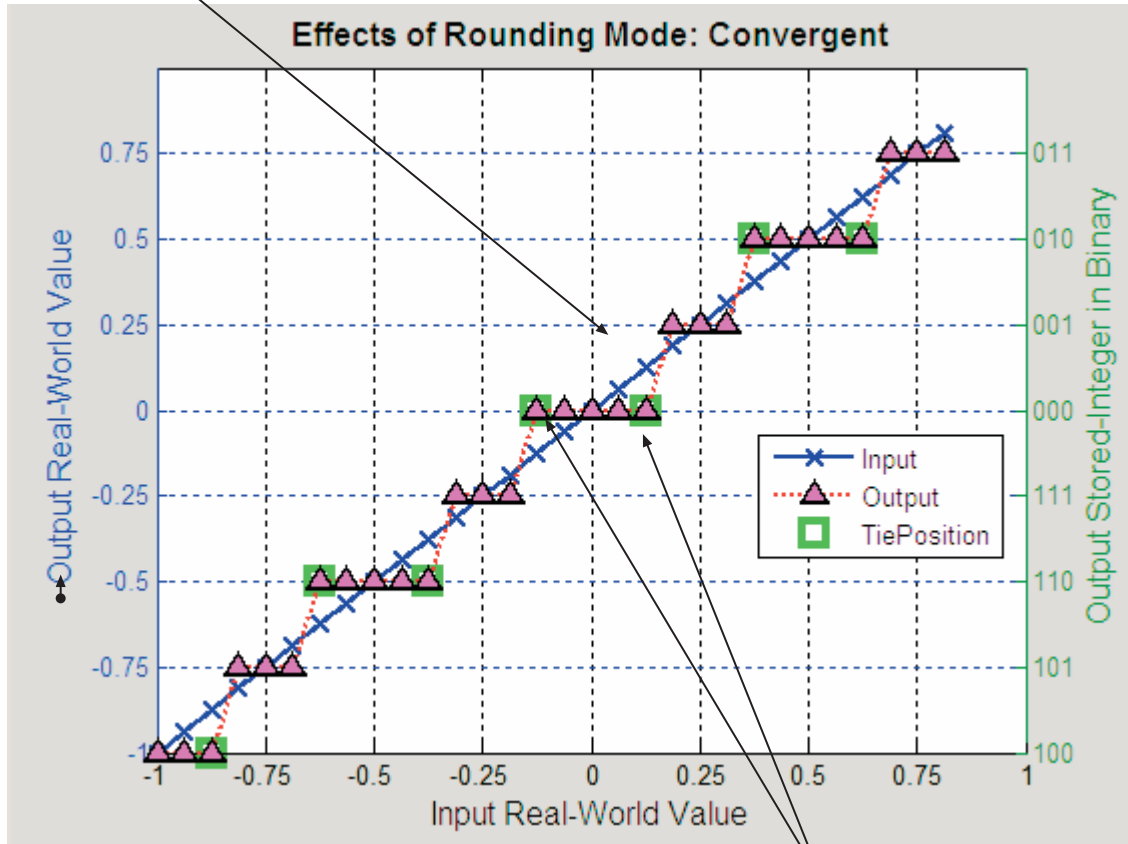


Rounding Mode: Convergent

Convergent rounds toward the nearest representable value with ties rounding toward the nearest even integer. It eliminates bias due to rounding. However, it introduces the possibility of overflow.

In the MATLAB software, you can perform convergent rounding using the `convergent` function. Convergent rounding is shown in the following figure.

All numbers are rounded to the nearest representable number

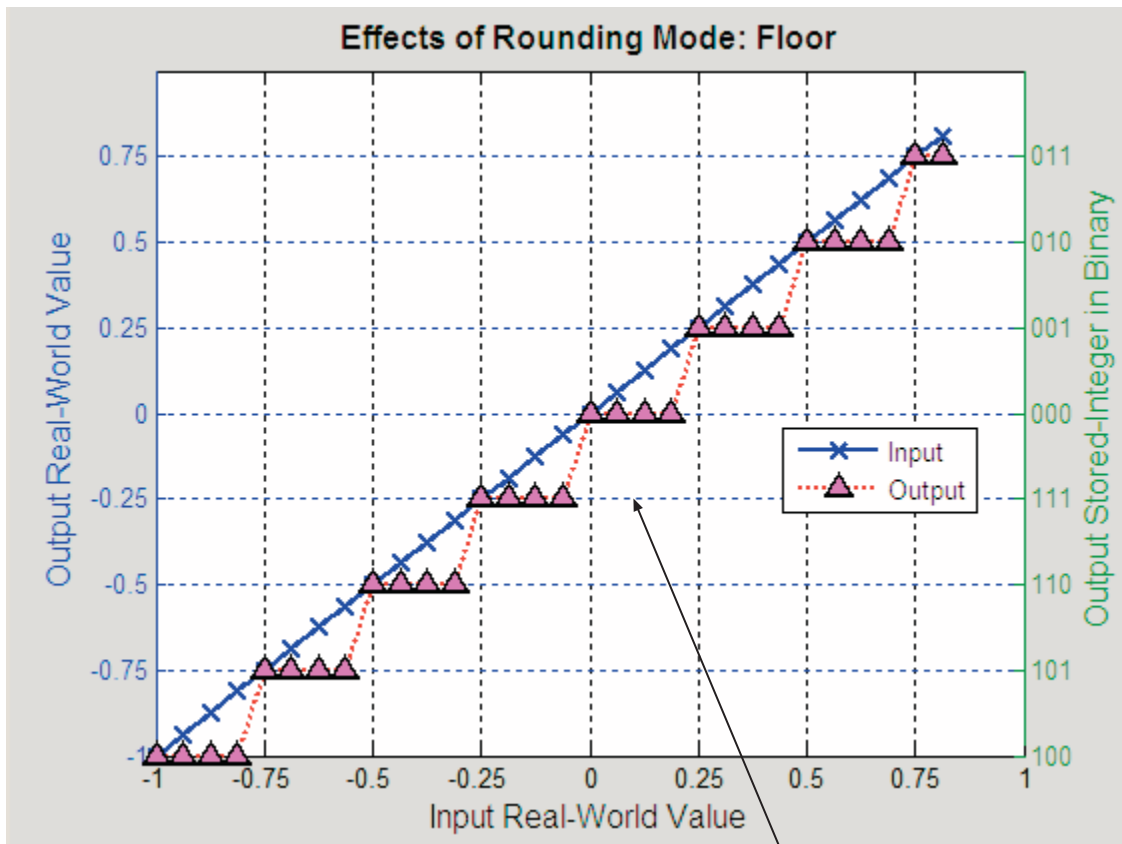


Ties are rounded to the nearest even number

Rounding Mode: Floor

When you round toward floor, both positive and negative numbers are rounded to negative infinity. As a result, a negative cumulative bias is introduced in the number.

In the MATLAB software, you can round to floor using the `floor` function. Rounding toward floor is shown in the following figure.



All numbers are rounded toward negative infinity

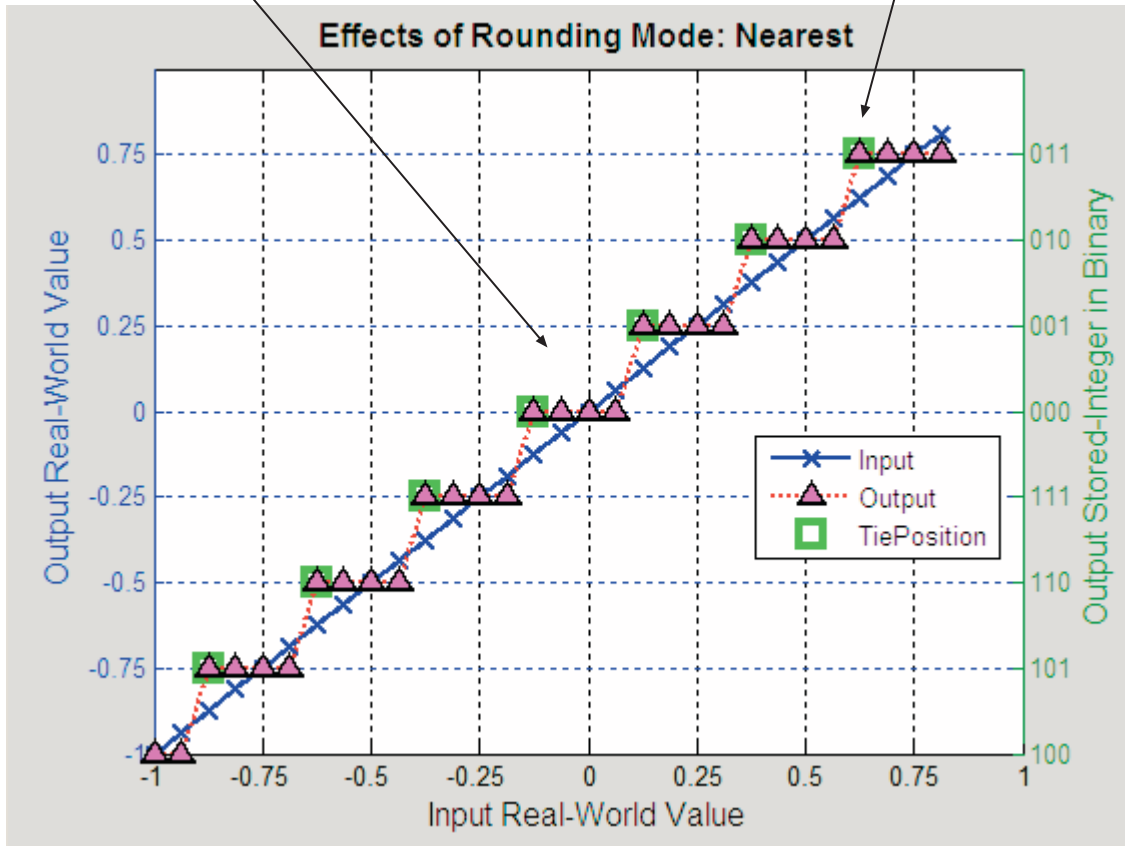
Rounding Mode: Nearest

When you round toward nearest, the number is rounded to the nearest representable value. In the case of a tie, **nearest** rounds to the closest representable number in the direction of positive infinity.

In the Fixed-Point Designer software, you can round to nearest using the **nearest** function. Rounding toward nearest is shown in the following figure.

All numbers are rounded to the nearest representable number

Ties are rounded to the closest representable number in the direction of positive infinity



Rounding Mode: Round

Round rounds to the closest representable number. In the case of a tie, it 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.

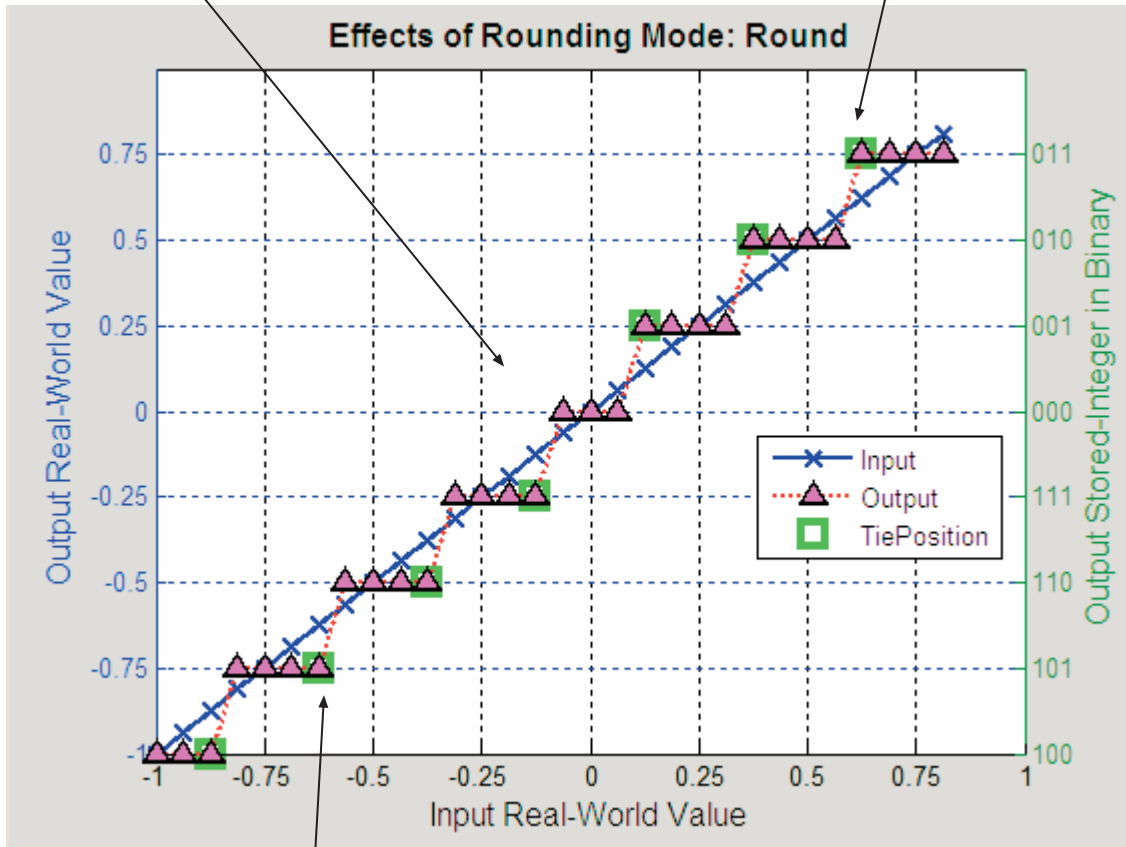
As a result:

- A small negative bias is introduced for negative samples.
- No bias is introduced for samples with evenly distributed positive and negative values.
- A small positive bias is introduced for positive samples.

In the MATLAB software, you can perform this type of rounding using the `round` function. The rounding mode `Round` is shown in the following figure.

All numbers are rounded to the nearest representable number

For positive numbers, ties are rounded to the closest representable number in the direction of positive infinity



For negative numbers, ties are rounded to the closest representable number in the direction of negative infinity

Rounding Mode: Simplest

The simplest rounding mode attempts to reduce or eliminate the need for extra rounding code in your generated code using a combination of techniques, discussed in the following sections:

- “Optimize Rounding for Casts” on page 33-16
- “Optimize Rounding for High-Level Arithmetic Operations” on page 33-16
- “Optimize Rounding for Intermediate Arithmetic Operations” on page 33-18

In nearly all cases, the simplest rounding mode produces the most efficient generated code. For a very specialized case of division that meets three specific criteria, round to floor might be more efficient. These three criteria are:

- Fixed-point/integer signed division
- Denominator is an invariant constant
- Denominator is an exact power of two

For this case, set the rounding mode to floor and the **Model Configuration Parameters > Hardware Implementation > Production Hardware > Signed integer division rounds to** parameter to describe the rounding behavior of your production target.

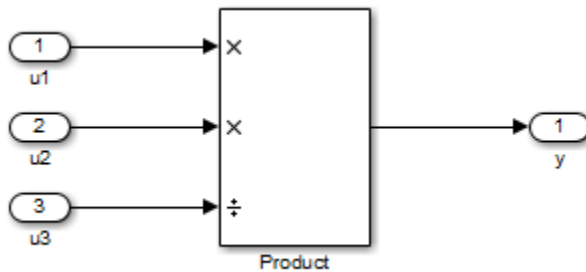
Optimize Rounding for Casts

The Data Type Conversion block casts a signal with one data type to another data type. When the block casts the signal to a data type with a shorter word length than the original data type, precision is lost and rounding occurs. The simplest rounding mode automatically chooses the best rounding for these cases based on the following rules:

- When casting from one integer or fixed-point data type to another, the simplest mode rounds toward floor.
- When casting from a floating-point data type to an integer or fixed-point data type, the simplest mode rounds toward zero.

Optimize Rounding for High-Level Arithmetic Operations

The simplest rounding mode chooses the best rounding for each high-level arithmetic operation. For example, consider the operation $y = u_1 \times u_2 / u_3$ implemented using a Product block:



As stated in the C standard, the most efficient rounding mode for multiplication operations is always floor. However, the C standard does not specify the rounding mode for division in cases where at least one of the operands is negative. Therefore, the most efficient rounding mode for a divide operation with signed data types can be floor or zero, depending on your production target.

The simplest rounding mode:

- Rounds to floor for all nondivision operations.
- Rounds to zero or floor for division, depending on the setting of the **Model Configuration Parameters > Hardware Implementation > Production Hardware > Signed integer division rounds to** parameter.

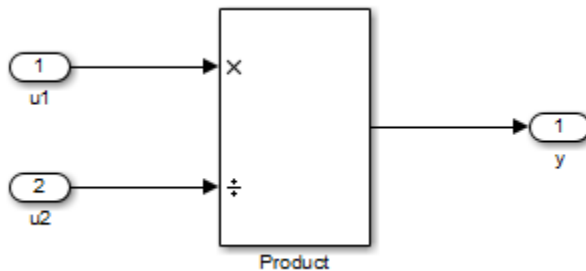
To get the most efficient code, you must set the **Signed integer division rounds to** parameter to specify whether your production target rounds to zero or to floor for integer division. Most production targets round to zero for integer division operations. Note that **Simplest** rounding enables “mixed-mode” rounding for such cases, as it rounds to floor for multiplication and to zero for division.

If the **Signed integer division rounds to** parameter is set to **Undefined**, the simplest rounding mode might not be able to produce the most efficient code. The simplest mode rounds to zero for division for this case, but it cannot rely on your production target to perform the rounding, because the parameter is **Undefined**. Therefore, you need additional rounding code to ensure rounding to zero behavior.

Note: For signed fixed-point division where the denominator is an invariant constant power of 2, the simplest rounding mode does not generate the most efficient code. In this case, set the rounding mode to floor.

Optimize Rounding for Intermediate Arithmetic Operations

For fixed-point arithmetic with nonzero slope and bias, the simplest rounding mode also chooses the best rounding for each intermediate arithmetic operation. For example, consider the operation $y = u_1 / u_2$ implemented using a Product block, where u_1 and u_2 are fixed-point quantities:



As discussed in “Data Types and Scaling in Digital Hardware” on page 32-2, each fixed-point quantity is calculated using its slope, bias, and stored integer. So in this example, not only is there the high-level divide called for by the block operation, but intermediate additions and multiplies are performed:

$$y = \frac{u_1}{u_2} = \frac{S_1 Q_1 + B_1}{S_2 Q_2 + B_2}$$

The simplest rounding mode performs the best rounding for each of these operations, high-level and intermediate, to produce the most efficient code. The rules used to select the appropriate rounding for intermediate arithmetic operations are the same as those described in “Optimize Rounding for High-Level Arithmetic Operations” on page 33-16. Again, this enables mixed-mode rounding, with the most common case being round toward floor used for additions, subtractions, and multiplies, and round toward zero used for divides.

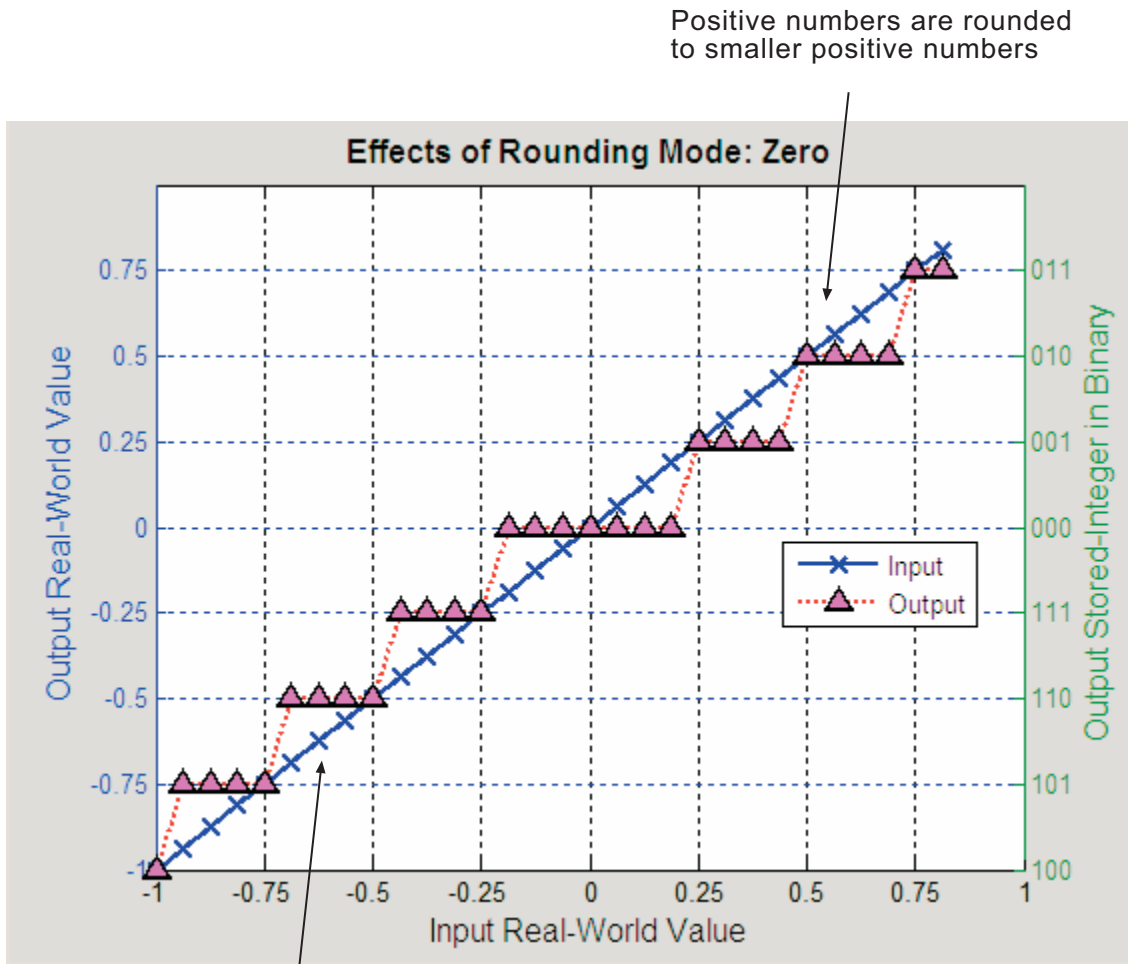
Remember that generating the most efficient code using the simplest rounding mode requires you to set the **Model Configuration Parameters > Hardware Implementation > Production Hardware > Signed integer division rounds to** parameter to describe the rounding behavior of your production target.

Note: For signed fixed-point division where the denominator is an invariant constant power of 2, the simplest rounding mode does not generate the most efficient code. In this case, set the rounding mode to floor.

Rounding Mode: Zero

Rounding towards zero is the simplest rounding mode computationally. All digits beyond the number required are dropped. Rounding towards zero results in a number whose magnitude is always less than or equal to the more precise original value. In the MATLAB software, you can round to zero using the `fix` function.

Rounding toward zero introduces a cumulative downward bias in the result for positive numbers and a cumulative upward bias in the result for negative numbers. That is, all positive numbers are rounded to smaller positive numbers, while all negative numbers are rounded to smaller negative numbers. Rounding toward zero is shown in the following figure.



Rounding to Zero Versus Truncation

Rounding to zero and *truncation* or *chopping* are sometimes thought to mean the same thing. However, the results produced by rounding to zero and truncation are different for unsigned and two's complement numbers. For this reason, the ambiguous term “truncation” is not used in this guide, and explicit rounding modes are used instead.

To illustrate this point, consider rounding a 5-bit unsigned number to zero by dropping (truncating) the two least significant bits. For example, the unsigned number $100.01 = 4.25$ is truncated to $100 = 4$. Therefore, truncating an unsigned number is equivalent to rounding to zero *or* rounding to floor.

Now consider rounding a 5-bit two's complement number by dropping the two least significant bits. At first glance, you may think truncating a two's complement number is the same as rounding to zero. For example, dropping the last two digits of -3.75 yields -3.00 . However, digital hardware performing two's complement arithmetic yields a different result. Specifically, the number $100.01 = -3.75$ truncates to $100 = -4$, which is rounding to floor.

Pad with Trailing Zeros

Padding with trailing zeros involves extending the least significant bit (LSB) of a number with extra bits. This method involves going from low precision to higher precision.

For example, suppose two numbers are subtracted from each other. First, the exponents must be aligned, which typically involves a right shift of the number with the smaller value. In performing this shift, significant digits can “fall off” to the right. However, when the appropriate number of extra bits is appended, the precision of the result is maximized. Consider two 8-bit fixed-point numbers that are close in value and subtracted from each other:

$$1.0000000 \times 2^q - 1.1111111 \times 2^{q-1},$$

where q is an integer. To perform this operation, the exponents must be equal:

$$\frac{1.0000000 \times 2^q - 0.1111111 \times 2^q}{0.0000001 \times 2^q}.$$

If the top number is padded by two zeros and the bottom number is padded with one zero, then the above equation becomes

$$\frac{1.000000000 \times 2^q - 0.111111110 \times 2^q}{0.00000010 \times 2^q},$$

which produces a more precise result. An example of padding with trailing zeros in a Simulink model is illustrated in “Digital Controller Realization” on page 39-38.

Limitations on Precision and Errors

Fixed-point variables have a limited precision because digital systems represent numbers with a finite number of bits. For example, suppose you must represent the real-world number 35.375 with a fixed-point number. Using the encoding scheme described in “Scaling” on page 32-6, the representation is

$$V \approx \tilde{V} = SQ + B = 2^{-2}Q + 32,$$

where $V = 35.375$.

The two closest approximations to the real-world value are $Q = 13$ and $Q = 14$:

$$\tilde{V} = 2^{-2}(13) + 32 = 35.25,$$

$$\tilde{V} = 2^{-2}(14) + 32 = 35.50.$$

In either case, the absolute error is the same:

$$|\tilde{V} - V| = 0.125 = \frac{S}{2} = \frac{F2^E}{2}.$$

For fixed-point values within the limited range, this represents the worst-case error if round-to-nearest is used. If other rounding modes are used, the worst-case error can be twice as large:

$$|\tilde{V} - V| < F2^E.$$

Maximize Precision

Precision is limited by slope. To achieve maximum precision, you should make the slope as small as possible while keeping the range adequately large. The bias is adjusted in coordination with the slope.

Assume the maximum and minimum real-world values are given by $\max(V)$ and $\min(V)$, respectively. These limits might be known based on physical principles or engineering considerations. To maximize the precision, you must decide upon a rounding scheme and whether overflows saturate or wrap. To simplify matters, this example assumes the minimum real-world value corresponds to the minimum encoded value, and the maximum real-world value corresponds to the maximum encoded value. Using the encoding scheme described in “Scaling” on page 32-6, these values are given by

$$\begin{aligned}\max(V) &= F 2^E (\max(Q)) + B \\ \min(V) &= F 2^E (\min(Q)) + B.\end{aligned}$$

Solving for the slope, you get

$$F 2^E = \frac{\max(V) - \min(V)}{\max(Q) - \min(Q)} = \frac{\max(V) - \min(V)}{2^{ws} - 1}.$$

This formula is independent of rounding and overflow issues, and depends only on the word size, ws .

Net Slope and Net Bias Precision

What are Net Slope and Net Bias?

You can represent a fixed-point number by a general slope and bias encoding scheme

$$V \approx \tilde{V} = SQ + B,$$

where:

- V is an arbitrarily precise real-world value.
- \tilde{V} is the approximate real-world value.
- Q , the stored value, is an integer that encodes V .
- $S = F2^E$ is the slope.
- B is the bias.

For a cast operation,

$$S_a Q_a + B_a = S_b Q_b + B_b$$

or

$$Q_a = \frac{S_b Q_b}{S_a} + \left(\frac{B_b - B_a}{S_a} \right),$$

where:

- $\frac{S_b}{S_a}$ is the net slope.
- $\frac{B_b - B_a}{S_a}$ is the net bias.

Detecting Net Slope and Net Bias Precision Issues

Precision issues might occur in the fixed-point constants, net slope and net bias, due to quantization errors when you convert from floating point to fixed point. These fixed-point constant precision issues can result in numerical inaccuracy in your model.

You can configure your model to alert you when fixed-point constant precision issues occur. For more information, see “Detect Net Slope and Bias Precision Issues” on page 33-29. The Fixed-Point Designer software provides the following information:

- The type of precision issue: underflow, overflow, or precision loss.
- The original value of the fixed-point constant.
- The quantized value of the fixed-point constant.
- The error in the value of the fixed-point constant.
- The block that introduced the error.

This information warns you that the outputs from this block are not accurate. If possible, change the data types in your model to fix the issue.

Fixed-Point Constant Underflow

Fixed-point constant underflow occurs when the Fixed-Point Designer software encounters a fixed-point constant whose data type does not have enough precision to represent the ideal value of the constant, because the ideal value is too close to zero. Casting the ideal value to the fixed-point data type causes the value of the fixed-point constant to become zero. Therefore the value of the fixed-point constant differs from its ideal value.

Fixed-Point Constant Overflow

Fixed-point constant overflow occurs when the Fixed-Point Designer software converts a fixed-point constant to a data type whose range is not large enough to accommodate the ideal value of the constant with reasonable precision. The data type cannot accurately represent the ideal value because the ideal value is either too large or too small. Casting the ideal value to the fixed-point data type causes overflow. For example, suppose the ideal value is 200 and the converted data type is `int8`. Overflow occurs in this case because the maximum value that `int8` can represent is 127.

The Fixed-Point Designer software reports an overflow error if the quantized value differs from the ideal value by more than the precision for the data type. The precision

for a data type is approximately equal to the default scaling (for more information, see “Fixed-Point Data Type Parameters” on page 32-12.) Therefore, for positive values, the Fixed-Point Designer software treats errors greater than the slope as overflows. For negative values, it treats errors greater than or equal to the slope as overflows.

For example, the maximum value that `int8` can represent is `127`. The precision for `int8` is `1.0`. An ideal value of `127.3` quantizes to `127` with an absolute error of `0.3`. Although the ideal value `127.3` is greater than the maximum representable value for `int8`, the quantization error is small relative to the precision of `int8`. Therefore the Fixed-Point Designer software does not report an overflow. However, an ideal value of `128.1` does cause an overflow because the quantization error is `1.1`, which is larger than the precision for `int8`.

Note: Fixed-point constant overflow differs from fixed-point constant precision loss. Precision loss occurs when the ideal fixed-point constant value is within the range of the current data type and scaling, but the software cannot represent this value exactly.

Fixed-Point Constant Precision Loss

Fixed-point constant precision loss occurs when the Fixed-Point Designer software converts a fixed-point constant to a data type without enough precision to represent the exact value of the constant. As a result, the quantized value differs from the ideal value. For an example of this behavior, see “Detect Fixed-Point Constant Precision Loss” on page 33-30.

Note: Fixed-point constant precision loss differs from fixed-point constant overflow. Overflow occurs when the range of the parameter data type, that is, the maximum value that it can represent, is smaller than the ideal value of the parameter.

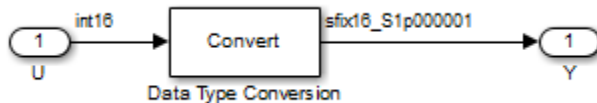
Detect Net Slope and Bias Precision Issues

To receive alerts when fixed-point constant precision issues occur, use these options available in the Simulink Configuration Parameters dialog box, on the **Diagnostics** > **Type Conversion** pane. Set the parameters to warning or error so that Simulink alerts you when precision issues occur.

Configuration Parameter	Specifies	Default
“Detect underflow” (Simulink)	Diagnostic action when a fixed-point constant underflow occurs during simulation	Does not generate a warning or error.
“Detect overflow” (Simulink)	Diagnostic action when a fixed-point constant overflow occurs during simulation	Does not generate a warning or error.
“Detect precision loss” (Simulink)	Diagnostic action when a fixed-point constant precision loss occurs during simulation	Does not generate a warning or error.

Detect Fixed-Point Constant Precision Loss

This example shows how to detect fixed-point constant precision loss. The example uses the following model.



For the Data Type Conversion block in this model, the:

- Input slope, $S_U = 1$
- Output slope, $S_Y = 1.000001$
- Net slope, $S_U/S_Y = 1/1.000001$

When you simulate the model, a net slope quantization error occurs.

To set up the model and run the simulation:

- 1 For the Inport block, set the **Output data type** to `int16`.
- 2 For the Data Type Conversion block, set the **Output data type** to `fixdt(1, 16, 1.000001, 0)`.
- 3 Set the **Diagnostics > Type Conversion > Detect precision loss** configuration parameter to `error`.
- 4 In your Simulink model window, select **Simulation > Run**.

The Fixed-Point Designer software generates an error informing you that net scaling quantization caused precision loss. The message provides the following information:

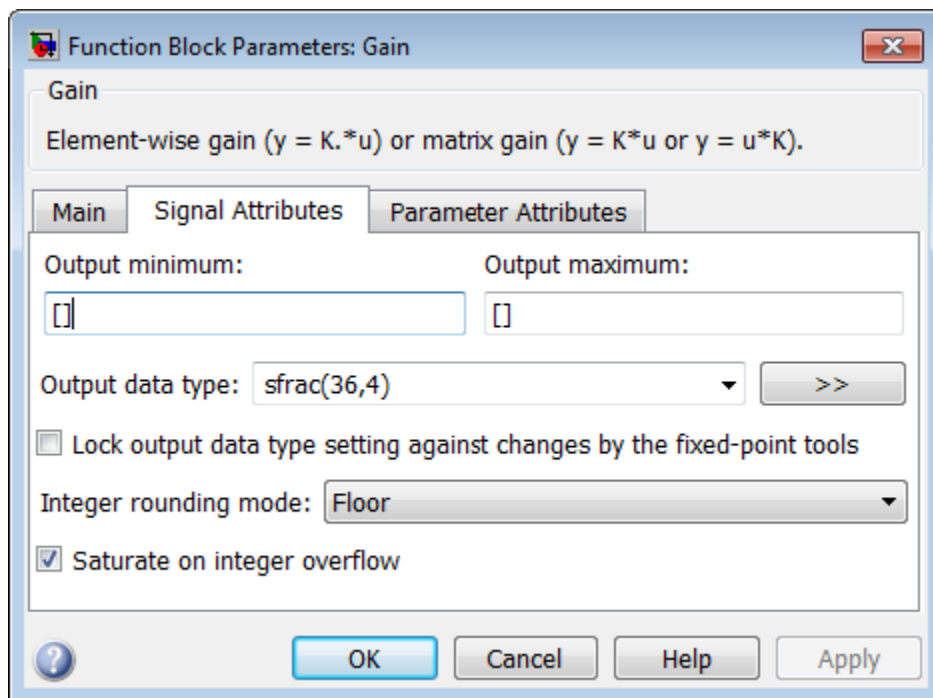
- The block that introduced the error.
- The original value of the net slope.
- The quantized value of the net slope.
- The error in the value of the net slope.

Limitations on Range

Limitations on the range of a fixed-point word occur for the same reason as limitations on its precision. Namely, fixed-point words have limited size. For a general discussion of range and precision, refer to “Range and Precision” on page 32-11.

In binary arithmetic, a processor might need to take an n -bit fixed-point number and store it in m bits, where $m \neq n$. If $m < n$, the range of the number has been reduced and an operation can produce an overflow condition. Some processors identify this condition as `Inf` or `NaN`. For other processors, especially digital signal processors (DSPs), the value *saturates* or *wraps*. If $m > n$, the range of the number has been extended. Extending the range of a word requires the inclusion of *guard bits*, which act to guard against potential overflow. In both cases, the range depends on the word's size and scaling.

The Simulink software supports saturation and wrapping for all fixed-point data types, while guard bits are supported only for fractional data types. As shown in the following figure, you can select saturation or wrapping for fixed-point Simulink blocks with the **Saturate on integer overflow** check box. By setting **Output data type** to `sfrac(36,4)`, you specify a 36-bit signed fractional data type with 4 guard bits (total word size is 40 bits).



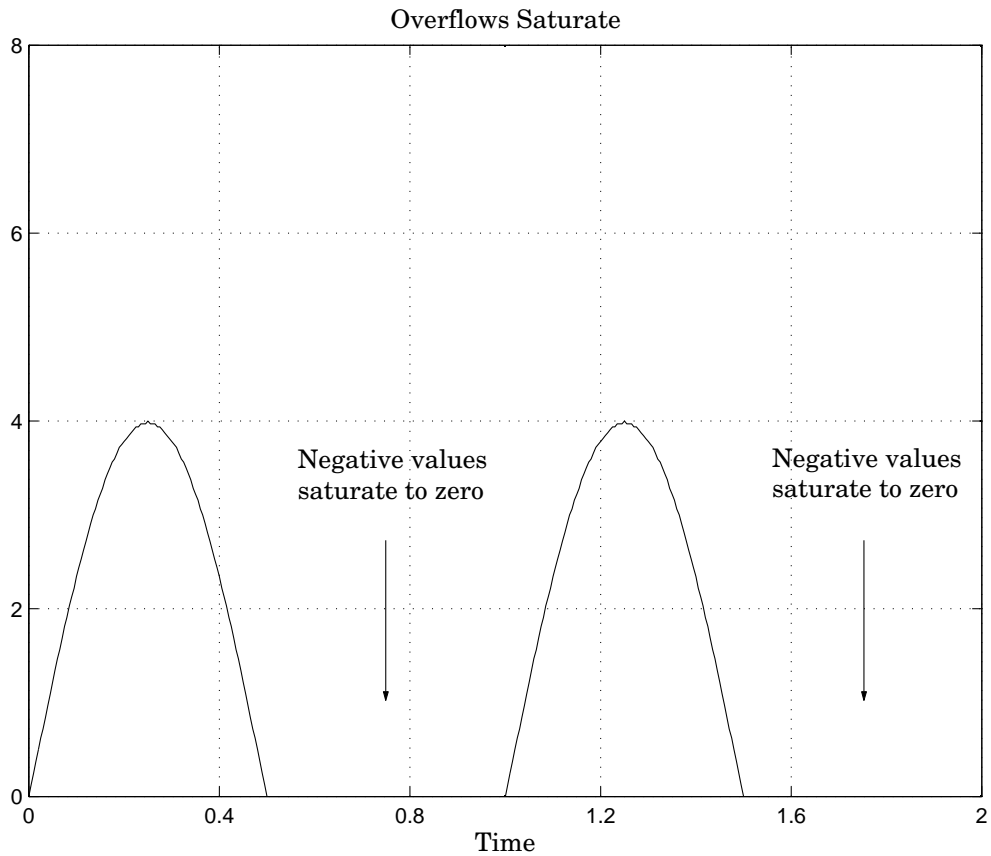
Saturation and Wrapping

What Are Saturation and Wrapping?

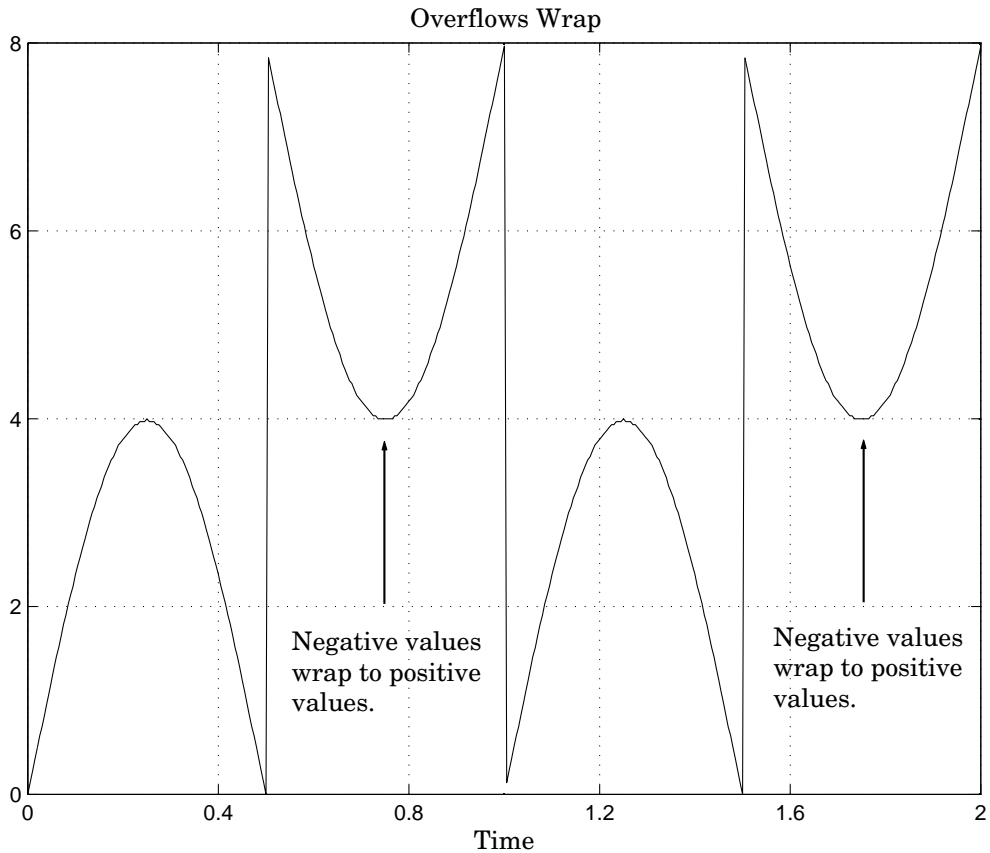
Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, the ADSP-2100 family of processors from Analog Devices® supports either of these modes. If a register has a saturation mode of operation, then an overflow condition is set to the maximum positive or negative value allowed. Conversely, if a register has a wrapping mode of operation, an overflow condition is set to the appropriate value within the range of the representation.

Saturation and Wrapping

Consider an 8-bit unsigned word with binary-point-only scaling of 2^{-5} . Suppose this data type must represent a sine wave that ranges from -4 to 4. For values between 0 and 4, the word can represent these numbers without regard to overflow. This is not the case with negative numbers. If overflows saturate, all negative values are set to zero, which is the smallest number representable by the data type. The saturation of overflows is shown in the following figure.



If overflows wrap, all negative values are set to the appropriate positive value. The wrapping of overflows is shown in the following figure.



Note: For most control applications, saturation is the safer way of dealing with fixed-point overflow. However, some processor architectures allow automatic saturation by hardware. If hardware saturation is not available, then extra software is required, resulting in larger, slower programs. This cost is justified in some designs—perhaps for safety reasons. Other designs accept wrapping to obtain the smallest, fastest software.

Guard Bits

You can eliminate the possibility of overflow by appending the appropriate number of guard bits to a binary word.

For a two's complement signed value, the guard bits are filled with either 0's or 1's depending on the value of the most significant bit (MSB). This is called *sign extension*. For example, consider a 4-bit two's complement number with value 1011. If this number is extended in range to 7 bits with sign extension, then the number becomes 1111101 and the value remains the same.

Guard bits are supported only for fractional data types. For both signed and unsigned fractionals, the guard bits lie to the left of the default binary point.

Determine the Range of Fixed-Point Numbers

Fixed-point variables have a limited range for the same reason they have limited precision—because digital systems represent numbers with a finite number of bits. As a general example, consider the case where an integer is represented as a fixed-point word of size ws . The range for signed and unsigned words is given by

$$\max(Q) - \min(Q),$$

where

$$\min(Q) = \begin{cases} 0 & \text{unsigned,} \\ -2^{ws-1} & \text{signed,} \end{cases}$$

$$\max(Q) = \begin{cases} 2^{ws} - 1 & \text{unsigned,} \\ 2^{ws-1} - 1 & \text{signed.} \end{cases}$$

Using the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6, the approximate real-world value has the range

$$\max(\tilde{V}) - \min(\tilde{V}),$$

where

$$\min(\tilde{V}) = \begin{cases} B & \text{unsigned,} \\ -F 2^E (2^{ws-1}) + B & \text{signed,} \end{cases}$$

$$\max(\tilde{V}) = \begin{cases} F 2^E (2^{ws} - 1) + B & \text{unsigned,} \\ F 2^E (2^{ws-1} - 1) + B & \text{signed.} \end{cases}$$

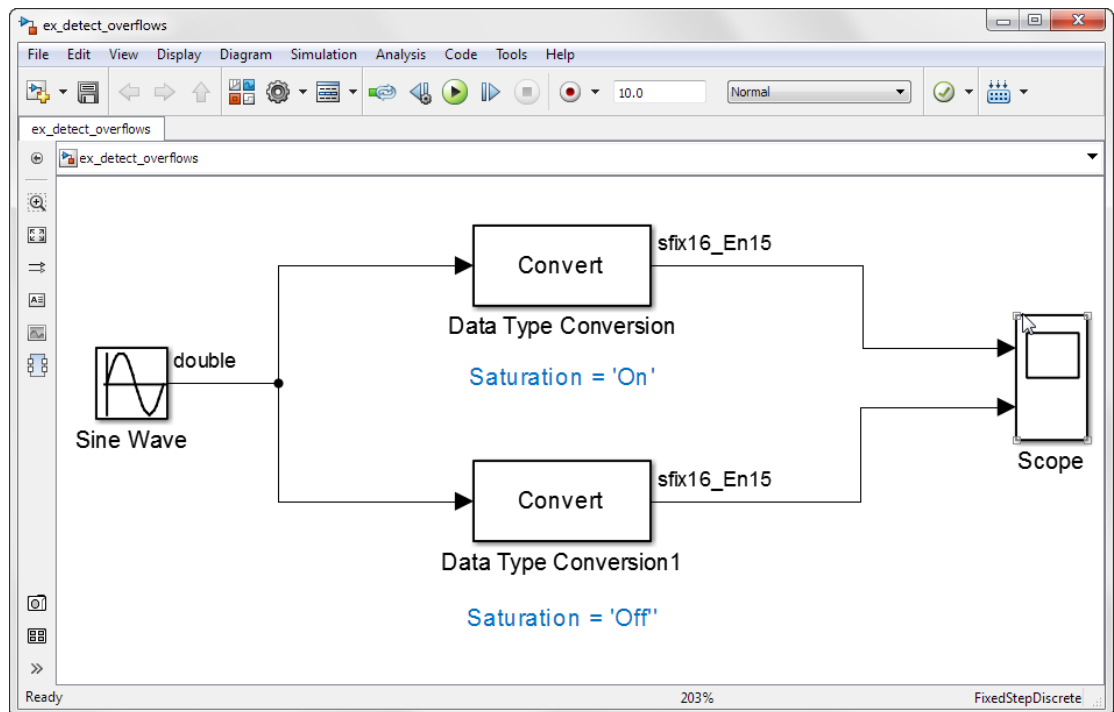
If the real-world value exceeds the limited range of the approximate value, then the accuracy of the representation can become significantly worse.

Handle Overflows in Simulink Models

This example shows how to control the warning messages you receive when a model contains an overflow. This diagnostic control can simplify debugging models in which only one type of overflow is of interest.

- 1 Open the `ex_detect_overflows` model.

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_detect_overflows
```



This model contains a sine wave with an amplitude of 1.5 passed through two Data Type Conversion blocks. In the Data Type Conversion block, the **Saturate on integer overflow** parameter is selected. The Data Type Conversion1 block wraps when the signal is too large to fit into the output data type.

- 2 Simulate the model.

The Diagnostic Viewer displays two overflow warnings. The first overflow saturated and the second overflow wrapped.

- 3** In the Configuration Parameters dialog box:
 - Set **Diagnostics > Data Validity > Wrap on overflow** to Error.
 - Set **Diagnostics > Data Validity > Saturate on overflow** to Warning.
- 4** Simulate the model again.

The Diagnostic Viewer displays an error message for the overflow that wrapped, and a warning message for the overflow that saturated.

For more information, see “Wrap on overflow” (Simulink), and “Saturate on overflow” (Simulink).

Recommendations for Arithmetic and Scaling

In this section...

“Arithmetic Operations and Fixed-Point Scaling” on page 33-40

“Addition” on page 33-41

“Accumulation” on page 33-44

“Multiplication” on page 33-44

“Gain” on page 33-47

“Division” on page 33-49

“Summary” on page 33-51

Arithmetic Operations and Fixed-Point Scaling

The sections that follow describe the relationship between arithmetic operations and fixed-point scaling, and offer some basic recommendations that may be appropriate for your fixed-point design. For each arithmetic operation,

- The general [Slope Bias] encoding scheme described in “Scaling” on page 32-6 is used.
- The scaling of the result is automatically selected based on the scaling of the two inputs. In other words, the scaling is *inherited*.
- Scaling choices are based on
 - Minimizing the number of arithmetic operations of the result
 - Maximizing the precision of the result

Additionally, binary-point-only scaling is presented as a special case of the general encoding scheme.

In embedded systems, the scaling of variables at the hardware interface (the ADC or DAC) is fixed. However for most other variables, the scaling is something you can choose to give the best design. When scaling fixed-point variables, it is important to remember that

- Your scaling choices depend on the particular design you are simulating.
- There is no best scaling approach. All choices have associated advantages and disadvantages. It is the goal of this section to expose these advantages and disadvantages to you.

Addition

Consider the addition of two real-world values:

$$V_a = V_b + V_c.$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

In a fixed-point system, the addition of values results in finding the variable Q_a :

$$Q_a = \frac{F_b}{F_a} 2^{E_b-E_a} Q_b + \frac{F_c}{F_a} 2^{E_c-E_a} Q_c + \frac{B_b + B_c - B_a}{F_a} 2^{-E_a}.$$

This formula shows

- In general, Q_a is not computed through a simple addition of Q_b and Q_c .
- In general, there are two multiplications of a constant and a variable, two additions, and some additional bit shifting.

Inherited Scaling for Speed

In the process of finding the scaling of the sum, one reasonable goal is to simplify the calculations. Simplifying the calculations should reduce the number of operations, thereby increasing execution speed. The following choices can help to minimize the number of arithmetic operations:

- Set $B_a = B_b + B_c$. This eliminates one addition.
- Set $F_a = F_b$ or $F_a = F_c$. Either choice eliminates one of the two constant times variable multiplications.

The resulting formula is

$$Q_a = 2^{E_b-E_a} Q_b + \frac{F_c}{F_a} 2^{E_c-E_a} Q_c$$

or

$$Q_a = \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + 2^{E_c - E_a} Q_c.$$

These equations appear to be equivalent. However, your choice of rounding and precision may make one choice stand out over the other. To further simplify matters, you could choose $E_a = E_c$ or $E_a = E_b$. This will eliminate some bit shifting.

Inherited Scaling for Maximum Precision

In the process of finding the scaling of the sum, one reasonable goal is maximum precision. You can determine the maximum-precision scaling if the range of the variable is known. “Maximize Precision” on page 33-25 shows that you can determine the range of a fixed-point operation from $\max(V_a)$ and $\min(V_a)$. For a summation, you can determine the range from

$$\begin{aligned} \min(\tilde{V}_a) &= \min(\tilde{V}_b) + \min(\tilde{V}_c), \\ \max(\tilde{V}_a) &= \max(\tilde{V}_b) + \max(\tilde{V}_c). \end{aligned}$$

You can now derive the maximum-precision slope:

$$\begin{aligned} F_a 2^{E_a} &= \frac{\max(\tilde{V}_a) - \min(\tilde{V}_a)}{2^{ws_a} - 1} \\ &= \frac{F_b 2^{E_b} (2^{ws_b} - 1) + F_c 2^{E_c} (2^{ws_c} - 1)}{2^{ws_a} - 1}. \end{aligned}$$

In most cases the input and output word sizes are much greater than one, and the slope becomes

$$F_a 2^{E_a} \approx F_b 2^{E_b + ws_b - ws_a} + F_c 2^{E_c + ws_c - ws_a},$$

which depends only on the size of the input and output words. The corresponding bias is

$$B_a = \min(\tilde{V}_a) - F_a 2^{E_a} \min(Q_a).$$

The value of the bias depends on whether the inputs and output are signed or unsigned numbers.

If the inputs and output are all unsigned, then the minimum values for these variables are all zero and the bias reduces to a particularly simple form:

$$B_a = B_b + B_c.$$

If the inputs and the output are all signed, then the bias becomes

$$B_a \approx B_b + B_c + F_b 2^{E_b} \left(-2^{ws_b-1} + 2^{ws_b-1} \right) + F_c 2^{E_c} \left(-2^{ws_c-1} + 2^{ws_c-1} \right),$$

$$B_a \approx B_b + B_c.$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = 2^{E_b-E_a} Q_b + 2^{E_c-E_a} Q_c.$$

This scaling choice results in only one addition and some bit shifting. The avoidance of any multiplications is a big advantage of binary-point-only scaling.

Note: The subtraction of values produces results that are analogous to those produced by the addition of values.

Accumulation

The accumulation of values is closely associated with addition:

$$V_{a_new} = V_{a_old} + V_b.$$

Finding Q_{a_new} involves one multiplication of a constant and a variable, two additions, and some bit shifting:

$$Q_{a_new} = Q_{a_old} + \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b}{F_a} 2^{-E_a}.$$

The important difference for fixed-point implementations is that the scaling of the output is identical to the scaling of the first input.

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_{a_new} results in this simple expression:

$$Q_{a_new} = Q_{a_old} + 2^{E_b - E_a} Q_b.$$

This scaling option only involves one addition and some bit shifting.

Note: The negative accumulation of values produces results that are analogous to those produced by the accumulation of values.

Multiplication

Consider the multiplication of two real-world values:

$$V_a = V_b V_c.$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

In a fixed-point system, the multiplication of values results in finding the variable Q_a :

$$Q_a = \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} 2^{E_b - E_a} Q_b + \frac{F_c B_b}{F_a} 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} 2^{-E_a}.$$

This formula shows

- In general, Q_a is not computed through a simple multiplication of Q_b and Q_c .
- In general, there is one multiplication of a constant and two variables, two multiplications of a constant and a variable, three additions, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = B_b B_c$. This eliminates one addition operation.
- Set $F_a = F_b F_c$. This simplifies the triple multiplication—certainly the most difficult part of the equation to implement.
- Set $E_a = E_b + E_c$. This eliminates some of the bit shifting.

The resulting formula is

$$Q_a = Q_b Q_c + \frac{B_c}{F_c} 2^{-E_c} Q_b + \frac{B_b}{F_b} 2^{-E_b} Q_c.$$

Inherited Scaling for Maximum Precision

You can determine the maximum-precision scaling if the range of the variable is known. “Maximize Precision” on page 33-25 shows that you can determine the range of a fixed-point operation from

$$\max(\tilde{V}_a)$$

and

$$\min(\tilde{V}_a).$$

For multiplication, you can determine the range from

$$\begin{aligned}\min(\tilde{V}_a) &= \min(V_{LL}, V_{LH}, V_{HL}, V_{HH}), \\ \max(\tilde{V}_a) &= \max(V_{LL}, V_{LH}, V_{HL}, V_{HH}),\end{aligned}$$

where

$$\begin{aligned}V_{LL} &= \min(\tilde{V}_b) \cdot \min(\tilde{V}_c), \\ V_{LH} &= \min(\tilde{V}_b) \cdot \max(\tilde{V}_c), \\ V_{HL} &= \max(\tilde{V}_b) \cdot \min(\tilde{V}_c), \\ V_{HH} &= \max(\tilde{V}_b) \cdot \max(\tilde{V}_c).\end{aligned}$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c.$$

Gain

Consider the multiplication of a constant and a variable

$$V_a = K V_b,$$

where K is a constant called the gain. Since V_a results from the multiplication of a constant and a variable, finding Q_a is a simplified version of the general fixed-point multiplication formula:

$$Q_a = \left(\frac{K F_b 2^{E_b}}{F_a 2^{E_a}} \right) Q_b + \left(\frac{K B_b - B_a}{F_a 2^{E_a}} \right).$$

Note that the terms in the parentheses can be calculated offline. Therefore, there is only one multiplication of a constant and a variable and one addition.

To implement the above equation without changing it to a more complicated form, the constants need to be encoded using a binary-point-only format. For each of these constants, the range is the trivial case of only one value. Despite the trivial range, the binary point formulas for maximum precision are still valid. The maximum-precision representations are the most useful choices unless there is an overriding need to avoid any shifting. The encoding of the constants is

$$\left(\frac{KF_b 2^{E_b}}{F_a 2^{E_a}} \right) = 2^{E_x} Q_X$$

$$\left(\frac{KB_b - B_a}{F_a 2^{E_a}} \right) = 2^{E_y} Q_Y$$

resulting in the formula

$$Q_a = 2^{E_x} Q_X Q_B + 2^{E_y} Q_Y.$$

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = KB_b$. This eliminates one constant term.
- Set $F_a = KF_b$ and $E_a = E_b$. This sets the other constant term to unity.

The resulting formula is simply

$$Q_a = Q_b.$$

If the number of bits is different, then either handling potential overflows or performing sign extensions is the only possible operation involved.

Inherited Scaling for Maximum Precision

The scaling for maximum precision does not need to be different from the scaling for speed unless the output has fewer bits than the input. If this is the case, then saturation

should be avoided by dividing the slope by 2 for each lost bit. This prevents saturation but causes rounding to occur.

Division

Division of values is an operation that should be avoided in fixed-point embedded systems, but it can occur in places. Therefore, consider the division of two real-world values:

$$V_a = V_b / V_c.$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

In a fixed-point system, the division of values results in finding the variable Q_a :

$$Q_a = \frac{F_b 2^{E_b} Q_b + B_b}{F_c F_a 2^{E_c + E_a} Q_c + B_c F_a 2^{E_a}} - \frac{B_a}{F_a} 2^{-E_a}.$$

This formula shows

- In general, Q_a is not computed through a simple division of Q_b by Q_c .
- In general, there are two multiplications of a constant and a variable, two additions, one division of a variable by a variable, one division of a constant by a variable, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = 0$. This eliminates one addition operation.
- If $B_c = 0$, then set the fractional slope $F_a = F_b / F_c$. This eliminates one constant times variable multiplication.

The resulting formula is

$$Q_a = \frac{Q_b}{Q_c} 2^{E_b - E_c - E_a} + \frac{(B_b / F_b)}{Q_c} 2^{-E_c - E_a}.$$

If $B_c \neq 0$, then no clear recommendation can be made.

Inherited Scaling for Maximum Precision

You can determine the maximum-precision scaling if the range of the variable is known. “Maximize Precision” on page 33-25 shows that you can determine the range of a fixed-point operation from

$$\max(\tilde{V}_a)$$

and

$$\min(\tilde{V}_a).$$

For division, you can determine the range from

$$\begin{aligned} \min(\tilde{V}_a) &= \min(V_{LL}, V_{LH}, V_{HL}, V_{HH}), \\ \max(\tilde{V}_a) &= \max(V_{LL}, V_{LH}, V_{HL}, V_{HH}), \end{aligned}$$

where for nonzero denominators

$$\begin{aligned} V_{LL} &= \min(\tilde{V}_b) / \min(\tilde{V}_c), \\ V_{LH} &= \min(\tilde{V}_b) / \max(\tilde{V}_c), \\ V_{HL} &= \max(\tilde{V}_b) / \min(\tilde{V}_c), \\ V_{HH} &= \max(\tilde{V}_b) / \max(\tilde{V}_c). \end{aligned}$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = \frac{Q_b}{Q_c} 2^{E_b - E_c - E_a}.$$

Note: For the last two formulas involving Q_a , a divide by zero and zero divided by zero are possible. In these cases, the hardware will give some default behavior but you must make sure that these default responses give meaningful results for the embedded system.

Summary

From the previous analysis of fixed-point variables scaled within the general [Slope Bias] encoding scheme, you can conclude

- Addition, subtraction, multiplication, and division can be very involved unless certain choices are made for the biases and slopes.
- Binary-point-only scaling guarantees simpler math, but generally sacrifices some precision.

Note that the previous formulas don't show the following:

- Constants and variables are represented with a finite number of bits.
- Variables are either signed or unsigned.
- Rounding and overflow handling schemes. You must make these decisions before an actual fixed-point realization is achieved.

Parameter and Signal Conversions

In this section...

“Introduction” on page 33-52

“Parameter Conversions” on page 33-53

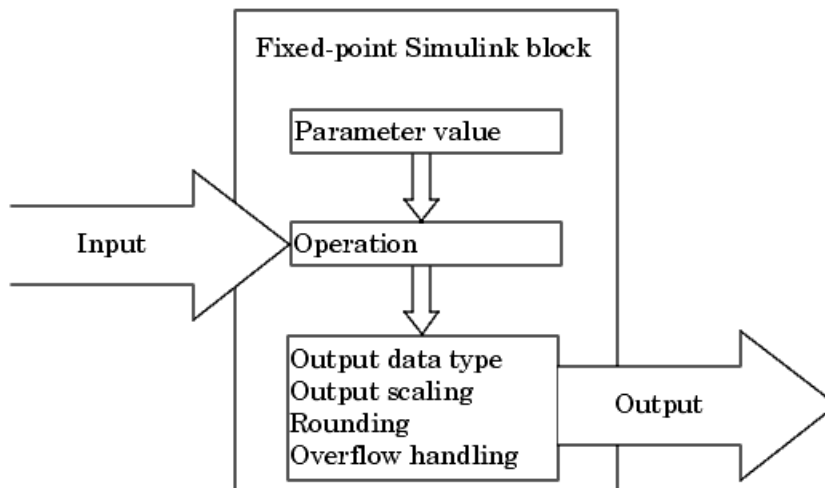
“Signal Conversions” on page 33-53

Introduction

To completely understand the results generated by fixed-point Simulink blocks, you must be aware of these issues:

- When numerical block parameters are converted from doubles to fixed-point data types
- When input signals are converted from one fixed-point data type to another (if at all)
- When arithmetic operations on input signals and parameters are performed

For example, suppose a fixed-point Simulink block performs an arithmetic operation on its input signal and a parameter, and then generates output having characteristics that are specified by the block. The following diagram illustrates how these issues are related.



The sections that follow describe parameter and signal conversions. “Rules for Arithmetic Operations” on page 33-56 discusses arithmetic operations.

Parameter Conversions

Parameters of fixed-point blocks that accept numerical values are always converted from `double` to a fixed-point data type. Parameters can be converted to the input data type, the output data type, or to a data type explicitly specified by the block. For example, the Discrete FIR Filter block converts its **Initial states** parameter to the input data type, and converts its **Numerator coefficient** parameter to a data type you explicitly specify via the block dialog box.

Parameters are always converted before any arithmetic operations are performed. Additionally, parameters are always converted *offline* using round-to-nearest and saturation. Offline conversions are discussed below.

Note: Because parameters of fixed-point blocks begin as `double`, they are never precise to more than 53 bits. Therefore, if the output of your fixed-point block is longer than 53 bits, your result might be less precise than you anticipated.

Offline Conversions

An offline conversion is a conversion performed by your development platform (for example, the processor on your PC), and not by the fixed-point processor you are targeting. For example, suppose you are using a PC to develop a program to run on a fixed-point processor, and you need the fixed-point processor to compute

$$y = \left(\frac{ab}{c} \right) u = C u$$

over and over again. If a , b , and c are constant parameters, it is inefficient for the fixed-point processor to compute ab/c every time. Instead, the PC's processor should compute ab/c offline one time, and the fixed-point processor computes only $C \cdot u$. This eliminates two costly fixed-point arithmetic operations.

Signal Conversions

Consider the conversion of a real-world value from one fixed-point data type to another. Ideally, the values before and after the conversion are equal.

$$V_a = V_b,$$

where V_b is the input value and V_a is the output value. To see how the conversion is implemented, the two ideal values are replaced by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

Solving for the output data type's stored integer value, Q_a is obtained:

$$\begin{aligned} Q_a &= \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b - B_a}{F_a} 2^{-E_a} \\ &= F_s 2^{E_b - E_a} Q_b + B_{net}, \end{aligned}$$

where F_s is the adjusted fractional slope and B_{net} is the net bias. The offline conversions and online conversions and operations are discussed below.

Offline Conversions

Both F_s and B_{net} are computed offline using round-to-nearest and saturation. B_{net} is then stored using the output data type and F_s is stored using an automatically selected data type.

Online Conversions and Operations

The remaining conversions and operations are performed *online* by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The conversions and operations are given by these steps:

- 1 The initial value for Q_a is given by the net bias, B_{net} :

$$Q_a = B_{net}.$$

- 2 The input integer value, Q_b , is multiplied by the adjusted slope, F_s :

$$Q_{RawProduct} = F_s Q_b.$$

- 3 The result of step 2 is converted to the modified output data type where the slope is one and bias is zero:

$$Q_{Temp} = \text{convert}(Q_{RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling.

- 4 The summation operation is performed:

$$Q_a = Q_{Temp} + Q_a.$$

This summation includes any necessary overflow handling.

Streamlining Simulations and Generated Code

Note that the maximum number of conversions and operations is performed when the slopes and biases of the input signal and output signal differ (are mismatched). If the scaling of these signals is identical (matched), the number of operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope and bias as the output, only step 3 is required:

$$Q_a = \text{convert}(Q_b).$$

Exclusive use of binary-point-only scaling for both input signals and output signals is a common way to eliminate mismatched slopes and biases, and results in the most efficient simulations and generated code.

Rules for Arithmetic Operations

Fixed-point arithmetic refers to how signed or unsigned binary words are operated on. The simplicity of fixed-point arithmetic functions such as addition and subtraction allows for cost-effective hardware implementations.

The sections that follow describe the rules that the Simulink software follows when arithmetic operations are performed on inputs and parameters. These rules are organized into four groups based on the operations involved: addition and subtraction, multiplication, division, and shifts. For each of these four groups, the rules for performing the specified operation are presented with an example using the rules.

Computational Units

The core architecture of many processors contains several computational units including arithmetic logic units (ALUs), multiply and accumulate units (MACs), and shifters. These computational units process the binary data directly and provide support for arithmetic computations of varying precision. The ALU performs a standard set of arithmetic and logic operations as well as division. The MAC performs multiply, multiply/add, and multiply/subtract operations. The shifter performs logical and arithmetic shifts, normalization, denormalization, and other operations.

Addition and Subtraction

Addition is the most common arithmetic operation a processor performs. When two n -bit numbers are added together, it is always possible to produce a result with $n + 1$ nonzero digits due to a carry from the leftmost digit. For two's complement addition of two numbers, there are three cases to consider:

- If both numbers are positive and the result of their addition has a sign bit of 1, then overflow has occurred; otherwise the result is correct.
- If both numbers are negative and the sign of the result is 0, then overflow has occurred; otherwise the result is correct.
- If the numbers are of unlike sign, overflow cannot occur and the result is always correct.

Fixed-Point Simulink Blocks Summation Process

Consider the summation of two numbers. Ideally, the real-world values obey the equation

$$V_a = \pm V_b \pm V_c,$$

where V_b and V_c are the input values and V_a is the output value. To see how the summation is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

The equation in “Addition” on page 33-41 gives the solution of the resulting equation for the stored integer, Q_a . Using shorthand notation, that equation becomes

$$Q_a = \pm F_{sb} 2^{E_b - E_a} Q_b \pm F_{sc} 2^{E_c - E_a} Q_c + B_{net},$$

where F_{sb} and F_{sc} are the adjusted fractional slopes and B_{net} is the net bias. The offline conversions and online conversions and operations are discussed below.

Offline Conversions

F_{sb} , F_{sc} , and B_{net} are computed offline using round-to-nearest and saturation. Furthermore, B_{net} is stored using the output data type.

Online Conversions and Operations

The remaining operations are performed online by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The worst (most inefficient) case occurs when the slopes and biases are mismatched. The worst-case conversions and operations are given by these steps:

- 1 The initial value for Q_a is given by the net bias, B_{net} :

$$Q_a = B_{net}.$$

- 2 The first input integer value, Q_b , is multiplied by the adjusted slope, F_{sb} :

$$Q_{RawProduct} = F_{sb} Q_b.$$

- 3 The previous product is converted to the modified output data type where the slope is one and the bias is zero:

$$Q_{Temp} = \text{convert}(Q_{RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling.

- 4 The summation operation is performed:

$$Q_a = Q_a \pm Q_{Temp}.$$

This summation includes any necessary overflow handling.

- 5 Steps 2 to 4 are repeated for every number to be summed.

It is important to note that bit shifting, rounding, and overflow handling are applied to the intermediate steps (3 and 4) and not to the overall sum.

For more information, see “The Summation Process” on page 33-66.

Streamlining Simulations and Generated Code

If the scaling of the input and output signals is matched, the number of summation operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope as the output, step 2 reduces to multiplication by one and can be eliminated. Trivial steps in the summation process are eliminated for both simulation and code generation. Exclusive use of binary-point-only scaling for both input signals and output signals is a common way to eliminate mismatched slopes and biases, and results in the most efficient simulations and generated code.

Multiplication

The multiplication of an n-bit binary number with an m-bit binary number results in a product that is up to m + n bits in length for both signed and unsigned words. Most processors perform n-bit by n-bit multiplication and produce a 2n-bit result (double bits) assuming there is no overflow condition.

Fixed-Point Simulink Blocks Multiplication Process

Consider the multiplication of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b V_c.$$

where V_b and V_c are the input values and V_a is the output value. To see how the multiplication is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

The solution of the resulting equation for the output stored integer, Q_a , is given below:

$$\begin{aligned} Q_a = & \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} 2^{E_b - E_a} Q_b \\ & + \frac{F_c B_b}{F_a} 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} 2^{-E_a}. \end{aligned}$$

Multiplication with Nonzero Biases and Mismatched Fractional Slopes

The worst-case implementation of the above equation occurs when the slopes and biases of the input and output signals are mismatched. In such cases, several low-level integer operations are required to carry out the high-level multiplication (or division). Implementation choices made about these low-level computations can affect the computational efficiency, rounding errors, and overflow.

In Simulink blocks, the actual multiplication or division operation is always performed on fixed-point variables that have zero biases. If an input has nonzero bias, it is converted to a representation that has binary-point-only scaling before the operation. If the result is to have nonzero bias, the operation is first performed with temporary variables that have binary-point-only scaling. The result is then converted to the data type and scaling of the final output.

If both the inputs and the output have nonzero biases, then the operation is broken down as follows:

$$\begin{aligned} V_{1Temp} &= V_1, \\ V_{2Temp} &= V_2, \\ V_{3Temp} &= V_{1Temp} V_{2Temp}, \\ V_3 &= V_{3Temp}, \end{aligned}$$

where

$$V_{1Temp} = 2^{E_{1Temp}} Q_{1Temp},$$

$$V_{2Temp} = 2^{E_{2Temp}} Q_{2Temp},$$

$$V_{3Temp} = 2^{E_{3Temp}} Q_{3Temp}.$$

These equations show that the temporary variables have binary-point-only scaling. However, the equations do not indicate the signedness, word lengths, or values of the fixed exponent of these variables. The Simulink software assigns these properties to the temporary variables based on the following goals:

- Represent the original value without overflow.

The data type and scaling of the original value define a maximum and minimum real-world value:

$$V_{Max} = F 2^E Q_{MaxInteger} + B,$$

$$V_{Min} = F 2^E Q_{MinInteger} + B.$$

The data type and scaling of the temporary value must be able to represent this range without overflow. Precision loss is possible, but overflow is never allowed.

- Use a data type that leads to efficient operations.

This goal is relative to the target that you will use for production deployment of your design. For example, suppose that you will implement the design on a 16-bit fixed-point processor that provides a 32-bit `long`, 16-bit `int`, and 8-bit `short` or `char`. For such a target, preserving efficiency means that no more than 32 bits are used, and the smaller sizes of 8 or 16 bits are used if they are sufficient to maintain precision.

- Maintain precision.

Ideally, every possible value defined by the original data type and scaling is represented perfectly by the temporary variable. However, this can require more bits than is efficient. Bits are discarded, resulting in a loss of precision, to the extent required to preserve efficiency.

For example, consider the following, assuming a 16-bit microprocessor target:

$$V_{Original} = Q_{Original} + -43.25,$$

where $Q_{Original}$ is an 8-bit, unsigned data type. For this data type,

$$Q_{MaxInteger} = 225,$$

$$Q_{MinInteger} = 0,$$

so

$$V_{Max} = 211.75,$$

$$V_{Min} = -43.25.$$

The minimum possible value is negative, so the temporary variable must be a signed integer data type. The original variable has a slope of 1, but the bias is expressed with greater precision with two digits after the binary point. To get full precision, the fixed exponent of the temporary variable has to be -2 or less. The Simulink software selects the least possible precision, which is generally the most efficient, unless overflow issues arise. For a scaling of 2^{-2} , selecting signed 16-bit or signed 32-bit avoids overflow. For efficiency, the Simulink software selects the smaller choice of 16 bits. If the original variable is an input, then the equations to convert to the temporary variable are

$$\text{uint8_T} = Q_{Original},$$

$$\text{uint16_T} = Q_{Temp},$$

$$Q_{Temp} = ((\text{uint16_T})Q_{Original} \ll 2) - 173.$$

Multiplication with Zero Biases and Mismatched Fractional Slopes

When the biases are zero and the fractional slopes are mismatched, the implementation reduces to

$$Q_a = \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c.$$

Offline Conversions

The quantity

$$F_{Net} = \frac{F_b F_c}{F_a}$$

is calculated offline using round-to-nearest and saturation. F_{Net} is stored using a fixed-point data type of the form

$$2^{E_{Net}} Q_{Net},$$

where E_{Net} and Q_{Net} are selected automatically to best represent F_{Net} .

Online Conversions and Operations

- 1 The integer values Q_b and Q_c are multiplied:

$$Q_{RawProduct} = Q_b Q_c.$$

To maintain the full precision of the product, the binary point of $Q_{RawProduct}$ is given by the sum of the binary points of Q_b and Q_c .

- 2 The previous product is converted to the output data type:

$$Q_{Temp} = \text{convert}(Q_{RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. “Signal Conversions” on page 33-53 discusses conversions.

- 3 The multiplication

$$Q_{2RawProduct} = Q_{Temp} Q_{Net}$$

is performed.

- 4 The previous product is converted to the output data type:

$$Q_a = \text{convert}(Q_{2RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. “Signal Conversions” on page 33-53 discusses conversions.

5 Steps 1 through 4 are repeated for each additional number to be multiplied.

Multiplication with Zero Biases and Matching Fractional Slopes

When the biases are zero and the fractional slopes match, the implementation reduces to

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c.$$

Offline Conversions

No offline conversions are performed.

Online Conversions and Operations

1 The integer values Q_b and Q_c are multiplied:

$$Q_{RawProduct} = Q_b Q_c.$$

To maintain the full precision of the product, the binary point of $Q_{RawProduct}$ is given by the sum of the binary points of Q_b and Q_c .

2 The previous product is converted to the output data type:

$$Q_a = \text{convert}(Q_{RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. “Signal Conversions” on page 33-53 discusses conversions.

3 Steps 1 and 2 are repeated for each additional number to be multiplied.

For more information, see “The Multiplication Process” on page 33-69.

Division

This section discusses the division of quantities with zero bias.

Note: When any input to a division calculation has nonzero bias, the operations performed exactly match those for multiplication described in “Multiplication with Nonzero Biases and Mismatched Fractional Slopes” on page 33-59.

Fixed-Point Simulink Blocks Division Process

Consider the division of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b / V_c,$$

where V_b and V_c are the input values and V_a is the output value. To see how the division is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

For the case where the slope adjustment factors are one and the biases are zero for all signals, the solution of the resulting equation for the output stored integer, Q_a , is given by the following equation:

$$Q_a = 2^{E_b - E_c - E_a} (Q_b / Q_c).$$

This equation involves an integer division and some bit shifts. If $E_a > E_b - E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b - E_c$, then the bit shifts are to the left and the implementation can be more complicated. The essential issue is that the output has more precision than the integer division provides. To get full precision, a *fractional* division is needed. The C programming language provides access to integer division only for fixed-point data types. Depending on the size of the numerator, you can obtain some of the fractional bits by performing a shift prior to the integer division. In the worst case, it might be necessary to resort to repeated subtractions in software.

In general, division of values is an operation that should be avoided in fixed-point embedded systems. Division where the output has more precision than the integer division (i.e., $E_a < E_b - E_c$) should be used with even greater reluctance.

For more information, see “The Division Process” on page 33-71.

Shifts

Nearly all microprocessors and digital signal processors support well-defined *bit-shift* (or simply *shift*) operations for integers. For example, consider the 8-bit unsigned integer 00110101. The results of a 2-bit shift to the left and a 2-bit shift to the right are shown in the following table.

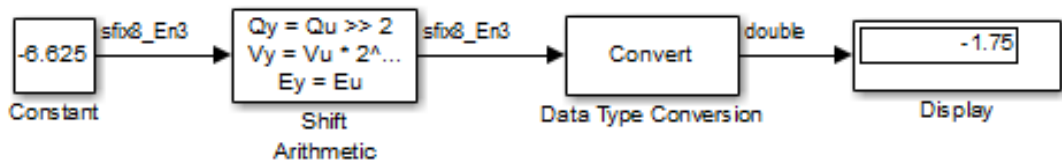
Shift Operation	Binary Value	Decimal Value
No shift (original number)	00110101	53
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

You can perform a shift using the Simulink Shift Arithmetic block. Use this block to perform a bit shift, a binary point shift, or both

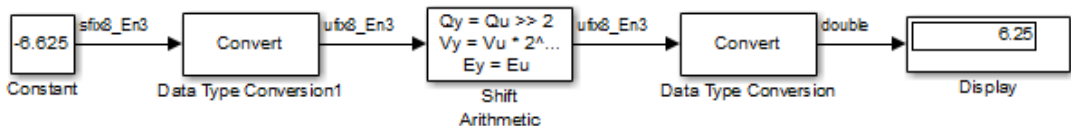
Shifting Bits to the Right

The special case of shifting bits to the right requires consideration of the treatment of the leftmost bit, which can contain sign information. A shift to the right can be classified either as a *logical* shift right or an *arithmetic* shift right. For a logical shift right, a 0 is incorporated into the most significant bit for each bit shift. For an arithmetic shift right, the most significant bit is recycled for each bit shift.

The Shift Arithmetic block performs an arithmetic shift right and, therefore, recycles the most significant bit for each bit shift right. For example, given the fixed-point number 11001.011 (-6.625), a bit shift two places to the right with the binary point unmoved yields the number 11110.010 (-1.75), as shown in the model below:



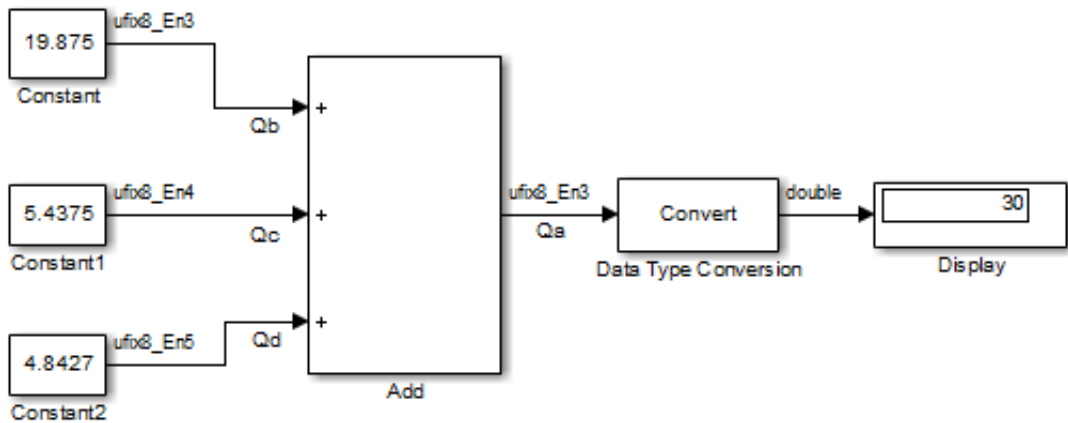
To perform a logical shift right on a signed number using the Shift Arithmetic block, use the Data Type Conversion block to cast the number as an unsigned number of equivalent length and scaling, as shown in the following model. The model shows that the fixed-point signed number 11001.001 (-6.625) becomes 00110.010 (6.25).



The Summation Process

Suppose you want to sum three numbers. Each of these numbers is represented by an 8-bit word, and each has a different binary-point-only scaling. Additionally, the output is restricted to an 8-bit word with binary-point-only scaling of 2^{-3} .

The summation is shown in the following model for the input values 19.875, 5.4375, and 4.84375.



Applying the rules from the previous section, the sum follows these steps:

- 1 Because the biases are matched, the initial value of Q_a is trivial:

$$Q_a = 00000.000.$$

- 2 The first number to be summed (19.875) has a fractional slope that matches the output fractional slope. Furthermore, the binary points and storage types are identical, so the conversion is trivial:

$$Q_b = 10011.111,$$

$$Q_{Temp} = Q_b.$$

- 3 The summation operation is performed:

$$Q_a = Q_a + Q_{Temp} = 10011.111.$$

- 4 The second number to be summed (5.4375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match, but the difference in binary points requires that both the bits and the binary point be shifted one place to the right:

$$Q_c = 0101.0111,$$

$$Q_{Temp} = \text{convert}(Q_c)$$

$$Q_{Temp} = 00101.011.$$

Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow cannot occur in this case because the bits and binary point are both shifted to the right.

- 5 The summation operation is performed:

$$\begin{aligned} Q_a &= Q_a + Q_{Temp} \\ &\quad 10011.111 \\ &= \frac{+00101.011}{11001.010} = 25.250. \end{aligned}$$

Note that overflow did not occur, but it is possible for this operation.

- 6 The third number to be summed (4.84375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match, but the difference in binary points requires that both the bits and the binary point be shifted two places to the right:

$$Q_d = 100.11011,$$

$$Q_{Temp} = \text{convert}(Q_d)$$

$$Q_{Temp} = 00100.110.$$

Note that a loss in precision of two bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow

cannot occur in this case because the bits and binary point are both shifted to the right.

- 7** The summation operation is performed:

$$\begin{aligned}
 Q_a &= Q_a + Q_{Temp} \\
 &\quad 11001.010 \\
 &= \frac{+00100.110}{11110.000} = 30.000.
 \end{aligned}$$

Note that overflow did not occur, but it is possible for this operation.

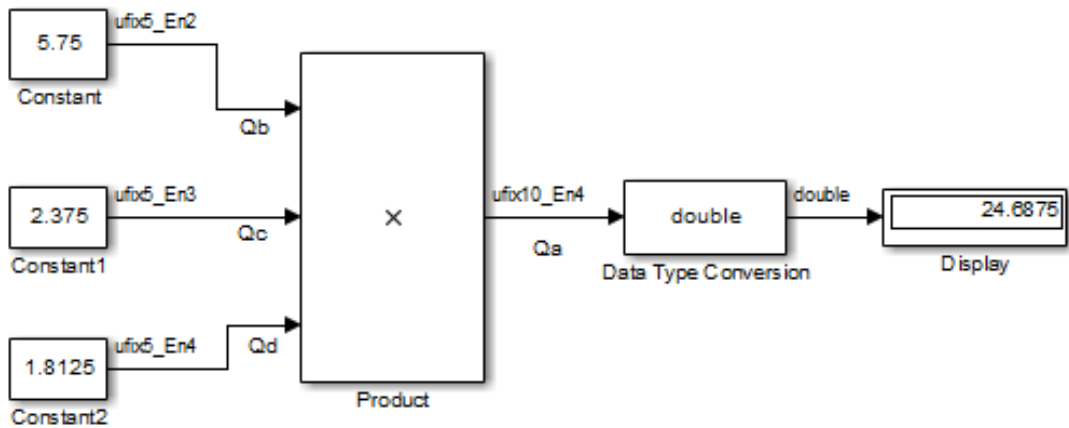
As shown here, the result of step 7 differs from the ideal sum:

$$\begin{aligned}
 &10011.111 \\
 &\quad 0101.0111 \\
 &= \frac{+100.11011}{11110.001} = 30.125.
 \end{aligned}$$

Blocks that perform addition and subtraction include the Sum, Gain, and Discrete FIR Filter blocks.

The Multiplication Process

Suppose you want to multiply three numbers. Each of these numbers is represented by a 5-bit word, and each has a different binary-point-only scaling. Additionally, the output is restricted to a 10-bit word with binary-point-only scaling of 2^{-4} . The multiplication is shown in the following model for the input values 5.75, 2.375, and 1.8125.



Applying the rules from the previous section, the multiplication follows these steps:

- 1 The first two numbers (5.75 and 2.375) are multiplied:

$$\begin{array}{r}
 Q_{RawProduct} = \quad 101.11 \\
 \quad \times 10.011 \\
 \hline
 101.11 \cdot 2^{-3} \\
 101.11 \cdot 2^{-2} \\
 + 101.11 \cdot 2^1 \\
 \hline
 01101.10101 = 13.65625.
 \end{array}$$

Note that the binary point of the product is given by the sum of the binary points of the multiplied numbers.

- 2 The result of step 1 is converted to the output data type:

$$\begin{aligned}
 Q_{Temp} &= \text{convert}(Q_{RawProduct}) \\
 &= 001101.1010 = 13.6250.
 \end{aligned}$$

“Signal Conversions” on page 33-53 discusses conversions. Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

- 3** The result of step 2 and the third number (1.8125) are multiplied:

$$\begin{array}{r}
 Q_{RawProduct} = \quad 01101.1010 \\
 \qquad \qquad \qquad \times 1.1101 \\
 \hline
 \qquad \qquad \qquad 1101.1010 \cdot 2^{-4} \\
 \qquad \qquad \qquad 1101.1010 \cdot 2^{-2} \\
 \qquad \qquad \qquad 1101.1010 \cdot 2^{-1} \\
 \qquad \qquad \qquad + 1101.1010 \cdot 2^0 \\
 \hline
 \qquad \qquad \qquad 0011000.10110010 = 24.6953125.
 \end{array}$$

Note that the binary point of the product is given by the sum of the binary points of the multiplied numbers.

- 4** The product is converted to the output data type:

$$\begin{aligned}
 Q_a &= \text{convert}(Q_{RawProduct}) \\
 &= 011000.1011 = 24.6875.
 \end{aligned}$$

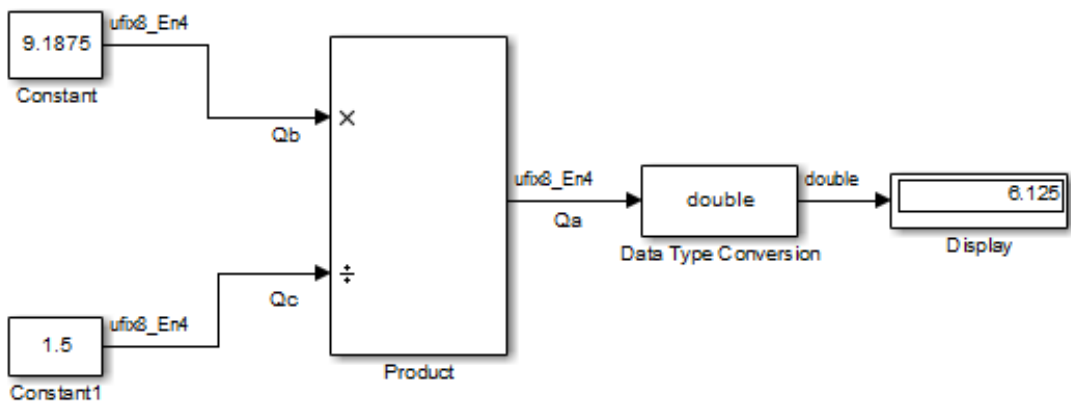
“Signal Conversions” on page 33-53 discusses conversions. Note that a loss in precision of 4 bits occurred, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

Blocks that perform multiplication include the Product, Discrete FIR Filter, and Gain blocks.

The Division Process

Suppose you want to divide two numbers. Each of these numbers is represented by an 8-bit word, and each has a binary-point-only scaling of 2^{-4} . Additionally, the output is restricted to an 8-bit word with binary-point-only scaling of 2^{-4} .

The division of 9.1875 by 1.5000 is shown in the following model.



For this example,

$$\begin{aligned} Q_a &= 2^{-4-(-4)-(-4)}(Q_b/Q_c) \\ &= 2^4(Q_b/Q_c). \end{aligned}$$

Assuming a large data type was available, this could be implemented as

$$Q_a = \frac{(2^4 Q_b)}{Q_c},$$

where the numerator uses the larger data type. If a larger data type was not available, integer division combined with four repeated subtractions would be used. Both approaches produce the same result, with the former being more efficient.

Shifts

Nearly all microprocessors and digital signal processors support well-defined *bit-shift* (or simply *shift*) operations for integers. For example, consider the 8-bit unsigned integer 00110101. The results of a 2-bit shift to the left and a 2-bit shift to the right are shown in the following table.

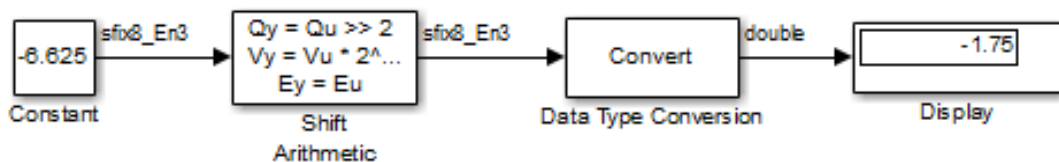
Shift Operation	Binary Value	Decimal Value
No shift (original number)	00110101	53
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

You can perform a shift using the Simulink Shift Arithmetic block. Use this block to perform a bit shift, a binary point shift, or both

Shifting Bits to the Right

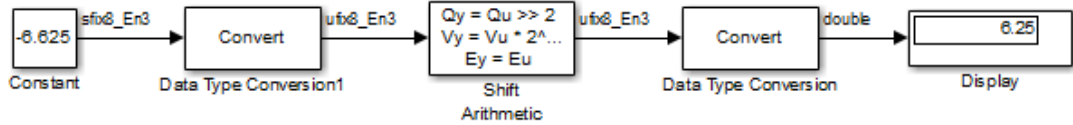
The special case of shifting bits to the right requires consideration of the treatment of the leftmost bit, which can contain sign information. A shift to the right can be classified either as a *logical* shift right or an *arithmetic* shift right. For a logical shift right, a 0 is incorporated into the most significant bit for each bit shift. For an arithmetic shift right, the most significant bit is recycled for each bit shift.

The Shift Arithmetic block performs an arithmetic shift right and, therefore, recycles the most significant bit for each bit shift right. For example, given the fixed-point number 11001.011 (-6.625), a bit shift two places to the right with the binary point unmoved yields the number 11110.010 (-1.75), as shown in the model below:



To perform a logical shift right on a signed number using the Shift Arithmetic block, use the Data Type Conversion block to cast the number as an unsigned number of equivalent

length and scaling, as shown in the following model. The model shows that the fixed-point signed number 11001.001 (-6.625) becomes 00110.010 (6.25).



Conversions and Arithmetic Operations

This example uses the Discrete FIR Filter block to illustrate when parameters are converted from a double to a fixed-point number, when the input data type is converted to the output data type, and when the rules for addition, subtraction, and multiplication are applied. For details about conversions and operations, refer to “Parameter and Signal Conversions” on page 33-52 and “Rules for Arithmetic Operations” on page 33-56.

Note: If a block can perform all four arithmetic operations, then the rules for multiplication and division are applied first. The Discrete FIR Filter block is an example of this.

Suppose you configure the Discrete FIR Filter block for two outputs, where the first output is given by

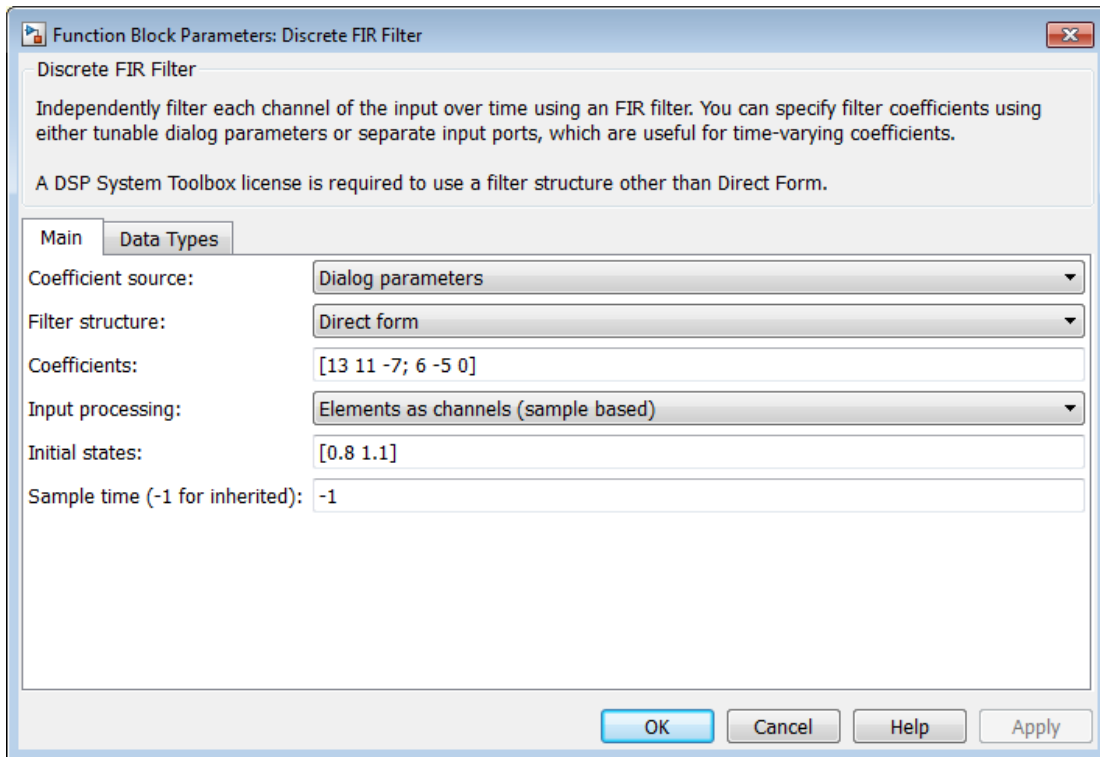
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2),$$

and the second output is given by

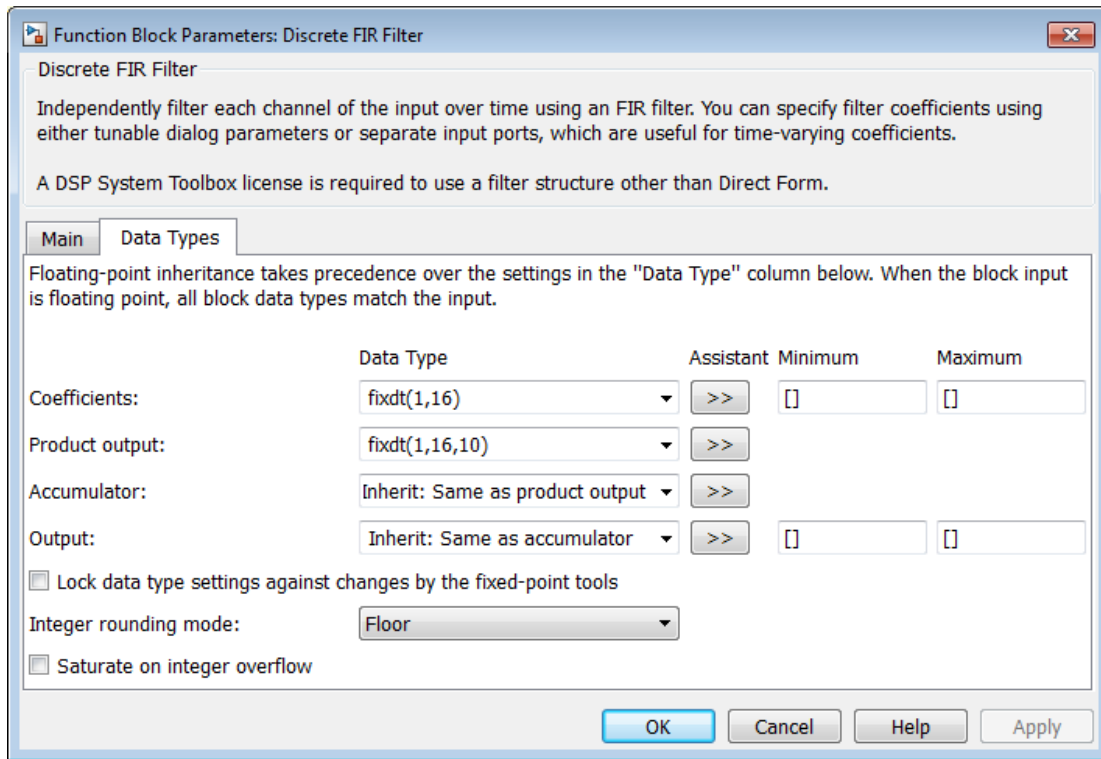
$$y_2(k) = 6 \cdot u(k) - 5 \cdot u(k-1).$$

Additionally, the initial values of $u(k-1)$ and $u(k-2)$ are given by 0.8 and 1.1, respectively, and all inputs, parameters, and outputs have binary-point-only scaling.

To configure the Discrete FIR Filter block for this situation, on the **Main** pane of its dialog box, you must specify the **Coefficients** parameter as [13 11 -7; 6 -5 0] and the **Initial states** parameter as [0.8 1.1], as shown here.



Similarly, configure the options on the **Data Types** pane of the block dialog box to appear as follows:



The Discrete FIR Filter block performs parameter conversions and block operations in the following order:

- 1 The **Coefficients** parameter is converted offline from doubles to the **Coefficients** data type using round-to-nearest and saturation.

The **Initial states** parameter is converted offline from doubles to the input data type using round-to-nearest and saturation.

- 2 The coefficients and inputs are multiplied together for the initial time step for both outputs. For $y_1(0)$, the operations $13 \cdot u(0)$, $11 \cdot 0.8$, and $-7 \cdot 1.1$ are performed, while for $y_2(0)$, the operations $6 \cdot u(0)$ and $-5 \cdot 0.8$ are performed.

The results of these operations are stored as **Product output**.

- 3 The sum is carried out in **Accumulator**. The final summation result is then converted to **Output**.

4 Steps 2 and 3 repeat for subsequent time steps.

Realization Structures

- “Realizing Fixed-Point Digital Filters” on page 34-2
- “Targeting an Embedded Processor” on page 34-4
- “Canonical Forms” on page 34-7
- “Direct Form II” on page 34-9
- “Series Cascade Form” on page 34-13
- “Parallel Form” on page 34-16

Realizing Fixed-Point Digital Filters

In this section...
“Introduction” on page 34-2
“Realizations and Data Types” on page 34-2

Introduction

This chapter investigates how you can realize fixed-point digital filters using Simulink blocks and the Fixed-Point Designer software.

The Fixed-Point Designer software addresses the needs of the control system, signal processing, and other fields where algorithms are implemented on fixed-point hardware. In signal processing, a digital filter is a computational algorithm that converts a sequence of input numbers to a sequence of output numbers. The algorithm is designed such that the output signal meets frequency-domain or time-domain constraints (desirable frequency components are passed, undesirable components are rejected).

In general terms, a discrete transfer function controller is a form of a digital filter. However, a digital controller can contain nonlinear functions such as lookup tables in addition to a discrete transfer function. This guide uses the term *digital filter* when referring to discrete transfer functions.

Note: To design and implement a wide variety of floating-point and fixed-point filters suitable for use in signal processing applications and for deployment on DSP chips, use the DSP System Toolbox software.

Realizations and Data Types

In an ideal world, where numbers, calculations, and storage of states have infinite precision and range, there are virtually an infinite number of realizations for the same system. In theory, these realizations are all identical.

In the more realistic world of double-precision numbers, calculations, and storage of states, small nonlinearities are introduced by the finite precision and range of floating-point data types. Therefore, each realization of a given system produces different results. In most cases however, these differences are small.

In the world of fixed-point numbers, where precision and range are limited, the differences in the realization results can be very large. Therefore, you must carefully select the data type, word size, and scaling for each realization element such that results are accurately represented. To assist you with this selection, design rules for modeling dynamic systems with fixed-point math are provided in “Targeting an Embedded Processor” on page 34-4.

Targeting an Embedded Processor

In this section...

“Introduction” on page 34-4

“Size Assumptions” on page 34-4

“Operation Assumptions” on page 34-4

“Design Rules” on page 34-5

Introduction

The sections that follow describe issues that often arise when targeting a fixed-point design for use on an embedded processor, such as some general assumptions about integer sizes and operations available on embedded processors. These assumptions lead to design issues and design rules that might be useful for your specific fixed-point design.

Size Assumptions

Embedded processors are typically characterized by a particular bit size. For example, the terms “8-bit micro,” “32-bit micro,” or “16-bit DSP” are common. It is generally safe to assume that the processor is predominantly geared to processing integers of the specified bit size. Integers of the specified bit size are referred to as the *base data type*. Additionally, the processor typically provides some support for integers that are twice as wide as the base data type. Integers consisting of double bits are referred to as the *accumulator data type*. For example a 16-bit micro has a 16-bit base data type and a 32-bit accumulator data type.

Although other data types may be supported by the embedded processor, this section describes only the base and accumulator data types.

Operation Assumptions

The embedded processor operations discussed in this section are limited to the needs of a basic simulation diagram. Basic simulations use multiplication, addition, subtraction, and delays. Fixed-point models also need shifts to do scaling conversions. For all these operations, the embedded processor should have native instructions that allow the

base data type as inputs. For accumulator-type inputs, the processor typically supports addition, subtraction, and delay (storage/retrieval from memory), but not multiplication.

Multiplication is typically not supported for accumulator-type inputs because of complexity and size issues. A difficulty with multiplication is that the output needs to be twice as big as the inputs for full precision. For example, multiplying two 16-bit numbers requires a 32-bit output for full precision. The need to handle the outputs from a multiplication operation is one of the reasons embedded processors include accumulator-type support. However, if multiplication of accumulator-type inputs is also supported, then there is a need to support a data type that is twice as big as the accumulator type. To restrict this additional complexity, multiplication is typically not supported for inputs of the accumulator type.

Design Rules

The important design rules that you should be aware of when modeling dynamic systems with fixed-point math follow.

Design Rule 1: Only Multiply Base Data Types

It is best to multiply only inputs of the base data type. Embedded processors typically provide an instruction for the multiplication of base-type inputs, but not for the multiplication of accumulator-type inputs. If necessary, you can combine several instructions to handle multiplication of accumulator-type inputs. However, this can lead to large, slow embedded code.

You can insert blocks to convert inputs from the accumulator type to the base type prior to Product or Gain blocks, if necessary.

Design Rule 2: Delays Should Use the Base Data Type

There are two general reasons why a Unit Delay should use only base-type numbers:

- The Unit Delay essentially stores a variable's value to RAM and, one time step later, retrieves that value from RAM. Because the value must be in memory from one time step to the next, the RAM must be exclusively dedicated to the variable and can't be shared or used for another purpose. Using accumulator-type numbers instead of the base data type doubles the RAM requirements, which can significantly increase the cost of the embedded system.
- The Unit Delay typically feeds into a Gain block. The multiplication design rule requires that the input (the unit delay signal) use the base data type.

Design Rule 3: Temporary Variables Can Use the Accumulator Data Type

Except for unit delay signals, most signals are not needed from one time step to the next. This means that the signal values can be temporarily stored in shared and reused memory. This shared and reused memory can be RAM or it can simply be registers in the CPU. In either case, storing the value as an accumulator data type is not much more costly than storing it as a base data type.

Design Rule 4: Summation Can Use the Accumulator Data Type

Addition and subtraction can use the accumulator data type if there is justification. The typical justification is reducing the buildup of errors due to roundoff or overflow.

For example, a common filter operation is a weighted sum of several variables. Multiplying a variable by a weight naturally produces a product of the accumulator type. Before summing, each product can be converted back to the base data type. This approach introduces round-off error into each part of the sum.

Alternatively, the products can be summed using the accumulator data type, and the final sum can be converted to the base data type. Round-off error is introduced in just one point and the precision is generally better. The cost of doing an addition or subtraction using accumulator-type numbers is slightly more expensive, but if there is justification, it is usually worth the cost.

Canonical Forms

The Fixed-Point Designer software does not attempt to standardize on one particular fixed-point digital filter design method. For example, you can produce a design in continuous time and then obtain an “equivalent” discrete-time digital filter using one of many transformation methods. Alternatively, you can design digital filters directly in discrete time. After you obtain a digital filter, it can be realized for fixed-point hardware using any number of canonical forms. Typical canonical forms are the direct form, series form, and parallel form, each of which is outlined in the sections that follow.

For a given digital filter, the canonical forms describe a set of fundamental operations for the processor. Because there are an infinite number of ways to realize a given digital filter, you must make the best realization on a per-system basis. The canonical forms presented in this chapter optimize the implementation with respect to some factor, such as minimum number of delay elements.

In general, when choosing a realization method, you must take these factors into consideration:

- **Cost**

The cost of the realization might rely on minimal code and data size.

- **Timing constraints**

Real-time systems must complete their compute cycle within a fixed amount of time. Some realizations might yield faster execution speed on different processors.

- **Output signal quality**

The limited range and precision of the binary words used to represent real-world numbers will introduce errors. Some realizations are more sensitive to these errors than others.

The Fixed-Point Designer software allows you to evaluate various digital filter realization methods in a simulation environment. Following the development cycle outlined in “Developing and Testing Fixed-Point Systems” on page 31-15, you can fine-tune the realizations with the goal of reducing the cost (code and data size) or increasing signal quality. After you have achieved the desired performance, you can use the Simulink Coder product to generate rapid prototyping C code and evaluate its performance with respect to your system's real-time timing constraints. You can then modify the model based upon feedback from the rapid prototyping system.

The presentation of the various realization structures takes into account that a summing junction is a fundamental operator, thus you may find that the structures presented here look different from those in the fixed-point filter design literature. For each realization form, an example is provided using the transfer function shown here:

$$\begin{aligned} H_{ex}(z) &= \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}} \\ &= \frac{(1 + 0.5z^{-1})(1 + 1.7z^{-1} + z^{-2})}{(1 + 0.1z^{-1})(1 - 0.6z^{-1} + 0.9z^{-2})} \\ &= 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}. \end{aligned}$$

Direct Form II

In general, a direct form realization refers to a structure where the coefficients of the transfer function appear directly as Gain blocks. The direct form II realization method is presented as using the minimal number of delay elements, which is equal to n , the order of the transfer function denominator.

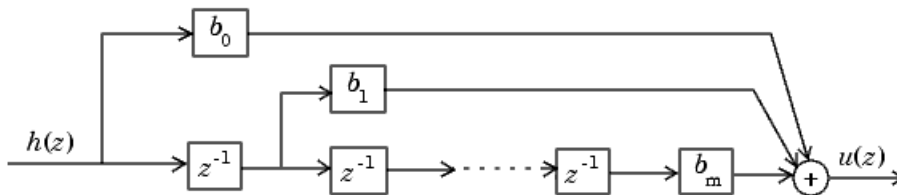
The canonical direct form II is presented as “Standard Programming” in *Discrete-Time Control Systems* by Ogata. It is known as the “Control Canonical Form” in *Digital Control of Dynamic Systems* by Franklin, Powell, and Workman.

You can derive the canonical direct form II realization by writing the discrete-time transfer function with input $e(z)$ and output $u(z)$ as

$$\frac{u(z)}{e(z)} = \frac{u(z)}{h(z)} \cdot \frac{h(z)}{e(z)}$$

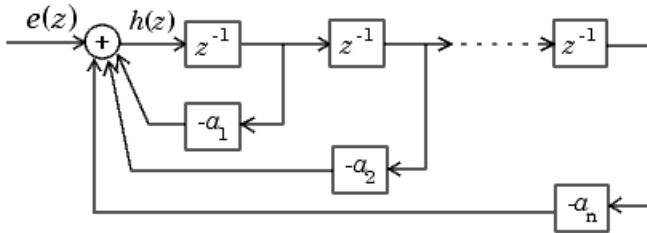
$$= \underbrace{(b_0 + b_1 z^{-1} + \dots + b_m z^{-m})}_{\frac{u(z)}{h(z)}} \cdot \underbrace{\frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_n z^{-n}}}_{\frac{h(z)}{e(z)}}$$

The block diagram for $u(z)/h(z)$ follows.



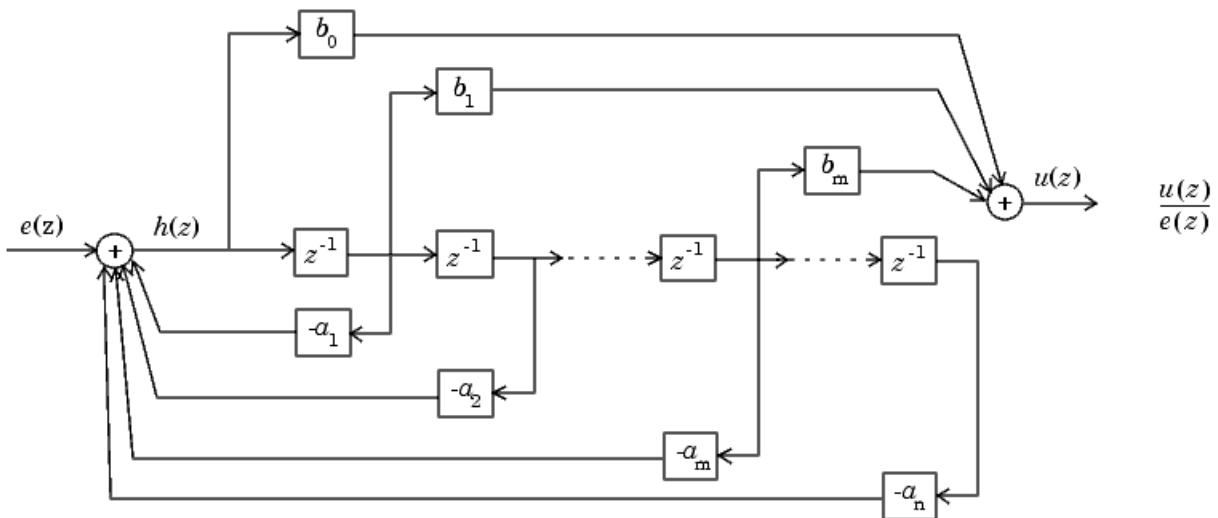
$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \dots + b_m z^{-m}$$

The block diagrams for $h(z)/e(z)$ follow.



$$\frac{h(z)}{e(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_n z^{-n}}$$

Combining these two block diagrams yields the direct form II diagram shown in the following figure. Notice that the feedforward part (top of block diagram) contains the numerator coefficients and the feedback part (bottom of block diagram) contains the denominator coefficients.



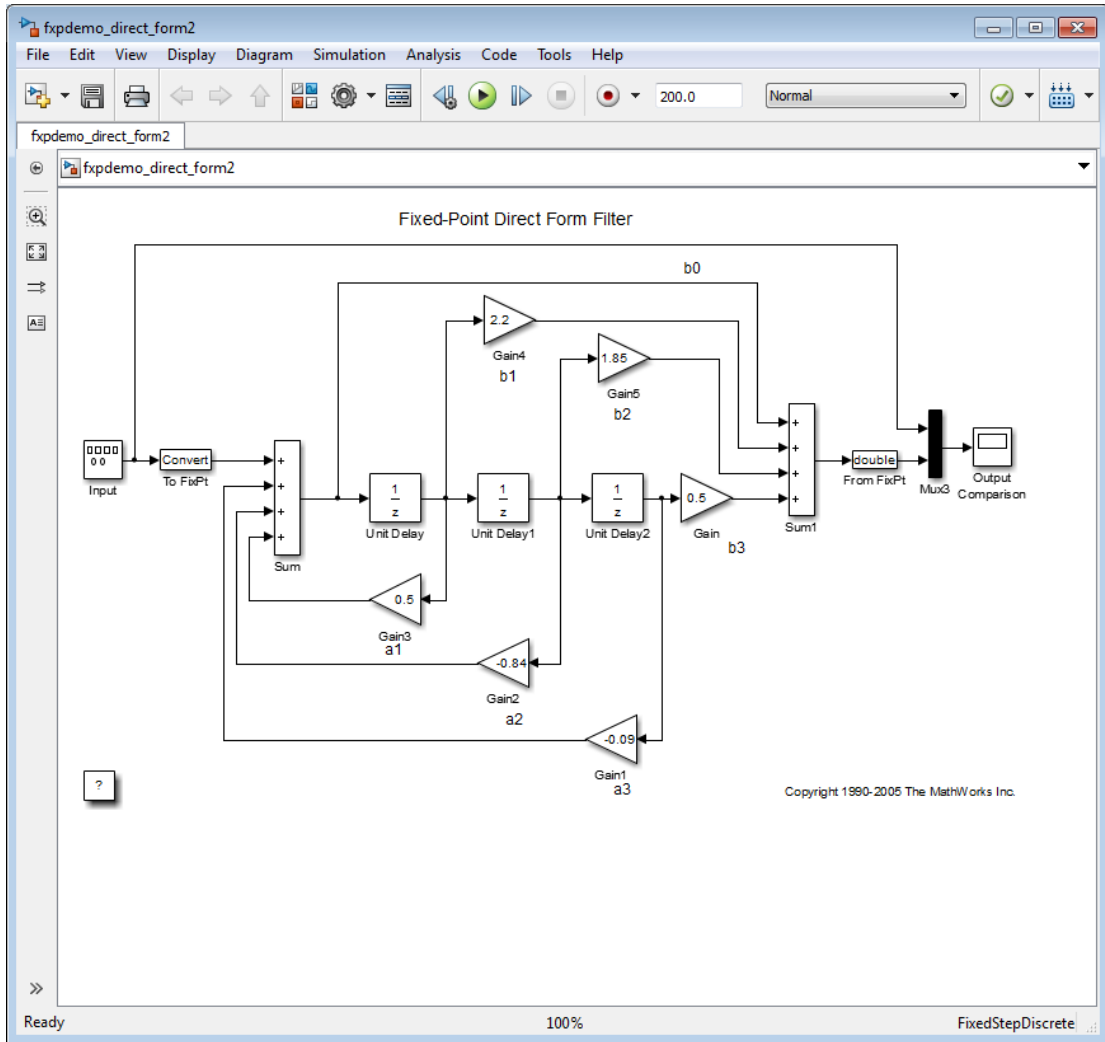
The direct form II example transfer function is given by

$$H_{ex}(z) = \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}}.$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

```
fxpdemo_direct_form2
```

at the MATLAB command line.

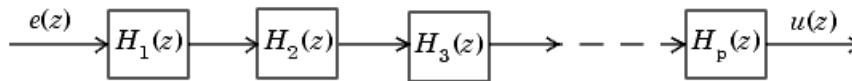


Series Cascade Form

In the canonical series cascade form, the transfer function $H(z)$ is written as a product of first-order and second-order transfer functions:

$$H_i(z) = \frac{u(z)}{e(z)} = H_1(z) \cdot H_2(z) \cdot H_3(z) \dots H_p(z).$$

This equation yields the canonical series cascade form.



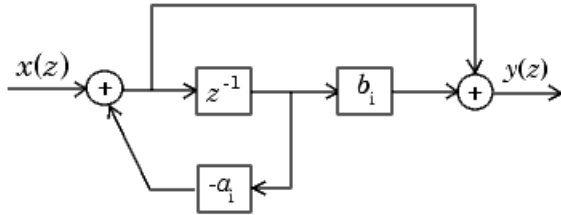
Factoring $H(z)$ into $H_i(z)$ where $i = 1, 2, 3, \dots, p$ can be done in a number of ways. Using the poles and zeros of $H(z)$, you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles and real zeros to produce either first-order or second-order transfer functions. You could also group two real zeros with a pair of conjugate complex poles or vice versa. Since there are many ways to obtain $H_i(z)$, you should compare the various groupings to see which produces the best results for the transfer function under consideration.

For example, one factorization of $H(z)$ might be

$$\begin{aligned} H(z) &= H_1(z) H_2(z) \dots H_p(z) \\ &= \prod_{i=1}^j \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^p \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}. \end{aligned}$$

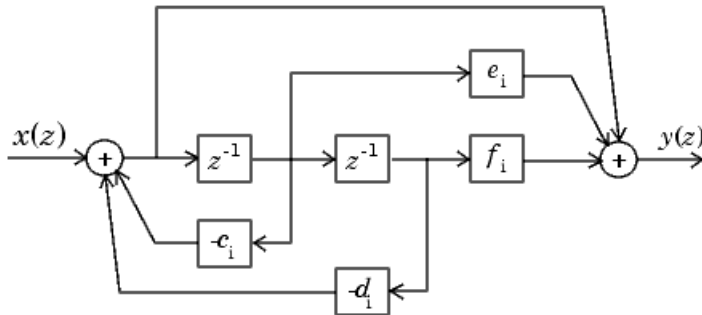
You must also take into consideration that the ordering of the individual $H_i(z)$'s will lead to systems with different numerical characteristics. You might want to try various orderings for a given set of $H_i(z)$'s to determine which gives the best numerical characteristics.

The first-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}}$$

The second-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

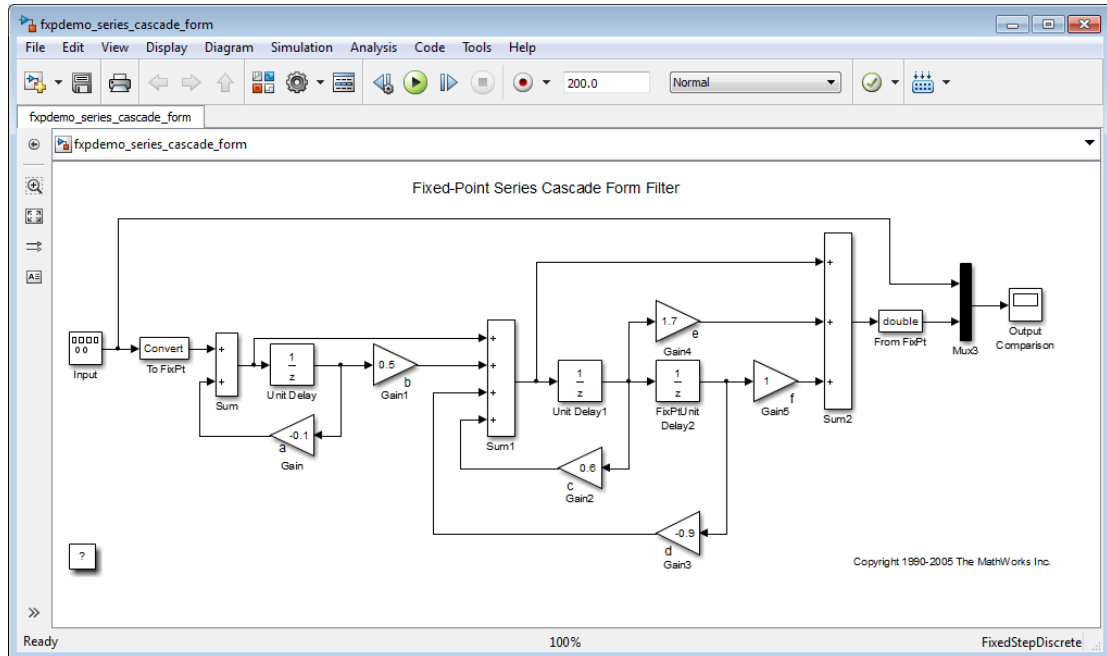
The series cascade form example transfer function is given by

$$H_{ex}(z) = \frac{(1 + 0.5z^{-1})(1 + 1.7z^{-1} + z^{-2})}{(1 + 0.1z^{-1})(1 - 0.6z^{-1} + 0.9z^{-2})}$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

`fxpdemo_series_cascade_form`

at the MATLAB command line.

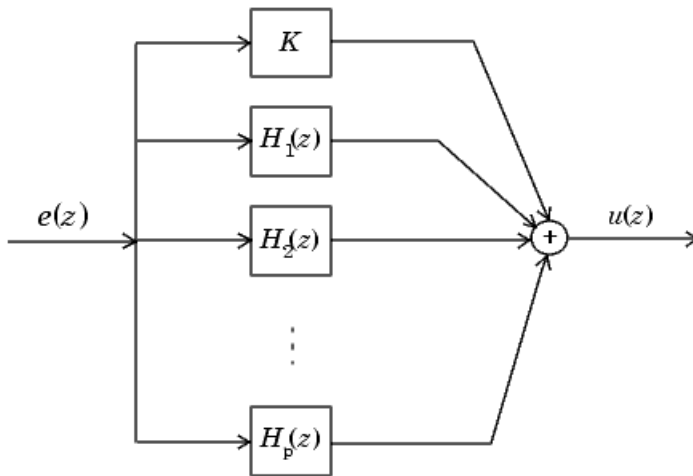


Parallel Form

In the canonical parallel form, the transfer function $H(z)$ is expanded into partial fractions. $H(z)$ is then realized as a sum of a constant, first-order, and second-order transfer functions, as shown:

$$H_i(z) = \frac{u(z)}{e(z)} = K + H_1(z) + H_2(z) + \dots + H_p(z).$$

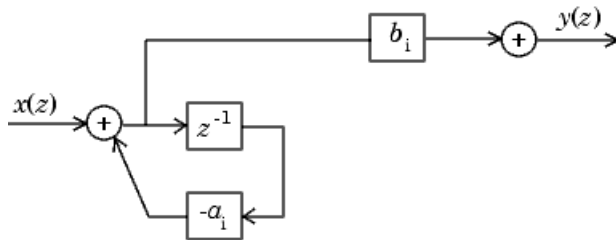
This expansion, where K is a constant and the $H_i(z)$ are the first- and second-order transfer functions, follows.



As in the series canonical form, there is no unique description for the first-order and second-order transfer function. Because of the nature of the Sum block, the ordering of the individual filters doesn't matter. However, because of the constant K , you can choose the first-order and second-order transfer functions such that their forms are simpler than those for the series cascade form described in the preceding section. This is done by expanding $H(z)$ as

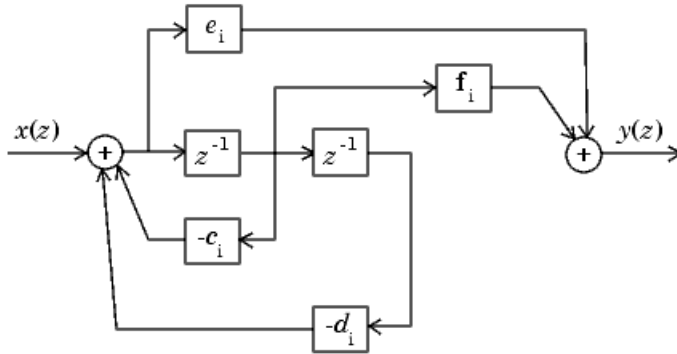
$$\begin{aligned}
 H(z) &= K + \sum_{i=1}^j H_i(z) + \sum_{i=j+1}^p H_i(z) \\
 &= K + \sum_{i=1}^j \frac{b_i}{1 + a_i z^{-1}} + \sum_{i=j+1}^p \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}.
 \end{aligned}$$

The first-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{b_i}{1 + a_i z^{-1}}$$

The second-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

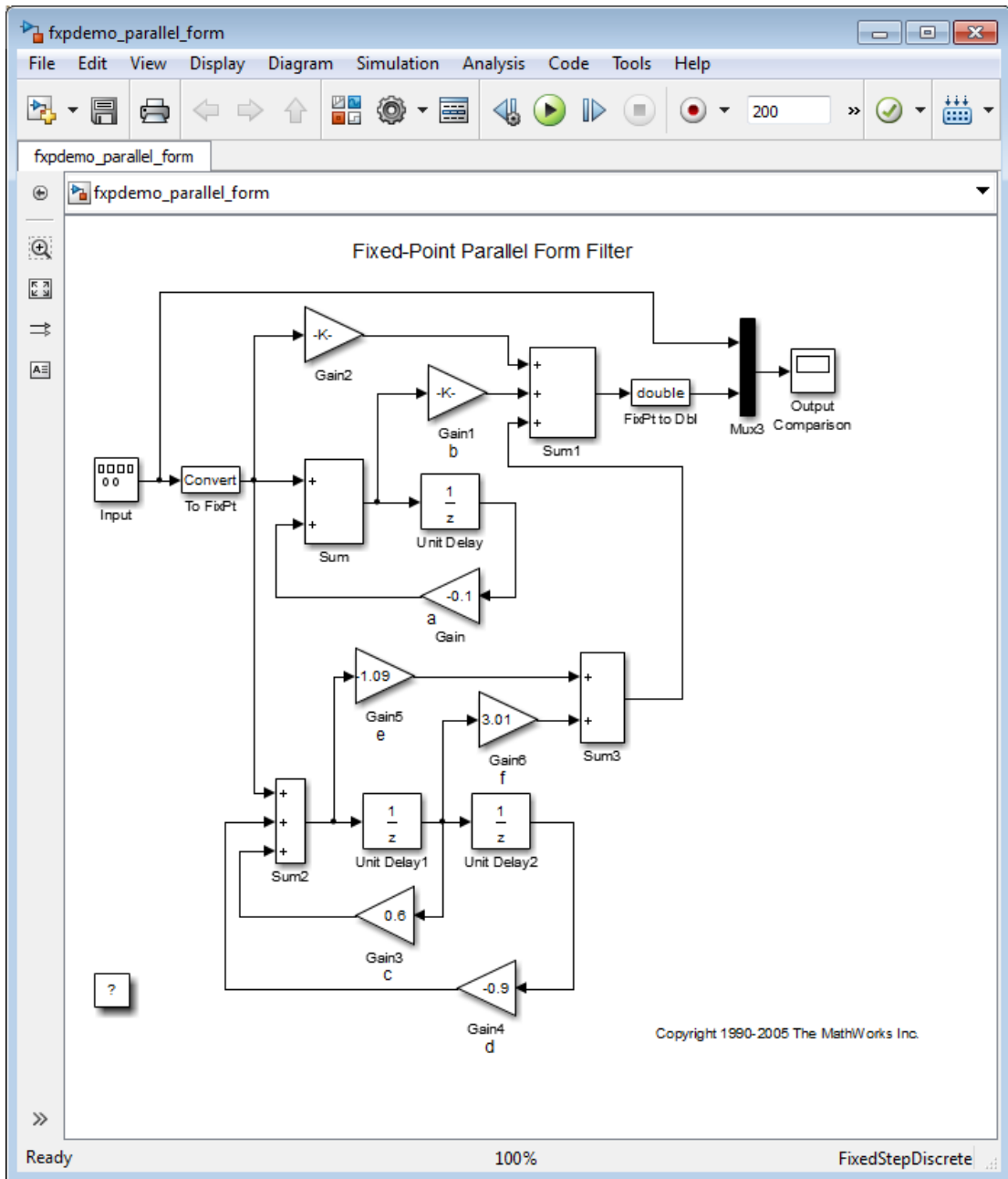
The parallel form example transfer function is given by

$$H_{ex}(z) = 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

`fxpdemo_parallel_form`

at the MATLAB command line.



Fixed-Point Advisor

Preparation for Fixed-Point Conversion

In this section...

- “Introduction” on page 35-2
- “Best Practices” on page 35-2
- “Run the Fixed-Point Advisor” on page 35-4
- “Fix a Task Failure” on page 35-5
- “Manually Fixing Failures” on page 35-5
- “Automatically Fixing Failures” on page 35-6
- “Batch Fixing Failures” on page 35-6
- “Restore Points” on page 35-7
- “Save a Restore Point” on page 35-7
- “Load a Restore Point” on page 35-8

Introduction

Using the Fixed-Point Advisor, you can prepare a model for conversion from a floating-point model or subsystem to an equivalent fixed-point representation. After preparing the model for conversion, use the Fixed-Point Tool to obtain initial fixed-point data types and then refine these data types.

Best Practices

Use a Known Working Model

Before using the Fixed-Point Advisor, verify that **update diagram** succeeds for your model. To update diagram, press **Ctrl+D**. If **update diagram** fails, before you start converting your model, fix the failure in your model.

Back Up Your Model

Back up your Simulink model first.

This practice provides you with a back up in case of error and a baseline for testing and validation.

Convert Small Models

The Fixed-Point Advisor is intended to assist in converting small models. Using larger models can result in long processing times.

Convert Subsystems

Convert subsystems within your model, rather than the entire model. This practice saves time and unnecessary conversions.

Specify Short Simulation Run Times

Specifying small simulation run times reduces task processing times. You can change the simulation run time in the Configuration Parameters dialog box. For more information, see “Start time” (Simulink) and “Stop time” (Simulink) in the *Simulink Reference*.

Make Small Changes to Your Model

Make small changes to your model so that you can identify where errors are accidentally introduced.

Isolate the System Under Conversion

If you encounter data type propagation issues with a particular subsystem, isolate this subsystem by placing Data Type Conversion blocks on the inputs and outputs of the system. The Data Type Conversion block converts an input signal of any Simulink software data type to the data type and scaling you specify for its **Output data type** parameter. This practice enables you to continue converting the rest of your model.

The ultimate goal is to replace all blocks that do not support fixed-point data types. You must eventually replace blocks that you isolate with Data Type Conversion blocks with blocks that do support fixed-point data types.

Use Lock Output Data Type Setting

You can prevent the Fixed-Point Advisor from replacing the current data type. Use the **Lock output data type setting against changes by the fixed-point tools** parameter available on many blocks. The default setting allows replacement. Use this setting when:

- You already know the fixed-point data types that you want to use for a particular block.

For example, the block is modeling a real-world component. Set up the block to allow for known hardware limitations, such as restricting outputs to integer values.

Specify the output data type of the block explicitly and select **Lock output data type setting against changes by the fixed-point tools**.

- You are debugging a model and know that a particular block accepts only certain data types.

Specify the output data type of upstream blocks explicitly and select **Lock output data type setting against changes by the fixed-point tools**.

Save Simulink Signal Objects

The Fixed-Point Advisor proposes data types for Simulink signal objects in your model. However, it does not automatically save Simulink signal objects. To preserve changes, before closing the model, save the Simulink signal objects in your workspace and model before closing the model.

Save Restore Point

Before making changes to your model that might cause subsequent update diagram failure, consider saving a restore point. For example, before applying proposed data types in task 3.1. For more information, see “Save a Restore Point” on page 35-7.

Run the Fixed-Point Advisor

- 1 Open a model.
- 2 Start the Fixed-Point Advisor by:
 - Type `fpcadvisor('model_name/subsystem_name')` at the MATLAB command line
 - Open the Fixed-Point Tool by selecting **Analysis > Data Type Design > Fixed-Point Tool**. Under **System under design**, select the system that you want to convert to fixed-point. Under **Fixed-point preparation**, click **Fixed-Point Advisor**.
- 3 In the Fixed-Point Advisor window, on the left pane, select the Fixed-Point Advisor folder.
- 4 Run the advisor by:

- Selecting **Run All** from the Run menu.
- Right-clicking the Fixed-Point Advisor folder and selecting **Run All** from the folder context menu.

The Fixed-Point Advisor runs the tasks in order until a task fails. A waitbar is displayed while each task runs.

- 5 Review the results. If a task fails because input parameters are not specified, select an **Input Parameter**. Then continue running to failure by right-clicking the task and selecting **Continue** from the context menu. If the task fails for a different reason, fix the task as described in “Fix a Task Failure” on page 35-5.

If your model contains referenced models, the Fixed-Point Advisor provides results for each referenced model instance.

Fix a Task Failure

Tasks fail when there is a step for you to take to convert your model from floating-point to fixed-point. The Fixed-Point Advisor provides guidance on how to fix the issues.

You can fix a failure using three different methods:

- Follow the instructions in the Analysis Result box. Use this method to fix failures individually. See “Manually Fixing Failures” on page 35-5.
- Use the Action box. Use this method to automatically fix all failures. See “Automatically Fixing Failures” on page 35-6.
- Use the Model Advisor Results Explorer. Use this method to batch fix failures. See “Batch Fixing Failures” on page 35-6

Note: A warning result is meant for your information. You can choose to fix the reported issue or move on to the next task.

Manually Fixing Failures

All checks have a **Result** pane that describes the recommended actions to manually fix failures.

To manually fix warnings or failures within a task:

- 1 Optionally, save a restore point so you can undo the changes that you make. For more information, see “Save a Restore Point” on page 35-7.
- 2 Under **Result**, review the recommended actions. Use the information to make changes to your model.
- 3 To verify that the task now passes, under **Analysis**, click **Run This Task**.

Automatically Fixing Failures

You can automatically fix failures using the **Action** box. The **Action** box applies all of the recommended actions listed in the **Result** box.

Caution: Prior to automatically fixing failures, review the **Analysis Result** box to ensure that you want to apply all of the recommended actions.

Automatically fix all failures within a task using the following steps:

- 1 Optionally, save a restore point so you can undo the changes that you make. For more information, see “Save a Restore Point” on page 35-7.
- 2 In the **Action** box, click **Modify All**.

The **Action Result** box displays a table of changes.

- 3 To verify that the task now passes, in the **Analysis** box, click **Run This Task**.

Batch Fixing Failures

If a task fails and you want to explore the results and make batch changes, use the following steps.

- 1 Optionally, save a restore point so you can undo the changes that you make. For more information, see “Save a Restore Point” on page 35-7.
- 2 In the **Analysis** box, click **Explore Result**.
- 3 Use the Model Advisor Result Explorer to modify block parameters.
- 4 When you finish making changes, in the Fixed-Point Advisor window, click **Run This Task** to see if the changes you made result in the task passing. Continue fixing failures and rerunning the task until the task passes.

Restore Points

The Fixed-Point Advisor provides a model and data restore point capability for reverting changes that you made in response to advice from the Fixed-Point Advisor. A restore point is a snapshot in time of the model, base workspace, and Fixed-Point Advisor.

Caution: A restore point saves only the current working model, base workspace variables, and Fixed-Point Advisor tree. It does not save other items, such as libraries and referenced submodels.

To learn how to save a restore point, see “Save a Restore Point” on page 35-7.

To learn how to load a restore point, see “Load a Restore Point” on page 35-8.

Save a Restore Point

When to Save a Restore Point

Consider saving a restore point:

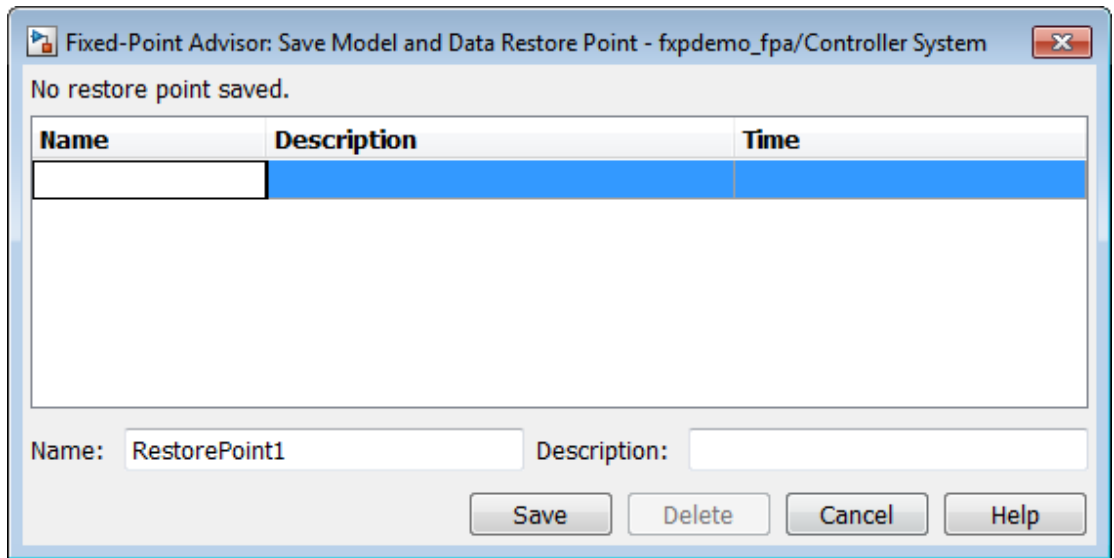
- Before applying changes to your model that might cause update diagram failure.
- Before attempting to fix failures.

How to Save a Restore Point

You can save a restore point and give it a name and optional description, or allow the Fixed Point Advisor to automatically name the restore point for you.

To save a restore point with a name and optional description:

- 1 From the main menu, select **File > Save Restore Point As**.



- 2 In the **Save Model and Data Restore Point** dialog box, in the **Name** field, enter a name for the restore point.
- 3 In the **Description** field, you can optionally add a description to help you identify the restore point.
- 4 Click **Save**.

The Fixed Point Advisor saves a restore point of the current model, base workspace, and Fixed Point Advisor status.

Note: To quickly save a restore point, go to **File > Save Restore Point**. The Fixed Advisor saves a restore point with the name `autosaven.n` where `n` is the sequential number of the restore point. If you use this method, you cannot change the name of, or add a description to, the restore point.

Load a Restore Point

When to Load a Restore Point

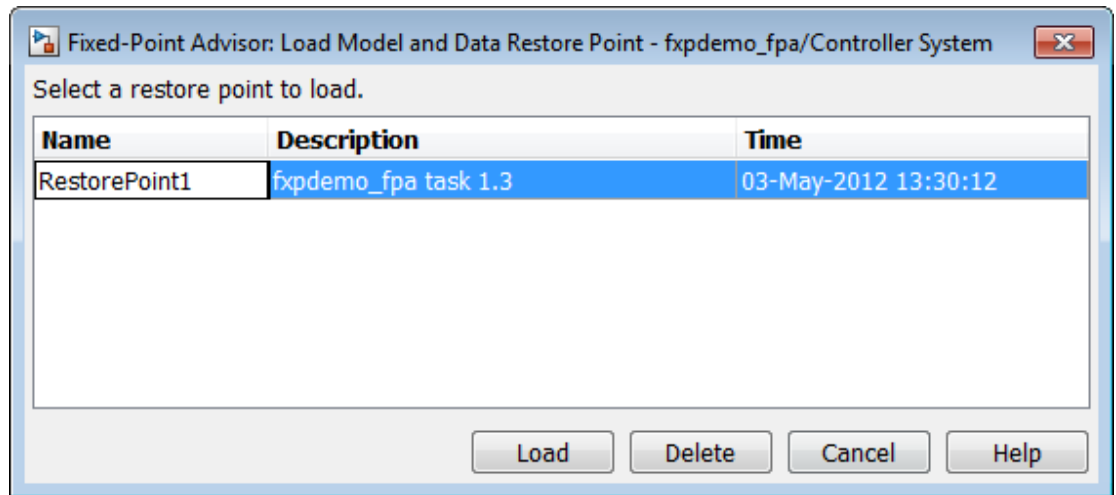
Load a restore point when:

- A task fails and you cannot continue the conversion. In this case, load a restore point saved earlier in the run to avoid rerunning all the previous tasks.
- You want to revert changes you made in response to advice from the Fixed-Point Advisor.

How to Load a Restore Point

To load a restore point:

- 1 Go to **File > Load Restore Point**.



- 2 In the **Load Model and Data Restore Point** dialog box, select the restore point that you want.
- 3 Click **Load**.

The Model Advisor issues a warning that the restoration will overwrite the current model and workspace.

- 4 Click **Load** to load the restore point that you selected.

The Fixed Point Advisor reverts the model, base workspace, and Fixed Point Advisor status.

Fixed-Point Tool

- “Fixed-Point Tool” on page 36-2
- “Run Management” on page 36-4
- “Use Shortcuts to Manage Runs” on page 36-6
- “Debug a Fixed-Point Model” on page 36-10
- “Logging Simulation Ranges for Referenced Models” on page 36-17
- “Log Simulation Ranges for Referenced Models” on page 36-21
- “Propose Data Types for a Referenced Model” on page 36-27
- “View Signal Names in Fixed-Point Tool” on page 36-30
- “Model Multiple Data Type Behaviors Using a Data Dictionary” on page 36-32

Fixed-Point Tool

In this section...

“Introduction to the Fixed-Point Tool” on page 36-2

“Using the Fixed-Point Tool” on page 36-2

Introduction to the Fixed-Point Tool

The Fixed-Point Tool is a graphical user interface that automates specifying fixed-point data types in a model. The tool collects range data for model objects. The range data comes from either design minimum and maximum values that objects specify explicitly, from logged minimum and maximum values that occur during simulation, or from minimum and maximum values derived using range analysis. Based on these values, the tool proposes fixed-point data types that maximize precision and cover the range. With this tool, you can review the data type proposals and then apply them selectively to objects in your model.

Fixed-Point Tool Capability	More Information
Deriving range information based on specified design range	“Derive Ranges” on page 40-7
Proposing data types based on simulation data	“Autoscaling Using the Fixed-Point Tool” on page 39-12
Proposing data types based on derived ranges	“Autoscaling Using the Fixed-Point Tool” on page 39-12
Proposing data types based on simulation data from multiple runs	“Propose Data Types Using Multiple Simulations” on page 39-58
Debugging fixed-point models	“Debug a Fixed-Point Model” on page 36-10

Using the Fixed-Point Tool

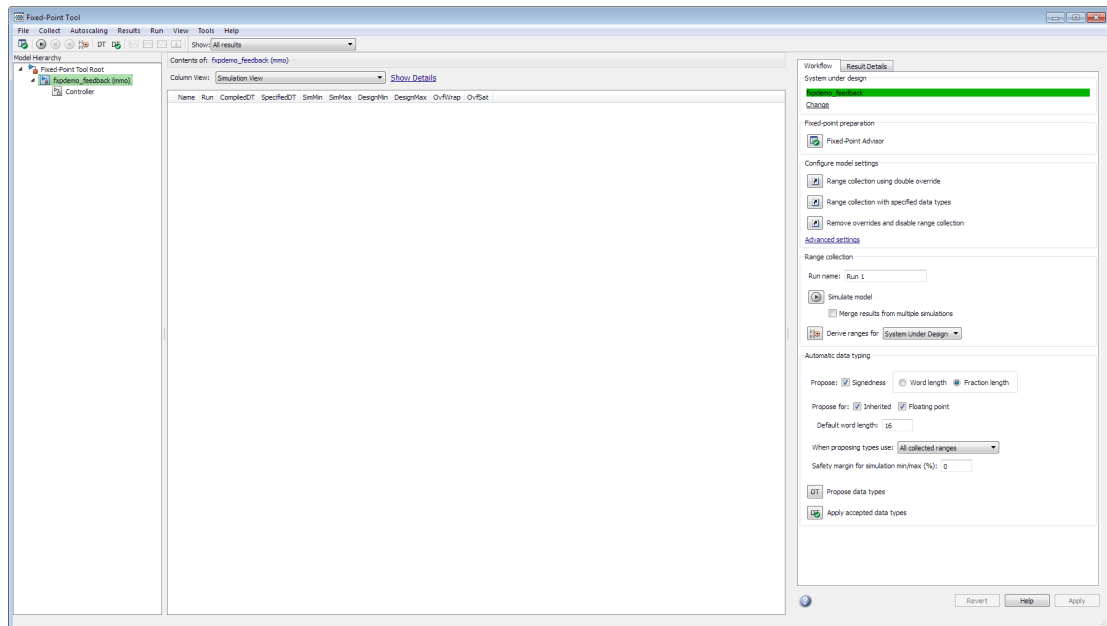
To open the Fixed-Point Tool, use any of the following methods:

- From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- From the model context menu, select **Fixed-Point Tool**.
- From a subsystem context menu, select **Fixed-Point Tool**.

If you want to open the tool programmatically, use the `fxptdlg` function. For more information, see `fxptdlg`.

The Fixed-Point Tool contains the following components:

- **Model Hierarchy** pane — Displays a tree-structured view of the Simulink model hierarchy.
- **Contents** pane — Displays a tabular view of objects that log fixed-point data in a system or subsystem.
- **Workflow** tab — Displays parameters for specifying particular attributes of a system or subsystem, such as its data type override and fixed-point instrumentation mode.
- **Toolbar** — Provides buttons for commonly used Fixed-Point Tool commands.
- **Advanced Settings** dialog— To open the **Advanced Settings** dialog far right-hand pane, under **Configure model settings** click the **Advanced settings** link. This editor provides the ability to configure shortcuts that set up the run name as well as model-wide data type override and instrumentation settings prior to simulation or range derivation. For more information, see “Run Management with the Shortcut Editor” on page 36-4.



Run Management

In this section...
“Run Management” on page 36-4
“Run Management with the Shortcut Editor” on page 36-4
“Manual Run Management” on page 36-5

Run Management

The Fixed-Point Tool supports multiple runs. Each run uses one set of model settings to simulate the model or to derive or propose data types. You can:

- Store multiple runs.
- Specify custom run names.
- Propose data types based on the results in any run.
- Apply data type proposals based on any run.
- Compare the results of any two runs.
- Rename runs directly in the Fixed-Point Tool **Contents** pane.

You can easily switch between different run setups using the factory default configuration settings. Alternatively, you can manually manage runs.

Run Management with the Shortcut Editor

You can use shortcuts prior to simulation to configure the run name as well as to configure model-wide data type override and instrumentation settings. The Fixed-Point Tool provides:

- Frequently used factory default configurations, such as **Range collection using double override**, which sets up your model so that you can override all fixed-point data types with double-precision numbers and logs the simulation minimum and maximum values and overflows.

Note: You can set up user-defined shortcuts across referenced model boundaries. The factory default shortcuts apply only to the top-level model and so do not affect the settings of any referenced model.

- The ability to add and edit custom shortcuts. The shortcuts are saved with the model so that you define them once and then reuse them multiple times. Use the Shortcut Editor in the Advanced Settings dialog to create or edit shortcuts and to add and organize shortcut buttons in the Fixed-Point Tool **Configure model settings** pane.

Note: You can use user-defined shortcuts across referenced model boundaries.

Manual Run Management

You can also manually manage runs using the following settings:

- In the **Range collection** pane, **Run name**.

Provide a new run name before a simulation or collecting derived minimum and maximum values so that you do not overwrite existing runs.

- Under **Configure model settings**, in the Advanced Settings dialog:
 - **Fixed-point instrumentation mode**
 - **Data type override**
 - **Data type override applies to**

More About

- “Use Shortcuts to Manage Runs” on page 36-6

Use Shortcuts to Manage Runs

In this section...

- “Why Use Shortcuts to Manage Runs” on page 36-6
- “When to Use Shortcuts to Manage Runs” on page 36-6
- “Add Shortcuts” on page 36-7
- “Edit Shortcuts” on page 36-7
- “Delete Shortcuts” on page 36-8
- “Capture Model Settings Using the Shortcut Editor” on page 36-9

Why Use Shortcuts to Manage Runs

Shortcuts provide a quick and easy way to set up data type override and fixed-point instrumentation settings prior to simulation or range derivation. You can associate a run name with each shortcut. When you apply a shortcut, you change the data type override and fixed-point instrumentation settings of multiple systems in your hierarchy simultaneously.

Shortcuts:

- Simplify the workflow. For example, you can collect a floating-point baseline in a clearly named run.
- Provide the ability to configure data type override and instrumentation settings on multiple subsystems in the model hierarchy at the same time. This capability is useful for models that have a complicated hierarchy.
- Are a convenient way to store frequently used settings and reuse them. This capability is useful when switching between different settings during debugging.
- Provide a way to store the original fixed-point instrumentation and data type override settings for the model. Preserving these settings in a shortcut provides a backup in case of failure and a baseline for testing and validation.

When to Use Shortcuts to Manage Runs

To ...	Use...
Autoscale your entire model	The factory default model configurations. These defaults provide an efficient way to override the model with floating-

To ...	Use...
	point data types or remove existing data type overrides. For more information, see “Propose Fraction Lengths Using Simulation Range Data” on page 39-41.
Debug a model	Configurations to switch between different data type override and fixed-point instrumentation modes. For more information, see “Debug a Fixed-Point Model” on page 36-10.
Manage the settings on multiple systems in a model. For example, if you are converting your model to fixed point one subsystem at a time.	The Advanced Settings Shortcuts tab to define your own shortcuts so that you can switch between different settings without manually changing individual settings each time.
Capture the initial settings of the model before making any changes to it.	The Advanced Settings Shortcuts tab to capture the model settings and save them in a named run. For more information, see “Capture Model Settings Using the Shortcut Editor” on page 36-9.

Add Shortcuts

- 1 On the Fixed-Point Tool, under **Configure model settings**, click **Advanced settings**.
- 2 For each subsystem that you want to specify a shortcut for, on the **Shortcuts** tab, in the **Model Hierarchy** pane, select the subsystem:
 - a In the **Name of shortcut** field, enter the shortcut name.

By default, if **Allow modification of run name** is selected, the software sets the **Run name** to the shortcut name. You can manually override the name.
 - b Edit the shortcut properties. See “Edit Shortcuts” on page 36-7.

Edit Shortcuts

- 1 On the Fixed-Point Tool, under **Configure model settings**, click **Advanced settings**.

- 2 In the **Shortcuts** tab, from the **Name of shortcut** list, select the shortcut that you want to edit.

The editor displays the run name, fixed-point instrumentation settings, and data type override settings defined by the shortcut.

Note: You cannot modify the factory default configurations.

- 3 If you do not want this shortcut to modify the existing fixed-point instrumentation settings on the model, clear **Allow modification of fixed-point instrumentation settings**.
- 4 If you do not want this shortcut to modify the existing data type override settings on the model, clear **Allow modification of data type override settings**.
- 5 If you do not want this shortcut to modify the run name on the model, clear **Allow modification of run name**.
- 6 If you want to modify the shortcut for a subsystem:
 - a Select the subsystem.
 - b If applicable, set the **Fixed-point instrumentation mode** to use when you apply this shortcut.
 - c If applicable, set the **Data type override** mode to use when you apply this shortcut.
 - d If applicable, set the **Run name** to use when you apply this shortcut.
 - e Click **Apply**.
- 7 Repeat step 6 to modify any subsystem shortcuts that you want.
- 8 Optionally, if you want the Fixed-Point Tool to display a button for this new shortcut, under **Manage shortcuts**, use the right arrow to move the shortcut to the list of shortcuts to display. Use the up and down arrows to change the order of the shortcut buttons.
- 9 Save the model to store the shortcut with the model.

Delete Shortcuts

To delete a shortcut from a model:

- 1 On the Fixed-Point Tool, under **Configure model settings**, click **Advanced settings**.

- 2 On the **Shortcuts** tab, under **Manage shortcuts**, in the **Shortcuts** table, select the shortcut that you want to delete.
- 3 Click the **Delete selected shortcut** button, .

Capture Model Settings Using the Shortcut Editor

- 1 On the Fixed-Point Tool, under **Configure model settings**, click **Advanced settings**.
- 2 In the **Shortcuts** tab, create a new shortcut, for example, **Initial subsystem settings**.

By default, if **Allow modification of run name** is selected, the software sets the **Run name** to the shortcut name. You can manually override the name.

- 3 Verify that **Allow modification of fixed-point instrumentation settings** and **Allow modification of data type override settings** are selected.
- 4 Click **Capture system settings**.

The software sets the **Fixed-point instrumentation mode**, **Data type override**, and, if appropriate, **Data type override applies to** for the systems in the model hierarchy.

- 5 Click **Apply**.
- 6 Save the model to store the shortcut with the model.

Debug a Fixed-Point Model

In this section...

“Simulating the Model to See the Initial Behavior” on page 36-10

“Debugging the Model” on page 36-12

“Simulating the Model Using a Different Input Stimulus” on page 36-14

“Debugging the Model with the New Input” on page 36-14

“Proposing Fraction Lengths for Math2 Based on Simulation Results” on page 36-15

“Verifying the New Settings” on page 36-15

This example shows how to:

- Identify which parts of a model cause numeric problems.

The current fixed-point settings on this model cause overflows. You debug the model by overriding the fixed-point settings on one subsystem at a time and simulating the model to determine how these fixed-point settings affect the model behavior.

- Create and use shortcuts to set up fixed-point instrumentation and data type override settings for different runs.

To optimize the model for two different inputs, you switch several times between different data type override and fixed-point instrumentation settings. Using shortcuts facilitates changing these settings.

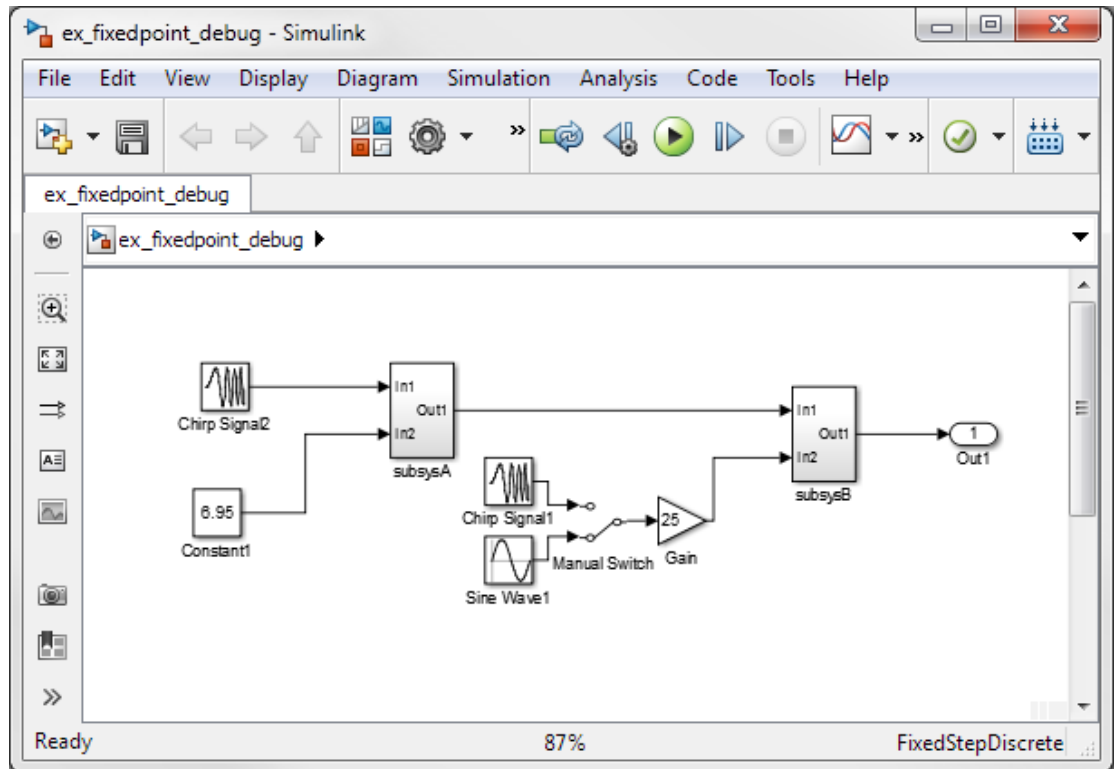
- Autoscale the model over the complete simulation range for both inputs.

Simulating the Model to See the Initial Behavior

Initially, the input to the Gain block is a sine wave of amplitude 7. Simulate the model using local system settings with logging enabled to see if any overflows or saturations occur.


- 1 Open the `ex_fixedpoint_debug` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_fixedpoint_debug
```



- 2 From the model **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to select `ex_fixedpoint_debug` as the system you want to convert.
- 4 In the Fixed-Point Tool, set up a shortcut for the initial system settings:
 - Under **Configure model settings**, click **Advanced settings**.
- 5 Click the **Shortcuts** tab. In the Shortcut Editor:
 - a On the **Model Hierarchy** pane, select `subsysA>Math1`.
 - b In the **Name of shortcut** field, enter **Setting A**.

The editor sets the **Run name** for this shortcut to **Setting A**.

- c** Set **Fixed-point instrumentation mode** to Minimums, maximums and overflows.
 - d** Set **Data type override** to Use local settings.
 - e** Click **Apply**.
 - f** On the **Model Hierarchy** pane, select `subsysA>Math2` and repeat steps (c) to (f).
 - g** On the **Manage shortcuts** pane, under **Shortcuts**, select **Setting A** then click the right arrow to move this shortcut to the list of shortcuts displayed in the Fixed-Point Tool.
- 6** Use this shortcut to set up a run. Use the settings to simulate the model.
 - a** In the Fixed-Point Tool, under **Model settings**, click **Setting A**.
 - b** Click the **Simulate** button, 

Simulink simulates the model using the fixed-point instrumentation and data type settings specified in **Setting A**. Afterward, on the **Contents** pane, the Fixed-Point Tool displays the simulation results for each block that logged fixed-point data. The tool stores the results in the run named **Setting A**. The Fixed-Point tool highlights `subsysB/Math2/Add1:Output` in red to indicate that there is an issue with this result. The **OvfWrap** column for this result shows that the block overflowed 51 times, which indicates a poor estimate for its scaling.

Debugging the Model

To debug the model, first simulate the model using local settings on the subsystem `Math1` while overriding the fixed-point settings on `Math2` with doubles. Simulating subsystem `Math2` with doubles override avoids quantization effects for this subsystem. If overflows occur, you can deduce that there are issues with the fixed-point settings in subsystem `Math1`.

Next, simulate the model using local settings on `Math2` and doubles override on `Math1`. If overflows occur for this simulation, there are problems with the fixed-point settings for subsystem `Math2`.

Setting Up Shortcuts

- 1** Use the Shortcut Editor to create the following new shortcuts.

Shortcut Name	Subsystem	Fixed-point instrumentation mode	Data type override	Data type override applies to
Setting B	Math1	MinMaxAndOverflow	Use local settings	N/A
	Math2	MinMaxAndOverflow	Double	All numeric types
Setting C	Math1	MinMaxAndOverflow	Double	All numeric types
	Math2	MinMaxAndOverflow	Use local settings	N/A

- 2 In the **Shortcuts** tab of the Advanced Settings dialog, under **Manage shortcuts**, add **Setting B** and **Setting C** to the list of buttons to display in the Fixed-Point Tool.

Testing Subsystem Math1 Settings

Simulate the model with original fixed-point settings on **Math1** while overriding the fixed-point settings with doubles on **Math2**.

- 1 In the Fixed-Point Tool, under **Configure model settings**, click **Setting B** to override fixed-point settings on **Math2**.
- 2 Click the **Simulate** button.

Simulink simulates the model using the fixed-point instrumentation and data type settings specified in **Setting B**, using fixed-point settings for **Math1** and overriding the fixed-point settings for **Math2**. No overflows occur, which indicates that the settings on **Math1** are not causing the overflows.

Testing Subsystem Math2 Settings

Simulate with original fixed-point settings on **Math2** while overriding the fixed-point settings with doubles on **Math1**.

- 1 Under **Configure model settings**, click **Setting C** to override the fixed-point settings on **Math1**.
- 2 Click the **Simulate** button.

Simulink simulates the model using the fixed-point instrumentation and data type settings specified in **Setting C**, using fixed-point settings for **Math2** and overriding

the fixed-point settings for `Math1`. Overflows occur in run **Setting C**, indicating that the settings on `Math2` are causing the overflows.

Simulating the Model Using a Different Input Stimulus

Simulate the model with a different input using the original fixed-point settings on subsystems `Math1` and `Math2`. Because you set up a shortcut for this initial set up, before rerunning the simulation, you can easily configure the model. Before simulating, select to merge the simulation results so that the tool gathers the simulation range for both inputs.

- 1 Under **Range collection**, select **Merge results from multiple simulations**.
- 2 In the `ex_fixedpoint_debug` model, double-click the Manual Switch block to select `Chirp Signal1` as the input to the Gain block.
- 3 In the Fixed-Point Tool, simulate using the original fixed-point settings for `Math1` and `Math2`.
 - a Under **Configure model settings**, click **Setting A**.
 - b Click the **Simulate** button.

Simulink simulates the model using the fixed-point instrumentation and data type settings specified in **Setting A**. Afterward, in the **Contents** pane, the Fixed-Point Tool displays the simulation results for each block that logged fixed-point data. The tool stores the results in the run named **Setting A**.

Tip: In the Fixed-Point Tool **Contents** pane, click **Run** to sort the results in this column.

Debugging the Model with the New Input

- 1 Simulate the model with original fixed-point settings on `Math1` while overriding the fixed-point settings with doubles on `Math2`.
 - a Under **Configure model settings**, click **Setting B**.
 - b Click the **Simulate** button.

No overflows occur, which indicates that the settings on `Math1` are not causing the overflows.

- 2 Simulate with original fixed-point settings on `Math2` while overriding the fixed-point settings with doubles on `Math1`.
 - a Under **Configure model settings**, click **Setting C**.
 - b Click the **Simulate** button.

Overflows occur, which indicates that the fixed-point settings on `Math2` are causing the overflows. Next, use the Fixed-Point Tool to propose new data types for this subsystem.

Proposing Fraction Lengths for `Math2` Based on Simulation Results

- 1 Under **System under design**, click **Change**. In the System Selector, select `subsysB/Math2`.
- 2 Under **Automatic data typing**, click **Propose data types**.
- 3 In the Propose Data Types dialog box, select **Setting B** as the run to use for proposing data types and click **OK**. This run simulated `Math2` with double override to obtain the 'ideal' behavior of the subsystem based on the simulation results for both input stimuli.

The Fixed-Point Tool proposes new fixed-point data types for the objects in subsystem `Math2` to avoid numerical issues such as overflows.

- 4 On the **Contents** pane **ProposedDT** column, examine the proposed data types for the objects in `Math2`. The tool proposed new fixed-point data types with reduced precision for the `Add1` block Output and Accumulator.
- 5 Because the Fixed-Point Tool marked all the proposed results with a green icon to indicate that the proposed data types pose no issues for these objects, accept the proposals.

Under **Automatic data typing**, click **Apply accepted data types**.

Verifying the New Settings

Verify that the new settings do not cause any numerical problems by simulating the model using local settings for subsystems `Math1` and `Math2` and logging the results. Use shortcut **Setting A** that you set up for these settings.

- 1 Under **System under design**, click **Change**. In the System Selector, select `ex_fixedpoint_debug`.

- 2** Under **Configure model settings**, click **Setting A**.
- 3** Under **Range collection**, set **Run name** to **Setting A2** so that the Fixed-Point Tool does not overwrite the previous results for this shortcut.
- 4** Click the **Simulate model** button.

Simulink simulates the model using the new fixed-point settings. Afterward, the Fixed-Point Tool displays the simulation results in run **Setting A2**. No overflows or saturations occur indicating that the model can now handle the full input range.

More About

- “Use Shortcuts to Manage Runs” on page 36-6

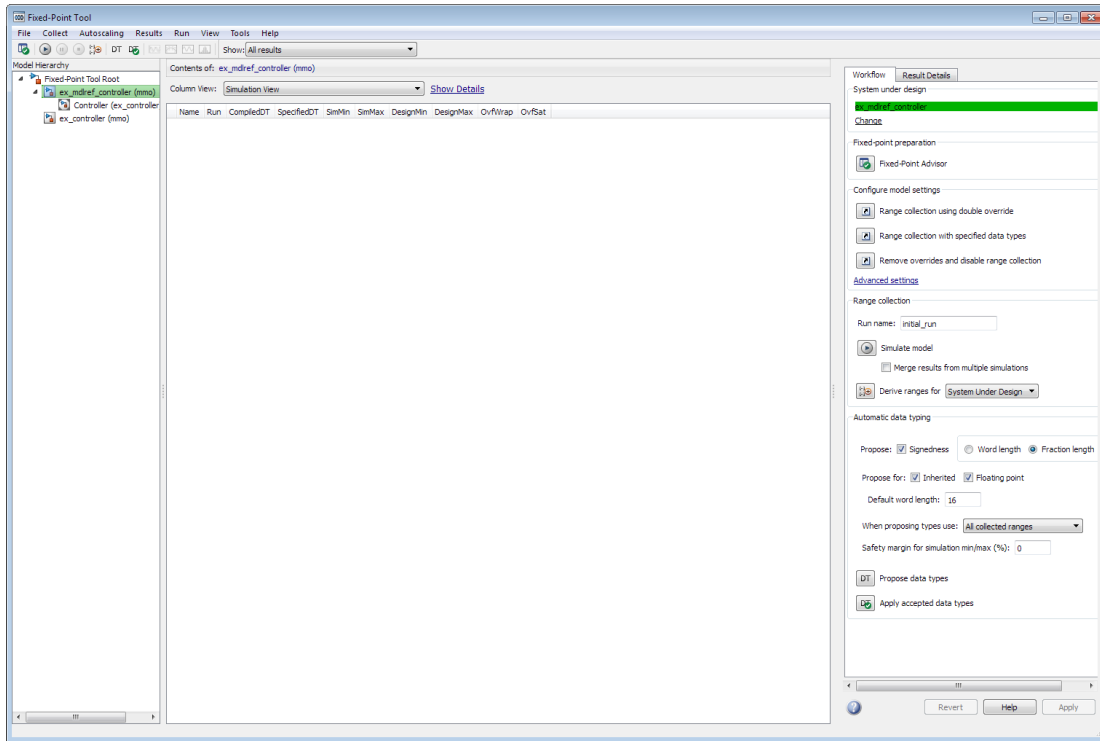
Logging Simulation Ranges for Referenced Models

In this section...
“Viewing Simulation Ranges for Referenced Models” on page 36-17
“Fixed-Point Instrumentation and Data Type Override Settings” on page 36-19

Viewing Simulation Ranges for Referenced Models

The Fixed-Point Tool logs simulation minimum and maximum values (ranges) for referenced models. The tool logs these values only for instances of the referenced model that are in Normal mode. It does not log simulation minimum and maximum values for instances of the referenced model that are in non-Normal modes. If your model contains multiple instances of a referenced model and some are instances are in Normal mode and some are not, the tool logs and displays data for those that are in Normal mode.

If a model contains a referenced model, the Fixed-Point Tool **Model Hierarchy** pane displays a subnode for the instance of the referenced model as well as a node for the referenced model. For example, the `ex_mdhref_controller` model contains a Model block that references the `ex_controller` model. The Fixed-Point Tool shows both models in the model hierarchy.



If a model contains multiple instances of a referenced model, the tool displays each instance of the referenced model in this model as well as a node for the referenced model. For example, the `ex_multi_instance` model contains two instances of the referenced model `ex_sum`. The Fixed-Point Tool displays both models and both instances of the referenced model in the model hierarchy.

The screenshot shows the Fixed-Point Tool interface. On the left, the Model Hierarchy displays a tree structure under 'Fixed-Point Tool Root' containing 'ex_multi_instance (mmo-off)', 'Model (ex_sum1) (mmo-off)', and 'Model1 (ex_sum1) (mmo-off)'. The main window shows the 'Contents of: ex_multi_instance (mmo-off)' with a 'Simulation View' column view. Below the column view, a table displays simulation results for the first instance of the referenced model `ex_sum1` in `ex_multi_instance`.

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax
Gain	run1		double	Inherit: Same as input	-15.93863374137345	15.993177648664082
Gain1	run1		double	Inherit: Same as input	-23.907950612060173	23.98976647299612
Out1	run1			Inherit: auto		
Out2	run1			Inherit: auto		

The tool logs and displays the results for each instance of the referenced model. For example, here are the results for the first instance of the referenced model `ex_sum1` in `ex_multi_instance`.

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax
Sum : Accumulator	run1		double	Inherit: Inherit via internal rule	-23.907950612060173	23.98976647299612
Sum : Output	run1		double	Inherit: Inherit via internal rule	-7.996588824332038	7.969316870686724

Here are the results for the second instance of `ex_sum1`.

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax
Sum : Accumulator	run1		double	Inherit: Inherit via internal rule	-19.92329217671681	19.9914720608301
Sum : Output	run1		double	Inherit: Inherit via internal rule	0	0

In the referenced model node, the tool displays the union of the results for each instance of the referenced model.

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax
Sum : Accumulator	run1		double	Inherit: Inherit via internal rule	-23.907950612060173	23.98976647299612
Sum : Output	run1		double	Inherit: Inherit via internal rule	-7.996588824332038	7.969316870686724

Fixed-Point Instrumentation and Data Type Override Settings

When you simulate a model that contains referenced models, the data type override and fixed-point instrumentation settings for the top-level model do not control the settings for the referenced models. You must specify these settings separately for the referenced model. If the settings are inconsistent, for example, if you set the top-level model data type override setting to double and the referenced model to use local settings and the referenced model uses fixed-point data types, data type propagation issues might occur.

You can set up user-defined shortcuts across referenced model boundaries. The factory default configurations apply only to the top-level model and so do not affect the settings of any referenced model.

When you change the fixed-point instrumentation and data type override settings for any instance of a referenced model, the settings change on all instances of the model and on the referenced model itself.

Related Examples

- “Log Simulation Ranges for Referenced Models” on page 36-21
- “Propose Data Types for a Referenced Model” on page 36-27

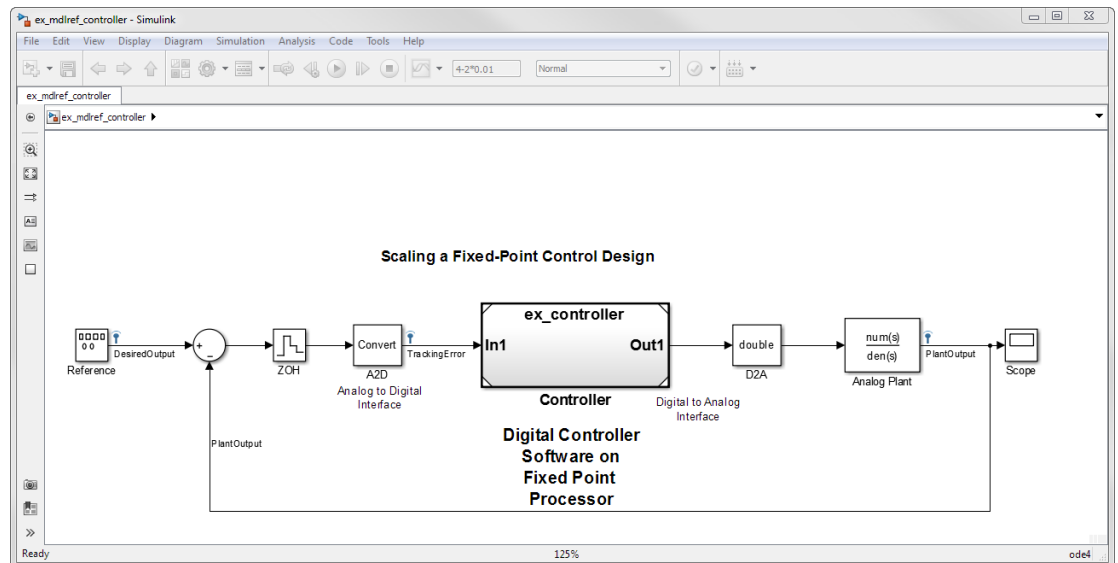
Log Simulation Ranges for Referenced Models

This example shows how to log simulation minimum and maximum values for a model that contains multiple instances of the same referenced model.

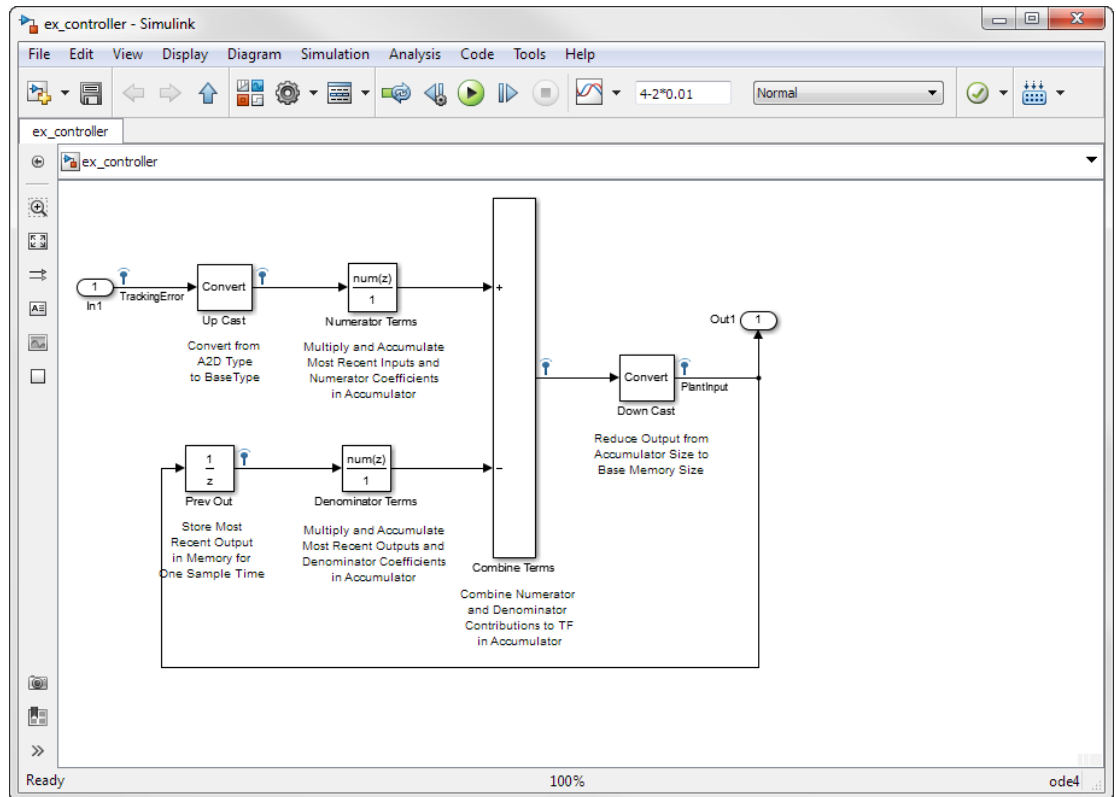
Simulate the Model Using Local Settings

- 1 Open the `ex_mdref_controller` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_mdref_controller
```



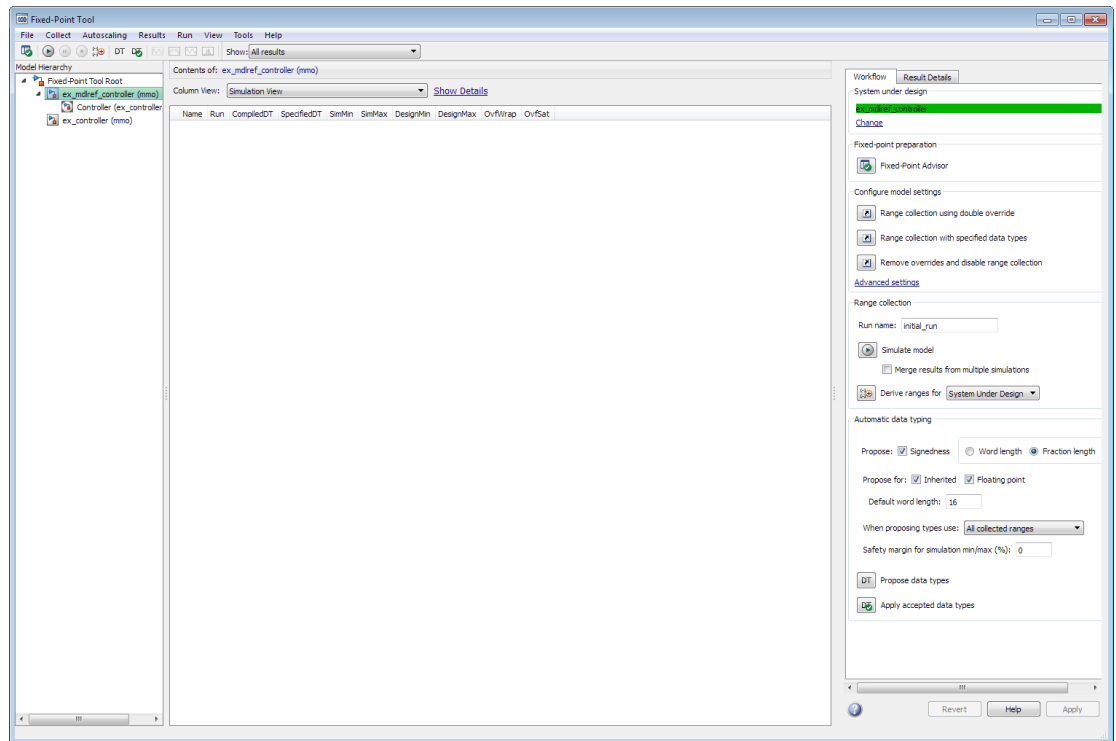
The model contains a Model block that references the `ex_controller` model. Using a referenced model isolates the controller from the rest of the system. This method is useful to help you configure a model to determine the effect of fixed-point data types on a system. Using this approach, you convert only the referenced model because this is the system of interest.



- 2 In the `ex_mdhref_controller` model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.

The Fixed-Point Tool opens. In its **Model Hierarchy** pane, the tool displays two model nodes, one for the `ex_mdhref_controller` model showing that this model contains a Model block that refers to the `ex_controller` model, and another for the `ex_controller` model itself.

- 3 Under **System under design**, click **Continue** to select `ex_mdhref_controller` as the subsystem you want to convert.



- 4 Under **Configure model settings**, click the **Range collection with specified data types** button. This configuration sets:
 - **Fixed-point instrumentation mode** to Minimums, maximums and overflows.
 - **Data type override** to Use local settings so the model will log simulation data using the data types set up on the model.
- 5 Under **Range collection**, set **Run name** to `initial_run` and then click **Apply**.
 Providing a unique name for the run avoids accidentally overwriting results from previous runs and enables you to identify the run more easily.
- 6 In the Fixed-Point Tool, click **Simulate model**.

Simulink simulates the model. Afterward, the Fixed-Point Tool displays in its **Contents** pane the simulation results for each block that logged fixed-point data. By default, it displays the Simulation View of these results.

- 7 In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_controller` model.

The Fixed-Point tool displays the results for the referenced model and highlights the **Up Cast** block in red to indicate that there is an issue with this result. The **OvfSat** column for this result shows that the block saturated 23 times, which indicates poor scaling.

The screenshot shows the Fixed-Point Tool interface. The 'Contents' pane displays a table of simulation results for the 'ex_controller' model. The table has the following columns: Name, Run, CompletedDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OvfWrap, and OvfSat. The 'Down Cast' block is highlighted in red, indicating a saturation issue. The 'OvfSat' column for this block shows 23 saturations.

Name	Run	CompletedDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax	OvfWrap	OvfSat
↑ Numerator	initial_run	fixed(1, 16, 14)	fixed(1, 16, 14)	-2	1.95993959464375				23
↓ Numerator Terms : Accumulator	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-2.8503241796875	2.85009769625				
↓ Numerator Terms : Output	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-1.77197265625	1.693603515625				
↓ Numerator Terms : Product output	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-2.8503241796875	2.85009769625				
↓ Denominator Terms : Accumulator	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-3.0791015625	4.859130859375				
↓ Denominator Terms : Output	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-0.700927734375	3.505859375				
↓ Denominator Terms : Product output	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-3.0791015625	4.859130859375				
↓ Combine Terms : Output	initial_run	fixed(1, 32, 12)	fixed(1, 32, 12)	-2.43920894375	0.661376953125				
↓ Combine Terms : Accumulator	initial_run	fixed(1, 32, 12)	Inherent: Inherent via internal rule	-2.43920894375	3.505859375				
↓ Down Cast	initial_run	fixed(1, 16, 5)	fixed(1, 16, 5)	-2.468375	0.65625				

The right-hand pane shows configuration options for range collection and automatic data typing. The 'Range collection' section includes options for 'Run name', 'Simulate model', and 'Merge results from multiple simulations'. The 'Automatic data typing' section includes options for 'Propose' (Signedness, Word length, Fracton length), 'Propose for' (Inherited, Floating point), 'Default word length', 'When proposing types use', 'Safety margin for simulation min/max (%)', 'Propose data types', and 'Apply accepted data types'.

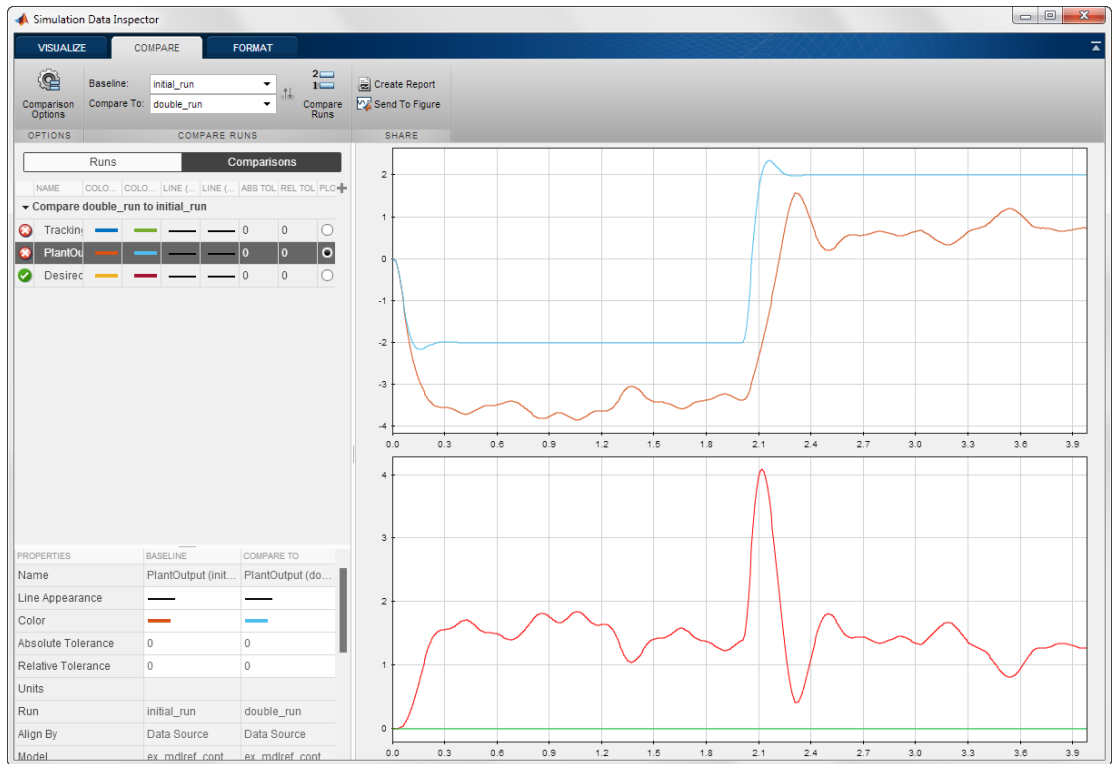
Next, use data type override mode to perform a global override of the fixed-point data types and scaling using double-precision numbers to avoid quantization effects. Later, you use these simulation results when performing automatic data typing.

Gather a Floating-Point Benchmark

- 1 Under **Configure model settings**, click the **Range collection using double override** button.
- 2 Under **Range collection**, set **Run name** to `double_run` and then click **Apply**.
- 3 In the Fixed-Point Tool, click **Simulate**.

Simulink simulates the `ex_mdhref_controller` model in data type override mode and stores the results in the run named `double_run`. Afterward, the Fixed-Point Tool displays in its **Contents** pane the results along with those of the run that you generated previously.

- 4 Use the Simulation Data Inspector to view the `initial_run` and `double_run` versions of the signal associated with the Analog Plant output. In the **Contents** pane, select the **Analog Plant** result. Click **Compare Runs** to launch the Simulation Data Inspector. The Simulation Data Inspector displays the two versions of the signal associated with the Analog Plant output (upper axes), and the difference between the signals (lower axes).




Now you are ready to propose data types based on the simulation results from the doubles override run.

Related Examples

- “Propose Data Types for a Referenced Model” on page 36-27

Propose Data Types for a Referenced Model

This example shows how to propose data types for a referenced model. To run this example, you must first run the “Log Simulation Minimum and Maximum Values for Referenced Models” example. See “Log Simulation Ranges for Referenced Models” on page 36-21

- 1 In the Fixed-Point Tool, under **System under design**, click **Change**. In the System Selector, select the `ex_controller` model.
- 2 Under **Automatic data typing**, specify the **Safety margin for simulation min/max (%)** parameter as 20 and click **Apply**.
- 3 Click **Propose data types**, .

Because no design minimum and maximum information is supplied, the simulation minimum and maximum data that was collected during the simulation run is used to propose data types. The **Safety margin for simulation min/max (%)** parameter value multiplies the “raw” simulation values by a factor of 1.2. Setting this parameter to a value greater than 1 decreases the likelihood that an overflow will occur when fixed-point data types are being used.

Because of the nonlinear effects of quantization, a fixed-point simulation will produce results that are different from an idealized, doubles-based simulation. Signals in a fixed-point simulation can cover a larger or smaller range than in a doubles-based simulation. If the range increases enough, overflows or saturations could occur. A safety margin decreases the likelihood of this happening, but it might also decrease the precision of the simulation.

- 4 In the Propose Data Types dialog box, select `double_run` and click **OK**.

The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.

The Fixed-Point Tool uses the minimum and maximum values stored in the selected run to propose each block's scaling such that the precision is maximized while the full range of simulation values is spanned. The tool displays the proposed scaling in its **Contents** pane. Now, it displays the Automatic Data Typing View to provide information, such as **ProposedDT**, **ProposedMin**, and **ProposedMax**, which are relevant at this stage of the fixed-point conversion.

Contents of: ex_controller* (mmo-dbi)

Column View: Automatic Data Typing View [Show Details](#)

Name	Run	Accept	ProposedDT	SpecifiedDT	CompiledDT	SimMin	SimMax	ProposedMin	ProposedMax
Up Cast	double_run	<input checked="" type="checkbox"/>	fixdt(1,16,12)	fixdt(1,16,14)	double	-2	3.9999999999711746	-8	7.999755859
Numerator Terms : Accumulator	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,28)	fixdt(1,32,12)	double	-5.677304459288715	5.700524518426912	-8	7.999999996
Numerator Terms : Output	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,28)	fixdt(1,32,12)	double	-3.367372640959928	3.5439615259925983	-8	7.999999996
Numerator Terms : Product output	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,28)	fixdt(1,32,12)	double	-5.677304459288715	5.700524518426912	-8	7.999999996
Denominator Terms : Accumulator	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,27)	fixdt(1,32,12)	double	-8.516638478410028	5.39648751512156	-16	15.99999999
Denominator Terms : Output	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,28)	fixdt(1,32,12)	double	-6.475416336873576	3.4877081684371363	-8	7.999999996
Denominator Terms : Product output	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,27)	fixdt(1,32,12)	double	-8.516638478410028	5.39648751512156	-16	15.99999999
Combine Terms : Output	double_run	<input checked="" type="checkbox"/>	fixdt(1,32,28)	fixdt(1,32,12)	double	-2.4135009037899806	4.32700180757929	-8	7.999999996
Combine Terms : Accumulator	double_run	<input type="checkbox"/>	n/a	Inherit: Inherit via internal rule	double	-6.475416336873576	4.32700180757929	-8	7.999755859
Down Cast	double_run	<input checked="" type="checkbox"/>	fixdt(1,16,12)	fixdt(1,16,5)	double	-2.4135009037899806	4.32700180757929	-8	7.999755859
In1	double_run	<input type="checkbox"/>	n/a	fixdt(1,8,4)					
Out1	double_run	<input checked="" type="checkbox"/>	fixdt(1,16,12)	fixdt(1,16,5)				-8	7.999755859

- Review the scaling that the Fixed-Point Tool proposes. You can choose to accept the scaling proposal for each block by selecting the corresponding **Accept** check box in the **Contents** pane. By default, the Fixed-Point Tool accepts all scaling proposals that differ from the current scaling. For this example, verify that the **Accept** check box associated with the active run is selected for each of the Controller system's blocks.

To view more information about a proposal, select the result and click the **Result Details** tab.

- In the Fixed-Point Tool, click the **Apply accepted data types** button .

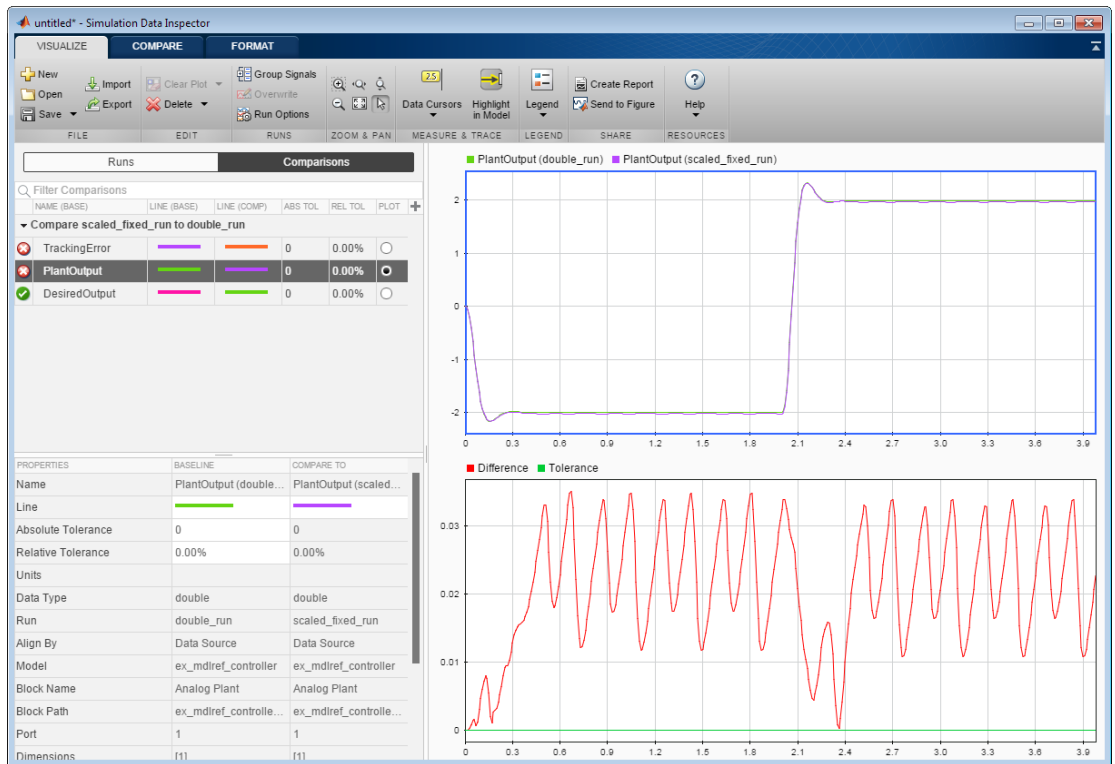
The Fixed-Point Tool applies the scaling proposals that you accepted in the previous step.

- Under **Configure model settings**, click the **Range collection with specified data types** button. Under **Range collection**, set the run name to `scaled_fixed_run`
- Click **Simulate model**.

Simulink simulates the `ex_mdhref_controller` model using the new scaling that you applied. Afterward, the Fixed-Point Tool displays in its **Contents** pane information about blocks that logged fixed-point data.

- Use the Simulation Data Inspector to plot the Analog Plant output for the floating-point and fixed-point runs and the difference between them.

The difference plot shows that the difference between the floating-point signal and the fixed-point signal is within the specified tolerance of **0.04**.



Related Examples

- “Log Simulation Ranges for Referenced Models” on page 36-21

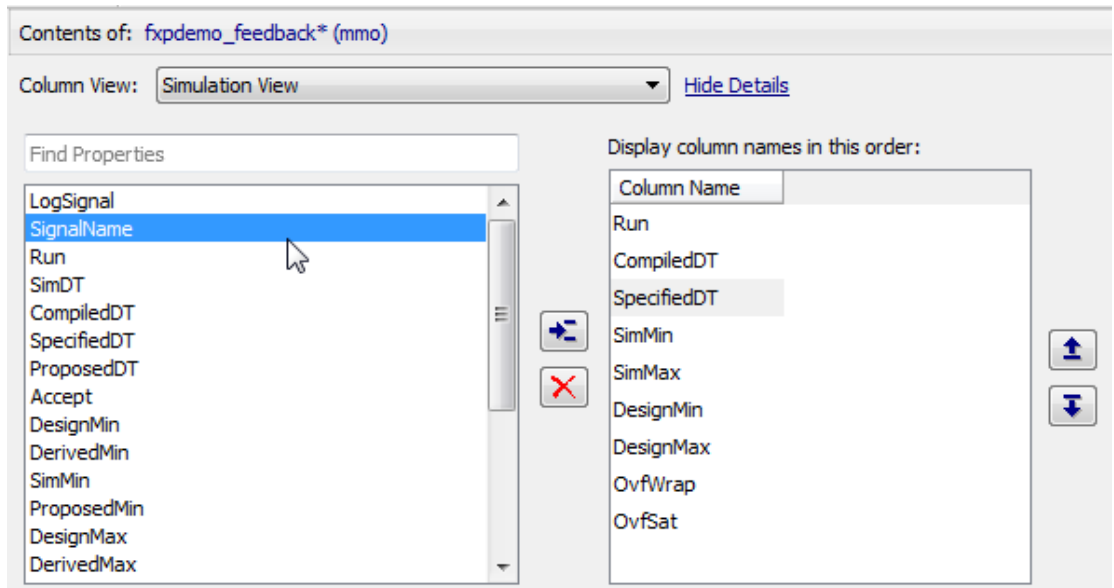
More About

- “Logging Simulation Ranges for Referenced Models” on page 36-17

View Signal Names in Fixed-Point Tool

To view signal names in the Fixed-Point Tool:

- 1 In the Fixed-Point Tool **Contents** pane, click **Show Details**.
- 2 In the list box of available columns, select **SignalName**.



- 3 Click .

The Fixed-Point Tool includes **SignalName** in the list box of columns to display.

- 4 Optionally, use the up and down arrow buttons to change the display order for the columns.
- 5 Click **Hide Details**.
- 6 If a signal has a name, the Fixed-Point Tool displays the name in the **Contents** pane.

Column View: Simulation View [St](#)

	Name	Run	SignalName	CompiledDT
✱	A2D	initial_run	TrackingError	fixdt(1,8,4)
✱	A2D	double_run	TrackingError	double
✱	A2D	scaled_fixed_run	TrackingError	fixdt(1,8,4)
✱	Analog Plant	initial_run	PlantOutput	
✱	Analog Plant	double_run	PlantOutput	
✱	Analog Plant	scaled_fixed_run	PlantOutput	

Model Multiple Data Type Behaviors Using a Data Dictionary

This example shows how to use referenced data dictionaries to store multiple sets of data types for a model. This example also shows how to change the data types by switching the referenced data dictionary.

- 1 Open the folder containing the example. At the MATLAB command line, enter:

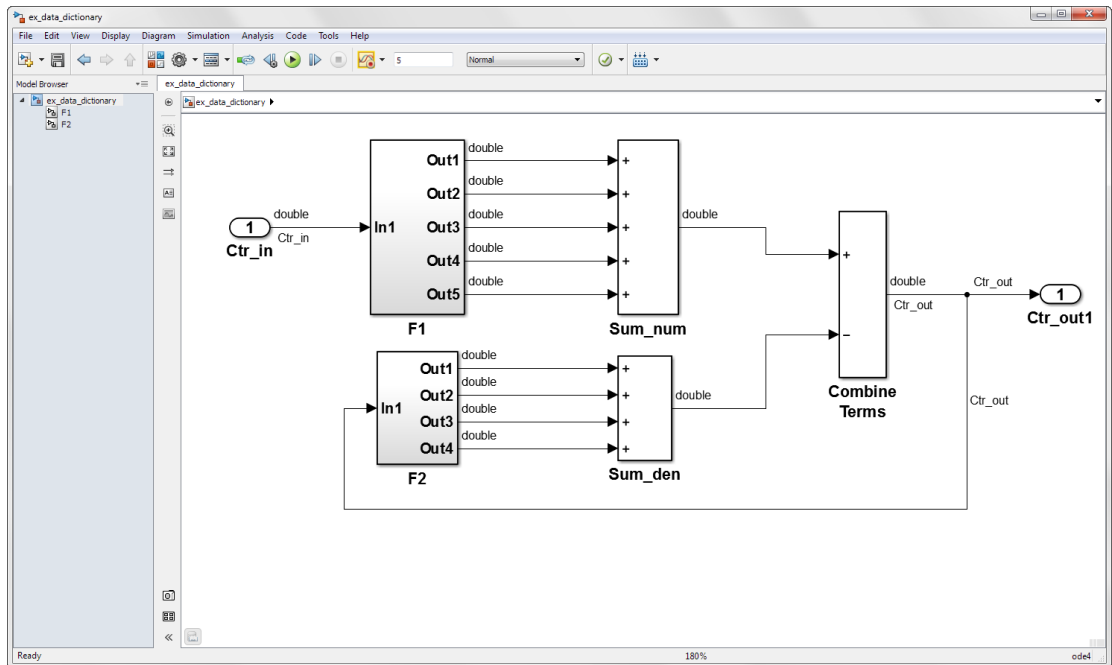
```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

- 2 Copy these files to a local writable folder:

- `ex_data_dictionary.slx` — Simulink model using a data dictionary to store its data types
- `mdl_dd.sldd` — Main data dictionary
- `flt_dd.sldd` — Referenced data dictionary using floating-point data types
- `fix_dd.sldd` — Referenced data dictionary using fixed-point data types

- 3 In the folder you copied the files to, open the `ex_data_dictionary` model.

```
ex_data_dictionary
```

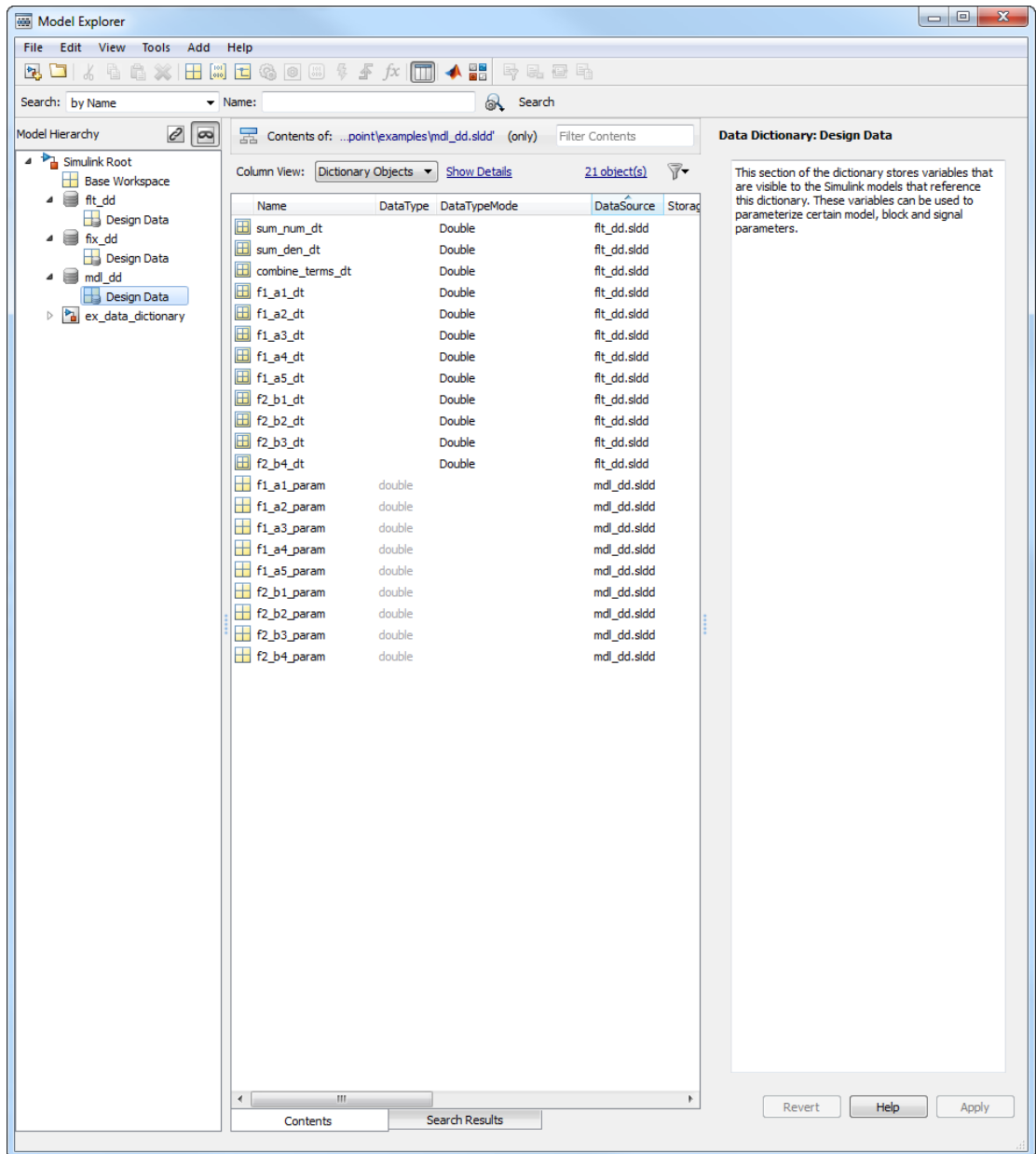



4

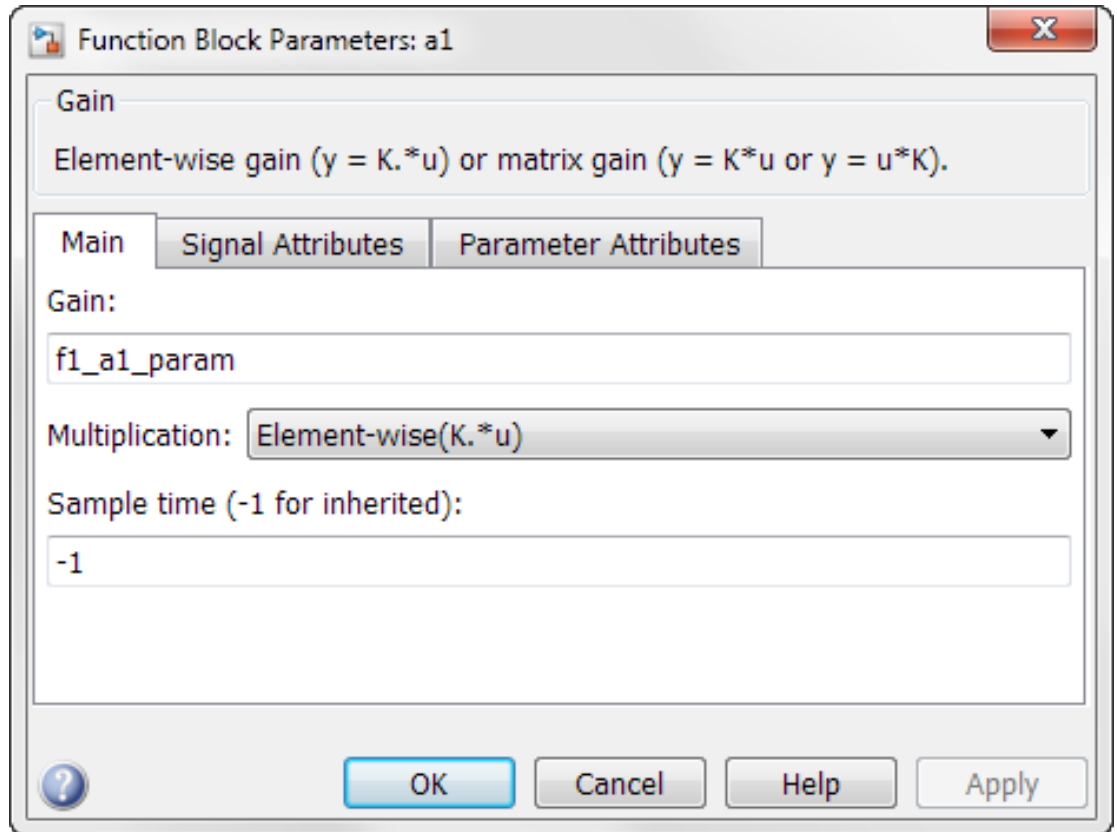
In the lower left corner of the Simulink Editor, click  to open the dictionary.

The data dictionary defines the parameters of the Gain blocks in the F1 and F2 subsystems. `mdl_dd` is associated with a referenced data dictionary, `fl_t_dd`, which defines the output data types of the gain blocks in the model's subsystems.

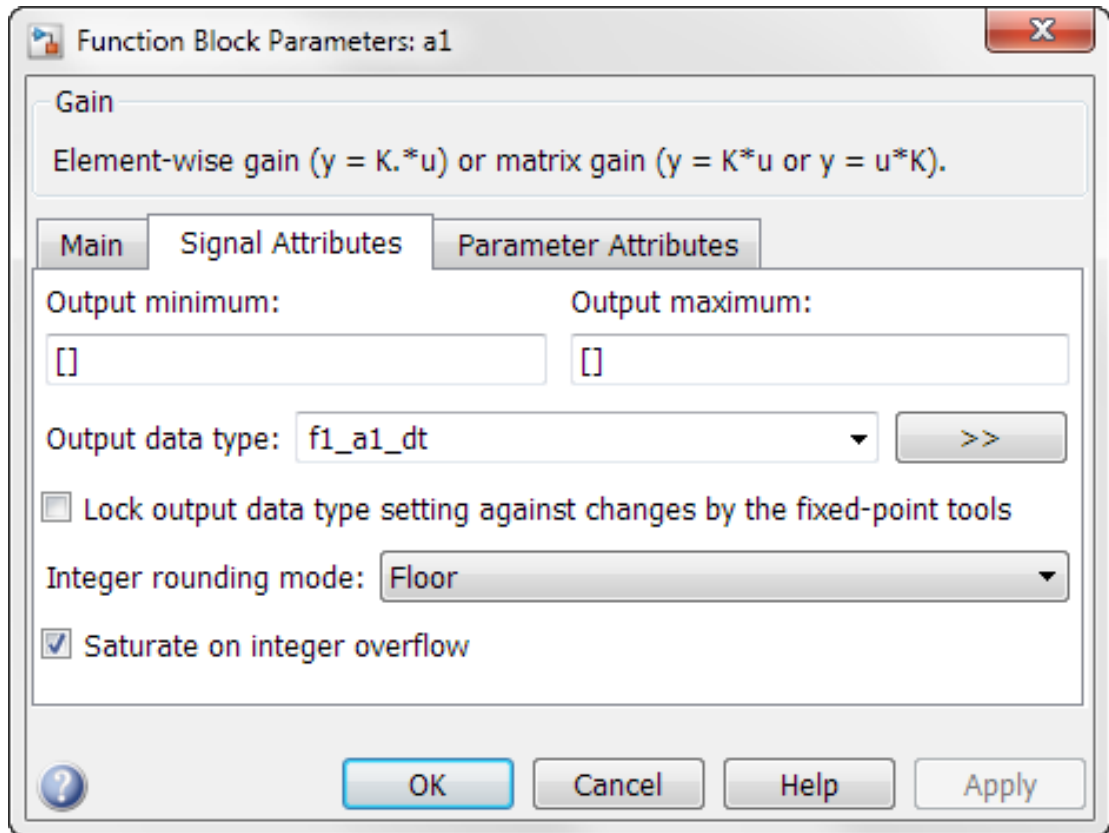
In the Model Explorer, in the **Contents** pane, the **Data Source** column shows the source data dictionary for each Gain block parameter.



- Return to the model. Open the F1 subsystem and double-click the a1 block. The block gain is specified as f1_a1_param, which is defined in the data dictionary.



In the **Signal Attributes** tab, the block output data type is specified as f1_a1_dt. The data type of f1_a1_dt is defined in the referenced data dictionary, f1t_dd.

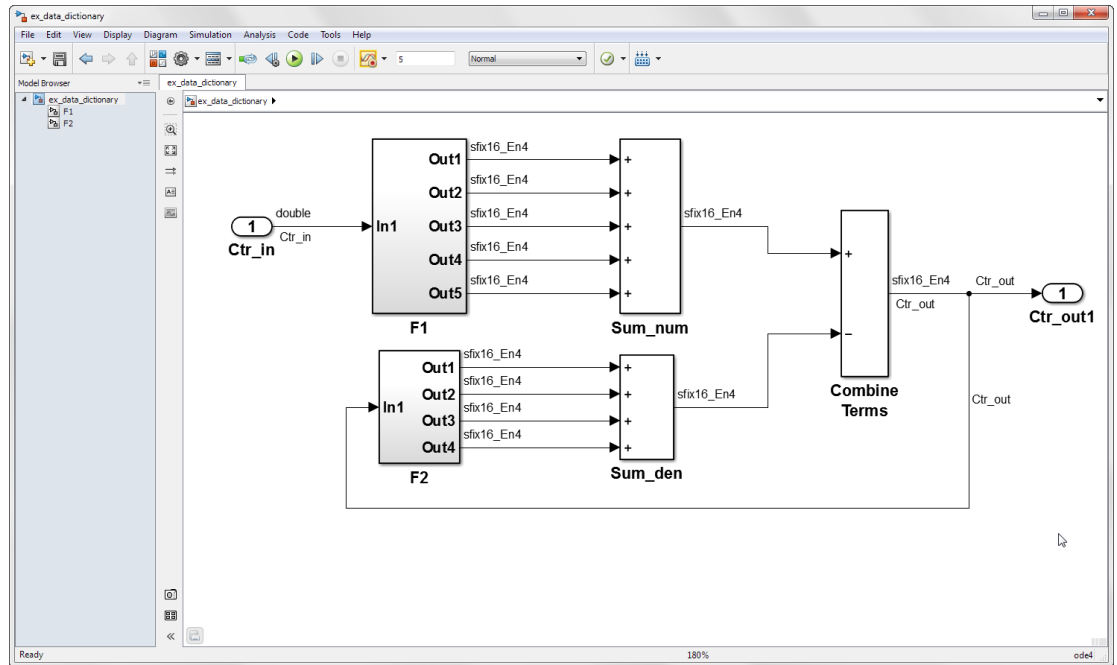


Change Data Types of Model Parameters

The `fix_dd` data dictionary contains the same entries as `flt_dd`, but defines fixed-point data types instead of floating-point data types. To use the fixed-point data types without changing the model, replace `flt_dd` with `fix_dd` as the referenced data dictionary of `mdl_dd`.

- 1 In the Model Explorer, in the **Model Hierarchy** pane, right-click `mdl_dd` and select **Properties**.
- 2 Remove the referenced floating-point data dictionary. In the Data Dictionary dialog box, in the **Referenced Dictionaries** pane, select `flt_dd` and click **Remove**.
- 3 Add a reference to the fixed-point data dictionary. Click **Add Reference** and select `fix_dd`. Click **OK** to close the dialog box.

- 4 In the Model Explorer, right-click mdl_dd and select **Save Changes**.
- 5 Return to the Simulink editor and update the model.



The model now uses fixed-point data types.

Related Examples

- “Migrate Single Model to Use Dictionary” (Simulink)

More About

- “What Is a Data Dictionary?” (Simulink)

Convert Floating-Point Model to Fixed Point

- “Learning Objectives” on page 37-2
- “Model Description” on page 37-4
- “Before You Begin” on page 37-7
- “Convert Floating-Point Model to Fixed Point” on page 37-8

Learning Objectives

In this example, you learn how to:

- Convert a floating-point system to an equivalent fixed-point representation.

This example shows the recommended workflow for conversion when using proposing fraction lengths based on simulation data.

- Use the Fixed-Point Advisor to prepare your model for conversion.

The Fixed-Point Advisor provides a set of tasks to help you convert a floating-point system to fixed point.

You use the Fixed-Point Advisor to:

- Set model-wide configuration options
- Set block-specific dialog parameters
- Check the model against fixed-point guidelines.
- Identify unsupported blocks.
- Remove output data type inheritance from blocks that use floating-point inheritance.
- Promote simulation minimum and maximum values to design minimum and maximum values. This capability is useful if you want to derive ranges for objects in the model and you have not specified design ranges but you have simulated the model with inputs that cover the full intended operating range. For more information, see “Specify block minimum and maximum values” on page 45-27.
- Use the Fixed-Point Tool to propose fixed-point data types.

The Fixed-Point Tool automates the task of specifying fixed-point data types in a system. In this example, the tool collects range data for model objects, either from design minimum and maximum values that you specify explicitly for signals and parameters, or from logged minimum and maximum values that occur during simulation. Based on these values, the tool proposes fixed-point data types that maximize precision and covers the range. The tool allows you to review the data type proposals and then apply them selectively to objects in your model.

- Handle floating-point inheritance blocks during conversion.

For floating-point inheritance blocks when inputs are floating point, all internal and output data types are floating point. The model in this example uses a Discrete Filter block, which is a floating-point inheritance block.

Model Description

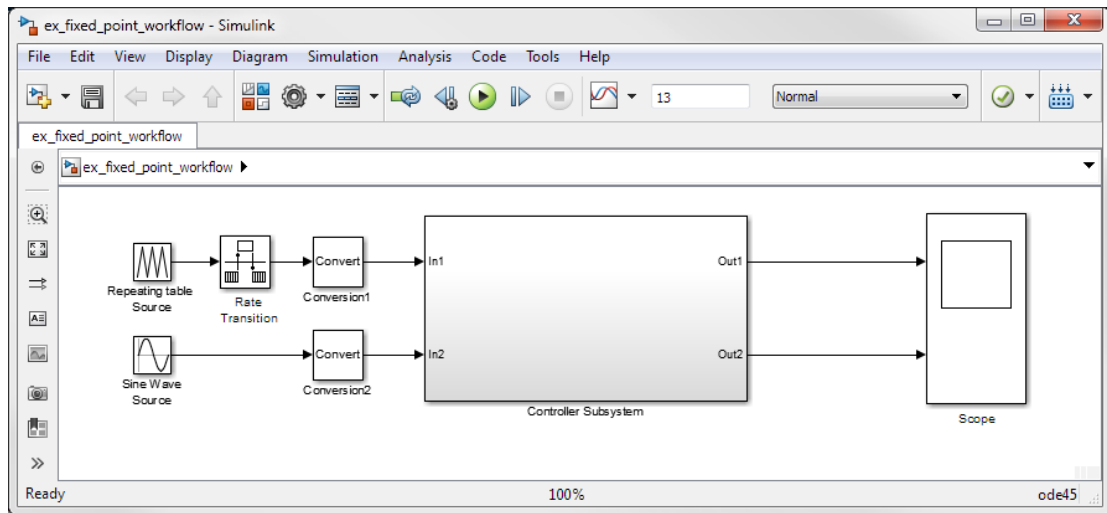
In this section...

“Model Overview” on page 37-4

“Model Set Up” on page 37-5

Model Overview

This example uses the `ex_fixed_point_workflow` model.



The model consists of a Source, a Controller Subsystem that you want to convert to fixed point, and a Scope to visualize the subsystem outputs. This method is how you configure a model to determine the effect of fixed-point data types on a system. Using this approach, you convert only the subsystem because this is the system of interest. There is no need to convert the Source or Scope to fixed point.

This configuration allows you to modify the inputs and collect simulation data for multiple stimuli. You can then examine the behavior of the subsystem with different input ranges and scale your fixed-point data types to provide maximum precision while accommodating the full simulation range.

Model Set Up

The model consists of the following blocks and subsystem.

Source

- **Repeating table Source**

A Repeating Sequence (Repeating Table) block provides the first input to the Controller Subsystem and periodically repeats the sequence of data specified in the mask.

- **Rate Transition**

A Rate Transition block outputs data from the Repeating table Source block at a different rate to the input.

- **Sine Wave Source**

A Sine Wave block provides the second input to the Controller Subsystem.

Initially, the amplitude of the Sine Wave block is 1. Later, you modify the amplitude to change the input range of the system.

- **Conversion1 and Conversion2**

These two Conversion blocks are set up so that the real-world values of their input and output are equal.

Controller Subsystem

The Controller Subsystem consists of:

- **Discrete Filter**

The Discrete Filter block filters the Repeating table Source signal. The Discrete Filter is a floating-point inheritance block. For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point.

- **Chart**

The Chart consists of aStateflow Chart block which converts the Sine Wave input to a positive output and multiplies it by 3.

- **Lookup Table for Chart**

The Lookup Table for Chart block is the first of two identical n-D Lookup Table blocks. This block receives the output from the Chart and, at each breakpoint, outputs the input multiplied by 10.

- **Gain**

The Gain block multiplies the Sine Wave input by -3.

- **Lookup Table for Gain**

The Lookup Table for Gain block is a n-D Lookup Table block. It receives the output from the Gain block and, at each breakpoint, outputs its input multiplied by 10.

- **Sum for Chart**

This Sum block adds the outputs from the Discrete Filter and Lookup Table for Chart blocks and outputs the result to the Scope block.

- **Sum for Gain**

This Sum block adds the outputs from the Discrete Filter and Lookup Table for Gain blocks and outputs the result to the Scope block.

Scope

- **Scope**

The model includes a Scope block that displays the Controller Subsystem output signals.

Before You Begin

This example shows the recommended workflow for converting a floating-point system to fixed point using design and simulation data. It shows you how to use the Fixed-Point Advisor to prepare a floating-point subsystem for conversion to an equivalent fixed-point representation, and then how to use the Fixed-Point Tool to propose the fixed-point data types in the subsystem.

The example uses the following recommended workflow:

- 1 “Prepare Floating-Point Model for Conversion to Fixed Point” on page 37-8.

Step through the Fixed-Point Advisor tasks that prepare the floating-point subsystem for conversion to an equivalent fixed-point representation.

Note: If your model contains referenced models, you must run the Fixed-Point Advisor on each instance of the referenced model as well as the parent model.

- 2 “Propose Data Types” on page 37-13.

Propose data types based on the simulation results. Examine the results to resolve any conflicts and to verify that you want to accept the proposed data type for each result.

- 3 “Apply Fixed-Point Data Types” on page 37-15.

Write the proposed data types to the model. Perform the automatic data typing procedure, which uses the double-precision simulation results to propose fixed-point data types for appropriately configured blocks. The Fixed-Point Tool allows you to accept and apply the proposals selectively.

- 4 “Verify Fixed-Point Settings” on page 37-15.

Simulate the model again using the fixed-point settings. Compare the ideal results for the double-precision run with the fixed-point results.

- 5 Test the fixed-point settings with a different input stimulus and, if necessary, propose new data types to accommodate the simulation range for this input.

Convert Floating-Point Model to Fixed Point

In this section...

“Open the Model” on page 37-8

“Prepare Floating-Point Model for Conversion to Fixed Point” on page 37-8

“Propose Data Types” on page 37-13

“Apply Fixed-Point Data Types” on page 37-15

“Verify Fixed-Point Settings” on page 37-15

“Test Fixed-Point Settings With New Input Data” on page 37-16

“Gather a Floating-Point Benchmark” on page 37-17

“Propose Data Types for the New Input” on page 37-18

“Apply the New Fixed-Point Data Types” on page 37-19

“Verify New Fixed-Point Settings” on page 37-19

“Clean Up Model Settings” on page 37-20

“Key Points to Remember” on page 37-20

“Where to Learn More” on page 37-21

Open the Model

Open the `ex_fixed_point_workflow` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_fixed_point_workflow
```

Prepare Floating-Point Model for Conversion to Fixed Point

The Fixed-Point Advisor provides a set of tasks that help you prepare a floating-point model or subsystem for conversion to an equivalent fixed-point representation. After preparing your model, you use the Fixed-Point Tool to perform the fixed-point conversion.

In this part of the example, you use the Fixed-Point Advisor to prepare the Controller Subsystem in the `ex_fixed_point_workflow` model for conversion.

Open the Fixed-Point Advisor

- 1 In the `ex_fixed_point_workflow` model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.

- 2 In the Fixed-Point Tool:
 - a Under **System Under Design**, click **Change**.
 - b In the System Selector dialog box, select **Controller Subsystem** and click **OK**.
 - c Under **Fixed-point preparation**, click the **Fixed-Point Advisor** button.

You run the Fixed-Point Advisor on the `ex_fixed_point_workflow` Controller Subsystem because this is the system of interest. There is no need to convert the system inputs or the display to fixed point.

Prepare Model for Conversion

- 1 In the Fixed-Point Advisor left pane, expand the **Prepare Model for Conversion** folder to view the tasks. For the purpose of this example, run the tasks in the this folder one at a time. Select **Verify model simulation settings** and, in the right pane, select **Run this task**.

This task validates that model simulation settings allow signal logging and disables data type override in the model and for `fi` objects or embedded numeric data types in your model or workspace. These settings facilitate conversion to fixed point in later tasks.

A waitbar appears while the task runs. When the run is complete, the result shows that the task passed.

- 2 Select and run **Verify update diagram status**.

Verify update diagram status runs. Your model must be able to successfully update diagram to run the checks in the Fixed-Point Advisor.

The task passes.

- 3 Select and run **Address unsupported blocks**. This task identifies blocks that do not support fixed-point data types.

The task passes because the subsystem contains no blocks that do not support fixed-point data.

- 4 Select and run **Set up signal logging**. Prior to simulation, you must specify at least one signal for the Fixed-Point Advisor to use for analysis and comparison in downstream checks. You should log, at minimum, the unique input and output signals.

The task generates a warning because signal logging is not specified for any signals.

5 Fix the warning using the Model Advisor Result Explorer:

- a** Click the **Explore Result** button.

The Model Advisor Result Explorer opens.

- b** In the middle pane, select each signal you want to log and, next to the signal, select the corresponding **EnableLogging** check box.

For this example, log these signals:

- Lookup Table for Gain
- Lookup Table for Chart
- Chart
- Discrete Filter

- c** Close the Model Advisor Result Explorer.

- d** In the Fixed-Point Advisor window, click **Run This Task**.

The task passes because signal logging is now enabled for at least one signal.

6 Select and run **Create simulation reference data**.

The Fixed-Point Advisor simulates the model using the current solver settings, and creates and archives reference signal data in a run named **FPA_Reference** to use for analysis and comparison in later conversion tasks. This task also validates that model simulation settings allow signal logging and that the **Fixed-point instrumentation mode** is set to **Minimums, maximums and overflows**.

The Fixed-Point Advisor issues a warning and provides information in the Analysis Result box that logging simulation minimum and maximum values failed.

Logging failed because the **Fixed-point instrumentation mode** is **Use local settings**, but the recommended setting is **Minimums, maximums and overflows**.

7 To fix the failure, in the **Action** pane, click **Modify All**.

The Fixed-Point Advisor configures the model to the settings recommended in the Analysis **Result** pane. The **Action** pane displays a table of changes showing that

the **Fixed-point instrumentation mode** is now **Minimums**, **maximums** and **overflows**

8 Click **Run This Task**.

Running the task after using the **Modify All** action verifies that you made the necessary changes. The Analysis **Result** pane updates to display a passed result and information about why the task passed.

Tip: You can view the reference run data in the Fixed-Point Tool **Contents** pane in the run named **FPA_Reference** or in the Simulation Data Inspector. Because you ran the simulation twice, the Simulation Data Inspector displays data for both runs using the same name (**FPA_Reference**).

9 In the **Verify Fixed-Point Conversion Guidelines** folder, select and run **Check model configuration data validity diagnostic parameters settings**. This task verifies that the **Model Configuration Parameters > Diagnostics > Data Validity > Parameters** options are all set to **warning**. If these options are set to **error**, the model update diagram action fails in downstream checks.

The task passes because none of these options are set to **error**.

10 Select and run **Implement logic signals as Boolean data**. This task verifies that **Model Configuration Parameters > Optimization > Implement logic signals as Boolean data** is selected. If it is cleared, the code generated in downstream checks is not optimized.

The task passes.

11 Select and run **Simulation range checking**. This task verifies that the **Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** option is not set to **none**.

The task generates a warning because the Simulation range checking option is **none**.

12 To fix the warning, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the Simulation range checking option to **warning**.

13 Rerun the task.

The task now passes because the **Simulation range checking** option is correct.

14 Select and run **Check for implicit signal resolution**. This task checks for models that use implicit signal resolution.

The task fails because implicit signal resolution is enabled.

- 15** To fix the failure, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the **Signal resolution** option to **Explicit only**.

- 16** Rerun the task.

The task now passes.

You have completed all the tasks for the **Prepare Model for Conversion** folder. At this point, you can review the results report found at the folder level, or continue to the next folder.

Prepare for Data Typing and Scaling

This folder contains tasks that set the block configuration options and set output minimum and maximum values for blocks. The block settings from this task simplify the initial data typing and scaling. Later tasks set optimal block configuration. The tasks in this folder prepare the model for automatic data typing in the Fixed-Point Tool.

- 1** For the purpose of this example, run the tasks in the **Prepare for Data Typing and Scaling** folder one at a time.

Open the **Prepare for Data Typing and Scaling** folder then select and run **Review locked data type settings**. This task identifies blocks that have their data type settings locked down which excludes them for automatic data typing.

This task passes because the model contains no blocks with locked data types.

- 2** Select and run **Verify Stateflow charts have strong data typing with Simulink**. This task verifies that the configuration of all Stateflow charts ensures strong data typing with Simulink I/O.

The task passes because the configuration of the Stateflow chart in the subsystem is correct.

- 3** Select and run **Specify block minimum and maximum values**. Ideally, you should specify block output and parameter minimum and maximum values for, at minimum, the Inport blocks in the system. You can specify the minimum and maximum values for any block in this step. Typically, you determine these values during the design process based on the system you are creating.

- 4 The tool advises you to specify minimum and maximum values for all blocks if possible. For the purpose of this example, specify the minimum and maximum values for the Inport blocks in the system.
 - a Click the **Explore Result** button.

The Model Advisor Result Explorer opens, showing that the Inport blocks, In1 and In2, do not have output minimum and maximum values specified.
 - b In the center pane, select In1. This block receives the output from Repeating table Source, which has a minimum value of 10 and a maximum value of 20. Therefore, set **OutMin** to 10 and set **OutMax** to 20 as follows:
 - i In the **OutMin** column for In1, select [] and replace with 10.
 - ii In the **OutMax** column for In1, select [] and replace with 20.
 - c Select In2. This block receives the output from Sine Wave block, which has a minimum value of -1 and a maximum value of 1. Therefore, set **OutMin** to -1 and set **OutMax** to 1.
 - d Close the Model Advisor Result Explorer.
 - e In the Fixed-Point Advisor, rerun the task.

The task passes because you specified minimum and maximum values for all Inport blocks.

You have completed all tasks in the **Prepare for Data Typing and Scaling** folder. At this point, you can review the results report found at the folder level, or continue to the next folder.


Return to Fixed-Point Tool to Perform Data Typing and Scaling

Select and run this task to close the Fixed-Point Advisor and return to the Fixed-Point Tool.

Propose Data Types

Use the Fixed-Point Tool to propose fixed-point data types for appropriately configured blocks based on the double-precision simulation results stored in the simulation reference run that the Fixed-Point Advisor created. These results are stored in the run named **FPA_Reference**. You can view the results in the Fixed-Point Tool **Contents** pane.

The tool proposes fixed-point data types and scaling based on the ranges of the Repeating table Source and Sine Wave inputs. You can then use the tool to accept and apply the proposed data types selectively. In this example, you propose fraction lengths for the specified word lengths.

- 1 In the Fixed-Point Tool, Under **Automatic data typing**, click the **Propose data types** button .

The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.

The Fixed-Point Tool uses the default proposal settings to propose data types with 16-bit word length and best-precision fraction length and updates the results in the **Contents** pane.

- 2 The tool displays the proposed data types in the **ProposedDT** column in the **Contents** pane.

By default, it selects the **Accept** check box for these signals because the proposed data type differs from the object's current data type. If you apply data types, the tool will apply the proposed data types to these signals. For more information, see “Apply Proposed Data Types” on page 39-27.

- 3 Examine the results to resolve any conflicts and to ensure that you want to accept the proposed data type for each result.


In the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

The Fixed-Point Tool detected no conflicts.

Tip: If the tool does detect conflicts, you must resolve these before applying data types. For more information, see “Examine Results to Resolve Conflicts” on page 39-22.

Now that you have reviewed the results and ensured that there are no issues, you are ready to apply the proposed data types to the model, as described in “Apply Fixed-Point Data Types” on page 37-15.

Apply Fixed-Point Data Types

- 1 Click the **Apply accepted data types** button to write the proposed data types to the model. 

The Fixed-Point Tool applies the data type proposals to the subsystem blocks.


- 2 In the Fixed-Point Tool toolbar, select **Show > All results**.

The tool has set all the specified data types to the proposed types.

You are now ready to check that the new data types are acceptable, as described in “Verify Fixed-Point Settings” on page 37-15.

Verify Fixed-Point Settings

Next, you simulate again using the new fixed-point settings. You then use the Fixed-Point Tool plotting capabilities to compare the results from the floating-point **FPA_Reference** run with the fixed-point results.

- 1 Under **Range collection**, set **Run name** to `Initial_fixed_point`. You specify a new run name to prevent the tool from overwriting the results that you want to retain in the **FPA_Reference** run.
- 2 Click the Fixed-Point Tool **Simulate model** button  to run the simulation.

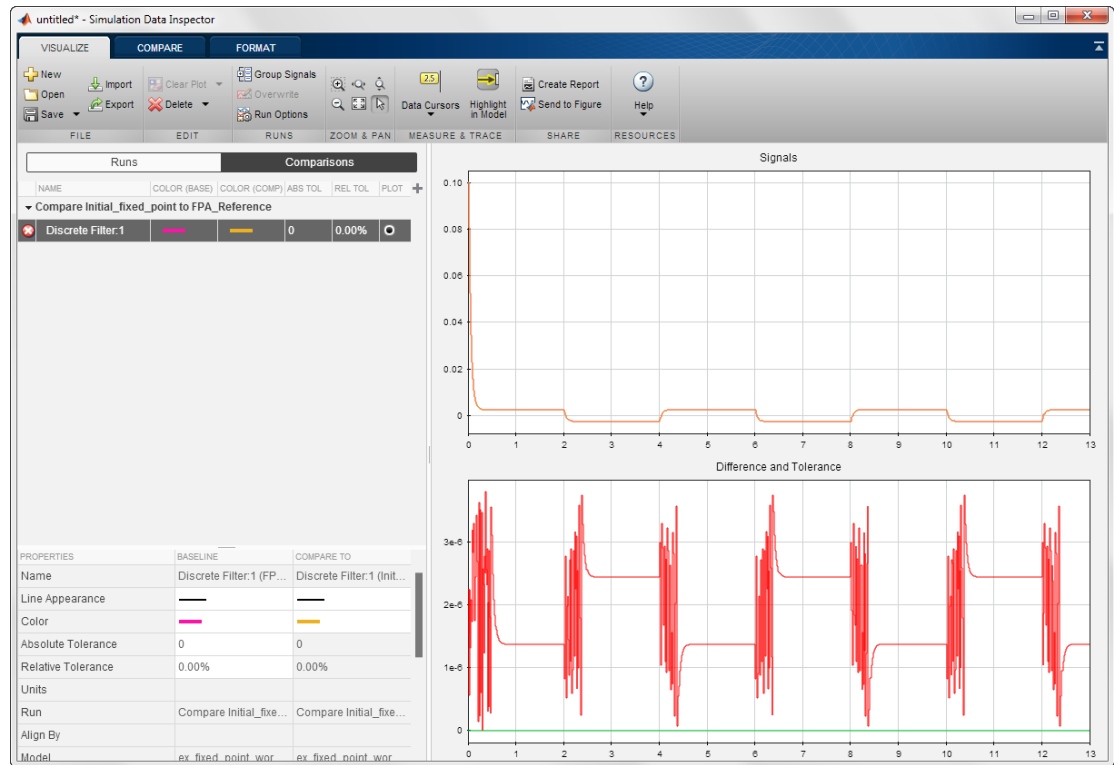
The Simulink software simulates using the new data types that you applied in the previous step. Afterward, the Fixed-Point Tool displays in its **Contents** pane information about blocks that logged fixed-point data. The **CompiledDT** (compiled data type) column for the run shows that the Controller Subsystem blocks use fixed-point data types with the new data types.

Tip: In the **Contents** pane, click the **Run** column heading to sort the runs.

- 3 Examine the results to verify that there are no overflows or saturations.
- 4 In the **Contents** pane, select the **Discrete Filter: Output** that corresponds to the **FPA_Reference** run, and then click the **Compare Signals** button.



The Fixed-Point Tool plots the signal for the FPA_Reference and Initial_fixed_point runs, as well as their difference. The difference plot shows that the floating-point signal and the fixed-point signal are almost identical, the difference is on the order of 10^{-5} .



Now you are ready to test the fixed-point settings with new the input data, as described in “Test Fixed-Point Settings With New Input Data” on page 37-16.

Test Fixed-Point Settings With New Input Data

You have successfully used the Fixed-Point Tool to propose fixed-point data types for your model. In the previous step, you saw that the numerical results for the double-precision system and the fixed-point system are very close. These results indicate that the fixed-point data types are suitable for the range of input data that you used. In

practice, you might need to run multiple simulations to cover the entire design range of your system and use the results of these simulations to refine the fixed-point data types in your model.


In this part of the example, you continue working on the model. First, you modify the range of the Sine Wave input and obtain simulation data based on this new range. Then, you use the Fixed-Point Tool to refine the model fixed-point settings based on the new simulation data. The Fixed-Point Tool proposes new data types that can accommodate the new input range.

To change the range of the input data and test the fixed-point settings:

- 1 In the `ex_fixed_point_workflow` model, double-click the Sine Wave Source block.

The **Source Block Parameters** dialog box opens.

- 2 In this dialog box, change the **Amplitude** to 2 and click **OK**.
- 3 In the Fixed-Point Tool **Range collection** pane, set **Run name** to `Input2`.

- 4 Click the Fixed-Point Tool **Simulate model** button  to run the simulation.

The Simulink software simulates the `ex_fixed_point_workflow` model. The Diagnostic Viewer warns of overflows that occurred during simulation.

- 5 The **Contents** pane displays the simulation results for each block in the subsystem that logged fixed-point data. The tool stores the results in the `Input2` run.

In the `Input2` run, the tool highlights in red the result for the Gain block, indicating that there are issues.

- 6 Examine the result for the Gain block.

The result shows that the Gain block output saturated, which indicates that the fixed-point data settings for this block are not suitable for the new input range.

Next, override the fixed-point data types with doubles and simulate the model again to obtain the ideal behavior of the subsystem, as described in “Gather a Floating-Point Benchmark” on page 37-17.

Gather a Floating-Point Benchmark

Run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This provides a floating-point benchmark that

represents the ideal output. The Simulink software logs the signal logging results to the MATLAB workspace. The Fixed-Point Tool displays the simulation results including minimum and maximum values that occur during the run.

- 1 Under **Configure model settings**, click **Range collection using double override**.

Using this shortcut setting, the Fixed-Point Tool performs a global override of the fixed-point data types and scaling using double-precision numbers, thus avoiding quantization effects.

- 2 Under **Range collection** pane, set **Run name** to `DTO_Input2`.


- 3 Click **Simulate model**  to run the simulation.

The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.

- 4 In the **Contents** pane, click the **Run** column to sort the runs. Verify that there were no overflows or saturations in the **DTO_Input2** run.

Propose Data Types for the New Input

Now, use the Fixed-Point Tool to propose fixed-point data types based on the double-precision simulation results for the new input stored in the **DTO_Input2** run.

- 1 In the Fixed-Point Tool, click **Propose data types** .
- 2 In the **Propose Data Types** dialog box, select `DTO_Input2` as the run to use for proposing data types, and then click **OK**.

The Fixed-Point Tool proposes new data types for all objects in the model and updates the results in the **Contents** pane.

- 3 The tool displays the proposed data types in the **ProposedDT** column in the **Contents** pane.


To accommodate the full simulation range, the Fixed-Point Tool proposes new data types with reduced precision for the Gain block output.

- 4 Examine the results to resolve any conflicts and to ensure that you want to accept the proposed data type for each result.

In the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

The Fixed-Point Tool detected no conflicts, so you are ready to apply the new data types as described in “Apply the New Fixed-Point Data Types” on page 37-19.

Apply the New Fixed-Point Data Types

- 1 Click **Apply accepted data types**  to write the proposed data types to the model.
- 2 In the **Apply Data Types** dialog box, select `DT0_Input2` as the run to use for applying proposed data types and then click **OK**.
- 3 In the Fixed-Point Tool toolbar, select **Show > All results**.

The tool has set all the specified data types to the proposed types.

Verify New Fixed-Point Settings

Finally, you simulate again using the new fixed-point settings. You then use the Fixed-Point Tool plotting capabilities to compare the results for the initial and final fixed-point settings.

- 1 Under **Configure model settings**, click **Range collection with specified data types** to remove the data type override on the model.
- 2 Under **Range collection**, set **Run name** to `Final_fixed_point`.
- 3 Click **Simulate model** to run the simulation.

The Simulink software simulates using the new data types that you applied in the previous step and stores the results in the `Final_fixed_point` run.

- 4 Examine the results to verify that there are no overflows or saturations.
- 5 In the **Contents** pane, select the `Discrete Filter: Output` that corresponds to the `Initial_fixed_point` run, and then click the **Compare Signals** button.



- 6 In the **Compare Runs Selector** dialog box, select `Final_fixed_point`, and then click **OK**.

The Fixed-Point Tool plots the signal for both runs, as well as their difference. The difference plot shows that the floating-point signal and the fixed-point signal are identical.

- 7 Optionally, you can zoom in to view the steady-state region with greater detail. From the **Tools** menu of the figure window, select **Zoom In** and then drag the pointer to draw a box around the area you want to view more closely.

Clean Up Model Settings

When you are satisfied with the results of the fixed-point conversion, under **Configure model settings**, click the **Remove overrides and disable range collection** button. This configuration cleans up settings after conversion to restore maximum simulation speed of your model. Close the Fixed-Point Tool.

Key Points to Remember

- Convert subsystems within your model, rather than the entire model. This practice saves time and avoids unnecessary conversions.
- Use the Fixed-Point Advisor to prepare your model for conversion to fixed point.
- Use the Fixed-Point Tool to propose fixed-point data types for your model or subsystem.
- When using the Fixed-Point Advisor, consider saving a restore point before applying recommendations.

A restore point provides a fallback in case the recommended data types causes subsequent update diagram failure. If you do not save a restore point and you encounter an update diagram failure, you must start the conversion from the beginning.

- Provide as much design minimum and maximum information as possible before starting the conversion to fixed point.

Providing this information enables the fixed-point tools to choose fixed-point data types that maximize precision and cover the range.

Specify minimum and maximum values for signals and parameters in the model for:

- Inport and Outport blocks

- Block outputs
- The interface between MATLAB Function and C Chart blocks and the Simulink model to ensure strong data typing
- Simulink.Signal objects
- Ensure that you simulate the system using the full range of inputs.

If you use simulation minimum and maximum values to scale fixed-point data types, the tools provide meaningful results when exercising the full range of values over which your design is meant to run.

Where to Learn More

To learn more about...	See...
Fixed-Point Advisor capabilities	“Preparation for Fixed-Point Conversion” on page 35-2
Best practices for using the Fixed-Point Advisor	“Best Practices” on page 35-2
Using restore points in the Fixed-Point Advisor	“Restore Points” on page 35-7
Fixed-Point Tool capabilities	“Fixed-Point Tool” on page 36-2 fxptdlg
Best practices for using the Fixed-Point Tool	“Best Practices for Fixed-Point Workflow” on page 39-7
Using the Fixed-Point Tool to merge multiple simulation results	“Propose Data Types Using Multiple Simulations” on page 39-58
Optimize model settings for code generation	“Optimize Generated Code with the Model Advisor” on page 44-28

Producing Lookup Table Data

- “Producing Lookup Table Data” on page 38-2
- “Worst-Case Error for a Lookup Table” on page 38-3
- “Create Lookup Tables for a Sine Function” on page 38-5
- “Use Lookup Table Approximation Functions” on page 38-20
- “Effects of Spacing on Speed, Error, and Memory Usage” on page 38-21

Producing Lookup Table Data

A function lookup table is a method by which you can approximate a function by a table with a finite number of points (X,Y). Function lookup tables are essential to many fixed-point applications. The function you want to approximate is called the *ideal function*. The X values of the lookup table are called the *breakpoints*. You approximate the value of the ideal function at a point by linearly interpolating between the two breakpoints closest to the point.

In creating the points for a function lookup table, you generally want to achieve one or both of the following goals:

- Minimize the worst-case error for a specified maximum number of breakpoints
- Minimize the number of breakpoints for a specified maximum allowed error

“Create Lookup Tables for a Sine Function” on page 38-5 shows you how to create function lookup tables using the function `fixpt_look1_func_approx`. You can optimize the lookup table to minimize the number of data points, the error, or both. You can also restrict the spacing of the breakpoints to be even or even powers of two to speed up computations using the table.

“Worst-Case Error for a Lookup Table” on page 38-3 explains how to use the function `fixpt_look1_func_plot` to find the worst-case error of a lookup table and plot the errors at all points.

Worst-Case Error for a Lookup Table

The error at any point of a function lookup table is the absolute value of the difference between the ideal function at the point and the corresponding Y value found by linearly interpolating between the adjacent breakpoints. The *worst-case error*, or *maximum absolute error*, of a lookup table is the maximum absolute value of all errors in the interval containing the breakpoints.

For example, if the ideal function is the square root, and the breakpoints of the lookup table are 0, 0.25, and 1, then in a perfect implementation of the lookup table, the worst-case error is $1/8 = 0.125$, which occurs at the point $1/16 = 0.0625$. In practice, the error could be greater, depending on the fixed-point quantization and other factors.

The section that follows shows how to use the function `fixpt_look1_func_plot` to find the worst-case error of a lookup table for the square root function.

Approximate the Square Root Function

This example shows how to use the function `fixpt_look1_func_plot` to find the maximum absolute error for the simple lookup table whose breakpoints are 0, 0.25, and 1. The corresponding Y data points of the lookup table, which you find by taking the square roots of the breakpoints, are 0, 0.5, and 1.

To use the function `fixpt_look1_func_plot`, you need to define its parameters first. To do so, type the following at the MATLAB prompt:

```
funcstr = 'sqrt(x)'; % Define the square root function
xdata = [0;.25;1]; % Set the breakpoints
ydata = sqrt(xdata); % Find the square root of the breakpoints
xmin = 0; % Set the minimum breakpoint
xmax = 1; % Set the maximum breakpoint
xdt = ufix(16); % Set the x data type
xscale = 2^-16; % Set the x data scaling
ydt = sfix(16); % Set the y data type
yscale = 2^-14; % Set the y data scaling
rndmeth = 'Floor'; % Set the rounding method
```

To get the worst-case error of the lookup table and a plot of the error, type:

```
errworst = fixpt_look1_func_plot(xdata,ydata,funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

```
errworst =
    0.1250
```

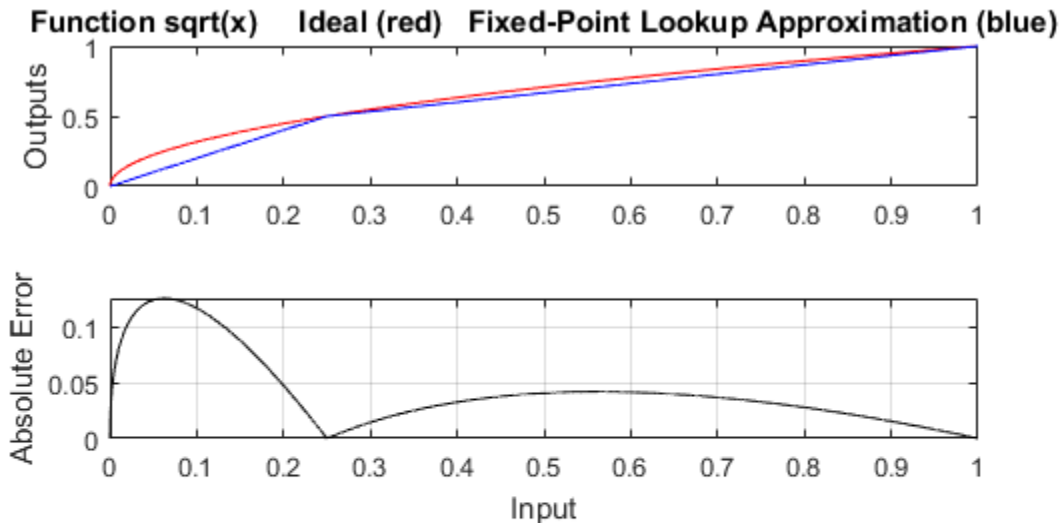


Table uses 3 unevenly spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.12503 $\log_2(\text{MAE}) = -2.9996$ MAE/yBit = 2048.5

The least significant 12 bits of the output can be inaccurate.

The most significant nonsign bit of the output is used.

The remaining 3 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

The upper box (Outputs) displays a plot of the square root function with a plot of the fixed-point lookup approximation underneath. The approximation is found by linear interpolation between the breakpoints. The lower box (Absolute Error) displays the errors at all points in the interval from 0 to 1. Notice that the maximum absolute error occurs at 0.0625. The error at the breakpoints is 0.

Create Lookup Tables for a Sine Function

In this section...

“Introduction” on page 38-5

“Parameters for `fixpt_look1_func_approx`” on page 38-5

“Setting Function Parameters for the Lookup Table” on page 38-7

“Using `errmax` with Unrestricted Spacing” on page 38-7

“Using `nptsmax` with Unrestricted Spacing” on page 38-10

“Using `errmax` with Even Spacing” on page 38-12

“Using `nptsmax` with Even Spacing” on page 38-13

“Using `errmax` with Power of Two Spacing” on page 38-14

“Using `nptsmax` with Power of Two Spacing” on page 38-16

“Specifying Both `errmax` and `nptsmax`” on page 38-17

“Comparison of Example Results” on page 38-18

Introduction

The sections that follow explain how to use the function `fixpt_look1_func_approx` to create lookup tables. It gives examples that show how to create lookup tables for the function $\sin(2\pi x)$ on the interval from 0 to 0.25.

Parameters for `fixpt_look1_func_approx`

To use the function `fixpt_look1_func_approx`, you must first define its parameters. The required parameters for the function are

- `funcstr` — Ideal function
- `xmin` — Minimum input of interest
- `xmax` — Maximum input of interest
- `xdt` — x data type
- `xscale` — x data scaling
- `ydt` — y data type

- `yscale` — y data scaling
- `rndmeth` — Rounding method

In addition there are three optional parameters:

- `errmax` — Maximum allowed error of the lookup table
- `nptsmax` — Maximum number of points of the lookup table
- `spacing` — Spacing allowed between breakpoints

You must use at least one of the parameters `errmax` and `nptsmax`. The next section, “Setting Function Parameters for the Lookup Table” on page 38-7, gives typical settings for these parameters.

Using Only `errmax`

If you use only the `errmax` parameter, without `nptsmax`, the function creates a lookup table with the fewest points, for which the worst-case error is at most `errmax`. See “Using `errmax` with Unrestricted Spacing” on page 38-7.

Using Only `nptsmax`

If you use only the `nptsmax` parameter without `errmax`, the function creates a lookup table with at most `nptsmax` points, which has the smallest worst case error. See “Using `nptsmax` with Unrestricted Spacing” on page 38-10.

The section “Specifying Both `errmax` and `nptsmax`” on page 38-17 describes how the function behaves when you specify both `errmax` and `nptsmax`.

Spacing

You can use the optional `spacing` parameter to restrict the spacing between breakpoints of the lookup table. The options are

- `'unrestricted'` — Default.
- `'even'` — Distance between any two adjacent breakpoints is the same.
- `'pow2'` — Distance between any two adjacent breakpoints is the same and the distance is a power of two.

The section “Restricting the Spacing” on page 38-11 and the examples that follow it explain how to use the `spacing` parameter.

Setting Function Parameters for the Lookup Table

To do the examples in this section, you must first set parameter values for the `fixpt_look1_func_approx` function. To do so, type the following at the MATLAB prompt:

```
funcstr = 'sin(2*pi*x)'; % Define the sine function
xmin = 0; % Set the minimum input of interest
xmax = 0.25; % Set the maximum input of interest
xdt = ufix(16); % Set the x data type
xscale = 2^-16; % Set the x data scaling
ydt = sfix(16); % Set the y data type
yscale = 2^-14; % Set the y data scaling
rndmeth = 'Floor'; % Set the rounding method
errmax = 2^-10; % Set the maximum allowed error
nptsmax = 21; % Specify the maximum number of points
```

If you exit the MATLAB software after typing these commands, you must retype them before trying any of the other examples in this section.

Using `errmax` with Unrestricted Spacing

The first example shows how to create a lookup table that has the fewest data points for a specified worst-case error, with unrestricted spacing. Before trying the example, enter the same parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session.

You specify the maximum allowed error by typing

```
errmax = 2^-10;
```

Creating the Lookup Table

To create the lookup table, type

```
[xdata, ydata, errworst] = fixpt_look1_func_approx(funcstr, ...
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth, errmax, []);
```

Note that the `nptsmax` and `spacing` parameters are not specified.

The function returns three variables:

- `xdata`, the vector of breakpoints of the lookup table
- `ydata`, the vector found by applying ideal function $\sin(2\pi x)$ to `xdata`
- `errworst`, which specifies the maximum possible error in the lookup table

The value of `errworst` is less than or equal to the value of `errmax`.

You can find the number of X data points by typing

```
length(xdata)
```

```
ans =
```

```
16
```

This means that 16 points are required to approximate $\sin(2\pi x)$ to within the tolerance specified by `errmax`.

You can display the maximum error by typing `errworst`. This returns

```
errworst
```

```
errworst =
```

```
9.7656e-04
```

Plotting the Results

You can plot the output of the function `fixpt_look1_func_plot` by typing

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...  
xscale,ydt,yscale,rndmeth);
```

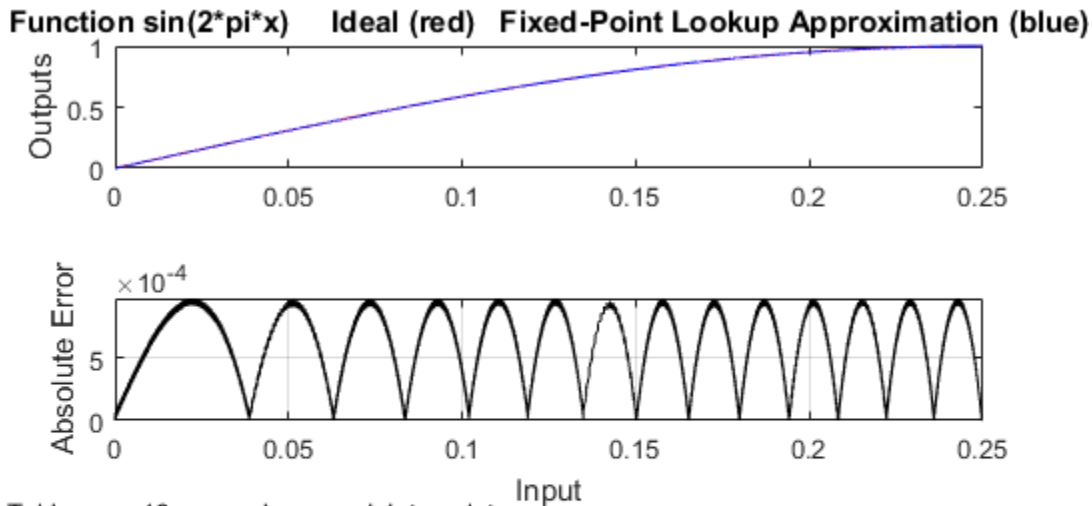


Table uses 16 unevenly spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00097656 $\log_2(\text{MAE}) = -10$ MAE/yBit = 16

The least significant 4 bits of the output can be inaccurate.

The most significant nonsign bit of the output is used.

The remaining 11 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

The upper plot shows the ideal function $\sin(2\pi x)$ and the fixed-point lookup approximation between the breakpoints. In this example, the ideal function and the approximation are so close together that the two graphs appear to coincide. The lower plot displays the errors.

In this example, the Y data points, returned by the function `fixpt_look1_func_approx` as `ydata`, are equal to the ideal function applied to the points in `xdata`. However, you can define a different set of values for `ydata` after running `fixpt_look1_func_plot`. This can sometimes reduce the maximum error.

You can also change the values of `xmin` and `xmax` in order to evaluate the lookup table on a subset of the original interval.

To find the new maximum error after changing `ydata`, `xmin` or `xmax`, type

```
errworst = fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax, ...  
xdt,xscale,ydt,yscale,rndmeth)
```

Using `nptsmax` with Unrestricted Spacing

The next example shows how to create a lookup table that minimizes the worst-case error for a specified maximum number of data points, with unrestricted spacing. Before starting the example, enter the same parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session.

Setting the Number of Breakpoints

You specify the number of breakpoints in the lookup table by typing

```
nptsmax = 21;
```

Creating the Lookup Table

Next, type

```
[xdata, ydata, errworst] = fixpt_look1_func_approx(funcstr, ...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax);
```

The empty brackets, `[]`, tell the function to ignore the parameter `errmax`, which is not used in this example. Omitting `errmax` causes the function `fixpt_look1_func_approx` to return a lookup table of size specified by `nptsmax`, with the smallest worst-case error.

The function returns a vector `xdata` with 21 points. You can find the maximum error for this set of points by typing `errworst` at the MATLAB prompt.

```
errworst
```

```
errworst =  
  
    5.1139e-04
```

Plotting the Results

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rdmeth);
```

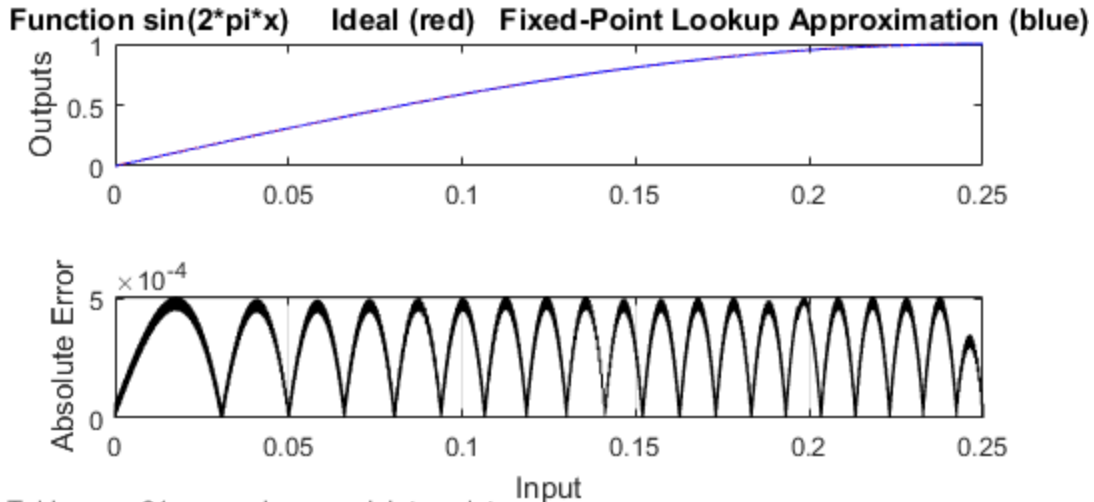


Table uses 21 unevenly spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00051139 $\log_2(\text{MAE}) = -10.9333$ MAE/yBit = 8.3785

The least significant 4 bits of the output can be inaccurate.

The most significant nonsign bit of the output is used.

The remaining 11 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

Restricting the Spacing

In the previous two examples, the function `fixpt_look1_func_approx` creates lookup tables with unrestricted spacing between the breakpoints. You can restrict the spacing to improve the computational efficiency of the lookup table, using the spacing parameter.

The options for spacing are

- 'unrestricted' — Default.
- 'even' — Distance between any two adjacent breakpoints is the same.

- 'pow2' — Distance between any two adjacent breakpoints is the same and is a power of two.

Both power of two and even spacing increase the computational speed of the lookup table and use less command read-only memory (ROM). However, specifying either of the spacing restrictions along with `errmax` usually requires more data points in the lookup table than does unrestricted spacing to achieve the same degree of accuracy. The section “Effects of Spacing on Speed, Error, and Memory Usage” on page 38-21 discusses the tradeoffs between different spacing options.

Using `errmax` with Even Spacing

The next example shows how to create a lookup table that has evenly spaced breakpoints and a specified worst-case error. To try the example, you must first enter the parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'even';  
[xdata, ydata, errworst] = fixpt_look1_func_approx(funcstr, ...  
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth, errmax, [], spacing);
```

You can find the number of points in the lookup table by typing:

```
length(xdata)
```

```
ans =
```

```
20
```

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata, ydata, funcstr, xmin, xmax, xdt, ...  
xscale, ydt, yscale, rndmeth);
```

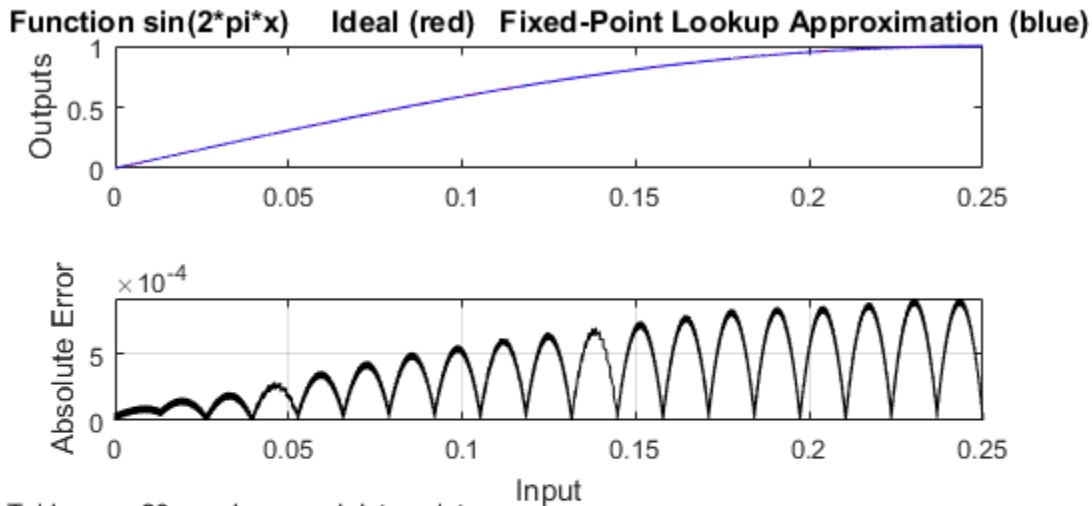



Table uses 20 evenly spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00092109 $\log_2(\text{MAE}) = -10.0844$ MAE/yBit = 15.0912

The least significant 4 bits of the output can be inaccurate.

The most significant nonsign bit of the output is not used.

The remaining 10 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

Using nptsmax with Even Spacing

The next example shows how to create a lookup table that has evenly spaced breakpoints and minimizes the worst-case error for a specified maximum number of points. To try the example, you must first enter the parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'even';
[xdata, ydata, errworst] = fixpt_look1_func_approx(funcstr, ...
xmin, xmax, xdt, xscale, ydt, yscale, rndmeth, [], nptsmax, spacing);
```

The result requires 21 evenly spaced points to achieve a maximum absolute error of $2^{-10.2209}$.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

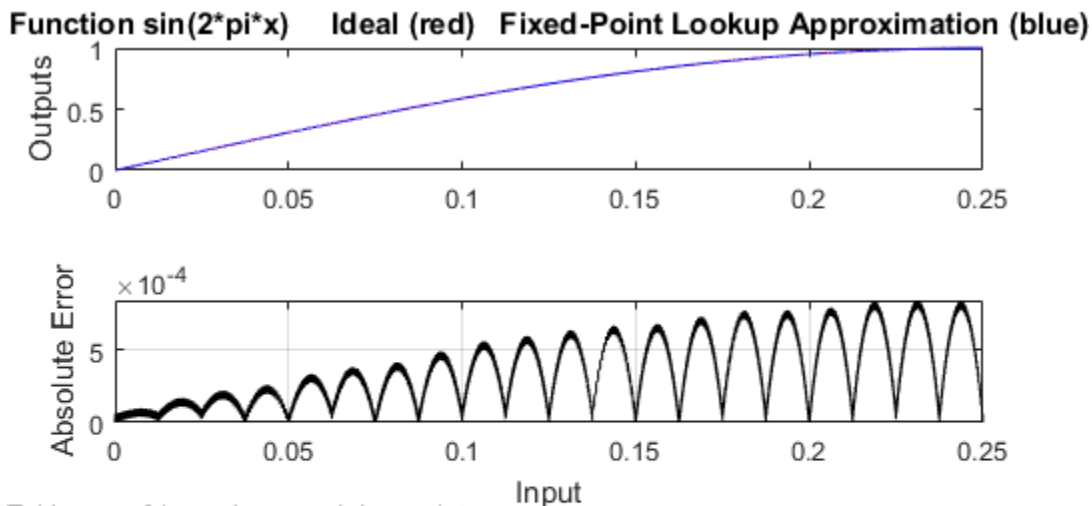


Table uses 21 evenly spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00083793 $\log_2(\text{MAE}) = -10.2209$ MAE/yBit = 13.7287

The least significant 4 bits of the output can be inaccurate.

The most significant nonsign bit of the output is not used.

The remaining 10 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

Using `errmax` with Power of Two Spacing

The next example shows how to construct a lookup table that has power of two spacing and a specified worst-case error. To try the example, you must first enter the parameter

values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'pow2';  
[xdata, ydata, errworst] = ...  
fixpt_look1_func_approx(funcstr, xmin, ...  
xmax, xdt, xscale, ydt, yscale, rndmeth, errmax, [], spacing);
```

To find out how many points are in the lookup table, type

```
length(xdata)
```

```
ans =
```

```
33
```

This means that 33 points are required to achieve the worst-case error specified by `errmax`. To verify that these points are evenly spaced, type

```
widths = diff(xdata)
```

This generates a vector whose entries are the differences between consecutive points in `xdata`. Every entry of `widths` is 2^{-7} .

To find the maximum error for the lookup table, type

```
errworst
```

```
errworst =
```

```
3.7209e-04
```

This is less than the value of `errmax`.

To plot the lookup table data along with the errors, type

```
fixpt_look1_func_plot(xdata, ydata, funcstr, xmin, xmax, xdt, ...
```

```
xscale,ydt,yscale,rndmeth);
```

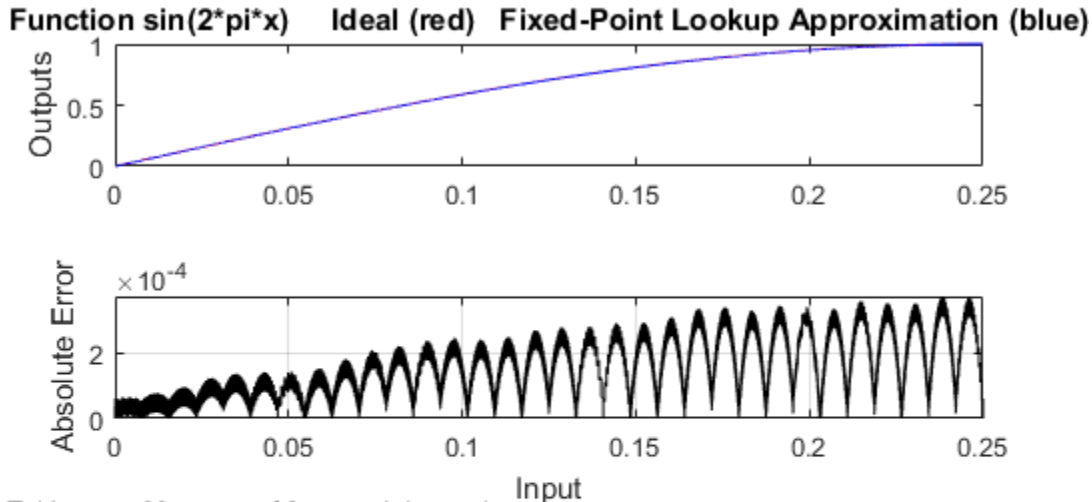


Table uses 33 power of 2 spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00037209 $\log_2(\text{MAE}) = -11.3921$ MAE/yBit = 6.0964

The least significant 3 bits of the output can be inaccurate.

The most significant nonsign bit of the output is used.

The remaining 12 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

Using nptsmx with Power of Two Spacing

The next example shows how to create a lookup table that has power of two spacing and minimizes the worst-case error for a specified maximum number of points. To try the example, you must first enter the parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 38-7, if you have not already done so in this MATLAB session:

```
spacing = 'pow2';
[xdata, ydata, errworst] = ...
fixpt_look1_func_approx(funcstr,xmin, ...
```

```
xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

The result requires 17 points to achieve a maximum absolute error of $2^{-9.6267}$.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

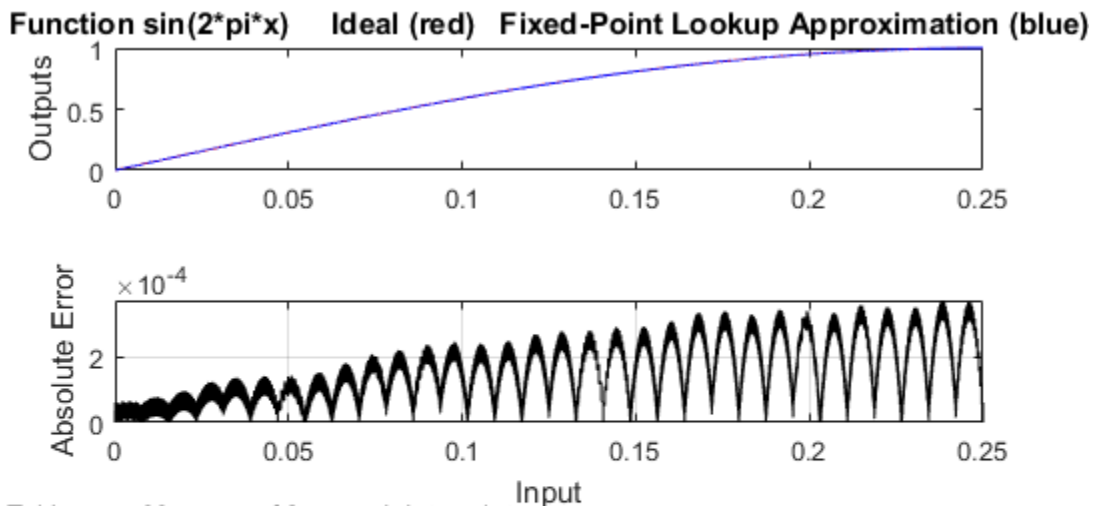


Table uses 33 power of 2 spaced data points.

The input is Unsigned 16 Bit with 16 bits right of binary point

The output is Signed 16 Bit with 14 bits right of binary point

Maximum Absolute Error 0.00037209 $\log_2(\text{MAE}) = -11.3921$ MAE/yBit = 6.0964

The least significant 3 bits of the output can be inaccurate.

The most significant nonsign bit of the output is used.

The remaining 12 nonsign bits of the output are used and always accurate.

The sign bit of the output is not used.

The rounding mode is to Floor

Specifying Both errmax and nptsmax

If you include both the `errmax` and the `nptsmax` parameters, the function `fixpt_look1_func_approx` tries to find a lookup table with at most `nptsmax` data

points, whose worst-case error is at most `errmax`. If it can find a lookup table meeting both conditions, it uses the following order of priority for spacing:

- 1 Power of two
- 2 Even
- 3 Unrestricted

If the function cannot find any lookup table satisfying both conditions, it ignores `nptsmax` and returns a lookup table with unrestricted spacing, whose worst-case error is at most `errmax`. In this case, the function behaves the same as if the `nptsmax` parameter were omitted.

Using the parameters described in the section “Setting Function Parameters for the Lookup Table” on page 38-7, the following examples illustrate the results of using different values for `nptsmax` when you enter

```
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax);
```

The results for three different settings for `nptsmax` are as follows:

- `nptsmax = 33`; — The function creates the lookup table with 33 points having power of two spacing, as in Example 3.
- `nptsmax = 21`; — Because the `errmax` and `nptsmax` conditions cannot be met with power of two spacing, the function creates the lookup table with 20 points having even spacing, as in Example 5.
- `nptsmax = 16`; — Because the `errmax` and `nptsmax` conditions cannot be met with either power of two or even spacing, the function creates the lookup table with 16 points having unrestricted spacing, as in Example 1.

Comparison of Example Results

The following table summarizes the results for the examples. Note that when you specify `errmax`, even spacing requires more data points than unrestricted, and power of two spacing requires more points than even spacing.

Example	Options	Spacing	Worst-Case Error	Number of Points in Table
1	<code>errmax=2⁻¹⁰</code>	'unrestricted'	<code>2⁻¹⁰</code>	16

Example	Options	Spacing	Worst-Case Error	Number of Points in Table
2	nptsmax=21	'unrestricted'	$2^{-10.933}$	21
3	errmax= 2^{-10}	'even'	$2^{-10.0844}$	20
4	nptsmax=21	'even'	$2^{-10.2209}$	21
5	errmax= 2^{-10}	'pow2'	$2^{-11.3921}$	33
6	nptsmax=21	'pow2'	$2^{-9.627}$	17

Use Lookup Table Approximation Functions

The following steps summarize how to use the lookup table approximation functions:

- 1 Define:
 - a The ideal function to approximate
 - b The range, `xmin` to `xmax`, over which to find X and Y data
 - c The fixed-point implementation: data type, scaling, and rounding method
 - d The maximum acceptable error, the maximum number of points, and the spacing
- 2 Run the `fixpt_look1_func_approx` function to generate X and Y data.
- 3 Use the `fixpt_look1_func_plot` function to plot the function and error between the ideal and approximated functions using the selected X and Y data, and to calculate the error and the number of points used.
- 4 Vary input criteria, such as `errmax`, `nptsmax`, and `spacing`, to produce sets of X and Y data that generate functions with varying worst-case error, number of points required, and spacing.
- 5 Compare results of the number of points required and maximum absolute error from various runs to choose the best set of X and Y data.

Effects of Spacing on Speed, Error, and Memory Usage

In this section...

“Criteria for Comparing Types of Breakpoint Spacing” on page 38-21

“Model That Illustrates Effects of Breakpoint Spacing” on page 38-21

“Data ROM Required for Each Lookup Table” on page 38-22

“Determination of Out-of-Range Inputs” on page 38-22

“How the Lookup Tables Determine Input Location” on page 38-23

“Interpolation for Each Lookup Table” on page 38-25

“Summary of the Effects of Breakpoint Spacing” on page 38-27

Criteria for Comparing Types of Breakpoint Spacing

The sections that follow compare implementations of lookup tables that use breakpoints whose spacing is uneven, even, and power of two. The comparison focuses on:

- Execution speed of commands
- Rounding error during interpolation
- The amount of read-only memory (ROM) for data
- The amount of ROM for commands

This comparison is valid only when the breakpoints are not tunable. If the breakpoints are tunable in the generated code, all three cases generate the same code. For a summary of the effects of breakpoint spacing on execution speed, error, and memory usage, see “Summary of the Effects of Breakpoint Spacing” on page 38-27.

Model That Illustrates Effects of Breakpoint Spacing

This comparison uses the model `fxpdemo_approx_sin`. Three fixed-point lookup tables appear in this model. All three tables approximate the function $\sin(2\pi u)$ over the first quadrant and achieve a worst-case error of less than 2^{-8} . However, they have different restrictions on their breakpoint spacing.

You can use the model `fxpdemo_approx`, which `fxpdemo_approx_sin` opens, to generate Simulink Coder code (Simulink Coder software license required). The sections that follow present several segments of generated code to emphasize key differences.

To open the model, type at the MATLAB prompt:

```
fxpdemo_approx_sin
```

Data ROM Required for Each Lookup Table

This section looks at the data ROM required by each of the three spacing options.

Uneven Case

Uneven spacing requires both Y data points and breakpoints:

```
int16_T yuneven[8];  
uint16_T xuneven[8];
```

The total bytes used is 32.

Even Case

Even spacing requires only Y data points:

```
int16_T yeven[10];
```

The total bytes used is 20. The breakpoints are not explicitly required. The code uses the spacing between the breakpoints, and might use the smallest and largest breakpoints. At most, three values related to the breakpoints are necessary.

Power of Two Case

Power of two spacing requires only Y data points:

```
int16_T ypow2[17];
```

The total bytes used is 34. The breakpoints are not explicitly required. The code uses the spacing between the breakpoints, and might use the smallest and largest breakpoints. At most, three values related to the breakpoints are necessary.

Determination of Out-of-Range Inputs

In all three cases, you must guard against the chance that the input is less than the smallest breakpoint or greater than the biggest breakpoint. There can be differences in how occurrences of these possibilities are handled. However, the differences are generally minor and are normally not a key factor in deciding to use one spacing method over

another. The subsequent sections assume that out-of-range inputs are impossible or have already been handled.

How the Lookup Tables Determine Input Location

This section describes how the three fixed-point lookup tables determine where the current input is relative to the breakpoints.

Uneven Case

Unevenly-spaced breakpoints require a general-purpose algorithm such as a binary search to determine where the input lies in relation to the breakpoints. The following code provides an example:

```
iLeft = 0;
iRight = 7; /* number of breakpoints minus 1 */

while ( ( iRight - iLeft ) > 1 )
{
    i = ( iLeft + iRight ) >> 1;

    if ( uAngle < xuneven[i] )
    {
        iRight = i;
    }
    else
    {
        iLeft = i;
    }
}
```

The while loop executes up to $\log_2(N)$ times, where N is the number of breakpoints.

Even Case

Evenly-spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints:

```
iLeft = uAngle / 455U;
```

The divisor `455U` represents the spacing between breakpoints. In general, the dividend would be `(uAngle - SmallestBreakPoint)`. In this example, the smallest breakpoint is zero, so the code optimizes out the subtraction.

Power of Two Case

Power of two spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints:

```
iLeft = uAngle >> 8;
```

The number of shifts is 8 because the breakpoints have spacing 2^8 . The smallest breakpoint is zero, so `uAngle` replaces the general case of `(uAngle - SmallestBreakPoint)`.

Comparison

To determine where the input lies with respect to the breakpoints, the unevenly-spaced case requires much more code than the other two cases. This code requires additional command ROM. If many lookup tables share the binary search algorithm as a function, you can reduce this ROM penalty. Even if the code is shared, the number of clock cycles required to determine the location of the input is much higher for the unevenly-spaced cases than the other two cases. If the code is shared, function-call overhead decreases the speed of execution a little more.

In the evenly-spaced case and the power of two spaced case, you can determine the location of the input with a single line of code. The evenly-spaced case uses a general integer division. The power of two case uses a shift instead of general division because the divisor is an exact power of two. Without knowing the specific processor, you cannot be certain that a shift is better than division.

Many processors can implement division with a single assembly language instruction, so the code will be small. However, this instruction often takes many clock cycles to complete. Many processors do not provide a division instruction. Division on these processors occurs through repeated subtractions. This process is slow and requires a lot of machine code, but this code can be shared.

Most processors provide a way to do logical and arithmetic shifts left and right. A key difference is whether the processor can do N shifts in one instruction (barrel shift) or requires N instructions that shift one bit at a time. The barrel shift requires less code. Whether the barrel shift also increases speed depends on the hardware that supports the operation.

The compiler can also complicate the comparison. In the previous example, the command `uAngle >> 8` essentially takes the upper 8 bits in a 16-bit word. The compiler can detect

this situation and replace the bit shifts with an instruction that takes the bits directly. If the number of shifts is some other value, such as 7, this optimization would not occur.

Interpolation for Each Lookup Table

In theory, you can calculate the interpolation with the following code:

$$y = (yData[iRight] - yData[iLeft]) * (u - xData[iLeft]) \dots \\ / (xData[iRight] - xData[iLeft]) + yData[iLeft]$$

The term $(xData[iRight] - xData[iLeft])$ is the spacing between neighboring breakpoints. If this value is constant, due to even spacing, some simplification is possible. If spacing is not just even but also a power of two, significant simplifications are possible for fixed-point implementations.

Uneven Case

For the uneven case, one possible implementation of the ideal interpolation in fixed point is as follows:

```
xNum = uAngle          - xuneven[iLeft];
xDen  = xuneven[iRight] - xuneven[iLeft];
yDiff = yuneven[iRight] - yuneven[iLeft];

MUL_S32_S16_U16( bigProd, yDiff, xNum );

DIV_NZP_S16_S32_U16_FL00R( yDiff, bigProd, xDen );

yUneven = yuneven[iLeft] + yDiff;
```

The multiplication and division routines are not shown here. These routines can be complex and depend on the target processor. For example, these routines look different for a 16-bit processor than for a 32-bit processor.

Even Case

Evenly-spaced breakpoints implement interpolation using slightly different calculations than the uneven case. The key difference is that the calculations do not directly use the breakpoints. When the breakpoints are not required in ROM, you can save a lot of memory:

```
xNum = uAngle - ( iLeft * 455U );
```

```
yDiff = yeven[iLeft+1] - yeven[iLeft];  
MUL_S32_S16_U16( bigProd, yDiff, xNum );  
DIV_NZP_S16_S32_U16_FLOOR( yDiff, bigProd, 455U );  
yEven = yeven[iLeft] + yDiff;
```

Power of Two Case

Power of two spaced breakpoints implement interpolation using very different calculations than the other two cases. As in the even case, breakpoints are not used in the generated code and therefore not required in ROM:

```
lambda = uAngle & 0x00FFU;  
yPow2 = ypow2[iLeft+1] - ypow2[iLeft];  
MUL_S16_U16_S16_SR8(yPow2, lambda, yPow2);  
yPow2 += ypow2[iLeft];
```

This implementation has significant advantages over the uneven and even implementations:

- A bitwise AND combined with a shift right at the end of the multiplication replaces a subtraction and a division.
- The term $(u - xData[iLeft]) / (xData[iRight] - xData[iLeft])$ results in no loss of precision, because the spacing is a power of two.

In contrast, the uneven and even cases usually introduce rounding error in this calculation.

Summary of the Effects of Breakpoint Spacing

The following table summarizes the effects of breakpoint spacing on execution speed, error, and memory usage.

Parameter	Even Power of 2 Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
Execution speed	The execution speed is the fastest. The position search and interpolation are the same as for evenly-spaced data. However, to increase the speed more, a bit shift replaces the position search, and a bit mask replaces the interpolation.	The execution speed is faster than that for unevenly-spaced data, because the position search is faster and the interpolation requires a simple division.	The execution speed is the slowest of the different spacings because the position search is slower, and the interpolation requires more operations.
Error	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be smaller because approximating a function with nonuniform curvature requires fewer points to achieve the same accuracy.
ROM usage	Uses less command ROM, but more data ROM.	Uses less command ROM, but more data ROM.	Uses more command ROM, but less data ROM.
RAM usage	Not significant.	Not significant.	Not significant.

The number of Y data points follows the expected pattern. For the same worst-case error, unrestricted spacing (uneven) requires the fewest data points, and power-of-two-spaced breakpoints require the most. However, the implementation for the evenly-spaced and the power of two cases does not need the breakpoints in the generated code. This reduces their data ROM requirements by half. As a result, the evenly-spaced case actually uses

less data ROM than the unevenly-spaced case. Also, the power of two case requires only slightly more ROM than the uneven case. Changing the worst-case error can change these rankings. Nonetheless, when you compare data ROM usage, you should always take into account the fact that the evenly-spaced and power of two spaced cases do not require their breakpoints in ROM.

The effort of determining where the current input is relative to the breakpoints strongly favors the evenly-spaced and power of two spaced cases. With uneven spacing, you use a binary search method that loops up to $\log_2(N)$ times. With even and power of two spacing, you can determine the location with the execution of one line of C code. But you cannot decide the relative advantages of power of two versus evenly spaced without detailed knowledge of the hardware and the C compiler.

The effort of calculating the interpolation favors the power of two case, which uses a bitwise AND operation and a shift to replace a subtraction and a division. The advantage of this behavior depends on the specific hardware, but you can expect an advantage in code size, speed, and also in accuracy. The evenly-spaced case calculates the interpolation with a minor improvement in efficiency over the unevenly-spaced case.

Automatic Data Typing

- “Choosing a Range Collection Method” on page 39-2
- “Before Proposing Data Types for Your Model” on page 39-5
- “Best Practices for Fixed-Point Workflow” on page 39-7
- “Models That Might Cause Data Type Propagation Errors” on page 39-10
- “Autoscaling Using the Fixed-Point Tool” on page 39-12
- “Set Up the Model” on page 39-14
- “Select System Under Design” on page 39-16
- “Collect Ranges” on page 39-17
- “Propose Data Types” on page 39-19
- “Examine Results to Resolve Conflicts” on page 39-22
- “Apply Proposed Data Types” on page 39-27
- “Verify New Settings” on page 39-29
- “Automatic Data Typing of Simulink Signal Objects” on page 39-31
- “Get Proposals for Results with Inherited Types” on page 39-32
- “Propose Fraction Lengths” on page 39-34
- “How the Fixed-Point Tool Proposes Word Lengths” on page 39-50
- “Propose Word Lengths Based on Simulation Data” on page 39-52
- “Propose Data Types Using Multiple Simulations” on page 39-58
- “View Simulation Results” on page 39-63
- “Viewing Results With the Simulation Data Inspector” on page 39-70
- “Fixed-Point Instrumentation and Data Type Override” on page 39-72

Choosing a Range Collection Method

The Fixed-Point Tool automates the task of specifying fixed-point data types in a Simulink model. This process is also known as *autoscaling*. The tool collects range data for model objects, either from design minimum and maximum values that objects specify explicitly, from logged minimum and maximum values that occur during simulation, or from minimum and maximum values derived using range analysis. Based on these values, the tool proposes fixed-point data types that maximize precision and cover the range. The tool allows you to review the data type proposals and then apply them selectively to objects in your model.

You can use the Fixed-Point Tool to select data types automatically for your model using the following methods.

Automatic Data Typing Method	Advantages	Disadvantages
Using simulation minimum and maximum values	<ul style="list-style-type: none"> Useful if you know the inputs to use for the model. You do not need to specify any design range information. 	<ul style="list-style-type: none"> Not always feasible to collect full simulation range. Simulation might take a very long time.
Using design minimum and maximum values	You can use this method if the model contains blocks that range analysis does not support. However, if possible, use simulation data to propose data types.	<ul style="list-style-type: none"> Design range often available only on some input and output signals. Can propose data types only for signals with specified design minimum and maximum values.
Using derived minimum and maximum values	You do not have to simulate multiple times to ensure that simulation data covers the full intended operating range.	<ul style="list-style-type: none"> Derivation might take a very long time.

Workflow Feature	Conversion Using Simulation Data	Conversion Using Range Analysis
Iterative workflow	Merge ranges of multiple simulations. This iterative process allows you to collect	Results will not change from one derivation to the next unless you manually change design ranges in the model.

Workflow Feature	Conversion Using Simulation Data	Conversion Using Range Analysis
	ranges from multiple input cases.	
Range coverage	Proposed data types are based on simulation ranges. The proposals provided by the Fixed-Point Tool are as good as the test bench provided.	Range analysis typically delivers a more conservative data type proposal.
Input to system	Comprehensive set of input signals that exercise the full range of your design are required for data type proposals.	Ranges reported from derivation based upon provided design ranges
Design ranges	Simulation ranges are verified against design range specification and violations are reported in the Diagnostic Viewer.	Design ranges must be specified on the model.
Supported Features	All model objects are supported for instrumentation and range collection	Range analysis supports a subset of model objects. For more information, see “Unsupported Simulink Software Features” on page 40-46.
Modeling constructs	Ranges always converge during simulation.	Some modeling constructs, such as feedback loops, may require more design range information before converging.
Tunable parameters with known ranges	You must exercise tunable range as an input	Design ranges of tunable parameters are reported.
Data type proposals	Data type proposals based on collected minimum and maximum values.	

Workflow Feature	Conversion Using Simulation Data	Conversion Using Range Analysis
Simulation mode	Instrumentation data is only collected during Normal mode. No instrumentation data is collected while a model is running in accelerated or rapid accelerated mode. If you know that simulation will take a long time, you may want to derive ranges for your model.	Simulation mode has no affect on range analysis.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Before Proposing Data Types for Your Model


Before you use the Fixed-Point Tool to propose data types for your Simulink model, consider how automatic data typing affects your model:

- The Fixed-Point Tool proposes new data types for objects in your model. If you choose to apply the proposed data types, the tool changes the data types in your model. Before using the Fixed-Point Tool, back up your model and workspace variables to ensure that you can recover your original data type settings and capture the fixed-point instrumentation and data type override settings using the Advanced System Settings dialog.



For more information, see “Best Practices for Fixed-Point Workflow” on page 39-7.

- Before proposing data types, verify that you can **update diagram** successfully . Sometimes, changing the data types in your model results in subsequent **update diagram** errors. Immediately before and after applying data type proposals, it is good practice to test **update diagram** again. This practice enables you to fix any errors before making further modifications to your model.

For more information, see “Update Diagram and Run Simulation” (Simulink).

- The Fixed-Point Tool alerts you to potential issues with proposed data types for each object in your model:
 - If the Fixed-Point Tool detects that the proposed data type introduces data type errors when applied to an object, the tool marks the object with an error, . You must inspect this proposal and fix the problem in the Simulink model. After fixing the problem, rerun the simulation and generate a proposal again to confirm that you have resolved the issue.

For more information, see “Examine Results to Resolve Conflicts” on page 39-22.

- If the Fixed-Point Tool detects that the proposed data type poses potential issues for an object, the tool marks the object with a yellow caution, . Review the proposal before accepting it.
- If the Fixed-Point Tool detects that the proposed data type poses no issues for an object, the tool marks the object with a green check, .

Caution: The Fixed-Point Tool does not detect all potential data type issues. If the Fixed-Point Tool does not detect any issues for your model, it is still possible to experience subsequent data type propagation issues. For more information, see “Models That Might Cause Data Type Propagation Errors” on page 39-10.

Related Examples

- “Propose Fraction Lengths” on page 39-34

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Best Practices for Fixed-Point Workflow

Use a Known Working Simulink Model

Before you begin automatic data typing, verify that **update diagram** succeeds for your model. To update diagram, press **Ctrl+D**. If **update diagram** fails, before automatic data typing to propose data types, fix the failure in your model.

Back Up Your Simulink Model

Before using the Fixed-Point Tool, back up your Simulink model and associated workspace variables.

Backing up your model provides a backup of your original model in case of error and a baseline for testing and validation.

Capture the Current Data Type Override Settings

Before changing the current fixed-point instrumentation and data type override settings, use the Fixed-Point Tool Shortcut Editor to create a shortcut for these settings. Creating a shortcut allows you to revert to the original model settings. For more information, see “Capture Model Settings Using the Shortcut Editor” on page 36-9.

Convert Individual Subsystems

Convert individual subsystems in your model one at a time. This practice facilitates debugging by isolating the source of fixed-point issues. For example, see “Debug a Fixed-Point Model” on page 36-10.

Isolate the System Under Design

If you encounter data type propagation issues with a particular subsystem during the conversion, isolate this subsystem by placing Data Type Conversion blocks on the inputs and outputs of the system. The Data Type Conversion block converts an input signal of any Simulink data type to the data type and scaling you specify for its **Output data type** parameter. This practice enables you to continue automatic data typing for the rest of your model.

Do Not Use “Save as” on Referenced Models and MATLAB Function blocks

During the fixed-point conversion process using the Fixed-Point Tool, do not use the “Save as” option to save referenced models or MATLAB Function blocks with a different name. If you do, you might lose existing results for the original model.

Use Lock Output Data Type Setting

You can prevent the Fixed-Point Tool from replacing the current data type. Use the **Lock output data type setting against changes by the fixed-point tools** parameter that is available on many blocks. The default setting allows for replacement. Use this setting when:

- You already know the fixed-point data types that you want to use for a particular block.

For example, the block is modeling a real-world component. Set up the block to allow for known hardware limitations, such as restricting outputs to integer values.

Explicitly specify the output data type of the block and select **Lock output data type setting against changes by the fixed-point tools**.

- You are debugging a model and know that a particular block accepts only certain input signal data types.

Explicitly specify the output data type of upstream blocks and select **Lock output data type setting against changes by the fixed-point tools**.

Save Simulink Signal Objects

If your model contains Simulink signal objects and you accept proposed data types, the Fixed-Point Tool automatically applies the changes to the signal objects. However, the Fixed-Point Tool does not automatically save changes that it makes to Simulink signal objects. To preserve changes, before closing your model, save the Simulink signal objects in your workspace and model.

Test Update Diagram Failure

Immediately after applying data type proposals, test **update diagram**. If **update diagram** fails, perform one of the following actions:

- Use the failure information to fix the errors in your model. After fixing the errors, test **update diagram** again.
- If you are unable to fix the errors, restore your back-up model. After restoring the model, try to fix the errors by, for example, locking output data type settings and isolating the system, as described in the preceding sections. After addressing the errors, test **update diagram** again.

Disable Fast Restart

The Fixed-Point Tool provides limited support when the model is simulated in Fast Restart. You must disable Fast Restart to collect simulation and derived ranges, and propose data types.

Related Examples

- “Convert Floating-Point Model to Fixed Point” on page 37-8

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Models That Might Cause Data Type Propagation Errors

When the Fixed-Point Tool proposes changes to the data types in your model, it alerts you to potential issues. If the Fixed-Point Tool alerts you to data type errors, you must diagnose the errors and fix the problems. For more information, see “Examine Results to Resolve Conflicts” on page 39-22.

The Fixed-Point Tool does not detect all potential data type issues. If the tool does not report any issues for your model, it is still possible to experience subsequent data type propagation errors. Before you use the Fixed-Point Tool, back up your model to ensure that you can recover your original data type settings. For more information, see “Best Practices for Fixed-Point Workflow” on page 39-7.

The following models are likely to cause data type propagation issues.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
Simulink parameter objects	Does not consider any data type information for Simulink parameter objects and does not use them for automatic data typing.	Fixed-Point Tool might propose data types that are inconsistent with the data types for the parameter object or generate proposals that cause overflows.
User-defined S-functions	Cannot detect the operation of user-defined S-functions.	<ul style="list-style-type: none"> The user-defined S-function accepts only certain input data types. The Fixed-Point Tool cannot detect this requirement and proposes a different data type upstream of the S-function. Update diagram fails on the model due to data type mismatch errors. The user-defined S-function specifies certain output data types. The Fixed-Point Tool is not aware of this requirement and does not use it for automatic data typing. Therefore, the tool might propose data types that are

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
		inconsistent with the data types for the S-function or generate proposals that cause overflows.
User-defined masked subsystems	Has no knowledge of the masked subsystem workspace and cannot take this subsystem into account when proposing data types.	Fixed-Point Tool might propose data types that are inconsistent with the requirements of the masked subsystem, particularly if the subsystem uses mask initialization. The proposed data types might cause data type mismatch errors or overflows.
Linked subsystems	Does not include linked subsystems when proposing data types.	Data type mismatch errors might occur at the linked subsystem boundaries.

More About

- “Examine Results to Resolve Conflicts” on page 39-22
- “Data Type Propagation Errors After Applying Proposed Data Types” on page 46-27

Autoscaling Using the Fixed-Point Tool

The Fixed-Point Tool automates the task of specifying fixed-point data types in a Simulink model. This process is also known as *autoscaling*. The tool can collect range data for model objects from:

- Design minimum and maximum values that objects specify explicitly
- Logged minimum and maximum values that occur during simulation
- Minimum and maximum values derived using range analysis

Based on these values, the tool proposes fixed-point data types that maximize precision and cover the range.

Workflow for Automatic Data Typing

Before you begin conversion, you need to set up the model in Simulink. For more detail, see *Set up the model*.

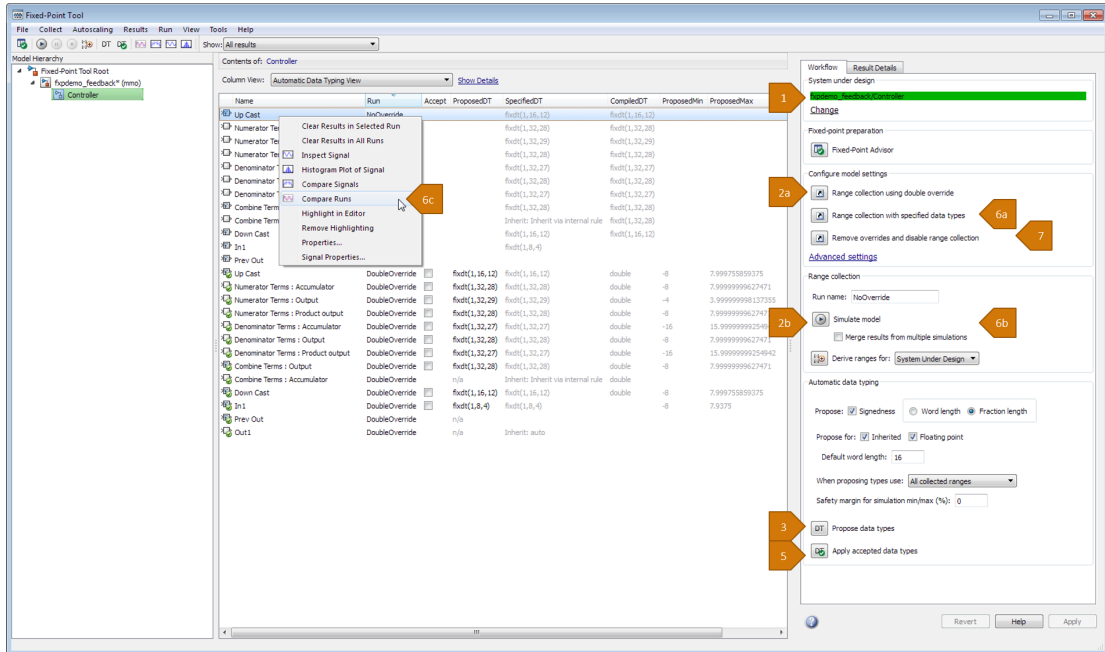
Data Typing

- 1 Select system under design
- 2 Collect ranges
 - a Range collection using simulation—Using this method, collect ideal ranges for objects in your model by overriding your model with double-precision data types, and then simulate the model.
 - b Range collection using derived range analysis—Using this method, the Fixed-Point Tool uses range analysis to derive minimum and maximum values for objects in your model. Range analysis works only for compatible models. For more information, see “Model Compatibility with Range Analysis” on page 40-6.
- 3 Propose data types
- 4 Examine results to resolve conflicts
- 5 Apply proposed data types

Validation

- 1 Verify new settings

- a Set range collection using specified types
 - b Simulate using fixed-point data types
 - c Compare runs in the Simulation Data Inspector
- 2 “Clean Up Model Settings” on page 39-30



To begin the fixed-point conversion process, see “Set Up the Model” on page 39-14.

Related Examples

- “Propose Fraction Lengths” on page 39-34

Set Up the Model

To use the Fixed-Point Tool to generate data type proposals for your model, first set up your model in Simulink.

- 1 Back up your model as a baseline for testing and validation.
- 2 Open your model in Simulink.
- 3 In the model, select **Simulation > Mode > Normal** so that the model runs in normal mode. The Fixed-Point Tool supports only normal mode.
- 4 If you are using design minimum and maximum range information, add this information to the blocks. To autoscale using derived data, you **must** specify design minimum and maximum values on at least the model inputs. The range analysis tries to narrow the derived range by using all the specified design ranges in the model. The more design range information you specify, the more likely the range analysis is to succeed. As the analysis is performed, it derives new range information for the model and then attempts to use this new information together with the specified ranges to derive ranges for the remaining objects in the model. For this reason, the analysis results might depend on block priorities because these priorities determine the order in which the software analyzes the blocks.

You specify a design range for model objects using parameters such as **Output minimum** and **Output maximum**. For a list of blocks in which you can specify these values, see “Blocks That Allow Signal Range Specification” (Simulink).

- 5 You can choose to lock some blocks against automatic data typing by selecting the **Lock output data type setting against changes by the fixed-point tools** parameter. If you select this parameter, the tool does not propose data types for the block.
- 6 Update the diagram to perform parameter range checking for all blocks in the model.

If updating the diagram fails, use the error messages to fix the errors in your model. After fixing the errors, update the diagram again. If you cannot fix the errors, restore your backup model.

- 7 If the model changed, back up the model again.

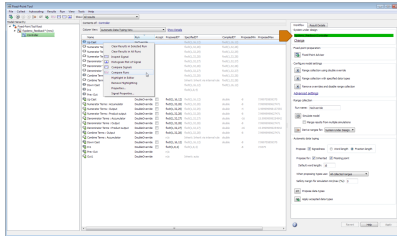
To learn about the next step in the conversion process, see “Select System Under Design” on page 39-16.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Select System Under Design

Select the system or subsystem you want to convert to fixed point. Convert individual subsystems in your model one at a time. This practice facilitates debugging by isolating the source of fixed-point issues.



- 1 In your model, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 2 Under **System Under Design**, specify the system or subsystem you want to convert. To continue with the conversion on the current system, click **Continue**. Otherwise, click **Change** and select the system.

You can also use the Fixed-Point Advisor to identify unsupported blocks in your model and prepare the model for conversion. For more information, see “Preparation for Fixed-Point Conversion” on page 35-2.

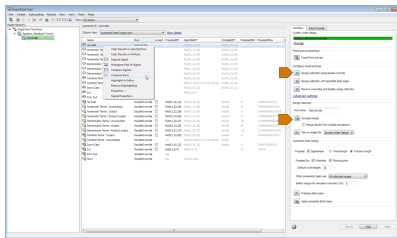
To learn about the next step in the conversion process, see “Collect Ranges” on page 39-17.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Collect Ranges


After selecting the system under design as described in “Select System Under Design” on page 39-16, collect ranges for the objects in your model using either simulation or derived range analysis. To decide which method of range collection is right for your application, see “Choosing a Range Collection Method” on page 39-2.



Collect Ranges Using Simulation Data

To view simulation results using the Simulation Data Inspector, you must enable signal logging for the system under design. The Fixed-Point Tool allows you to enable signal logging for multiple signals simultaneously. You can choose to plot results using the Simulation Data Inspector only for signals that have signal logging enabled.

- 1 On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem.
- 2 Right-click the selected system and select **Enable Signal Logging**.

After simulation, the **Contents** pane of the Fixed-Point Tool displays an antenna icon  on items that have signal logging enabled.

- 3 Under **Configure model settings**, click the **Range collection using double override** button.

The Fixed-Point Tool performs a global override of the fixed-point data types with double-precision data types, avoiding quantization effects. This setting provides a floating-point benchmark that represents the ideal output.

Note: Data type override does not apply to Boolean or enumerated data types.

- 4 Click the **Simulate model** button .

The Fixed-Point Tool collects range information for the objects in your model and displays the ranges in the **Contents** pane of the Fixed-Point Tool. The tool highlights any simulation results that have issues, such as overflows due to wrap or saturations.

Collect Ranges Using Range Analysis

If you choose to collect ranges for objects in your model through derived range analysis, you do not need to simulate the model. However, to compare floating-point and fixed-point behavior using the Simulation Data Inspector, simulation is required. You can collect a floating-point baseline run to use for comparison later in the workflow by performing the steps described in “Collect Ranges Using Simulation Data” on page 39-17.

- 1 Under **Collect Ranges**, specify if you want to derive ranges for the selected system under design, or for the whole model in the **Derive ranges for** field.

If you select **System under design**, the tool uses only design ranges specified within the system under design to collect ranges. If you select **model**, the tool uses all design range information specified on the model to derive ranges for objects in the system under design.

- 2 Click **Derive ranges**.

The analysis runs and tries to derive range information for objects in the selected system.

If the analysis successfully derives range data for the model, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the selected system. Before proposing data types, review the results.

If the analysis fails, examine the error messages and resolve the issues. See “Resolve Range Analysis Issues” on page 46-30.

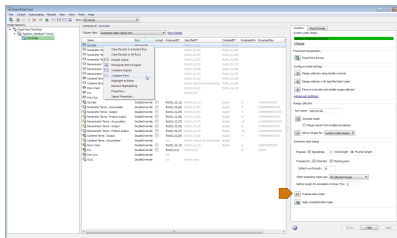
To complete the next step in the conversion process, see “Propose Data Types” on page 39-19.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Propose Data Types

After collecting ranges as described in “Collect Ranges” on page 39-17, propose data types for objects in your model based on the collected ideal ranges. The Fixed-Point Tool proposes a data type for all objects in the system under design whose **Lock output data type setting against changes by the fixed-point tools** parameter is cleared.



When proposing data types, the Fixed-Point Tool uses these types of range data for model objects:

- Design minimum or maximum values — You specify a design range for model objects using parameters such as **Output minimum** and **Output maximum**. For a list of blocks for which you can specify these values, see “Blocks That Allow Signal Range Specification” (Simulink).
- Simulation minimum or maximum values — When simulating a system with instrumentation enabled, the Fixed-Point Tool logs the minimum and maximum values generated by model objects. For more information about the **Fixed-point instrumentation mode** parameter, see “Fixed-point instrumentation mode”.
- Derived minimum or maximum values — When deriving minimum and maximum values for a selected system, the Fixed-Point Tool uses the design minimum and maximum values that you specify on the blocks to derive range information for signals in your model. For more information, see “Derive Ranges” on page 40-7.

The Fixed-Point Tool uses available range data to calculate data type proposals. You can specify the range data that the Fixed-Point Tool uses for data type proposals with the **When proposing types use** option.

When proposing types use Setting	Fixed-Point Tool Action
All collected ranges	The Fixed-Point Tool uses design ranges in addition to derived and simulation ranges to propose data types.


When proposing types use Setting	Fixed-Point Tool Action
	Design minimum and maximum values take precedence over the simulation and derived range.
Design and simulation ranges	<p>The Fixed-Point Tool uses the design ranges in conjunction with collected simulation ranges to propose data types. Design ranges take precedence over simulation ranges.</p> <p>The Safety margin for simulation min/max (%) parameter specifies a range that differs from the range defined by the simulation range.</p>
Design and derived ranges	The Fixed-Point Tool uses the design ranges in conjunction with derived ranges to propose data types. Design ranges take precedence over derived ranges.

Under **Automatic data typing**, configure the proposal settings and generate proposals. When the Fixed-Point Tool proposes data types, it does not modify your model.

- 1 Select the **Signedness** check box to have the Fixed-Point Tool propose signedness.

When this box is selected, the Fixed-Point Tool proposes signedness based on range information.

- 2 Select the type of length to propose.
 - When you select **Word length**, the Fixed-Point Tool uses range information and the specified **Default fraction length** value to propose word lengths for the objects in your model.
 - When you select **Fraction length**, the Fixed-Point Tool uses the range information and the specified **Default word length** value to propose best-precision fraction lengths for the objects in your model.
- 3 Specify whether to propose new data types for objects that use inherited and floating-point output data types by selecting the corresponding check boxes.
- 4 Specify the default word length or default fraction length to use for proposals.

- 5 Select the type of ranges for the Fixed-Point Tool to use for proposals from the **When proposing types use** menu.
- 6 If you have safety margins to apply, enter the value in the **Safety margin for simulation min/max (%)** field. For example, enter **10** for a 10 % safety margin.
- 7 To generate proposals, click the **Propose data types** button .

To complete the next step in the conversion process, see “Examine Results to Resolve Conflicts” on page 39-22.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Examine Results to Resolve Conflicts

After proposing data types with the Fixed-Point Tool as described in “Propose Data Types” on page 39-19, you can examine each proposal using the **Result Details** tab. This tab displays the rationale for the proposed data types and a histogram plot of the signal. This tab also describes potential issues or errors and suggests methods for resolving them. To view the details, on the **Contents** pane, select an object that has a proposed data type. Then select the **Result Details** tab in the rightmost pane of the Fixed-Point Tool.



Proposed Data Type Summary

The summary describes how the proposal differs from the currently specified data type of the object. For cases when the Fixed-Point Tool does not propose data types, this section provides a rationale. For example, the data type might be locked against changes by the fixed-point tools.

This section of the **Result Details** tab also informs you if the selected object must use the same data type as other objects in the model because of data type propagation rules. For example, the inputs to a Merge block must have the same data type. Therefore, the outputs of blocks that connect to these inputs must use the same data type. Similarly, blocks that are connected by the same element of a virtual bus must use the same data type.

In the tab, click the link to highlight the objects that share data types in the model. To clear this highlighting, right-click in the model and select **Remove Highlighting**.

The Fixed-Point Tool allocates a tag to objects that must use the same data type. The tool displays this tag in the **DTGroup** column for the object. To display only the objects that must share data types, from the Fixed-Point Tool toolbar, select the corresponding **Show** option.


Some Simulink blocks accept only certain data types on some ports. This section of the **Result Details** tab also informs you when a block that connects to the selected object has data type constraints that affect the proposed data type of the selected object.


The **Proposed Data Type Summary** section of the **Result Details** tab provides a table with the proposed data type information.

Item	Description
ProposedDT	Data type that the Fixed-Point Tool proposes for this object and the minimum and maximum values that the proposed data type can represent
SpecifiedDT	Data type that an object specifies

Needs Attention

This section lists potential issues and errors associated with data type proposals and describes the issues and suggests methods for resolving them.

 Indicates a warning message

 Indicates an error message

Range Information

This section provides a table with model object attributes that influence the data type proposal.

Item	Description
Design	Design maximum and minimum values that an object specifies, e.g., its Output maximum and Output minimum parameters
Simulation	Maximum and minimum values that occur during simulation

Shared Values

When proposing data types, the Fixed-Point Tool attempts to satisfy data type requirements that model objects impose on one another. For example, the Sum block has an option that requires all its inputs to have the same data type. As a result, the table might also list attributes of other model objects that affect the proposal for the selected object. In such cases, the table displays these types of shared values:

- **Initial Values**—Some model objects have parameters that allow you to specify the initial values of their signals. For example, the Constant block has a **Constant value** parameter that initializes the block output signal. The Fixed-Point Tool uses initial values to propose data types for model objects whose design and simulation ranges are unavailable. With data type dependencies, the tool determines how initial values impact the proposals for neighboring objects.
- **Model-Required Parameters**—Some model objects require you to specify numeric parameters to compute the value of their outputs. For example, the **Table data** parameter of an n-D Lookup Table block specifies values that the block requires to perform a lookup operation and generate output. When proposing data types, the Fixed-Point Tool considers how this parameter value required by the model impacts the proposals for neighboring objects.

Examine the Results and Resolve Conflicts

- 1 On the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

The Fixed-Point Tool lists the data type proposals on the **Contents** pane under the **ProposedDT** column. Potential issues for each object appear coded by color in the list.



The proposed data type poses no issues for this object




The proposed data type poses potential issues for this object



The proposed data type will introduce data type errors if applied to this object

- 2 Review and fix each error. Select the result with the error, right-click, and select **Highlight in Editor** from the context menu to identify the conflict. Use the information in the **Needs Attention** section of the **Result Details** tab to resolve the conflict.
- 3 Review the **Result Details** for the warnings and correct the problem if necessary.
- 4 You have changed the Simulink model, so the benchmark data is not up to date.

Click the Fixed-Point Tool **Simulate model** button  to rerun the simulation, or click **Derive ranges** to derive new ranges.

If the Fixed-Point Tool warns you that you have not applied proposals, click **Ignore and Simulate** to continue.

- 5 To generate a proposal, click **Propose data types** .
- 6 On the Fixed-Point Tool toolbar, select **Show > All results**.

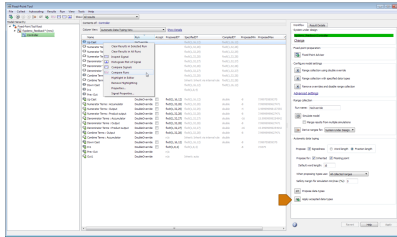
To complete the next step in the conversion process, see “Apply Proposed Data Types” on page 39-27.

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Apply Proposed Data Types

After reviewing the data type proposals as described in “Examine Results to Resolve Conflicts” on page 39-22, apply the proposed data types to your model.




The Fixed-Point Tool allows you to apply data type proposals selectively to objects in your model. On the **Contents** pane, use the **Accept** check box to specify the proposals that you want to assign to model objects.

- The Fixed-Point Tool applies the proposed data type to this object. By default, the tool selects the **Accept** check box when a proposal differs from the current data type of the object.
- The Fixed-Point Tool ignores the proposed data type and leaves the current data type intact for this object.
- No proposal exists for this object, for example, the object is locked against automatic data typing.

- 1 Examine each result. For more information about a particular result, select the result and then click the **Result Details** tab.
- 2 If you do **not** want to accept the proposal for a result, on the Fixed-Point Tool **Contents** pane, clear the **Accept** check box for that result.

Before applying proposals to your model, you can customize them. On the **Contents** pane, click a **ProposedDT** cell and edit the data type expression. For information about specifying fixed-point data types, see `fixdt`.

- 3 To write the proposed data types to the model, click the **Apply accepted data types** button .

If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.

To complete the next step in the conversion process, see “Verify New Settings” on page 39-29.

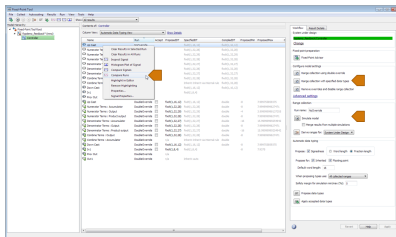
More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Verify New Settings

After applying proposed data types to your model as described in “Apply Proposed Data Types” on page 39-27, simulate the model using the applied fixed-point data types.

Note: This step requires that you ran a floating-point simulation during the “Collect Ranges” on page 39-17 phase of the conversion. If you used range analysis to collect ideal ranges for your system under design and did not run a floating-point simulation, you will not have a run to compare at this step.






- 1 Under **Configure model settings**, click the **Range collection with specified data types** button to use the locally specified data type settings.

This configuration sets:

- **Data type override** to Use local settings
- **Fixed-point instrumentation mode** to Minimums, maximums and overflows
- The run name (in the **Range collection** pane **Run name** field) to NoOverride

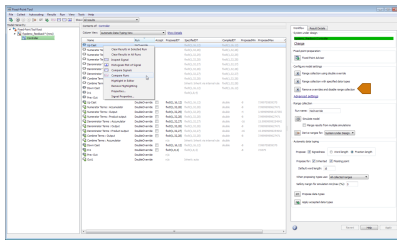
Using these settings, the Fixed-Point Tool simulates the model using the new fixed-point settings. It logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run NoOverride.

- 2 Click the Fixed-Point Tool **Simulate model** button .
- 3 Compare the ideal results stored in the DoubleOverride run, collected during the “Collect Ranges” on page 39-17 step of the workflow, with the fixed-point results in the NoOverride run:

- a On the **Contents** pane, select a result that has logged signal data. These results are annotated with the  icon.
- b Click the **Compare Runs** button  to view the difference between the fixed-point and double override runs for the selected result. If you have more than two runs, in the **Compare Runs Selector** dialog box, select the two runs that you want to compare.

Clean Up Model Settings

When you are satisfied with the results of the fixed-point conversion, under **Configure model settings**, click the **Remove overrides and disable range collection** button. This configuration cleans up the settings after conversion to restore the maximum simulation speed to your model.



More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Automatic Data Typing of Simulink Signal Objects

The Fixed-Point Tool can propose new data types for Simulink signal objects in the base or model workspace. If you accept the proposed data types, the Fixed-Point Tool applies them to the Simulink signal objects.

Caution: The Fixed-Point Tool does not save the changes to the signal object. Before closing the model, you must save the changes.

After automatic data typing, if you delete or manipulate a signal object in the base workspace, you must rerun the automatic data typing.

Get Proposals for Results with Inherited Types

Blocks can inherit data types from a variety of sources, including signals to which they are connected and particular block parameters. The following table lists the types of inheritance rules that a block might specify.

Inheritance Rule	Description
Inherit: Inherit via back propagation	Simulink automatically determines the output data type of the block during data type propagation. In this case, the block uses the data type of a downstream block or signal object.
Inherit: Same as input	The block uses the data type of its sole input signal for its output signal.
Inherit: Same as first input	The block uses the data type of its first input signal for its output signal.
Inherit: Same as second input	The block uses the data type of its second input signal for its output signal.
Inherit: Inherit via internal rule	The block uses an internal rule to determine its output data type. The internal rule chooses a data type that optimizes numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. It is not always possible for the software to optimize efficiency and numerical accuracy at the same time.

How to Get Proposals for Objects That Use an Inherited Output Data Type

To enable proposals for results that specify an inherited output data type, in the Fixed-Point Tool, under **Automatic data typing**, select the **Inherited** check box.

For objects that specify an inherited output data type, the Fixed-Point Tool proposes a new data type based on collected ranges and the proposal settings specified in the **Automatic data typing** pane.

When the Fixed-Point Tool Will Not Propose for Inherited Data Types

The Fixed-Point Tool proposes data types only for the **Output data type** parameter of a block or model object. It will not propose for other block data types, such as the **Accumulator data type** of a Sum block, or the **Gain** parameter in a Gain block.

The Fixed-Point Tool will also not propose for the following model objects if they use an inherited output data type.

- Signal objects
- Stateflow charts
- Bus objects
- MATLAB variables

More About

- “Data Type Propagation Errors After Applying Proposed Data Types” on page 46-27

Propose Fraction Lengths


In this section...

“How to Propose Fraction Lengths” on page 39-34

“About the Feedback Controller Example Model” on page 39-34

“Propose Fraction Lengths Using Simulation Range Data” on page 39-41

How to Propose Fraction Lengths

- 1 In the Fixed-Point Tool, under **Automatic data typing**, next to **Propose**, select **Fraction Length**. Select the **Signedness** check box if you also want proposals for signedness.
- 2 You can select to have the Fixed-Point Tool propose data types for objects that use inherited and floating-point data types by selecting the corresponding check boxes.
- 3 Set the **Default word length** for the Fixed-Point Tool to use when proposing data types.
- 4 Select the types of ranges to use for proposals by selecting from the **When proposing types use** drop down menu.
- 5 If you have safety margins to apply, set **Safety margin for simulation min/max (%)**.
- 6 Click the **Propose data types** button, .

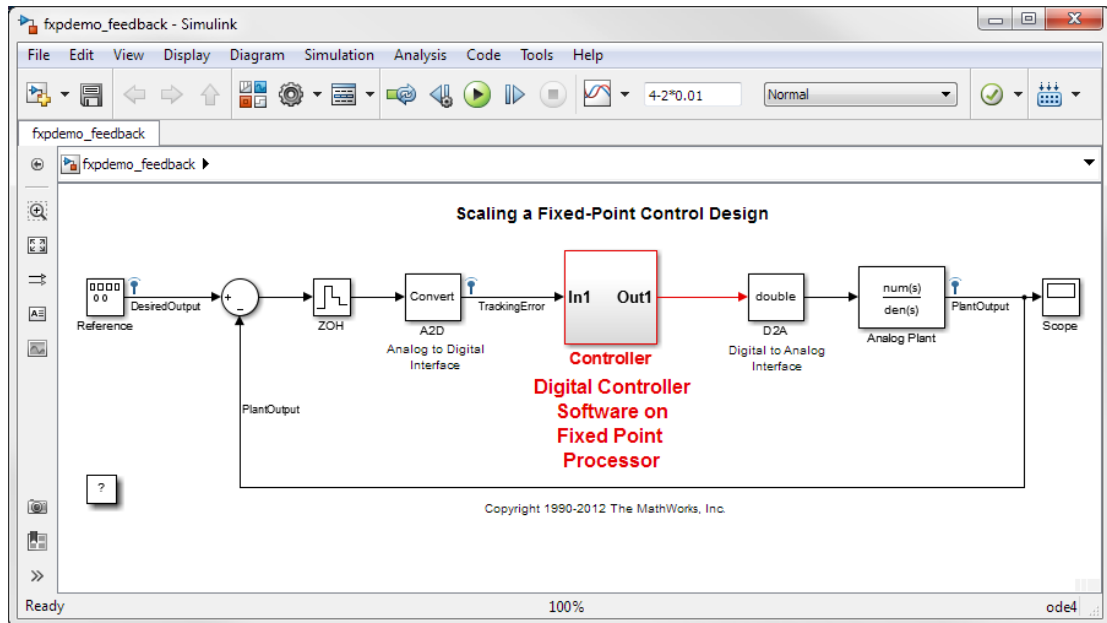
Note: When the Fixed-Point Tool proposes data types, it does not alter your model.

About the Feedback Controller Example Model

- “Opening the Feedback Controller Model” on page 39-34
- “Simulation Setup” on page 39-36
- “Idealized Feedback Design” on page 39-37
- “Digital Controller Realization” on page 39-38

Opening the Feedback Controller Model

To open the Simulink feedback design model for this tutorial, at the MATLAB command line, type `fxpdemo_feedback`.



The Simulink model of the feedback design consists of the following blocks and subsystems:

- **Reference**

This Signal Generator block generates a continuous-time reference signal. It is configured to output a square wave.

- **Sum**

This Sum block subtracts the plant output from the reference signal.

- **ZOH**

The Zero-Order Hold block samples and holds the continuous signal. This block is configured so that it quantizes the signal in time by 0.01 seconds.

- **Analog to Digital Interface**

The analog to digital (A/D) interface consists of a Data Type Conversion block that converts a double to a fixed-point data type. It represents any hardware that

digitizes the amplitude of the analog input signal. In the real world, its characteristics are fixed.

- **Controller**

The digital controller is a subsystem that represents the software running on the hardware target. Refer to “Digital Controller Realization” on page 39-38.

- **Digital to Analog Interface**

The digital to analog (D/A) interface consists of a Data Type Conversion block that converts a fixed-point data type into a `double`. It represents any hardware that converts a digitized signal into an analog signal. In the real world, its characteristics are fixed.

- **Analog Plant**

The analog plant is described by a transfer function, and is controlled by the digital controller. In the real world, its characteristics are fixed.

- **Scope**

The model includes a Scope block that displays the plant output signal.

Simulation Setup

To set up this kind of fixed-point feedback controller simulation:

- 1 Identify all design components.

In the real world, there are design components with fixed characteristics (the hardware) and design components with characteristics that you can change (the software). In this feedback design, the main hardware components are the A/D hardware, the D/A hardware, and the analog plant. The main software component is the digital controller.

- 2 Develop a theoretical model of the plant and controller.

For the feedback design in this tutorial, the plant is characterized by a transfer function.

The digital controller model in this tutorial is described by a z-domain transfer function and is implemented using a direct-form realization.

- 3 Evaluate the behavior of the plant and controller.

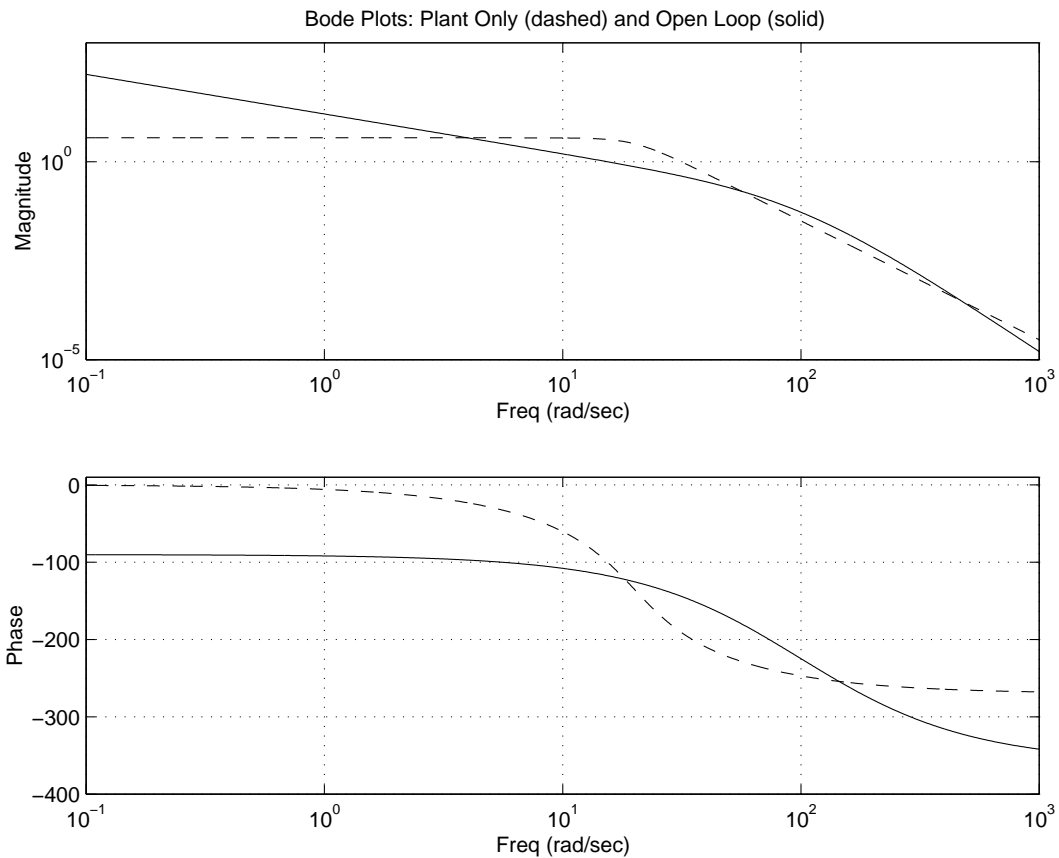
You evaluate the behavior of the plant and the controller with a Bode plot. This evaluation is idealized, because all numbers, operations, and states are double-precision.

4 Simulate the system.

You simulate the feedback controller design using Simulink and Fixed-Point Designer software. In a simulation environment, you can treat all components (software *and* hardware) as though their characteristics are not fixed.

Idealized Feedback Design

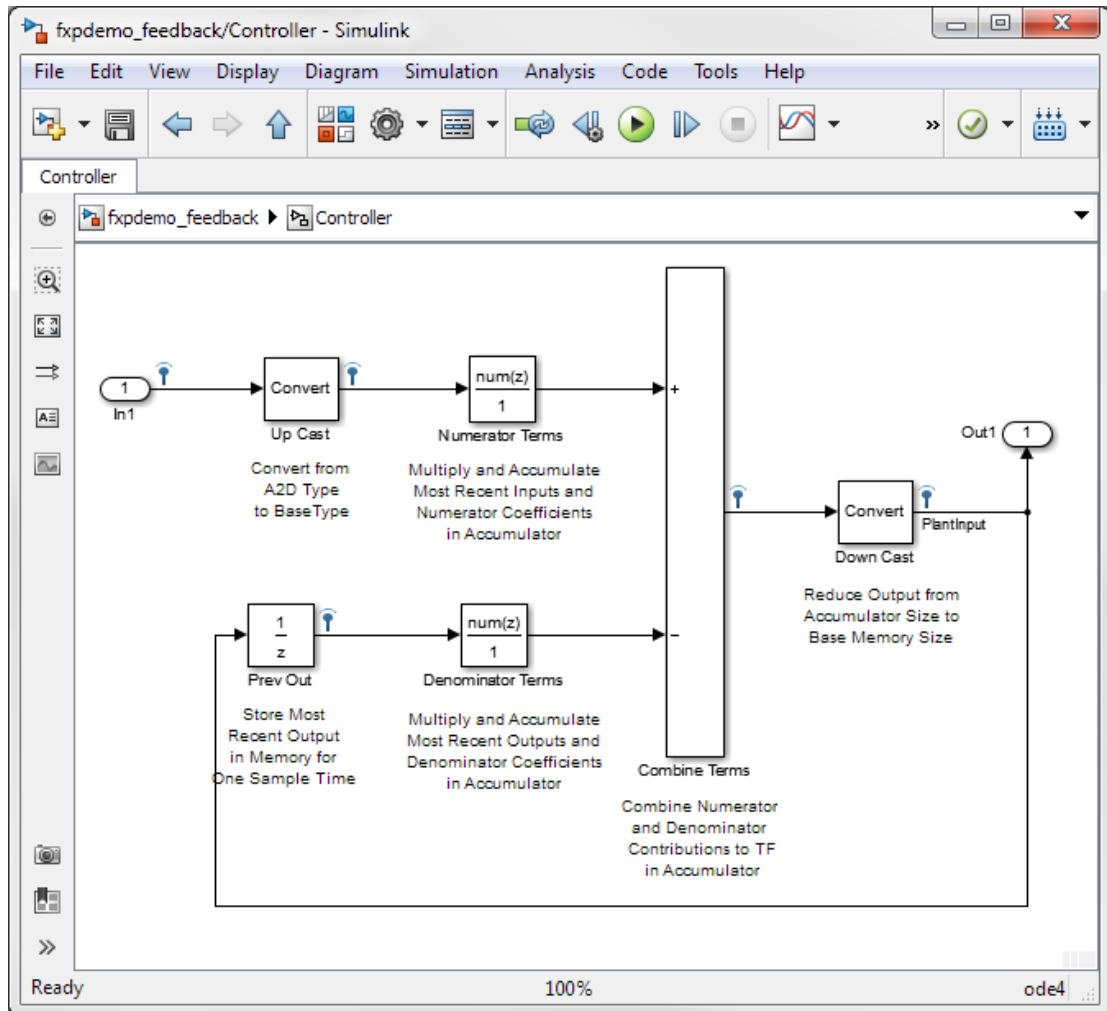
Open loop (controller and plant) and plant-only Bode plots for the “Scaling a Fixed-Point Control Design” model are shown in the following figure. The open loop Bode plot results from a digital controller described in the idealized world of continuous time, double-precision coefficients, storage of states, and math operations.



The Bode plots were created using workspace variables produced by a script named `preload_feedback.m`.

Digital Controller Realization

In this simulation, the digital controller is implemented using the fixed-point direct form realization shown in the following diagram. The hardware target is a 16-bit processor. Variables and coefficients are generally represented using 16 bits, especially if these quantities are stored in ROM or global RAM. Use of 32-bit numbers is limited to temporary variables that exist briefly in CPU registers or in a stack.



The realization consists of these blocks:

- **Up Cast**

Up Cast is a Data Type Conversion block that connects the A/D hardware with the digital controller. It pads the output word size of the A/D hardware with trailing zeros to a 16-bit number (the base data type).

- **Numerator Terms and Denominator Terms**

Each of these Discrete FIR Filter blocks represents a weighted sum carried out in the CPU target. The word size and precision in the calculations reflect those of the accumulator. Numerator Terms multiplies and accumulates the most recent inputs with the FIR numerator coefficients. Denominator Terms multiples and accumulates the most recent delayed outputs with the FIR denominator coefficients. The coefficients are stored in ROM using the base data type. The most recent inputs are stored in global RAM using the base data type.

- **Combine Terms**

Combine Terms is a Sum block that represents the accumulator in the CPU. Its word size and precision are twice that of the RAM (double bits).

- **Down Cast**

Down Cast is a Data Type Conversion block that represents taking the number from the CPU and storing it in RAM. The word size and precision are reduced to half that of the accumulator when converted back to the base data type.

- **Prev Out**

Prev Out is a Unit Delay block that delays the feedback signal in memory by one sample period. The signals are stored in global RAM using the base data type.

Direct Form Realization

The controller directly implements this equation:

$$y(k) = \sum_{i=0}^N b_i u(k-1) - \sum_{i=1}^N a_i y(k-1),$$

- $u(k-1)$ represents the *input* from the previous time step.
- $y(k)$ represents the current output, and $y(k-1)$ represents the output from the previous time step.
- b_i represents the FIR numerator coefficients.
- a_i represents the FIR denominator coefficients.

The first summation in $y(k)$ represents multiplication and accumulation of the most recent inputs and numerator coefficients in the accumulator. The second summation

in $y(k)$ represents multiplication and accumulation of the most recent outputs and denominator coefficients in the accumulator. Because the FIR coefficients, inputs, and outputs are all represented by 16-bit numbers (the base data type), any multiplication involving these numbers produces a 32-bit output (the accumulator data type).

Propose Fraction Lengths Using Simulation Range Data

- “Initial Guess at Scaling” on page 39-41
- “Data Type Override” on page 39-44
- “Automatic Data Typing” on page 39-45
- “Clean Up Settings” on page 39-48

This example shows you how to use the Fixed-Point Tool to refine the scaling of fixed-point data types associated with a feedback controller model (see “About the Feedback Controller Example Model” on page 39-34). Although the tool enables multiple workflows for converting a digital controller described in ideal double-precision numbers to one realized in fixed-point numbers, this example uses the following approach:

- “Initial Guess at Scaling” on page 39-41. Run an initial simulation using a reasonable guess at the fixed-point word size and scaling. This task illustrates how difficult it is to guess the best scaling.
- “Data Type Override” on page 39-44. Perform a global override of the fixed-point data types using double-precision numbers. The Simulink software logs the simulation results to the MATLAB workspace, and the Fixed-Point Tool displays them.
- “Automatic Data Typing” on page 39-45. Perform the automatic data typing procedure, which uses the double-precision simulation results to propose fixed-point scaling for appropriately configured blocks. The Fixed-Point Tool allows you to accept and apply the scaling proposals selectively. Afterward, you determine the quality of the results by examining the input and output of the model's analog plant.

Initial Guess at Scaling


Initial guesses for the scaling of each block are already specified in each block mask in the model. This task illustrates the difficulty of guessing the best scaling.

- 1 Open the `fxpdemo_feedback` model.
- 2 Open the Fixed-Point Tool. In the model, select **Analysis > Data Type Design > Fixed-Point Tool**.

- 3 In the Fixed-Point Tool, under **System under design**, click **Change**.

In the System Selector, select **Controller** as the system under design for conversion. Click **OK**.

- 4 Under **Configure model settings**, click the **Range collection with specified data types** button to configure the model to collect ranges using the currently specified data types. This configuration also sets the **Run name** to **NoOverride**.

- 5 In the Fixed-Point Tool, click **Simulate model** .

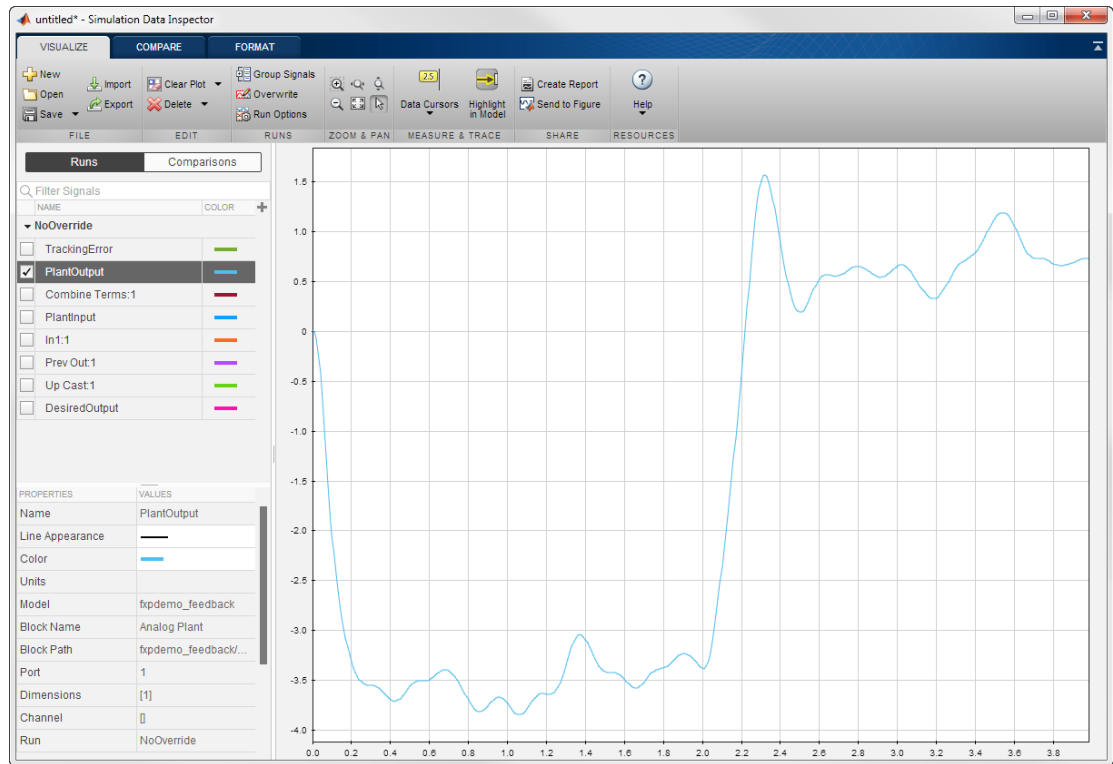
Simulink simulates the `fxpdemo_feedback` model. On its **Contents** pane, the Fixed-Point Tool displays the simulation results for each block that logged fixed-point data. By default, it displays the Simulation View of these results. You can customize this view by clicking **Show Details**. For more information about the standard views provided by the Fixed-Point Tool, see “Customizing the Contents Pane View”. For more information about customizing views, see “Customize Model Explorer Views” (Simulink).

The tool stores the results in the **NoOverride** run. The Fixed-Point tool highlights the **Up Cast** block to indicate that there is an issue with this result. The **OvfSat** column for this result shows that the block saturated 23 times, which indicates a poor guess for its scaling.

Tip: To view only blocks that have overflows, in the toolbar, select **Show > Overflows**.

- 6 On the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named **Analog Plant** and then click the **Inspect Signal** button .

The Fixed-Point Tool plots the signal associated with the plant output.




The plot of the plant output signal reflects the initial guess at scaling. The Bode plot design sought to produce a well-behaved linear response for the closed-loop system. Clearly, the response is nonlinear. Significant quantization effects cause the nonlinear features. An important part of fixed-point design is finding a scaling that reduces quantization effects to acceptable levels.


Tip: Use the Fixed-Point Tool plotting tools to plot simulation results associated with logged signal data. To view a list of all logged signals, in the toolbar, select **Show > Signal logging results**.

Data Type Override

Data type override enables you to perform a global override of the fixed-point data types with double-precision data types, thereby avoiding quantization effects. When performing automatic scaling to propose higher fidelity fixed-point scaling, the Fixed-Point Tool uses these simulation results.

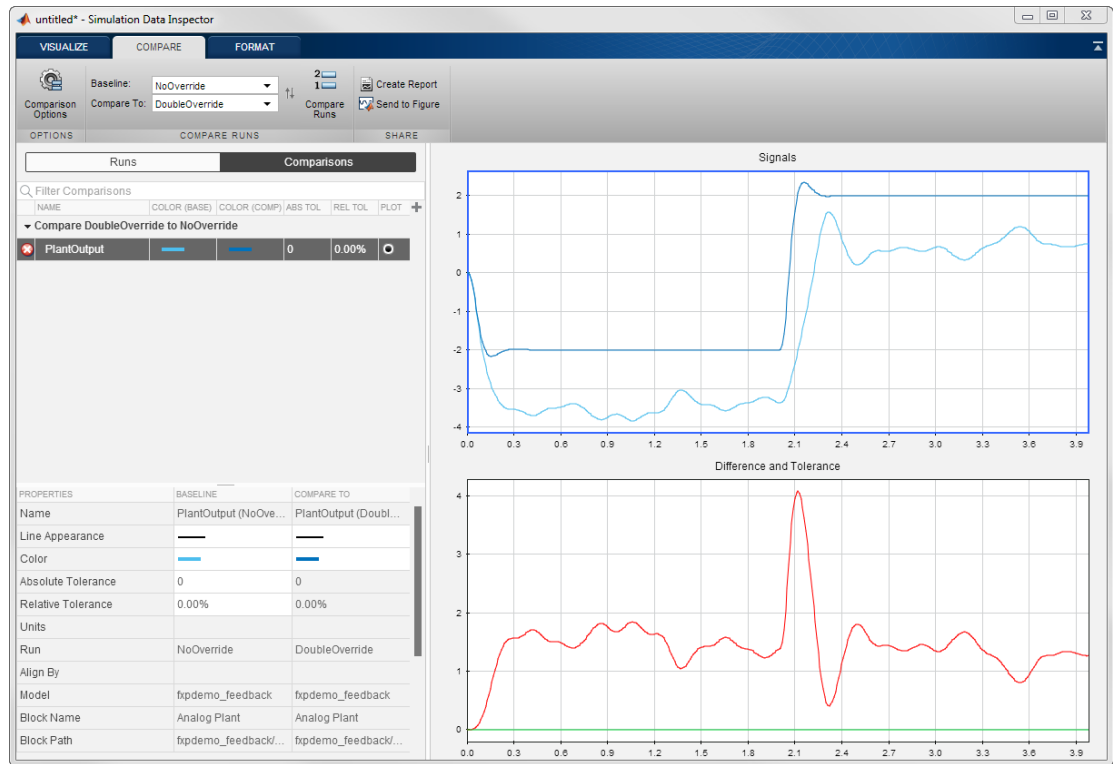
- 1 Under **Configure model settings**, click **Range collection using double override** to override the model with double-precision data types. This configuration also sets the **Run name** to **DoubleOverride**.
- 2 In the Fixed-Point Tool, click the **Simulate model** button .

The Simulink software simulates the `fxpdemo_feedback` model in data type override mode and stores the results in the **DoubleOverride** run. Afterward, on its **Contents** pane, the Fixed-Point Tool displays the **DoubleOverride** run results along with those of the **NoOverride** run that you generated previously (see “Initial Guess at Scaling” on page 39-41). The compiled data type (**CompiledDT**) column for the **DoubleOverride** run shows that the model's blocks used a **double** data type during simulation.

- 3 On the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named **Analog Plant** in the **NoOverride** run, and then click the **Compare Signals** button .

The Fixed-Point Tool plots both the **DoubleOverride** and **NoOverride** versions of the signal associated with the plant output (upper axes), and plots the difference between the active and reference versions of that signal (lower axes). Compare the ideal (double data type) plant output signal with its fixed-point version.

Tip: From the Simulation Data Inspector menu bar, use the zoom tools to zoom in on an area.



Automatic Data Typing


Using the automatic data typing procedure, you can maximize the precision of the output data type while spanning the full simulation range.

Because no design range information is supplied, The Fixed-Point Tool uses the simulation range data that was collected during the simulation run for proposing data types. The **Safety margin for simulation min/max (%)** parameter value multiplies the “raw” simulation values. Setting this parameter to a value greater than 1 decreases the likelihood that an overflow will occur when fixed-point data types are being used. For more information about how the Fixed-Point Tool calculates data type proposals, see “How the Fixed-Point Tool Proposes Word Lengths” on page 39-50.

Because of the nonlinear effects of quantization, a fixed-point simulation produces results that are different from an idealized, doubles-based simulation. Signals in a fixed-point simulation can cover a larger or smaller range than in a doubles-based simulation. If the range increases enough, overflows or saturations could occur. A safety margin decreases this likelihood, but it might also decrease the precision of the simulation.

Note: When the maximum and minimum simulation values cover the full, intended operating range of your design, the Fixed-Point Tool yields meaningful automatic data typing results.

Perform automatic data typing for the **Controller** subsystem. This subsystem represents software running on the target, and requires optimization.

- 1 Under **Automatic data typing**, set the **Safety margin for simulation min/max (%)** parameter to 20. Use the default settings for all other parameters.
- 2 Click **Propose data types** .
- 3 In the **Propose Data Types** dialog box, select **DoubleOverride**, and then click **OK**.

The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose **Lock output data type setting against changes by the fixed-point tools** parameter is cleared.

The Fixed-Point Tool uses the minimum and maximum values stored in the **DoubleOverride** run to propose each block's data types such that the precision is maximized while the full range of simulation values is spanned. The tool displays the proposed data types on its **Contents** pane. Now, it displays the **Automatic Data Typing View** to provide information, such as **ProposedDT**, **ProposedMin**, and **ProposedMax**, which are relevant at this stage of the fixed-point conversion.

Tip: In the main toolbar, use the **Show** option to view the groups that must share data types. For more information, see “Main Toolbar”.

- 4 Review the scaling that the Fixed-Point Tool proposes. You can choose to accept the scaling proposal for each block. On the **Contents** pane, select the corresponding **Accept** check box. By default, the Fixed-Point Tool accepts all scaling proposals that differ from the current scaling. For this example, ensure that the **Accept** check


box associated with the `DoubleOverride` run is selected for each of the Controller subsystem's blocks.

- 5 In the Fixed-Point Tool, click the **Apply accepted data types** button .

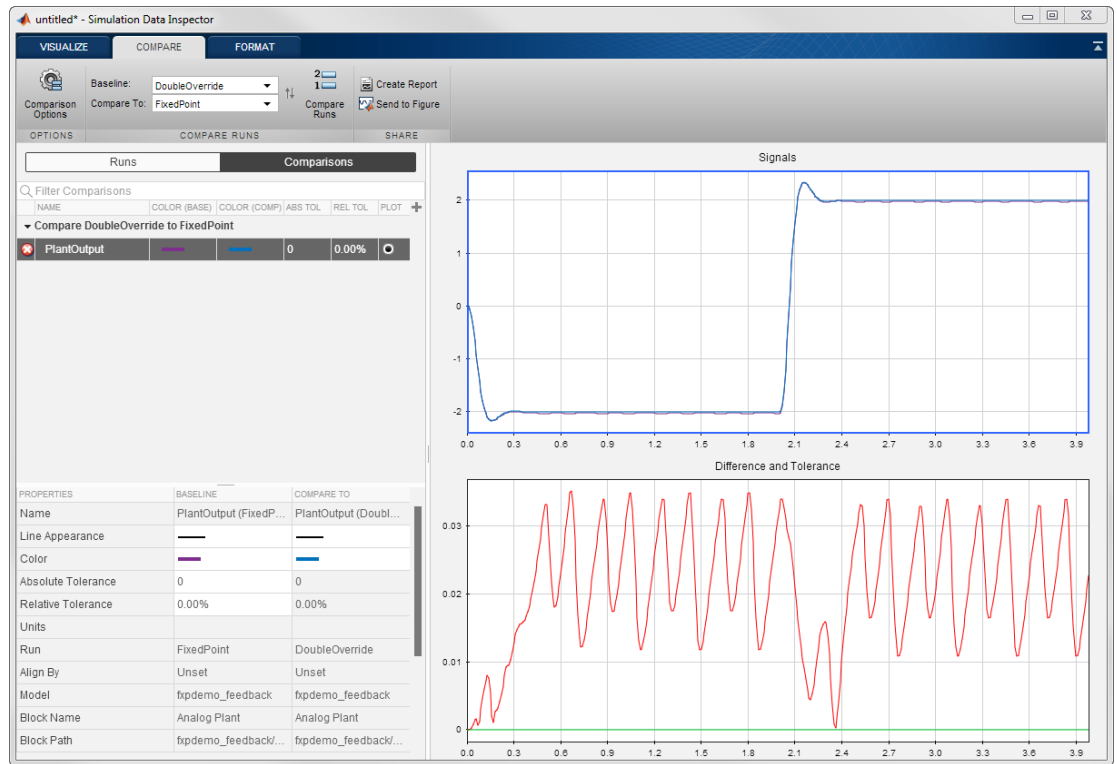
The Fixed-Point Tool applies the scaling proposals that you accepted in the previous step to the `Controller` subsystem's blocks.

- 6 Under **Configure model settings**, click the **Range collection with specified data types** button to collect ranges using the newly applied, locally specified data type settings.
- 7 Under **Range collection** pane, set **Run name** to `FixedPoint` so that the Fixed-Point Tool stores the results with a new run name and does not overwrite the results for the initial fixed-point set up. Storing the results in different runs allows you to compare the initial system behavior with the behavior of the autoscaled model.
- 8 In the Fixed-Point Tool, click **Simulate model**.

Simulink simulates the `fxpdemo_feedback` model using the new scaling that you applied. Afterward, in its **Contents** pane, the Fixed-Point Tool displays information about blocks that logged fixed-point data. The compiled data type (**CompiledDT**) column for the `FixedPoint` run shows that the Controller subsystem's blocks used fixed-point data types with the new scaling.

- 9 On the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system.
 - a On the **Contents** pane, select the Transfer Fcn block named `Analog Plant` for the `FixedPoint` run, and then click the **Compare Signals** button .
 - b In the **Compare Runs Selector** dialog box, select `DoubleOverride`, and then click **OK**.

The Fixed-Point Tool plots the fixed-point and double override versions of the plant output signal, as well as their difference.



The plant output signal represented by the fixed-point run achieves a steady state, but a small limit cycle is present because of poor A/D design.

Clean Up Settings

When you are satisfied with the results of the fixed-point conversion, under **Configure model settings**, click the **Remove overrides and disable range collection** button. This configuration cleans up settings after conversion to restore maximum simulation speed of your model.

Related Examples

- “Propose Word Lengths Based on Simulation Data” on page 39-52

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

How the Fixed-Point Tool Proposes Word Lengths

To use the Fixed-Point Tool to propose word lengths, you must specify the fraction length requirements for data types in the model. Select the fraction lengths based on the precision required for the system that you are modeling. If you do not specify fraction lengths, the Fixed-Point Tool sets the **Default fraction length** to four. The Fixed-Point Tool uses these specified fraction lengths to recommend the minimum word length for objects in the selected model or subsystem to avoid overflow for the collected range information.

The proposed word length is based on:

- Design range information and range information that the Fixed-Point Tool or Fixed-Point Advisor collects. This collected range information can be either simulation or derived range data.
- The signedness and fraction lengths of data types that you specify on blocks, signal objects.
- The production hardware implementation settings specified in the Configuration Parameters.

How the Fixed-Point Tool Uses Range Information

The Fixed-Point Tool determines whether to use different types of range information based on its availability and on the Fixed-Point Tool **When proposing types use** setting.

Design range information always takes precedence over both simulation and derived range data. When there is no design range information, the Fixed-Point Tool uses the union of available simulation and derived range data. If you specify a safety margin, the Fixed-Point Tool takes the safety margin into account.

For example, if a signal has a design range of $[-10, 10]$, the Fixed-Point Tool uses this range for the proposal and ignores all simulation and derived range information.

If the signal has no specified design information, but does have a simulation range of $[-8, 8]$ and a derived range of $[-2, 2]$, the proposal uses the union of the ranges, $[-8, 8]$. If you specify a safety margin of 50%, the proposal uses a range of $[-12, 12]$.

How the Fixed-Point Tool Uses Target Hardware Information

The Fixed-Point Tool calculates the ideal word length and then checks this length against the production hardware implementation settings for the target hardware.

- If the target hardware is an FPGA/ASIC, then the Fixed-Point Tool proposes the ideal word length. If the ideal word length is greater than 128, then the Fixed-Point Tool proposes 128.
- If the target hardware is an embedded processor, then the Fixed-Point Tool rounds the ideal word length up and proposes the nearest supported data type of your processor.

Related Examples

- “Propose Word Lengths Based on Simulation Data” on page 39-52

More About

- “Hardware Implementation Pane” (Simulink)


Propose Word Lengths Based on Simulation Data

In this section...

“How to Propose Word Lengths” on page 39-52

“Propose Word Lengths Based on Simulation Data” on page 39-52

How to Propose Word Lengths

- 1 Specify the target hardware. In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, specify the **Device vendor** and **Device type**, and then click **Apply**.
- 2 On the Fixed-Point Tool, under **Automatic data typing for selected system**, next to **Propose**, select **Word Length**. Optionally, select **Signedness** if you want the Fixed-Point Tool to propose signedness for objects in the system.
- 3 Set the **Default fraction length** for objects in your model.
- 4 On the same pane, specify the ranges to use for type proposals in the **When proposing types use** dropdown menu.
- 5 If you have safety margins to apply, set **Safety margin for simulation min/max (%)**.
- 6 Click the **Propose data types** button, .

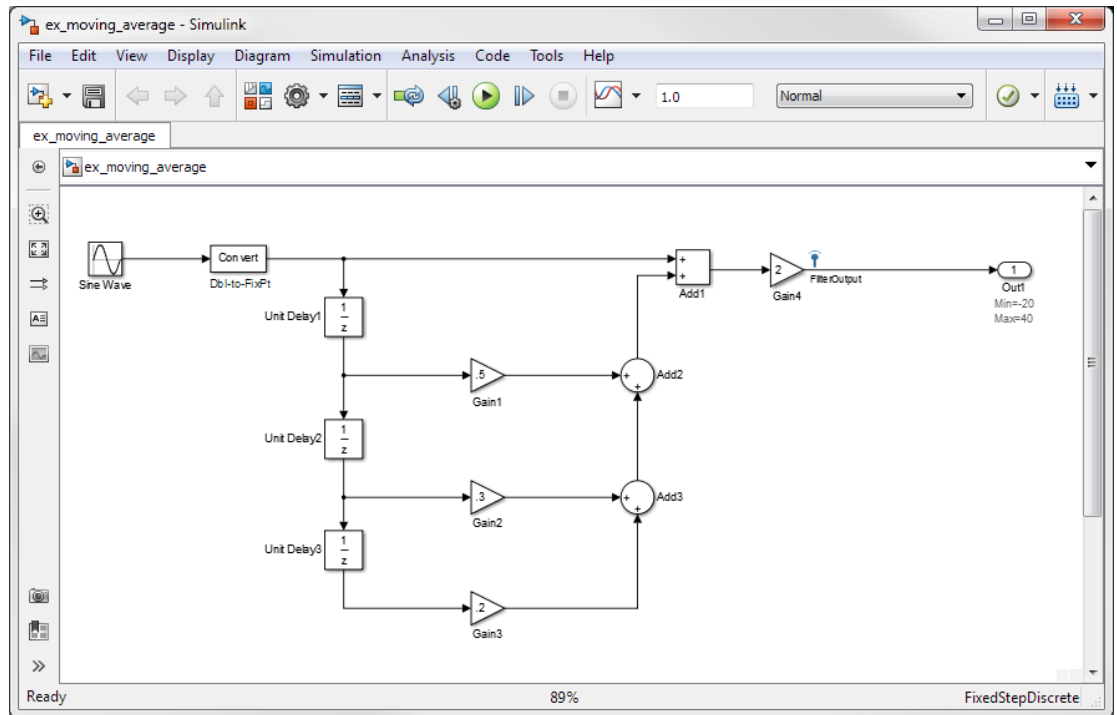
Note: When the Fixed-Point Tool proposes data types, it does not alter your model.

Propose Word Lengths Based on Simulation Data

This example shows how to use the Fixed-Point Tool to propose word lengths for a model that implements a simple moving average algorithm. The model already uses fixed-point data types, but they are not optimal. Simulate the model and propose data types based on simulation data. To see how the target hardware affects the word length proposals, first set the target hardware to an embedded processor and propose word lengths. Then, set the target hardware to an FPGA and propose word lengths.

- 1 Open the `ex_moving_average` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_moving_average
```




Some blocks in the model already have specified fixed-point data types.

Block	Data Type Specified on Block
Db12Fixpt	fixdt(1,16,10)
Gain1	fixdt(1,32,17)
Gain2	fixdt(1,32,17)
Gain3	fixdt(1,32,17)
Gain4	fixdt(1,16,1)
Add1	fixdt(1,32,17)
Add2	fixdt(1,32,17)
Add3	fixdt(1,32,17)

- 2 Verify that the target hardware is an embedded processor. In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the **Device vendor** to **Custom Processor**. Close the Configuration Parameters dialog box.
- 3 In the model, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 4 In the Fixed-Point Tool, under **System under design**, click **Continue** to set `ex_moving_average` as the system to convert.
- 5 Under **Configure model settings**, click the **Range collection using double override** button to set:
 - **Data type override** to `Double`
 - **Data type override applies to** to `All numeric types`
 - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`
 - The run name (in the **Run name** field under **Range collection**) to `DoubleOverride`

Using these settings, the Fixed-Point Tool performs a global override of the fixed-point data types with double-precision data types, avoiding quantization effects. During simulation, the tool logs minimum value, maximum value, and overflow data for all blocks in the system under design in the run `DoubleOverride`.

- 6 Click the Fixed-Point Tool **Simulate model** button  to run the simulation.

The Fixed-Point Tool simulates the model and displays the results on the **Contents** pane in the run named `DoubleOverride`.

Contents of: `ex_moving_average*` (mmo-dbl)

Column View: Simulation View [Show Details](#)

Name	Run	CompiledDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax
↳ Add1 : Accumulator	DoubleOverride	double	Inherit: Inherit via internal rule	0	10.086854311623828		
↳ Add1 : Output	DoubleOverride	double	fixdt(1,32,17)	0	10.086854311623828		
↳ Add2 : Accumulator	DoubleOverride	double	Inherit: Inherit via internal rule	0	5.038028402776449		
↳ Add2 : Output	DoubleOverride	double	fixdt(1,32,17)	0	5.038028402776449	-8	15
↳ Add3 : Accumulator	DoubleOverride	double	Inherit: Inherit via internal rule	0	2.5167860950974905		
↳ Add3 : Output	DoubleOverride	double	fixdt(1,32,17)	0	2.5167860950974905	-4	8
↳ Data Type Conversion1	DoubleOverride	double	fixdt(1,16,10)	0	5.048825908847379		
↳ Gain1	DoubleOverride	double	fixdt(1,32,17)	0	2.521242307678959		
↳ Gain2	DoubleOverride	double	fixdt(1,32,17)	0	1.5108372258966896		
↳ Gain3	DoubleOverride	double	fixdt(1,32,17)	0	1.005948869200801		
↳ Gain4	DoubleOverride	double	fixdt(1,16,1)	0	20.173708623247656		
↳ Out1	DoubleOverride		Inherit: auto			-20	40

7 Under **Automatic data typing**, next to **Propose**, select **Word Length**.

8 Click the **Propose data types** button.

The Fixed-Point Tool uses available range data to calculate data type proposals according to the following rules:

- Design minimum and maximum values take precedence over the simulation range.
- The tool observes the simulation range because you selected the **Simulation min/max** option.

The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. In this example, no safety margins are set.

The Fixed-Point Tool analyzes the data types of all fixed-point blocks whose **Lock output data type setting against changes by the fixed-point tools** parameter is cleared.

For each object in the model, the Fixed-Point Tool proposes the minimum word length that avoids overflow for the collected range information. Because the target hardware is a 16-bit embedded processor, the Fixed-Point tool proposes word lengths

based on the number of bits used by the processor for each data type. For more information, see “How the Fixed-Point Tool Uses Target Hardware Information” on page 39-51.

The tool proposes smaller word lengths for **Gain4** and **Gain4:Gain**. The tool calculated that their ideal word length is less than or equal to the character bit length for the embedded processor (8), so the tool rounds up the word length to 8.

Contents of: ex_moving_average* (mmo-dbl)

Column View: Automatic Data Typing View [Show Details](#)

Name	Run	DTGroup	Accept	ProposedDT	SpecifiedDT	CompiledDT	SimMin	SimMax	ProposedMin	ProposedMax
Data Type Conversion 1	DoubleOverride	G2	<input checked="" type="checkbox"/>	fixdt(0,16,10)	fixdt(1,16,10)	double	0	5.048825908847379	0	63.9990234375
Gain1	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,32,17)	fixdt(1,32,17)	double	0	2.521242307678959	0	32767.999992370605
Gain2	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,32,17)	fixdt(1,32,17)	double	0	1.5108372258966896	0	32767.999992370605
Gain3	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,32,17)	fixdt(1,32,17)	double	0	1.005948869200801	0	32767.999992370605
Add3 : Output	DoubleOverride		<input type="checkbox"/>	fixdt(1,32,17)	fixdt(1,32,17)	double	0	2.5167860950974905	-16384	16383.999992370605
Add3 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	2.5167860950974905		
Add2 : Output	DoubleOverride		<input type="checkbox"/>	fixdt(1,32,17)	fixdt(1,32,17)	double	0	5.038028402776449	-16384	16383.999992370605
Add2 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	5.038028402776449		
Add1 : Output	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,32,17)	fixdt(1,32,17)	double	0	10.086854311623828	0	32767.999992370605
Add1 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	10.086854311623828		
Gain4	DoubleOverride	G1	<input checked="" type="checkbox"/>	fixdt(1,8,1)	fixdt(1,16,1)	double	0	20.173708623247656	-64	63.5
Out1	DoubleOverride	G1		n/a	Inherit: auto					
Gain1 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule					
Gain2 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule					
Gain3 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule					
Gain4 : Gain	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,8,0)	fixdt(1,16,0)				0	255
Unit Delay1	DoubleOverride	G2		n/a						
Unit Delay2	DoubleOverride	G2		n/a						

- 9 To see how the target hardware affects the word length proposal, change the target hardware to FPGA/ASIC.
 - a In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set **Device vendor** to ASIC/FPGA.
 - b Click **Apply** and close the Configuration Parameters dialog box.
- 10 On the Fixed-Point Tool, under **Automatic data typing**, click **Propose data types**.

Because the target hardware is an FPGA, there are no constraints on the word lengths that the Fixed-Point Tool proposes. The word length for **Gain4:Gain** is now 2.

Contents of: ex_moving_average* (mmo-db)

Column View: Automatic Data Typing View [Show Details](#)

Name	Run	DTGroup	Accept	ProposedDT	SpecifiedDT	CompiledDT	SimMin	SimMax
Data Type Conversion1	DoubleOverride	G2	<input checked="" type="checkbox"/>	fixdt(0,13,10)	fixdt(1,16,10)	double	0	5.048825908847379
Gain1	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,19,17)	fixdt(1,32,17)	double	0	2.521242307678959
Gain2	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,18,17)	fixdt(1,32,17)	double	0	1.5108372258966896
Gain3	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,18,17)	fixdt(1,32,17)	double	0	1.005948869200801
Add3 : Output	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(1,22,17)	fixdt(1,32,17)	double	0	2.5167860950974905
Add3 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	2.5167860950974905
Add2 : Output	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(1,22,17)	fixdt(1,32,17)	double	0	5.038028402776449
Add2 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	5.038028402776449
Add1 : Output	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,21,17)	fixdt(1,32,17)	double	0	10.086854311623828
Add1 : Accumulator	DoubleOverride			n/a	Inherit: Inherit via internal rule	double	0	10.086854311623828
Gain4	DoubleOverride	G1	<input checked="" type="checkbox"/>	fixdt(1,8,1)	fixdt(1,16,1)	double	0	20.173708623247656
Out1	DoubleOverride	G1		n/a	Inherit: auto			
Gain1 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule			
Gain2 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule			
Gain3 : Gain	DoubleOverride			n/a	Inherit: Inherit via internal rule			
Gain4 : Gain	DoubleOverride		<input checked="" type="checkbox"/>	fixdt(0,2,0)	fixdt(1,16,0)			
Unit Delay1	DoubleOverride	G2		n/a				
Unit Delay2	DoubleOverride	G2		n/a				

Related Examples

- “Propose Fraction Lengths” on page 39-34

More About

- “How the Fixed-Point Tool Proposes Word Lengths” on page 39-50

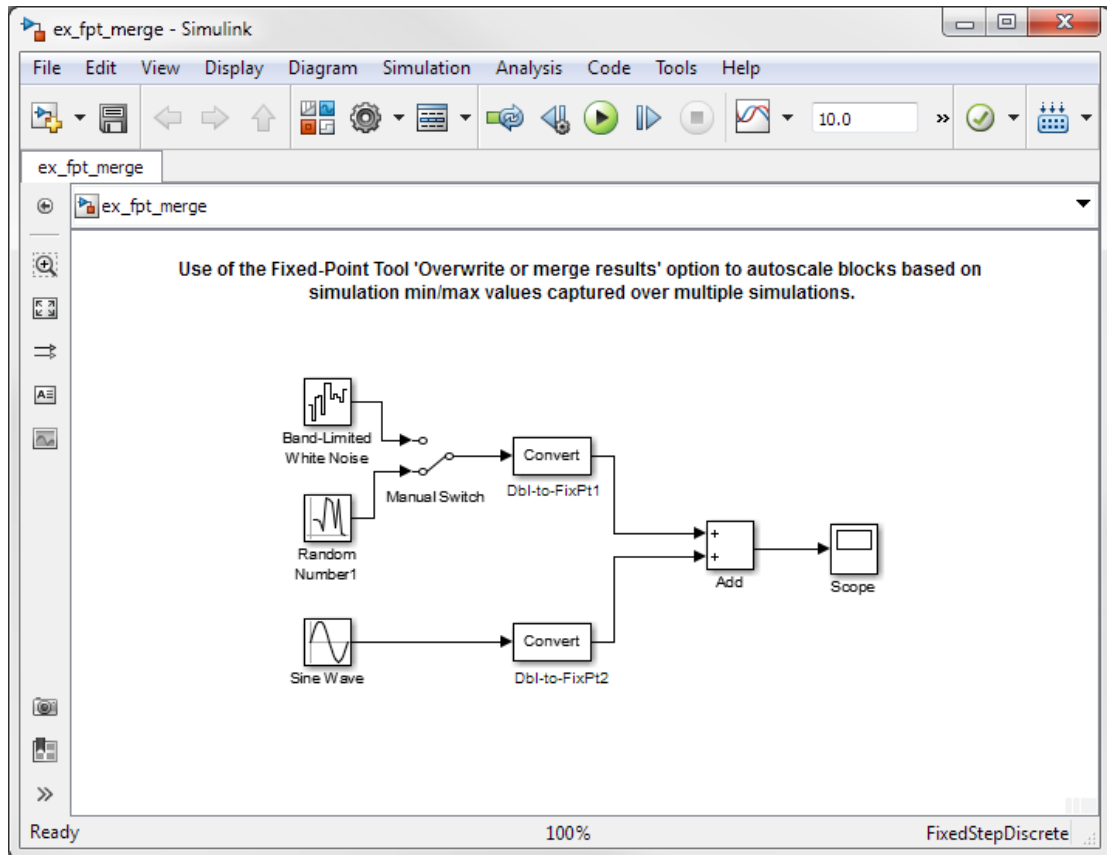
Propose Data Types Using Multiple Simulations

In this section...
“About This Example” on page 39-58
“Running the Simulation” on page 39-60

About This Example

This example shows how to use the Fixed-Point Tool to propose fraction lengths for a model based on the simulation minimum and maximum values captured over multiple simulations.

This example uses the `ex_fpt_merge` model.



About the Model

The model contains a sine wave input and two alternate noise sources, band-limited white noise and random uniform noise. The software converts the sine wave input and selected noise signal to fixed point and then adds them.

- The Data Type Conversion block `Dbl-to-FixPt1` converts the double-precision noise input to the fixed-point data type `fixdt(1, 16, 15)`.
- The Data Type Conversion block `Dbl-to-FixPt2` converts the double-precision sine wave input to the fixed-point data type `fixdt(1, 16, 10)`.
- The Add block **Accumulator data type** is `fixdt(1, 32, 30)` and **Output data type** is `fixdt(1, 16, 14)`.

Merging Results from Two Simulation Runs

In this example, you use the Fixed-Point Tool to merge the results from two simulation runs. Merging results allows you to autoscale your model over the complete simulation range.

- 1 “Simulate the Model Using Random Uniform Noise” on page 39-60. Using the Fixed-Point Tool, you simulate the model with the random uniform noise signal and observe the simulation minimum and maximum values for the Add block.

This run provides the simulation results for the random uniform noise input only.

- 2 “Simulate Model Using Band-Limited White Noise” on page 39-61. You select the band-limited white noise signal and run another simulation using the same Fixed-Point Tool simulation settings. The Fixed-Point Tool overwrites the results of the previous run.

This run provides the simulation range for the band-limited white noise input only.

- 3 “Merge Results” on page 39-61. You configure the Fixed-Point Tool to merge results. Select the random uniform noise input again, rerun the simulation, and observe the simulation results for the Add block.

This run provides the simulation range based on the entire set of input data for both noise sources.


- 4 “Propose Fraction Lengths Based on Merged Results” on page 39-62. The Fixed-Point Tool uses the merged simulation minimum and maximum values to propose scaling for each block to ensure maximum precision while spanning the full range of simulation values.

Running the Simulation

Simulate the Model Using Random Uniform Noise

- 1 Open the `ex_fpt_merge` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_fpt_merge
```
- 2 In the model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to set `ex_fpt_merge` as the system to convert.

- 4 Under **Configure model settings**, click the **Range collection using double override** button to override the model's data types with doubles to observe the ideal numeric behavior of the model and enable range collection.
- 5 In the Fixed-Point Tool, click the **Simulate model** button .

Simulink simulates the `ex_fpt_merge` model, using the random uniform noise signal. Afterward, the Fixed-Point Tool **Contents** pane displays the simulation results for each block that logged fixed-point data. The tool stores the results in a run named `DoubleOverride`.

- 6 The **SimMin** and **SimMax** values for the Add block are:

SimMin is -3.5822

SimMax is 2.7598

Simulate Model Using Band-Limited White Noise

- 1 In the model, double-click the switch to select the band-limited white noise signal.
- 2 In the Fixed-Point Tool, click the **Simulate** button.

Simulink simulates the `ex_fpt_merge` model, now using the band-limited white noise signal.

- 3 The changed values for **SimMin** and **SimMax** for the Add block are:

SimMin is now -2.5317

SimMax is now 3.1542

Merge Results

- 1 In the model, double-click the switch to select the random uniform noise signal.
- 2 In the Fixed-Point Tool, under **Range collection**, select **Merge results from multiple simulations**, click **Apply** and rerun the simulation.
- 3 The **SimMin** and **SimMax** values for the Add block now cover the entire simulation range for both the random uniform and band-limited white noise signals.

SimMin is -3.5822

SimMax is 3.1542

Propose Fraction Lengths Based on Merged Results

- Under **Automatic data typing**, click the **Propose data types** button.

The Fixed-Point Tool analyzes the data types of all fixed-point blocks whose **Lock output data type setting against changes by the fixed-point tools** parameter is cleared.

The Fixed-Point Tool uses the merged minimum and maximum values to propose fraction lengths for each block. These values ensure maximum precision while spanning the full range of simulation values. The tool displays the proposed data types in the **Contents** pane.

Name	Run	Accept	ProposedDT	SpecifiedDT	CompiledDT	SimMin	SimMax	ProposedMin	ProposedMax
Add : Accumulator	DoubleOverride	<input checked="" type="checkbox"/>	fixdt(1,32,29)	fixdt(1,32,30)	double	-3.58218376...	3.15421092...	-4	3.99999999...
Add : Output	DoubleOverride	<input checked="" type="checkbox"/>	fixdt(1,16,13)	fixdt(1,16,14)	double	-3.58218376...	3.15421092...	-4	3.99987792...
Band-Limited White Noise/Output	DoubleOverride		n/a	Inherit: Same as input	double	-1.60588986...	2.18629125...		
Band-Limited White Noise/Output : Gain	DoubleOverride		n/a	Inherit: Same as input					
Band-Limited White Noise/y	DoubleOverride		n/a	Inherit: auto					
Data Type Conversion	DoubleOverride	<input checked="" type="checkbox"/>	fixdt(1,16,13)	fixdt(1,16,15)	double	-2.89441760...	2.18629125...	-4	3.99987792...
Data Type Conversion1	DoubleOverride	<input checked="" type="checkbox"/>	fixdt(1,16,15)	fixdt(1,16,10)	double	-0.99992325...	0.99957360...	-1	0.99996948...
Manual Switch	DoubleOverride		n/a		double	-2.89441760...	2.18629125...		

More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

View Simulation Results

In this section...

“Compare Runs” on page 39-63


“Compare Signals” on page 39-64

“Inspect Signals” on page 39-65

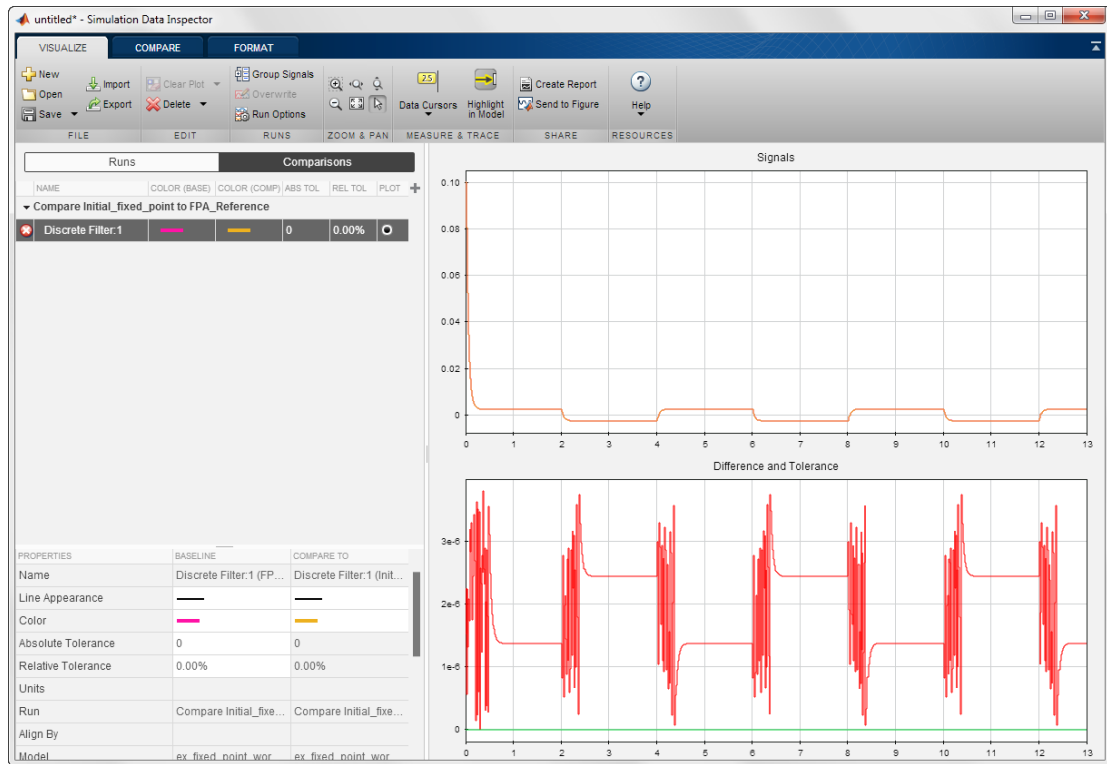
“Histogram Plot of Signal” on page 39-66

Compare Runs

To compare runs:


- 1 In one of the runs that you want to compare, select a logged signal.
- 2 From the Fixed-Point Tool menu, select **Results > Compare Runs** or click .
- 3 If there are more than two runs, in the **Compare Runs Selector** dialog box, select the run that you want to compare, and then click **OK**.

On the upper axes, the Simulation Data Inspector plots the signal in both selected runs. On the lower axes, the Simulation Data Inspector plots the difference between those runs.

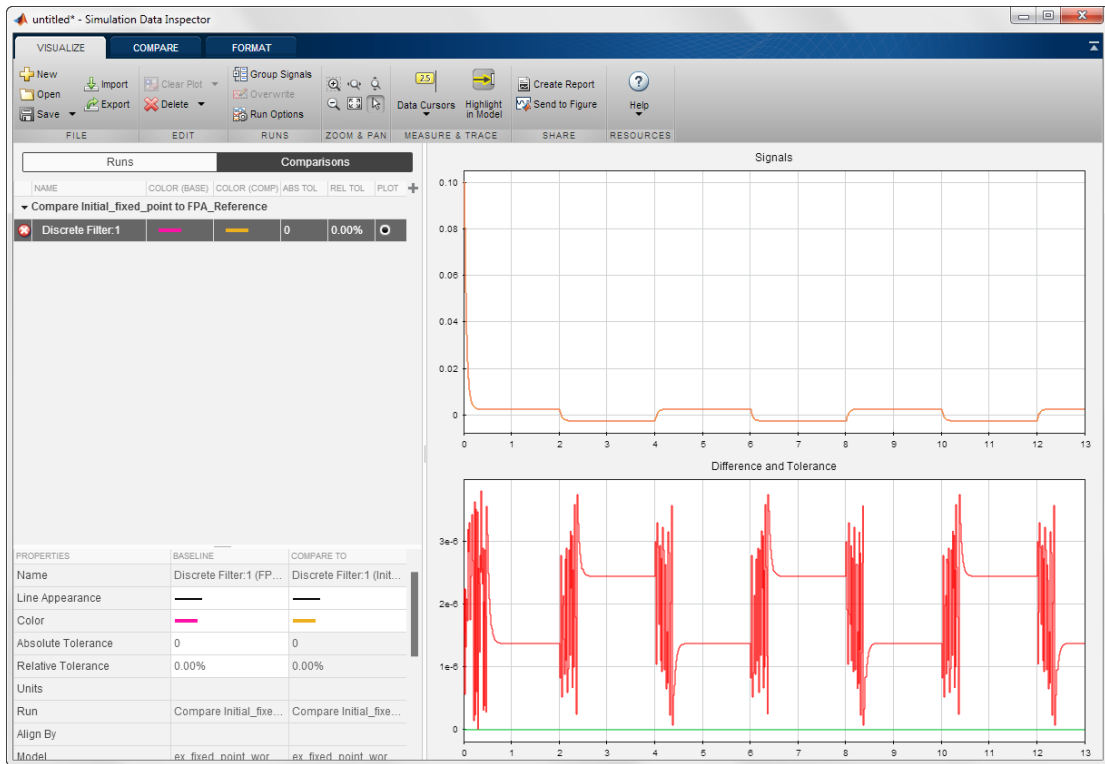


Compare Signals

To compare signals:


- 1 In one of the runs that you want to compare, select a logged signal.
- 2 From the Fixed-Point Tool menu, select **Results > Compare Signals** or click .
- 3 If there are more than two runs, in the **Compare Runs Selector** dialog box, select the run that you want to compare, and then click **OK**.

On the upper axes, the Simulation Data Inspector plots the signal in both selected runs. On the lower axes, the Simulation Data Inspector plots the difference between those runs.

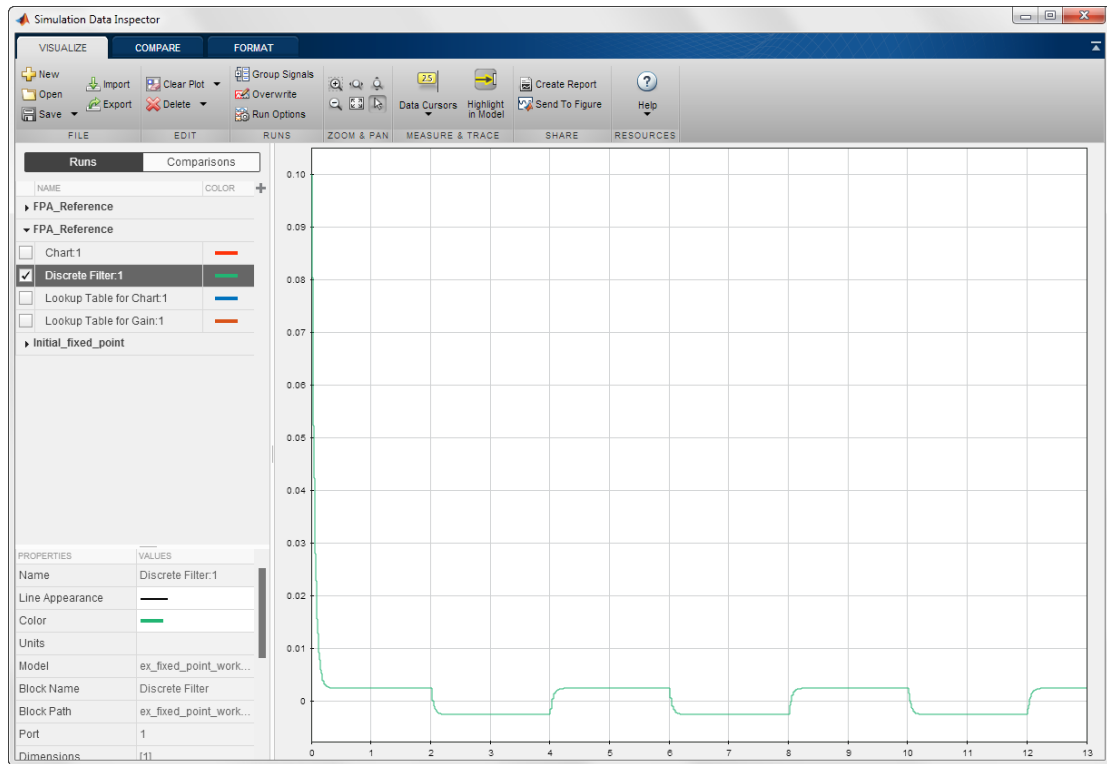


Inspect Signals

To inspect a signal:

- 1 Select the logged signal that you want to inspect.
- 2 From the Fixed-Point Tool menu, select **Results > Inspect Signal** or click .

The Simulation Data Inspector plots data as a function of time.



Histogram Plot of Signal

To view the histogram plot of a signal:

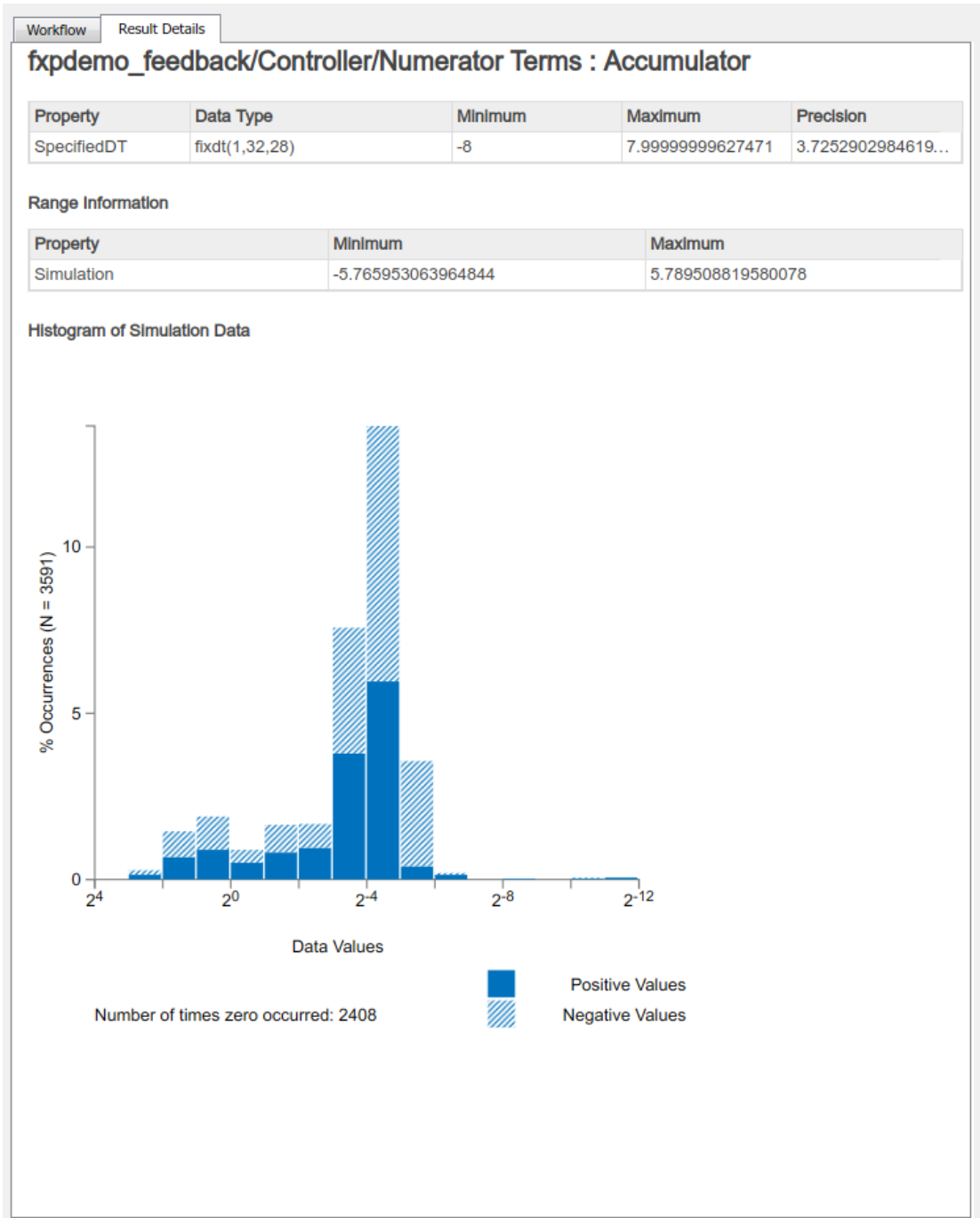
- 1 Select the logged signal that you want to plot.
- 2 Click the **Result Details** tab.

The histogram plot helps you visualize the dynamic range of a signal. It provides information about the:

- Total number of samples (N).
- Maximum number of bits to prevent overflow.
- Number of times each bit has represented the data (as a percentage of the total number of samples).

- Number of times that exact zero occurred (without the effect of quantization). This number does not include the number of zeroes that occurred due to rounding.

You can use this information to estimate the word size required to represent the signal.



Related Examples

- “Viewing Results With the Simulation Data Inspector” on page 39-70
- “Propose Fraction Lengths Using Simulation Range Data” on page 39-41

Viewing Results With the Simulation Data Inspector

Why Use the Simulation Data Inspector

Using the Simulation Data Inspector to inspect and compare data after converting your floating-point model to fixed point facilitates tracking numerical error propagation.

When to Use the Simulation Data Inspector

Use the Simulation Data Inspector to:

- Plot multiple signals in one or more axes
- Compare a signal in different runs
- Compare all logged signal data from different runs
- Export signal logging results to a MAT-file
- Specify tolerances for signal comparison
- Create a report of the current view and data in the Simulation Data Inspector

What You Can Inspect in the Simulation Data Inspector

The Fixed-Point Tool uses the Simulation Data Inspector (Simulink) tool plotting capabilities that enable you to plot signals for graphical analysis. The tool can access signal data that resides in the MATLAB workspace, allowing you to plot simulation results associated with:

- Scope blocks whose **Save data to workspace** parameter is selected
- To Workspace blocks
- Root-level Outport blocks, when the **Output** check box on the **Data Import/Export** pane of the Configuration Parameters dialog box is selected
- Logged signal data

Tip: The **Contents** pane of the Fixed-Point Tool displays an antenna icon  next to items that you can plot.

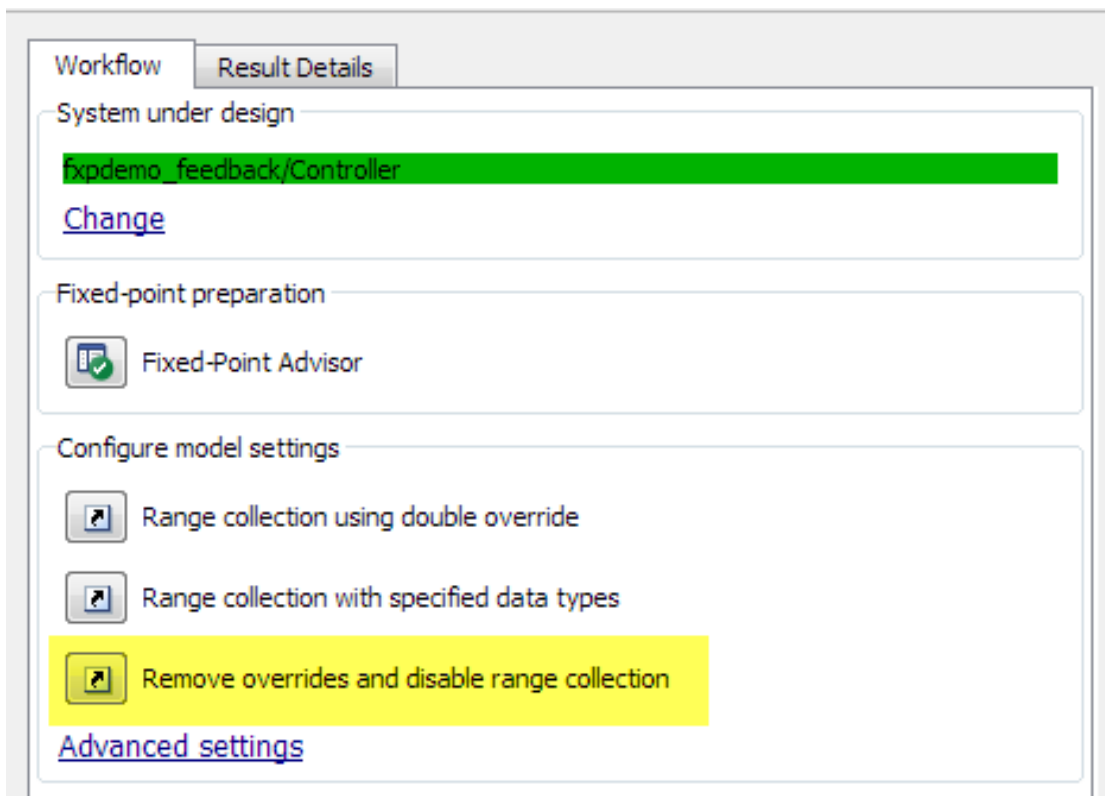
Related Examples

- “View Simulation Results” on page 39-63

Fixed-Point Instrumentation and Data Type Override

The conversion of a model from floating point to fixed point requires configuring fixed-point instrumentation and data type overrides. However, leaving these settings on after the conversion can lead to unexpected results.

To turn off instrumentation and data type override for your entire model, open the Fixed-Point Tool, and under **Configure model settings**, click the **Remove overrides and disable range collection** button.



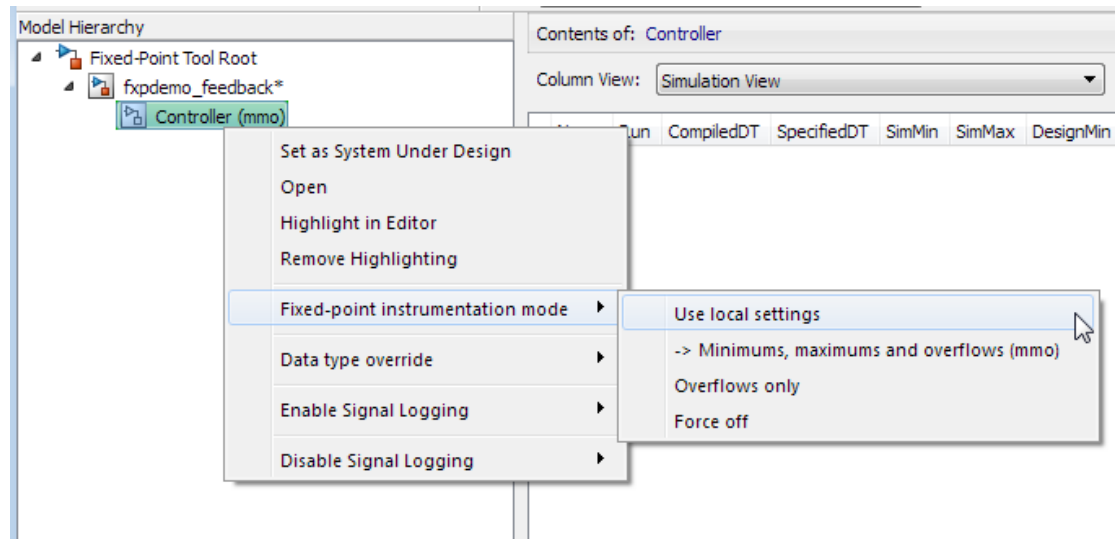
Control Instrumentation Settings

The fixed-point instrumentation mode controls which objects log minimum, maximum, and overflow data during simulation. Instrumentation is required to collect simulation

ranges using the fixed-point tool. These ranges are used to propose data types for the model. When you are not actively converting your model to fixed point, disable the fixed-point instrumentation to restore the maximum simulation speed to your model.

- 1 From the model **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 2 Under **System under design**, click **Continue**.
- 3 In the Model Hierarchy pane, the Fixed-Point Tool denotes systems that currently have instrumentation turned on with (mmo), or (o). Right-click the system in the model hierarchy and, under **Fixed-point instrumentation mode**, select the mode you want your model to use.

For the maximum simulation speed when you are not actively converting your model, select **Use local settings** or **Force off**.

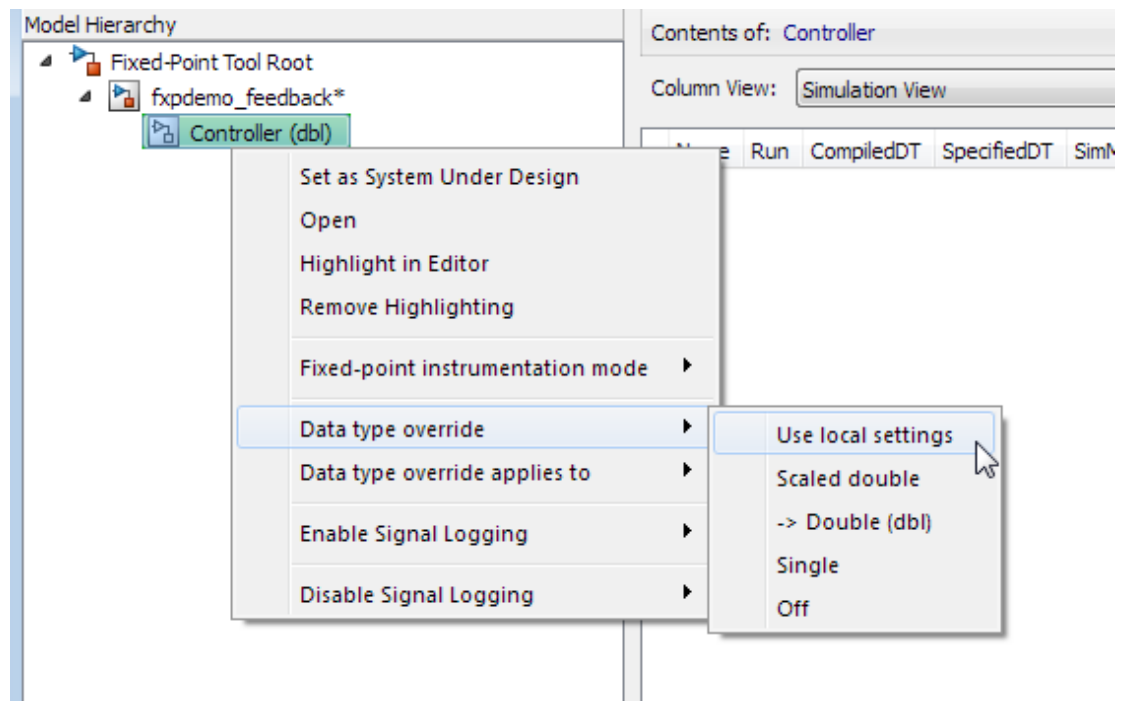


Control Data Type Override

Use data type override to simulate your model using double, single, or scaled double data types. If you do not have Fixed-Point Designer software, you can still configure data type override settings to simulate a model that specifies fixed-point data types. Using this setting, the software temporarily overrides fixed-point data types with floating-point data types during simulation. Turn off data type override to observe the true behavior of your model.

- 1 From the model **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 2 Under **System under design**, click **Continue**.
- 3 In the Model Hierarchy pane, the Fixed-Point Tool denotes systems that currently have data type override turned on with (dbl), (sgl), or (scl). Right-click the system in the Model Hierarchy pane and under **Data type override**, select the mode you want your model to use.

To observe the true behavior of your model, set data type override to **Use local settings** or **Force off**.



More About

- “Autoscaling Using the Fixed-Point Tool” on page 39-12

Range Analysis

- “How Range Analysis Works” on page 40-2
- “Derive Ranges” on page 40-7
- “Derive Ranges at the Subsystem Level” on page 40-9
- “View Derived Ranges in the Fixed-Point Tool” on page 40-11
- “Derive Ranges Using Design Ranges” on page 40-13
- “Derive Ranges Using Block Initial Conditions” on page 40-16
- “Derive Ranges for Simulink.Parameter Objects” on page 40-19
- “Insufficient Design Range Information” on page 40-22
- “Troubleshoot Range Analysis of System Objects” on page 40-25
- “Providing More Design Range Information” on page 40-28
- “Fixing Design Range Conflicts” on page 40-31
- “Derive Ranges for a Referenced Model” on page 40-34
- “Propose Data Types for a Referenced Model” on page 40-40
- “Deriving Ranges for a Referenced Model” on page 40-41
- “Intermediate Range Results” on page 40-43
- “Unsupported Simulink Software Features” on page 40-46
- “Simulink Blocks Supported for Range Analysis” on page 40-48

How Range Analysis Works

In this section...

“Analyzing a Model with Range Analysis” on page 40-2

“Automatic Stubbing” on page 40-5

“Model Compatibility with Range Analysis” on page 40-6

Analyzing a Model with Range Analysis

The model that you want to analyze **must** be compatible with range analysis. If your model is not compatible, either replace unsupported blocks or divide the model so that you can analyze the parts of the model that are compatible. For more information, see “Model Compatibility with Range Analysis” on page 40-6.

The Fixed-Point Designer software performs a static range analysis of your model to derive minimum and maximum range values for signals in the model. The software analyzes the model behavior and computes the values that can occur during simulation for each block Output. The range of these values is called a *derived range*.

The software statically analyzes the ranges of the individual computations in the model based on:

- Specified design ranges, known as *design minimum and maximum* values, for example, minimum and maximum values specified for:
 - Inport and Outport blocks
 - Block outputs
 - Input, output, and local data used in MATLAB Function and Stateflow Chart blocks
 - Simulink data objects (`Simulink.Signal` and `Simulink.Parameter` objects)
- Inputs
- The semantics of each calculation in the blocks

If the model contains objects that the analysis cannot support, where possible, the software uses automatic stubbing. For more information, see “Automatic Stubbing” on page 40-5.

The range analysis tries to narrow the derived range by using all the specified design ranges in the model. The more design range information you specify, the more likely the range analysis is to succeed. As the software performs the analysis, it derives new range information for the model. The software then attempts to use this new information, together with the specified ranges, to derive ranges for the remaining objects in the model.

For models that contain floating-point operations, range analysis might report a range that is slightly larger than expected. This difference is due to rounding errors because the software approximates floating-point numbers with infinite-precision rational numbers for analysis and then converts to floating point for reporting.

The following table summarizes how the analysis derives range information and provides links to examples.

When...	How the Analysis Works	Examples
<p>You specify design minimum and maximum data for a block output.</p>	<p>The derived range at the block output is based on these specified values and on the following values for blocks connected to its inputs and outputs:</p> <ul style="list-style-type: none"> • Specified minimum and maximum values • Derived minimum and maximum values 	<p>“Derive Ranges Using Design Ranges” on page 40-13</p>
<p>A parameter on a block has initial conditions and a design range.</p>	<p>The analysis takes both factors into account by taking the union of the design range and the initial conditions.</p>	<p>“Derive Ranges Using Block Initial Conditions” on page 40-16</p>
<p>The model contains a parameter with a specified range and the parameter storage class is set to Auto.</p>	<p>The analysis does not take into account the range specified for the parameter. Instead, it uses the parameter value.</p>	<p>“Derive Ranges for Simulink.Parameter Objects” on page 40-19</p>
<p>The model contains a parameter with a specified range and the parameter storage class is not set to Auto.</p>	<p>The analysis takes into account the range specified for the parameter and ignores the value.</p>	<p>“Derive Ranges for Simulink.Parameter Objects” on page 40-19</p>
<p>The model contains insufficient design range information.</p>	<p>The analysis cannot determine derived ranges. Specify more design range information and rerun the analysis.</p>	<p>“Troubleshoot Range Analysis of System Objects” on page 40-25</p> <p>The analysis results might depend on block sorting order which determines the order in which the</p>

When...	How the Analysis Works	Examples
		software analyzes the blocks. For more information, see “Control and Display the Sorted Order” (Simulink).
The model contains conflicting design range information.	The analysis cannot determine the derived minimum or derived maximum value for an object. The Fixed-Point Tool generates an error. To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent.	“Fixing Design Range Conflicts” on page 40-31

Automatic Stubbing

What is Automatic Stubbing?

Automatic stubbing is when the software considers only the interface of the unsupported objects in a model, not their actual behavior. Automatic stubbing lets you analyze a model that contains objects that the Fixed-Point Designer software does not support. However, if any unsupported model element affects the derivation results, the analysis might achieve only partial results.

How Automatic Stubbing Works

With automatic stubbing, when the range analysis comes to an unsupported block, the software ignores ("stubs") that block. The analysis ignores the behavior of the block. As a result, the block output can take any value.

The software cannot “stub” all Simulink blocks, such as the Integrator block. See the blocks marked “not stubbable” in “Simulink Blocks Supported for Range Analysis” on page 40-48.

Model Compatibility with Range Analysis

To verify that your model is compatible with range analysis, see:

- “Unsupported Simulink Software Features” on page 40-46
- “Simulink Blocks Supported for Range Analysis” on page 40-48
- “Limitations of Support for Model Blocks” on page 40-58

Derive Ranges

- 1 Verify that your model is compatible with range analysis on page 40-6. See “Model Compatibility with Range Analysis” on page 40-6.
- 2 In Simulink, open your model and set it up for use with the Fixed-Point Tool. For more information, see Set Up the Model on page 39-14.
- 3 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 4 In the Fixed-Point Tool, under **System under design**, select the system or subsystem of interest.
- 5 Optionally, you can use the Fixed-Point Advisor to prepare the model for conversion.
 - a Under **Fixed-point preparation for selected system**, click the **Fixed-Point Advisor** button.
 - b Run each task in the Fixed-Point Advisor. For more information, see “Preparation for Fixed-Point Conversion” on page 35-2.

The Fixed-Point Advisor:

- Checks the model against fixed-point guidelines.
 - Identifies unsupported blocks.
 - Allows you to promote simulation minimum and maximum values to design minimum and maximum values. This capability is useful if you have not specified design ranges and you have simulated the model with inputs that cover the full intended operating range. For more information, see “Specify block minimum and maximum values” on page 45-27.
- 6 Under **Configure model settings**, click the **Range collection using double override** button.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 7 Optionally, under **Range collection**, specify a run name. Specifying a unique run name avoids overwriting results from previous runs.
- 8 In the Fixed-Point Tool, click the **Derive ranges for System Under Design** button.

The analysis runs and tries to derive range information for objects in the selected system. Your next steps depend on the analysis results.

Analysis Results	Fixed-Point Tool Behavior	Next Steps	For More Information
Successfully derives range data for the model.	Displays the derived minimum and maximum values for the blocks in the selected system.	Review the derived ranges to determine if the results are suitable for proposing data types. If not, you must specify additional design information and rerun the analysis.	“View Derived Ranges in the Fixed-Point Tool” on page 40-11
Fails because the model contains blocks that the software does not support.	Generates an error and provides information about the unsupported blocks.	To fix the error, review the error message information and replace the unsupported blocks.	“Model Compatibility with Range Analysis” on page 40-6
Cannot derive range data because the model contains conflicting design range information.	Generates an error.	To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent.	“Fixing Design Range Conflicts” on page 40-31
Cannot derive range data for an object because there is insufficient design range information specified on the model.	Highlights the results for the object.	Examine the model to determine which design range information is missing.	“Troubleshoot Range Analysis of System Objects” on page 40-25

Derive Ranges at the Subsystem Level

In this section...

“Deriving Ranges at the Subsystem Level” on page 40-9

“Derive Ranges at the Subsystem Level” on page 40-10

Deriving Ranges at the Subsystem Level

You can derive range information for individual atomic subsystems and atomic charts. When you derive ranges at the model level, the software takes into account all information in the scope of the model. When you derive ranges at the subsystem level only, the software treats the subsystem as a standalone unit and the derived ranges are based on only the local design range information specified in the subsystem or chart. Therefore, when you derive ranges at the subsystem level, the analysis results might differ from the results of the analysis at the model level.

For example, consider a subsystem that has an input with a design minimum of -10 and a design maximum of 10 that is connected to an input signal with a constant value of 1. When you derive ranges at the model level, the range analysis software uses the constant value 1 as the input. When you derive ranges at the subsystem level, the range analysis software does not take the constant value into account and instead uses [-10..10] as the range.

When to Derive Ranges at the Subsystem Level

Derive ranges at the subsystem level to facilitate:

- System validation

It is a best practice to analyze individual subsystems in your model one at a time. This practice makes it easier to understand the atomic behavior of the subsystem. It also makes debugging easier by isolating the source of any issues.

- Calibration

The results from the analysis at subsystem level are based only on the settings specified within the subsystem. The proposed data types cover the full intended design range of the subsystem. Based on these results, you can determine whether you can reuse the subsystem in other parts of your model.

Derive Ranges at the Subsystem Level

The complete procedure for deriving ranges is described in “Derive Ranges” on page 40-7.

To derive ranges at the subsystem level, the key points to remember are:

- The subsystem or subchart must be atomic.
- In the Fixed-Point Tool, under **System under design**, select the system or subsystem of interest.
- Under **Configure model settings**, click the **Range collection using double override** button.

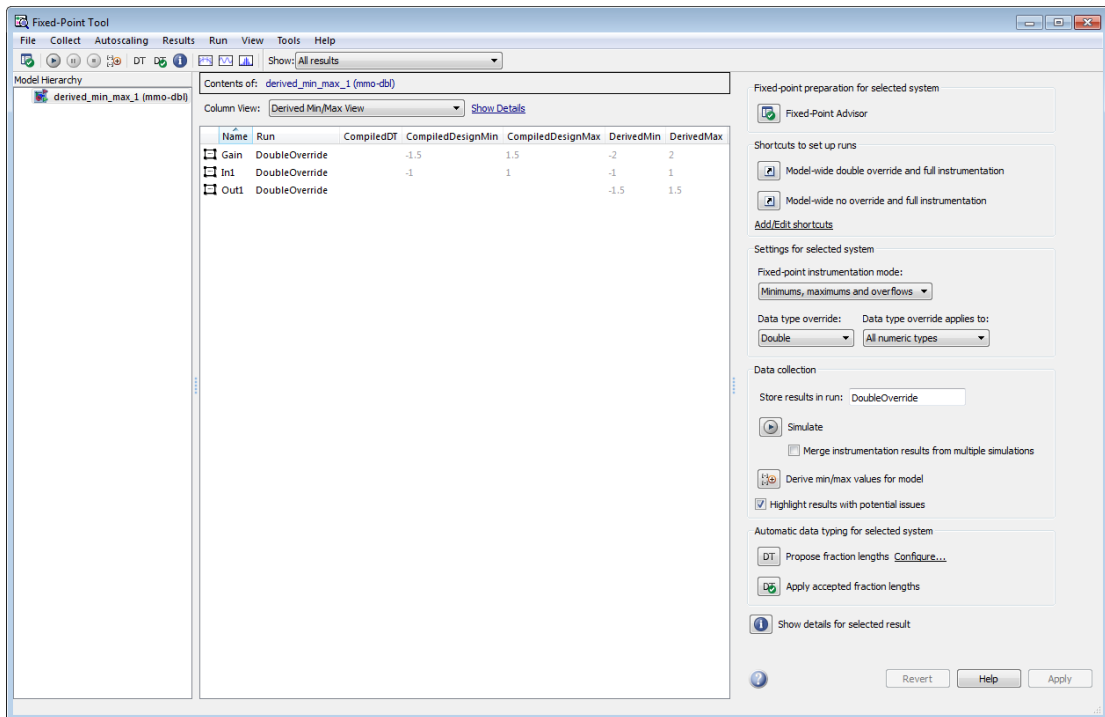
Tip: If the parent of the selected subsystem controls the data type override setting of the subsystem, first set the parent **Data type override** to **Use local settings** and then set the subsystem **Data type override** to **Double**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- Under **Range collection**, specify a run name. Specifying a unique run name avoids overwriting results from previous runs. This run contains derived minimum and maximum values that take into account the full intended design range of the subsystem.

View Derived Ranges in the Fixed-Point Tool

After you use the Fixed-Point Tool to derive ranges for a model, the Fixed-Point Tool **Contents** pane displays the derived minimum and maximum values for each object in the selected system.



If the analysis cannot derive a minimum or maximum value, the Fixed-Point Tool highlights the result. To fix the issue, examine the model to identify which objects have no specified design ranges and add this information. See “Insufficient Design Range Information” on page 40-22.

Contents of: ex_derived_min_max_4* (mmo-dbl)

Column View: [Show Details](#)

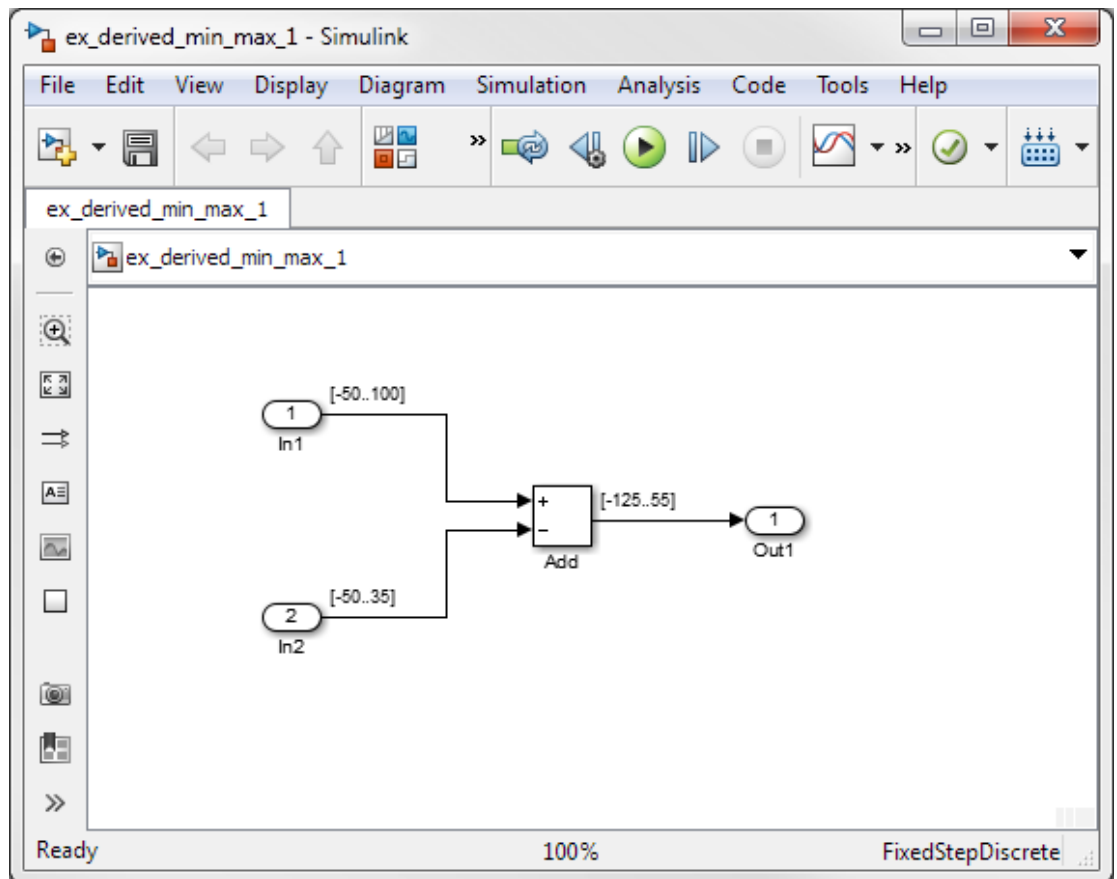
	Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
▶	In1	DoubleOverride	double	-1		-1	Inf
▶	Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
▶	Out1	DoubleOverride				-1.5	1.5

Derive Ranges Using Design Ranges

This example shows how the range analysis narrows the derived range for the Output block. This range is based on the range derived for the **Add** block using the design ranges specified on the two Inport blocks and the design range specified for the Add block.

- 1 Open the `ex_derived_min_max_1` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_derived_min_max_1
```



The model displays the specified design minimum and maximum values for each block.

- In1 design range is [-50 . . 100].
- In2 design range is [-50 . . 35].
- Add block design range is [-125 . . 55].

Tip: To display design ranges in your model, from the model menu, select **Display > Signals & Ports** and select **Design Ranges**.

- 2 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to select `ex_derived_min_max_1` as the system you want to convert.
- 4 Under **Configure model settings**, click **Range collection using double override**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 5 Under **Range collection**, click the **Derive ranges for System Under Design** button.

To calculate the derived range at the Add block input, the software uses the design minimum and maximum values specified for the Inport blocks, [-50 . . 100] and [-50 . . 35]. The derived range at the Add block input is [-85 . . 150].

In the **Contents** pane, the Fixed-Point Tool displays the derived and design minimum and maximum values for the blocks in the selected system.

- The derived range for the Add block output signal is narrowed to [-85 . . 55]. This derived range is the intersection of the range derived from the block inputs, [-85 . . 150], and the design minimum and maximum values specified for the block output, [-125 . . 55].
- The derived range for the Outport block Out1 is [-85 . . 55], the same as the Add block output.

Related Examples

- “Insufficient Design Range Information” on page 40-22

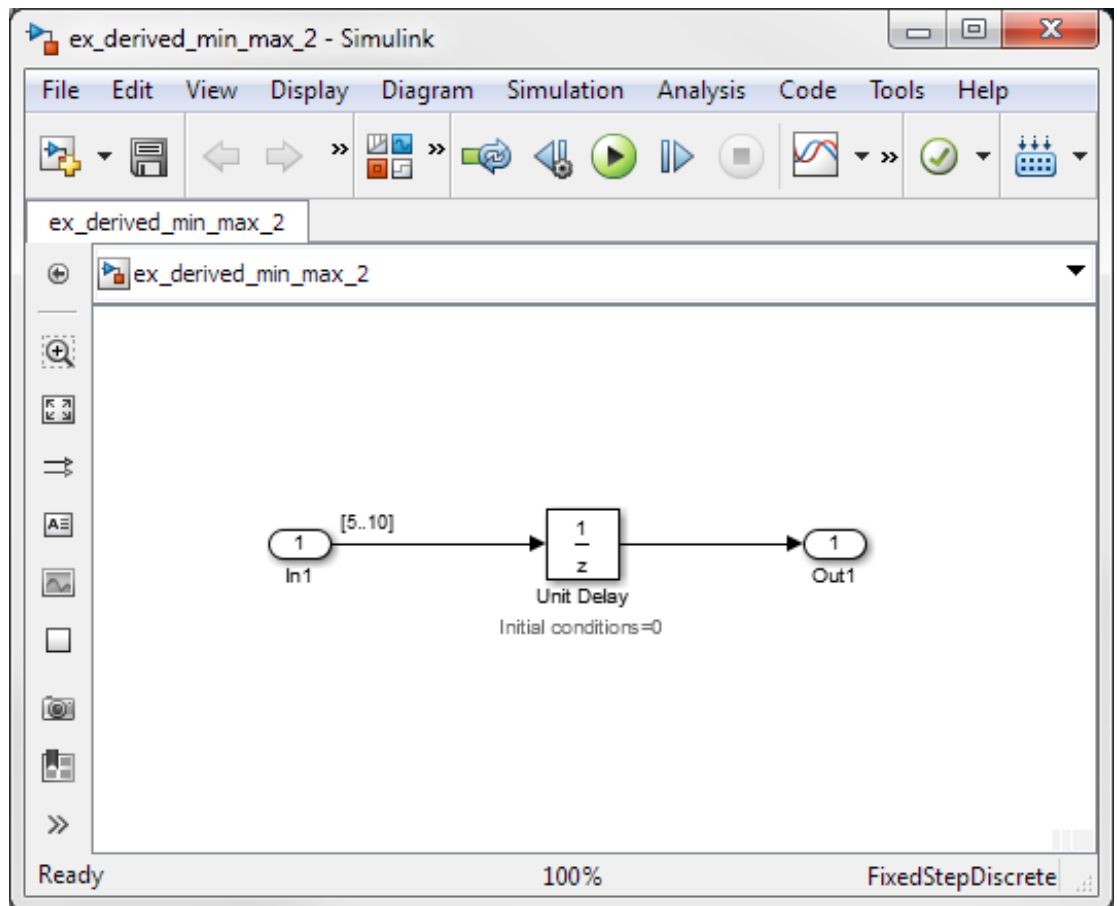
- “Troubleshoot Range Analysis of System Objects” on page 40-25

Derive Ranges Using Block Initial Conditions

This example shows how range analysis takes into account block initial conditions.

- 1 Open the `ex_derived_min_max_2` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_derived_min_max_2
```



The model uses block annotations to display the specified design minimum and maximum values for the Inport block and the initial conditions for the Unit Delay block.

- In1 design range is [5 . . 10].
- Unit Delay block initial condition is 0.

Tip: To display design ranges in your model, from the model menu, select **Display > Signals & Ports** and select **Design Ranges**.

- 2** In the model, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 3** Under **System under design**, click **Continue** to select `ex_derived_min_max_2` as the system you want to convert.
- 4** Under **Configure model settings**, click the **Range collection using double override** button.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 5** Under **Range collection**, click the **Derive ranges for System Under Design** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model.

The derived minimum and maximum range for the Outport block, `Out1`, is [0 . . 10]. The range analysis derives this range by taking the union of the initial value, 0, on the Unit Delay block and the design range on the block, [5 . . 10].

- 6** Change the initial value of the Unit Delay block to 7.
 - a** In the model, double-click the Unit Delay block.
 - b** In the **Block Parameters** dialog box, set **Initial condition** to 7, then click **OK**.
 - c** In the Fixed-Point Tool, click the **Derive ranges for System Under Design** button.

Because the analysis takes the union of the initial conditions, 7, and the design range, [5 . . 10], on the Unit Delay block, the derived range for the block is now [5 . . 10].

More About

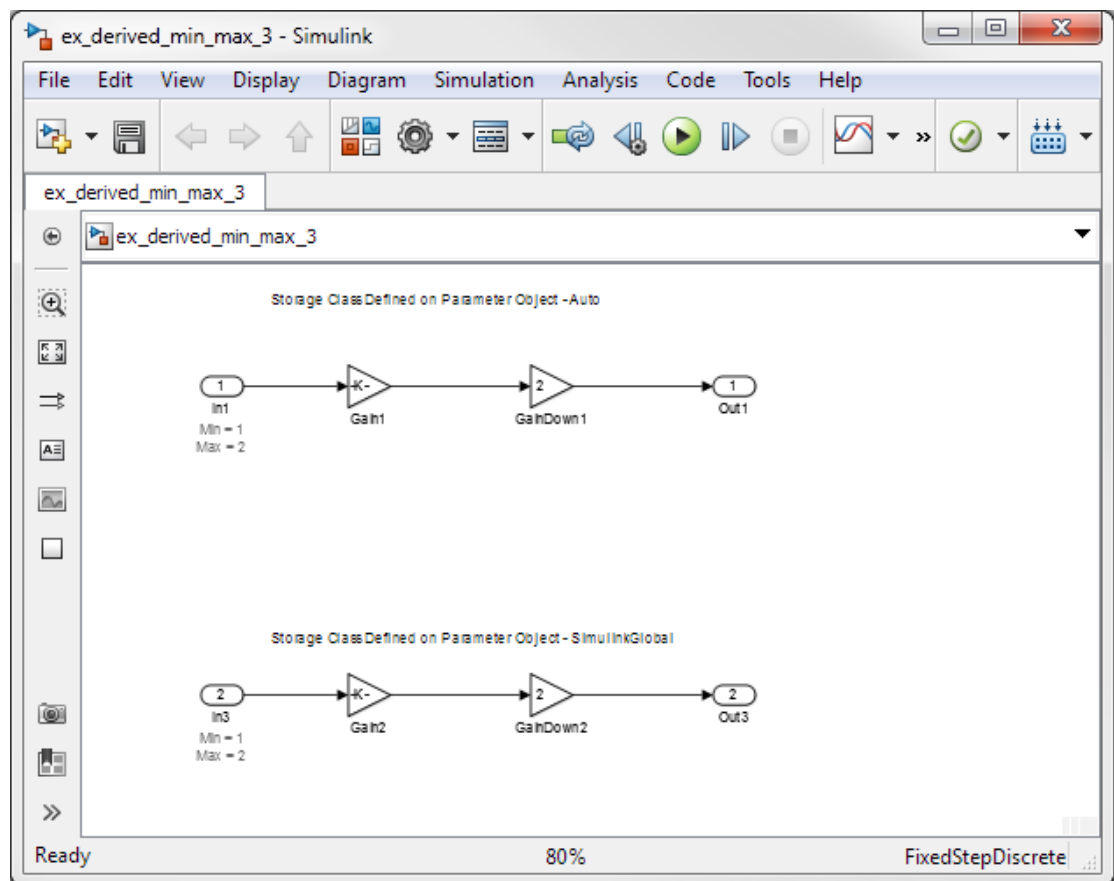
- “How Range Analysis Works” on page 40-2

Derive Ranges for Simulink.Parameter Objects

This example shows how the range analysis takes into account design range information for Simulink.Parameter objects unless the parameter storage class is set to Auto. If the parameter storage class is set to Auto, the analysis uses the value of the parameter.

- 1 Open the `ex_derived_min_max_3` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_derived_min_max_3
```



The model displays the specified design minimum and maximum values for the Inport blocks. The design range for both Inport blocks is [1 . . 2].

Tip: To display design ranges in your model, from the model menu, select **Display > Signals & Ports** and select **Design Ranges**.

- 2 Examine the gain parameters for the Gain blocks.
 - a Double-click each Gain block and note the name of the Gain parameter on the Main tab.

Gain Block	Gain Parameter
Gain1	paramObjOne
Gain2	paramObjTwo

- b From the model menu, select **Tools > Model Explorer**.
- c In Model Explorer window, select the base workspace and view information for each of the gain parameters used in the model.

Gain Parameter	Type Information
paramObjOne	Simulink.Parameter object. Value 2. Storage class set to Auto.
paramObjTwo	Simulink.Parameter object. Value 2. Storage class set to SimulinkGlobal.

- 3 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 4 Under **System under design**, click **Continue** to select `ex_derived_min_max_3` as the system you want to convert.
- 5 Under **Configure model settings**, click **Range collection using double override**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 6 Under **Range collection**, click the **Derive ranges for System Under Design** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model.

Block	Derived Range	Reason
Gain1	[2..4]	The gain parameter, <code>paramObjOne</code> , specified on Gain block <code>Gain1</code> is a <code>Simulink.Parameter</code> object that has its storage class specified as <code>Auto</code> . The range analysis uses the <code>Value</code> property of the <code>Simulink.Parameter</code> object, whose value is 2, and ignores the design range specified for these parameters.
Gain2	[1..20]	The gain parameter, <code>paramObjTwo</code> , specified on Gain block <code>Gain2</code> is a <code>Simulink.Parameter</code> object that has its storage class specified as <code>SimulinkGlobal</code> . The range analysis takes into account the design range, [1..10], specified for this parameter.

More About

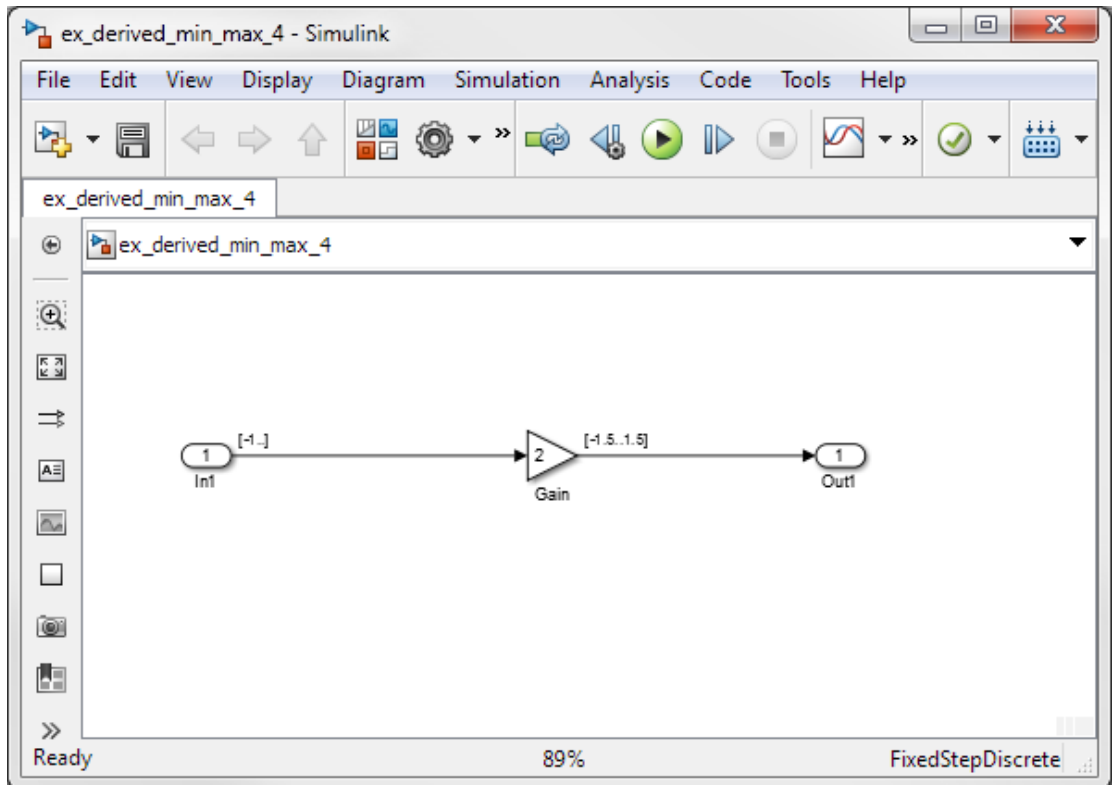
- “How Range Analysis Works” on page 40-2

Insufficient Design Range Information

This example shows that if the analysis cannot derive range information because there is insufficient design range information, you can fix the issue by providing additional input design minimum and maximum values.

- 1 Open the `ex_derived_min_max_4` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_derived_min_max_4
```



The model displays the specified design minimum and maximum values for the blocks in the model.

- The Inport block **In1** has a design minimum of -1 but no specified maximum value, as shown by the annotation, **Max=[]**.
- The Gain block has a design range of **[-1.5..1.5]**.
- The Outport block **Out1** has no design range specified, as shown by the annotations, **Min=[]**, **Max=[]**.

Tip: To display design ranges in your model, from the model menu select **Display > Signals & Ports** and select **Design Ranges**.

- 2 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to select **ex_derived_min_max_4** as the system you want to convert.
- 4 Under **Configure model settings**, click **Range collection using double override**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 5 Under **Range collection**, click the **Derive ranges for System Under Design** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model. The range analysis is unable to derive a maximum value for the Inport block, **In1**. The tool highlights this result.

Contents of: **ex_derived_min_max_4* (mmo-dbl)**

Column View: **Derived Min/Max View** [Show Details](#)

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
In1	DoubleOverride	double	-1		-1	Inf
Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
Out1	DoubleOverride				-1.5	1.5

- 6 To fix the issue, specify a design maximum value for **In1**:
 - a In the model, double-click the Inport block, **In1**.

- b** In the block parameters dialog box, select the **Signal Attributes** tab.
- c** On this tab, set **Maximum** to 1 and click **OK**.

The model displays the updated maximum value in the block annotation for In1.

- 7 In the Fixed-Point Tool, click the **Derive ranges for System Under Design** button to rerun the range analysis.

The range analysis can now derive ranges for the Inport and Gain blocks.

Block	Derived Range	Reason
Inport In1	[-1 . . 1]	Uses specified design range on the block.
Gain	[-1.5 . . 1.5]	The design range specified on the Gain block is [-1.5 . . 1.5]. The derived range at the block input is [-1 . . 1] (the derived range at the output of In1). Therefore, because the gain is 2, the derived range at the Gain block output is the intersection of the propagated range, [-2 . . 2], and the design range, [-1.5 . . 1.5].
Outport In2	[-1.5 . . 1.5]	Same as Gain block output because no locally specified design range on Outport block.

Related Examples

- “Troubleshoot Range Analysis of System Objects” on page 40-25

Troubleshoot Range Analysis of System Objects

When deriving ranges for a model that uses a system object, the analysis fails if the model contains variables that can refer to multiple handle objects. The following example shows how to reconfigure the code so that the Fixed-Point Tool can derive ranges for the model.

In this example, range analysis of the first model `ex_HandleVariableRefersToMultipleObjects` produces an error because there is a variable in the code that can refer to different system objects depending on other conditions. The model, `ex_HandleVariableRefersToSingleObject` is a rewrite of the first model with the same functionality, but the Fixed-Point Tool is able to derive ranges for the model.

- 1 Open the first model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_HandleVariableRefersToMultipleObjects
```

The code inside the MATLAB Function block refers to the custom System Object `fAddConstant`.

```
function y = fcn(u, c)
%#codegen

persistent hSysObjAddTen
persistent hSysObjAddNegTen
persistent hSysObjForStep

if isempty(hSysObjAddTen)
    hSysObjAddTen = fAddConstant(10);
end

if isempty(hSysObjAddNegTen)
    hSysObjAddNegTen = fAddConstant(-10);
end

if c > 0
    hSysObjForStep = hSysObjAddTen;
else
    hSysObjForStep = hSysObjAddNegTen;
end
```

```
y = step(hSysObjForStep, u);
```

- 2 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, select **Continue**.
- 4 Under **Range collection**, click the **Collect derived ranges** button.

The analysis fails because there is a handle variable in the code that can refer to different system objects depending on the value of `c`.

- 5 You can alter the code inside the MATLAB Function block so that the Fixed-Point Tool is able to derive ranges for the System Object by rewriting the code like the following:

```
function y = fcn(u, c)
    %#codegen

    persistent hSysObjAddTen
    persistent hSysObjAddNegTen

    if isempty(hSysObjAddTen)
        hSysObjAddTen = fAddConstant(10);
    end
    if isempty(hSysObjAddNegTen)
        hSysObjAddNegTen = fAddConstant(-10);
    end

    if c > 0
        y = step(hSysObjAddTen, u);
    else
        y = step(hSysObjAddNegTen, u);
    end
end
```

- 6 Close the Fixed-Point Tool and the `ex_HandleVariableRefersToMultipleObjects` model. Open the `ex_HandleVariableRefersToSingleObject` model, which contains the rewritten code. At the MATLAB command line, enter:

```
ex_HandleVariableRefersToSingleObject
```

- 7 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 8 Under **System under design**, select **Continue**.
- 9 Under **Range collection**, click the **Collect derived ranges** button.

This time, the Fixed-Point Tool successfully derives ranges for the variables used in the model.

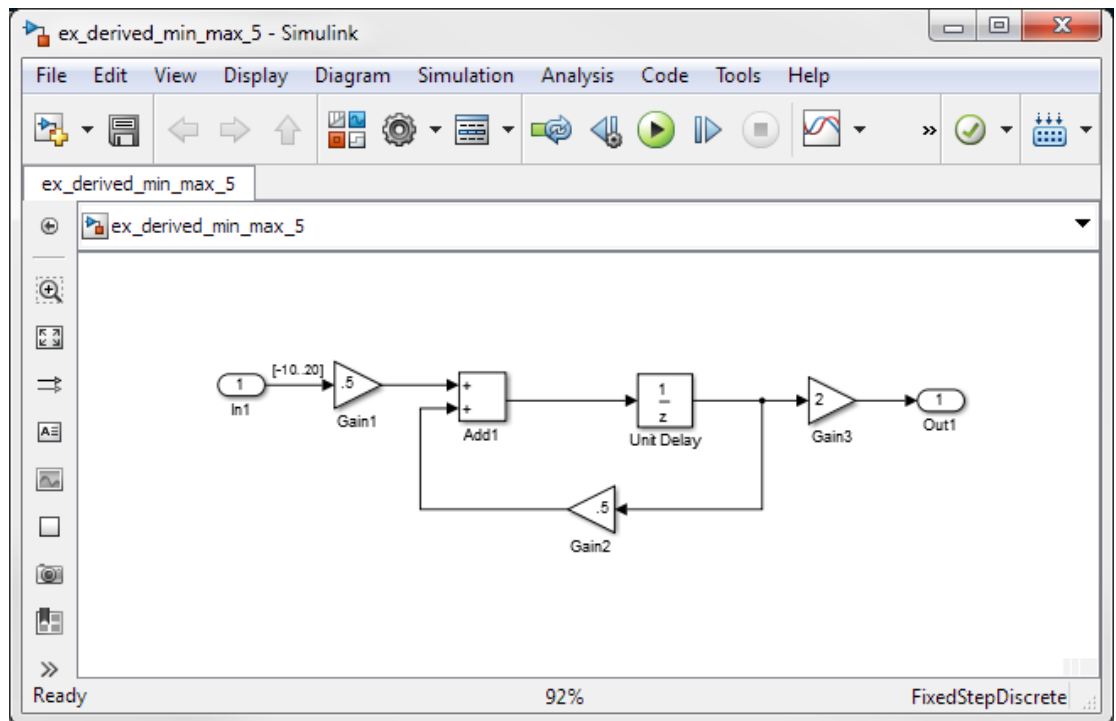
Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
In1	DoubleOverride	double	-10	10	-10	10
In2	DoubleOverride	double	-10	10	-10	10
MATLAB Function.y	DoubleOverride	double			-20	20
Out1	DoubleOverride				-20	20
MATLAB Function/fAddConstant : con...	DoubleOverride	double			-10	10
MATLAB Function/fcn : u	DoubleOverride	double			-10	10
MATLAB Function/fcn : c	DoubleOverride	double			-10	10
MATLAB Function/fAddConstant>step...	DoubleOverride	double			-10	10
MATLAB Function/fAddConstant>step...	DoubleOverride	double			-20	20
MATLAB Function/fcn : y	DoubleOverride	double			-20	20

Providing More Design Range Information

This example shows that if the analysis cannot derive range information because there is insufficient design range information, you can fix the issue by providing additional design range information.

- 1 Open the `ex_derived_min_max_5` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_derived_min_max_5
```



The model displays the specified design minimum and maximum values for the blocks in the model.

- The Inport block `In1` has a design range of `-10,20`.
- The rest of the blocks in the model have no specified design range.

Tip: To display design ranges in your model, from the model menu select **Display > Signals & Ports** and select **Design Ranges**.

- 2 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **Range collection**, click the **Derive ranges** button.

The Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model in the **Contents** pane. Because the model contains a feedback loop, the analysis is unable to derive an output range for the Add block or for any of the blocks connected to this output. The Fixed-Point Tool highlights these results.

Contents of: ex_derived_min_max_5* (mmo-dbl)

Column View: Derived Min/Max View [Show Details](#)

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Unit Delay	DoubleOverride	double			-Inf	Inf
Gain3	DoubleOverride	double			-Inf	Inf
Out1	DoubleOverride				-Inf	Inf
In1	DoubleOverride	double	-10	20	-10	20
Gain1	DoubleOverride	double			-5	10
Gain2	DoubleOverride	double			-Inf	Inf
Add1 : Output	DoubleOverride	double			-Inf	Inf

- 4 To fix the issue, specify design minimum and maximum values inside the feedback loop. For this example, specify the range for the Gain2 block:
 - a In the model, double-click the Gain2 block.
 - b In the block parameters dialog box, select the **Signal Attributes** tab.
 - c On this tab, set **Minimum** to -20 and **Maximum** to 40 and click **OK**.
- 5 In the Fixed-Point Tool, click the **Derive ranges for System Under Design** button to rerun the range analysis.

The range analysis uses the minimum and maximum values specified for Gain2 and In1 to derive ranges for all objects in the model.

Related Examples

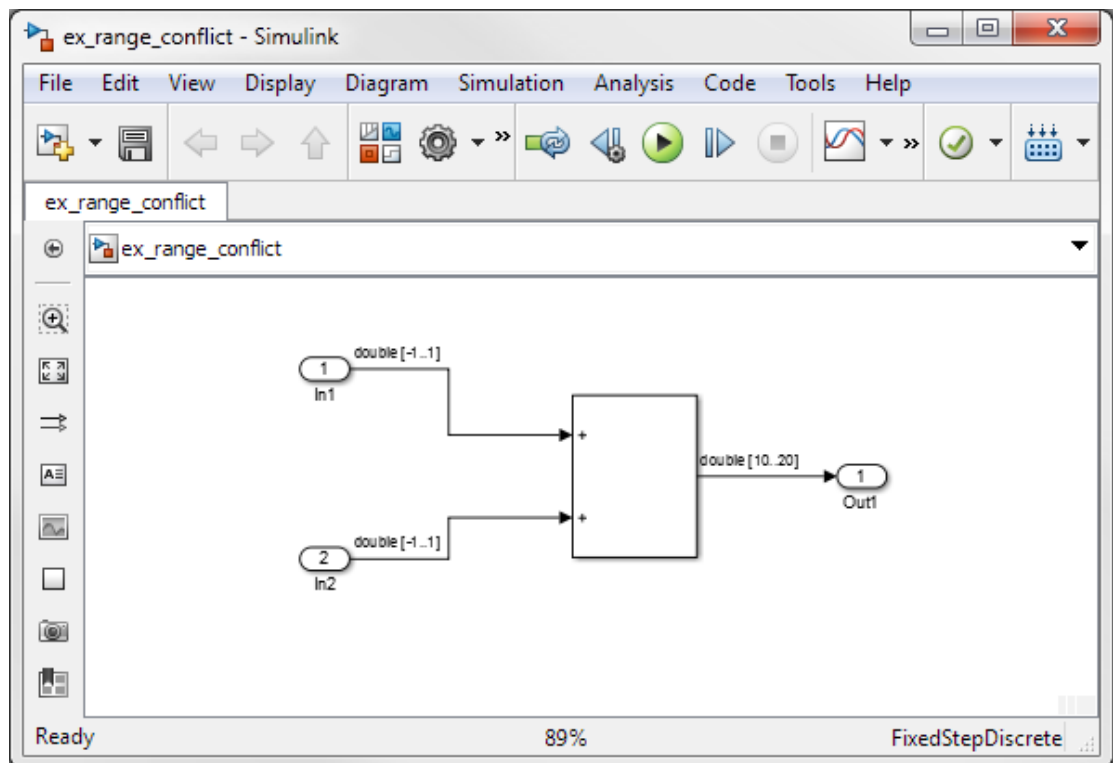
- “Insufficient Design Range Information” on page 40-22

Fixing Design Range Conflicts

This example shows how to fix design range conflicts. If you specify conflicting design minimum and maximum values in your model, the range analysis software reports an error. To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent. In this example, the output design range specified on the Outport block conflicts with the input design ranges specified on the Inport blocks.

- 1 Open the `ex_range_conflict` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))  
ex_range_conflict
```



The model displays the specified design minimum and maximum values for the blocks in the model.

- The Inport blocks `In1` and `In2` have a design range of `[-1..1]`.
- The Outport block `Out1` has a design range of `[10..20]`.

Tip: To display design ranges in your model, from the model menu select **Display > Signals & Ports** and select **Design Ranges**.

- 2 From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to select `ex_range_conflict` as the system you want to convert.
- 4 Under **Configure model settings**, click **Range collection using double override**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 5 Under **Range collection**, click the **Derive ranges for System Under Design** button.

The Fixed-Point Tool generates an error because the range analysis fails. It reports an error because the derived range for the Sum block, `[-2..2]` is outside the specified design range for the Outport block, `[10..20]`.

- 6 Close the error dialog box.
- 7 To fix the conflict, change the design range on the Outport block to `[-10..20]` so that this range includes the derived range for the Sum block.
 - a In the model, double-click the Outport block.
 - b In the block parameters dialog box, click the **Signal Attributes** tab.
 - c On this tab, set **Minimum** to `-10` and click **OK**.
- 8 In the Fixed-Point Tool, click the **Derive ranges for System Under Design** button to rerun the range analysis.

The range analysis derives a minimum value of `-2` and a maximum value of `2` for the Outport block.

More About

- “How Range Analysis Works” on page 40-2

Derive Ranges for a Referenced Model

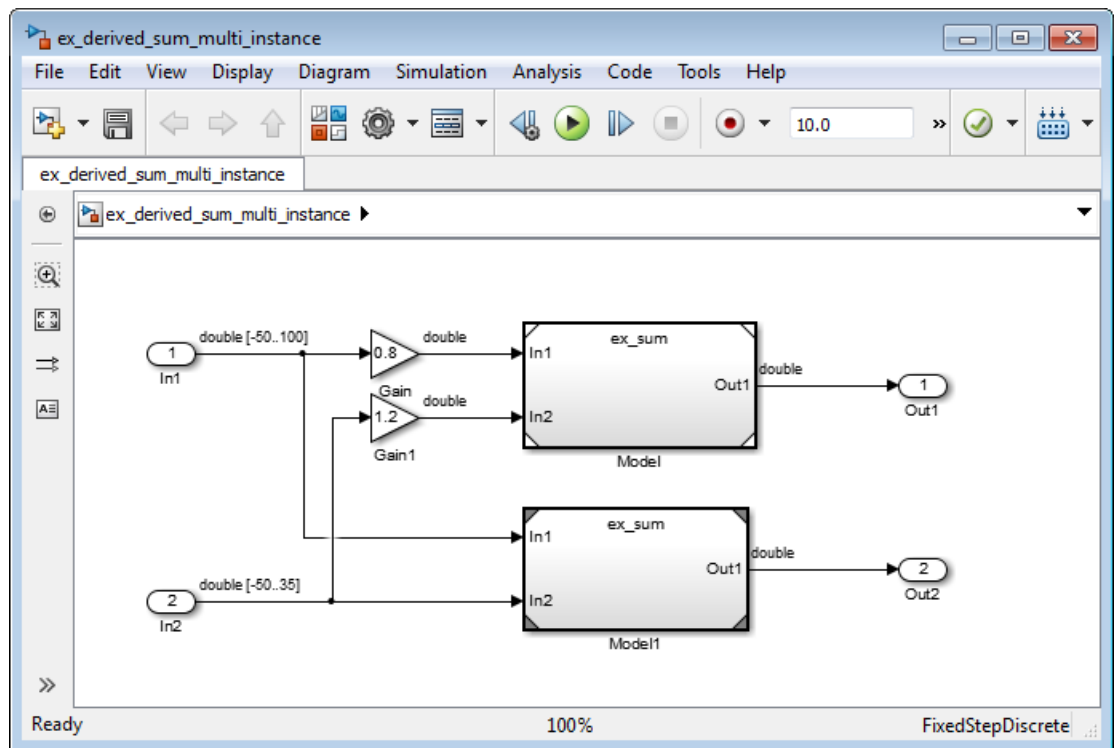
This example shows how to derive ranges for a model that contains multiple instances of the same referenced model.

Derive Ranges

- 1 Open the `ex_derived_sum_multi_instance` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_derived_sum_multi_instance
```

Save the model and the referenced model, `ex_sum`, in a local writable folder.

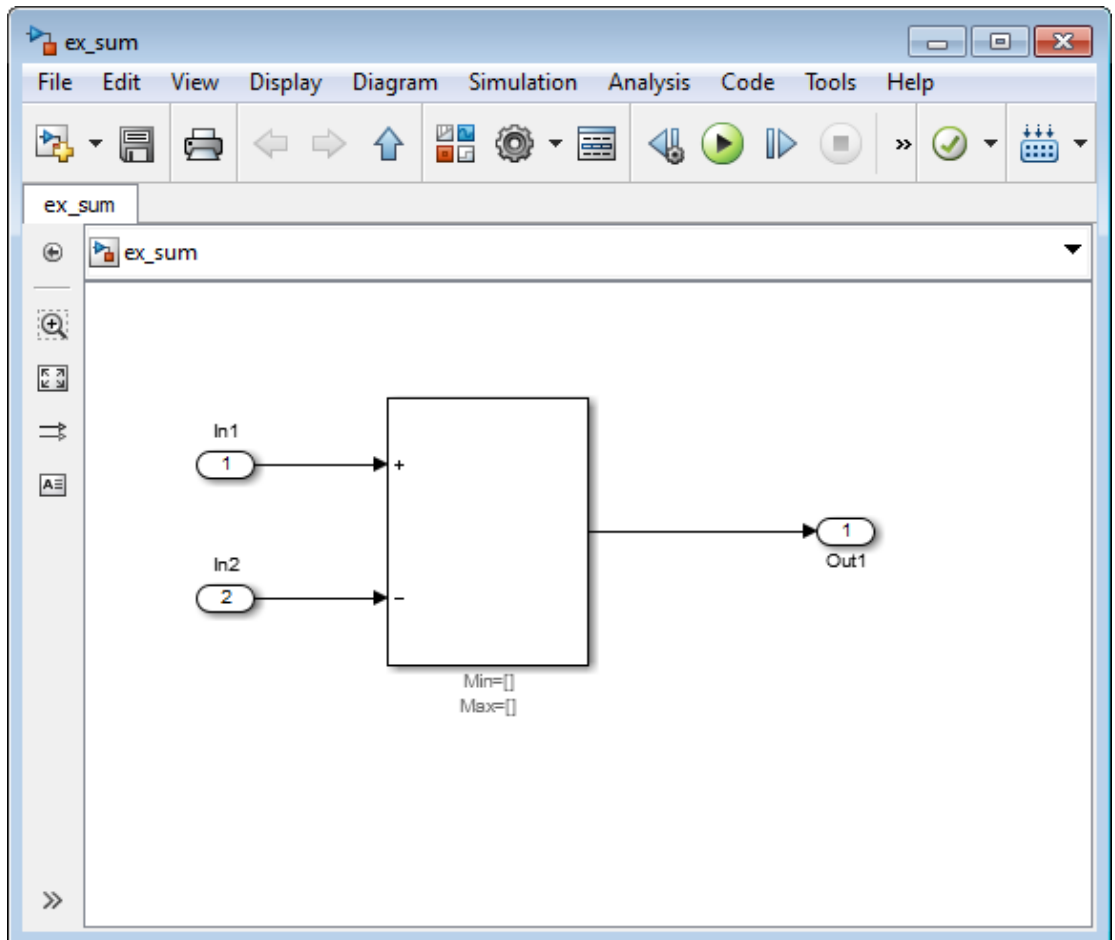


The model displays the specified design minimum and maximum values for the blocks in the model.

- The Inport block In1 has a design range of [-50 . . 100].
- The Inport block In2 has a design range of [-50 . . 35].

Tip: To display design ranges in your model, from the model menu select **Display > Signals & Ports** and select **Design Ranges**.

The model contains two Model blocks that both reference the ex_sum model.



Initially, the Sum block has no design range information.

- 2 From the `ex_derived_sum_multi_instance` model, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 3 Under **System under design**, click **Continue** to select `ex_derived_sum_multi_instance` as the system you want to convert.
- 4 Under **Configure model settings**, click **Range collection using double override**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

- 5 Under **Range collection**, set **Run name** to `double_run` and then click **Apply**.

Providing a unique name for the run avoids accidentally overwriting results from previous runs and enables you to identify the run more easily.

- 6 Save the models.
- 7 In the Fixed-Point Tool, under **Range collection**, click the **Derive ranges for System Under Design** button.

To calculate the derived ranges, the software uses the design minimum and maximum values specified for the Inport blocks in the top-level model, `In1` and `In2`.

In the **Contents** pane, the Fixed-Point Tool displays the derived and design minimum and maximum values for the blocks and referenced models in the `ex_derived_sum_multi_instance` model. Some of the derived values that the Fixed-Point Tool reports are slightly larger than expected. This difference is due to rounding errors because the software approximates floating-point numbers with infinite-precision rational numbers for analysis and then converts them to floating point for reporting.

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
In1	double_run	double	-50	100	-50	100
Gain	double_run	double			-40.000000000000001	80.000000000000001
In2	double_run	double	-50	35	-50	35
Gain1	double_run	double			-60	42
Out1	double_run				-82.000000000000001	140.000000000000003
Out2	double_run				-85	150

View Derived Ranges for Referenced Model

- 1 In the **Model Hierarchy** pane, select the first instance of the referenced model, `Model1 (ex_sum)`.

The tool displays the derived minimum and maximum values for this instance of the referenced model, $[-82.001 \dots 140.001]$. This range is derived from the outputs of the two Gain blocks, $[-40, 80]$ and $[-60 \dots 42.001]$.

- 2 Select the second instance of the referenced model, `Model1 (ex_sum)`.

The tool displays the derived values for the second instance, $[-85..150]$. This range is derived from the referenced model inputs, `In1` and `In2`, $[-50..100]$ and $[-50..35]$ respectively.

- 3 Select the node for the referenced model, `ex_sum`.

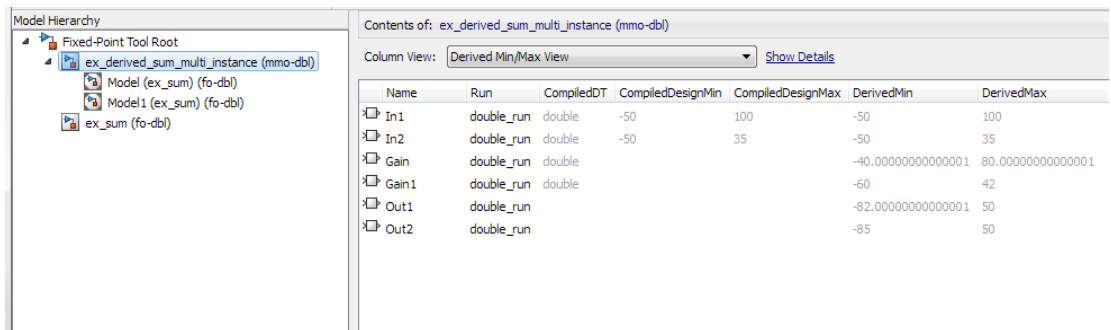
For this node, the Fixed-Point Tool displays the merged results for the derived range for the referenced model which is the union of the results for each instance of the model, $[-85..150]$.

Next, you set design range on `Sum` block in referenced model to see how the range analysis takes this information into account.

Add Design Range for Sum Block and Derive Ranges

- 1 In the `ex_sum` model, double-click the `Sum` block.
- 2 In the block parameters dialog box, on the **Signal Attributes** tab, set **Output minimum** to `-125` and **Output maximum** to `50`, click **OK**, and then save the model.
- 3 Under **Range collection**, click the **Derive ranges for System Under Design** button.

This time, to calculate the derived ranges, the software uses the design minimum and maximum values specified for the `Inport` blocks in the parent model, `In1` and `In2`, and the design minimum and maximum values specified for the `Sum` block in the referenced model.



Model Hierarchy

- Fixed-Point Tool Root
 - ex_derived_sum_multi_instance (mno-dbl)
 - Model (ex_sum) (fo-dbl)
 - Model1 (ex_sum) (fo-dbl)
 - ex_sum (fo-dbl)

Contents of: ex_derived_sum_multi_instance (mno-dbl)

Column View: Derived Min/Max View [Show Details](#)

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
In1	double_run	double	-50	100	-50	100
In2	double_run	double	-50	35	-50	35
Gain	double_run	double			-40.000000000000001	80.000000000000001
Gain1	double_run	double			-60	42
Out1	double_run	double			-82.000000000000001	50
Out2	double_run	double			-85	50

- 4 You can now propose data types for the model based on these derived minimum and maximum values.

Related Examples

- “Propose Data Types for a Referenced Model” on page 40-40

More About

- “Deriving Ranges for a Referenced Model” on page 40-41

Propose Data Types for a Referenced Model

This example shows how to propose data types for a referenced model. To run this example, you must first run the “Derive Ranges for a Referenced Model” on page 40-34 example.

- 1 In the Fixed-Point Tool, under **Automatic data typing**, click **Propose data types**.

The Fixed-Point Tool proposes fraction lengths for the inputs In1 and In2 based on the design minimum and maximum values specified on the blocks in the model and on the derived minimum and maximum values.

Contents of: ex_derived_sum_multi_instance (mmo-dbl)

Column View: Automatic Data Typing View [Show Details](#)

Name	Run	DTGroup	Accept	ProposedDT	SpecifiedDT	CompiledDT	SimMin	SimMax	ProposedMin	ProposedMax
In1	double_run	G1	<input checked="" type="checkbox"/>	fixdt(1,16,8)	double	double			-128	127.99609375
In2	double_run	G2	<input checked="" type="checkbox"/>	fixdt(1,16,9)	double	double			-64	63.998046875
Gain	double_run	G1	<input checked="" type="checkbox"/>	fixdt(1,16,8)	Inherit: Same as input	double			-128	127.99609375
Gain1	double_run	G2	<input checked="" type="checkbox"/>	fixdt(1,16,9)	Inherit: Same as input	double			-64	63.998046875
Out1	double_run	G3		n/a	Inherit: auto					
Out2	double_run	G4		n/a	Inherit: auto					
Gain : Gain	double_run			n/a	Inherit: Inherit via internal rule					
Gain1 : Gain	double_run			n/a	Inherit: Inherit via internal rule					

- 2 After reviewing the data type proposals, click **Apply accepted fraction lengths** to apply the proposed data types to your model.

Related Examples

- “Derive Ranges for a Referenced Model” on page 40-34

More About

- “Deriving Ranges for a Referenced Model” on page 40-41

Deriving Ranges for a Referenced Model

In this section...

“Viewing Derived Minimum and Maximum Values for Referenced Models” on page 40-41

“Data Type Override Settings” on page 40-42

Viewing Derived Minimum and Maximum Values for Referenced Models

The Fixed-Point Tool derives minimum and maximum values for referenced models. The simulation mode is not relevant for the analysis — instances of the referenced models can be in any simulation mode.

The Fixed-Point Tool displays the top-level model that contains the referenced models and the referenced models in its **Model Hierarchy** pane. For example, the `ex_derived_sum_multi_instance` model contains two instances of the referenced model `ex_sum`. The Fixed-Point Tool displays both models and both instances of the referenced model in the model hierarchy.

Model Hierarchy

- Fixed-Point Tool Root
 - ex_derived_sum_multi_instance (mmo-dbl)
 - Model (ex_sum) (fo-dbl)
 - Model1 (ex_sum) (fo-dbl)
 - ex_sum (fo-dbl)

Contents of: ex_derived_sum_multi_instance (mmo-dbl)

Column View: Derived Min/Max View [Show Details](#)

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
In1	double_run	double	-50	100	-50	100
Gain	double_run	double			-40.000000000000001	80.000000000000001
In2	double_run	double	-50	35	-50	35
Gain1	double_run	double			-60	42
Out1	double_run				-82.000000000000001	140.000000000000003
Out2	double_run				-85	150

If a model contains multiple instances of the referenced model, the tool displays each instance of the referenced model as well as a node for the referenced model. For example, here are the results for the first instance of the referenced model `ex_sum` in `ex_derived_sum_multi_instance`.

Model Hierarchy

- Fixed-Point Tool Root
 - ex_derived_sum_multi_instance (fo-dbl)
 - Model (ex_sum) (fo-dbl)
 - Model1 (ex_sum) (fo-dbl)
 - ex_sum (fo-dbl)

Contents of: Model (ex_sum) (fo-dbl)

Column View: Derived Min/Max View [Show Details](#)

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Sum : Output	double_run	double			-82.001	140.01

Here are the results for the second instance of `ex_sum`.

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Sum : Output	double_run	double			-85	150

In the referenced model node, the tool displays the union of the results for each instance of the referenced model.

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Sum : Output	double_run	double			-85	150

Data Type Override Settings

You can set up user-defined shortcuts across referenced model boundaries. The factory default shortcuts apply only to the top-level model and so do not affect the settings of any referenced model.

When you change the fixed-point instrumentation and data type override settings for any instance of a referenced model, the settings change on all instances of the model and on the referenced model itself.

Related Examples

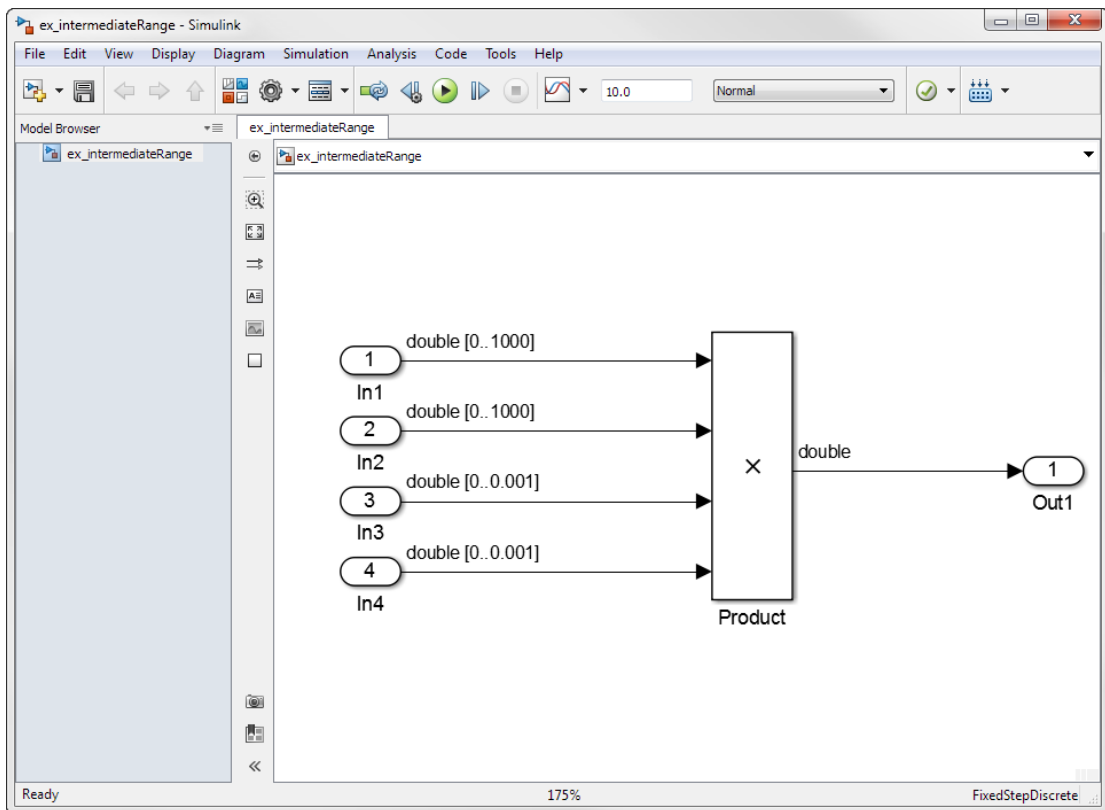
- “Derive Ranges for a Referenced Model” on page 40-34

Intermediate Range Results

This example shows how to interpret the Intermediate Maximum and Intermediate Minimum results in the **Result Details** tab.


Open the model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))  
ex_intermediateRange
```




- 1 Update the diagram. Notice the design range information for each of the input ports.
- 2 Open the Fixed-Point Tool. From the model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.

3 Under **System under design**, click **Continue** to select `ex_intermediateRange` as the system you want to convert.

4 Click the **Derive ranges for System Under Design** button .

The Fixed-Point Tool **Contents** pane displays the derived minimum and maximum values for each object in the `ex_intermediateRange` model.

5 Under **Automatic data typing**, in the **Default word length** field, enter 32.

6 Click the **Propose data types** button .

The proposed data types appear in the **Contents** pane of the Fixed-Point Tool.

7 Look at the proposed data type of the Product block. The Fixed-Point Tool proposed a data type with 32-bit word length and 12-bit fraction length. The derived maximum value of the Product block is 1, but the maximum representable value for the proposed data type is approximately 1,048,575.

To learn more about the data type proposal, in the **Contents** pane, select the product block and then click the **Result Details** tab.

8 In the Result Details tab, in the **Range Information** table, notice the row labeled **Intermediate**. After the first two inputs to the Product block are multiplied, the block has a maximum value of 1000000 before being multiplied by the next two inputs for a final maximum value of 1. The data type proposal for the Product block in this model is based on the intermediate minimum and maximum values. It is not based on the derived minimum and maximum values to prevent overflows at the intermediate stages of the block.

Workflow Result Details

ex_intermediateRange/Product

Proposed Data Type Summary

- There is a requirement for the data type of this result to match the data type of other results.
 - [Highlight Elements Sharing Same Data Type](#)

Property	Data Type	Minimum	Maximum	Precision
ProposedDT	fixdt(0,32,12)	0	1048575.99975...	0.000244140625
SpecifiedDT	double	-1.7976931348...	1.79769313486...	4.94065645841...

Range Information

Property	Minimum	Maximum
Shared derived	0	1.0000000000000002
Derived	0	1.0000000000000002
Intermediate	0	1000000

More About

- “How Range Analysis Works” on page 40-2

Unsupported Simulink Software Features

Range analysis does not support the following Simulink software features. Avoid using these unsupported features.

Not Supported	Description
Variable-step solvers	<p>The software supports only fixed-step solvers.</p> <p>For more information, see “Choose a Fixed-Step Solver” (Simulink).</p>
Callback functions	<p>The software does not execute model callback functions during the analysis. The results that the analysis generates may behave inconsistently with the expected behavior.</p> <ul style="list-style-type: none"> • If a model or any referenced model calls a callback function that changes any block parameters, model parameters, or workspace variables, the analysis does not reflect those changes. • Changing the storage class of base workspace variables on model callback functions or mask initializations is not supported. • Callback functions called prior to analysis, such as the <code>PreLoadFcn</code> or <code>PostLoadFcn</code> model callbacks, are fully supported.
Model callback functions	<p>The software only supports model callback functions if the <code>InitFcn</code> callback of the model is empty.</p>
Algebraic loops	<p>The software does not support models that contain algebraic loops.</p> <p>For more information, see “Algebraic Loops” (Simulink).</p>
Masked subsystem initialization functions	<p>The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.</p>
Variable-size signals	<p>The software does not support variable-size signals. A variable-size signal is a signal whose size (number of elements in a dimension), in addition to its values, can change during model execution.</p>

Not Supported	Description
Arrays of buses	<p>The software does not support arrays of buses.</p> <p>For more information, see “Combine Buses into an Array of Buses” (Simulink).</p>
Multiword fixed-point data types	<p>The software does not support multiword fixed-point data types.</p>
Nonfinite data	<p>The software does not support nonfinite data (for example, NaN and Inf) and related operations.</p>
Signals with nonzero sample time offset	<p>The software does not support models with signals that have nonzero sample time offsets.</p>
Models with no output ports	<p>The software only supports models that have one or more output ports.</p>

Note: The software does not report initial or intermediate values for Stateflow variables. Range analysis will only report the ranges at the output of the block.

Simulink Blocks Supported for Range Analysis

Overview of Simulink Block Support

The following tables summarize range analysis support for Simulink blocks. Each table lists all the blocks in each Simulink library and describes support information for that particular block. If the software does not support a given block, where possible, automatic stubbing considers the interface of the unsupported blocks, but not their behavior, during the analysis. However, if any of the unsupported blocks affect the simulation outcome, the analysis may achieve only partial results. If the analysis cannot use automatic stubbing for a block, the block is marked as “not stubbable”. For more information, see “Automatic Stubbing” on page 40-5.

Not all blocks that are supported for range analysis are supported for fixed-point conversion. To check if a block supports fixed-point data types, see “Using Blocks that Don’t Support Fixed-Point Data Types” on page 46-21.

Additional Math and Discrete Library

The software supports all blocks in the Additional Math and Discrete library.

Commonly Used Blocks Library

The Commonly Used Blocks library includes blocks from other libraries. Those blocks are listed under their respective libraries.

Continuous Library

Block	Support Notes
Derivative	Not supported
Integrator	Not supported and not stubbable
Integrator Limited (Simulink)	Not supported and not stubbable
PID Controller	Not supported
PID Controller (2 DOF)	Not supported
Second Order Integrator (Simulink)	Not supported and not stubbable
Second Order Integrator Limited	Not supported and not stubbable
State-Space	Not supported and not stubbable
Transfer Fcn	Not supported and not stubbable

Block	Support Notes
Transport Delay	Not supported
Variable Time Delay	Not supported
Variable Transport Delay	Not supported
Zero-Pole	Not supported and not stubbable

Discontinuities Library

The software supports all blocks in the Discontinuities library.

Discrete Library

Block	Support Notes
Delay	Supported
Difference	Supported
Discrete Derivative	Supported
Discrete Filter	The software analyzes through the filter. It does not derive any range information for the filter.
Discrete FIR Filter	Supported
Discrete PID Controller (Simulink)	Supported
Discrete PID Controller (2 DOF) (Simulink)	Supported
Discrete State-Space	Not supported
Discrete Transfer Fcn	Supported
Discrete Zero-Pole	Not supported
Discrete-Time Integrator	Supported
First-Order Hold	Supported
Memory	Supported
Tapped Delay	Supported
Transfer Fcn First Order	Supported
Transfer Fcn Lead or Lag	Supported
Transfer Fcn Real Zero	Supported

Block	Support Notes
Unit Delay	Supported
Zero-Order Hold	Supported

Logic and Bit Operations Library

The software supports all blocks in the Logic and Bit Operations library.

Lookup Tables Library

Block	Support Notes
Cosine	Supported
Direct Lookup Table (n-D)	Supported
Interpolation Using Prelookup	Not supported when: <ul style="list-style-type: none"> The Interpolation method parameter is Linear and the Number of table dimensions parameter is greater than 4. <p>or</p> <ul style="list-style-type: none"> The Interpolation method parameter is Linear and the Number of sub-table selection dimensions parameter is not 0.
1-D Lookup Table	Not supported when the Interpolation method or the Extrapolation method parameter is Cubic Spline .
2-D Lookup Table	Not supported when the Interpolation method or the Extrapolation method parameter is Cubic Spline .
n-D Lookup Table	Not supported when: <ul style="list-style-type: none"> The Interpolation method or the Extrapolation method parameter is Cubic Spline. <p>or</p> <ul style="list-style-type: none"> The Interpolation method parameter is Linear and the Number of table dimensions parameter is greater than 5.

Block	Support Notes
Lookup Table Dynamic	Supported
Prelookup	Supported
Sine (Simulink)	Supported

Math Operations Library

Block	Support Notes														
Abs	Supported														
Add	Supported														
Algebraic Constraint	Supported														
Assignment	Supported														
Bias	Supported														
Complex to Magnitude-Angle	Not supported														
Complex to Real-Imag	Supported														
Divide	Supported														
Dot Product	Supported														
Find Nonzero Elements (Simulink)	Supported														
Gain	Supported														
Magnitude-Angle to Complex	Supported														
Math Function	<p>All signal types support the following Function parameter settings.</p> <table border="1"> <tbody> <tr> <td>conj</td> <td>hermitian</td> <td>magnitude^2</td> <td>mod</td> </tr> <tr> <td>rem</td> <td>reciprocal</td> <td>square</td> <td>transpose</td> </tr> </tbody> </table> <p>The software does not support the following Function parameter settings.</p> <table border="1"> <tbody> <tr> <td>10^u</td> <td>exp</td> <td>hypot</td> </tr> <tr> <td>log</td> <td>log10</td> <td>pow</td> </tr> </tbody> </table>	conj	hermitian	magnitude^2	mod	rem	reciprocal	square	transpose	10^u	exp	hypot	log	log10	pow
conj	hermitian	magnitude^2	mod												
rem	reciprocal	square	transpose												
10^u	exp	hypot													
log	log10	pow													
Matrix Concatenate (Simulink)	Supported														

Block	Support Notes
MinMax	Supported
MinMax Running Resetable	Supported
Permute Dimensions	Supported
Polynomial	Supported
Product	Supported
Product of Elements	Supported
Real-Imag to Complex	Supported
Reciprocal Sqrt	Not supported
Reshape	Supported
Rounding Function	Supported
Sign	Supported
Signed Sqrt	Not supported
Sine Wave Function	Not supported
Slider Gain	Supported
Sqrt	Supported
Squeeze	Supported
Subtract	Supported
Sum	Supported
Sum of Elements	Supported
Trigonometric Function	Supported if Function is <code>sin</code> , <code>cos</code> , or <code>sincos</code> , and Approximation method is <code>CORDIC</code> .
Unary Minus	Supported
Vector Concatenate (Simulink)	Supported
Weighted Sample Time Math	Supported

Model Verification Library

The software supports all blocks in the Model Verification library.

Model-Wide Utilities Library

Block	Support Notes
Block Support Table	Supported
DocBlock	Supported
Model Info	Supported
Timed-Based Linearization	Not supported
Trigger-Based Linearization	Not supported

Ports & Subsystems Library

Block	Support Notes
Atomic Subsystem	Supported
Code Reuse Subsystem	Supported
Configurable Subsystem	Supported
Enable	Supported
Enabled Subsystem	Range analysis does not consider the design minimum and maximum values specified for blocks connected to the output of the subsystem.
Enabled and Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Range analysis does not consider the design minimum and maximum values specified for blocks connected to the output of the subsystem.
For Each	Supported with the following limitations: <ul style="list-style-type: none"> • When For Each Subsystem contains another For Each Subsystem, not supported. • When For Each Subsystem contains one or more Simulink Design Verifier™ Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported.
For Each Subsystem	Supported with the following limitations:

Block	Support Notes
	<ul style="list-style-type: none"> • When For Each Subsystem contains another For Each Subsystem, not supported. • When For Each Subsystem contains one or more Simulink Design Verifier Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported.
For Iterator Subsystem	Supported
Function-Call Feedback Latch	Supported
Function-Call Generator	Supported
Function-Call Split	Supported
Function-Call Subsystem	Range analysis does not consider the design minimum and maximum values specified for blocks connected to the output of the subsystem.
If	Supported
If Action Subsystem	Supported
Inport	—
Model	Supported except for the limitations described in “Limitations of Support for Model Blocks” on page 40-58.
Model Variants	Supported except for the limitations described in “Limitations of Support for Model Blocks” on page 40-58.
Output	Supported
Subsystem	Supported
Switch Case	Supported
Switch Case Action Subsystem	Supported
Trigger	Supported

Block	Support Notes
Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Range analysis does not consider the design minimum and maximum values specified for blocks connected to the output of the subsystem.
Variant Subsystem	Supported
While Iterator Subsystem	Supported

Signal Attributes Library

The software supports all blocks in the Signal Attributes library.

Signal Routing Library

Block	Support Notes
Bus Assignment	Supported
Bus Creator	Supported
Bus Selector	Supported
Data Store Memory	<ul style="list-style-type: none"> When the Data Store Memory variable is tunable, range analysis considers the design ranges specified on the block, and ignores local model writes. When the Data Store Memory variable is not tunable, or Auto, the analysis considers only local model writes. The derived range is the range of the last write to the variable. When the Data Store Memory variable is defined outside of the analyzed system, range analysis uses design ranges.
Data Store Read	Supported
Data Store Write	Supported
Demux	Supported
Environment Controller	Supported
From	Supported

Block	Support Notes
Goto	Supported
Goto Tag Visibility	Supported
Index Vector	Supported
Manual Switch	The Manual Switch block is compatible with the software, but the analysis ignores this block in a model.
Merge	Supported
Multiport Switch	Supported
Mux	Supported
Selector	Supported
Switch	Supported
Vector Concatenate	Supported

Sinks Library

Block	Support Notes
Display	Supported
Floating Scope	Supported
Outport (Out1)	Supported
Scope	Supported
Stop Simulation	Not supported and not stubbable
Terminator	Supported
To File	Supported
To Workspace	Supported
XY Graph	Supported

Sources Library

Block	Support Notes
Band-Limited White Noise	Not supported
Chirp Signal	Not supported

Block	Support Notes
Clock	Supported
Constant	Supported unless Constant value is <code>inf</code> .
Counter Free-Running	Supported
Counter Limited	Supported
Digital Clock	Supported
Enumerated Constant	Supported
From File	Not supported. When MAT-file data is stored in MATLAB <code>timeseries</code> format, not stubbable.
From Workspace	Not supported
Ground	Supported
Inport (In1)	Supported
Pulse Generator	Supported
Ramp	Supported
Random Number	Not supported and not stubbable
Repeating Sequence	Not supported
Repeating Sequence Interpolated	Not supported
Repeating Sequence Stair	Supported
Signal Builder	Not supported
Signal Generator	Not supported
Sine Wave	Not supported
Step	Supported
Uniform Random Number	Not supported and not stubbable

User-Defined Functions Library

Block	Support Notes
Fcn	Supports all operators except <code>^</code> . Supports only the mathematical functions <code>abs</code> , <code>ceil</code> , <code>fabs</code> , <code>floor</code> , <code>rem</code> , and <code>sgn</code> .
Interpreted MATLAB Function	Not supported

Block	Support Notes
MATLAB Function	<p>The software uses the specified design minimum and maximum values and returned derived minimum and maximum values for instances of variables that correspond to input and output ports. It does not consider intermediate instances of these variables. For example, consider a MATLAB Function block that contains the following code:</p> <pre data-bbox="605 552 908 663">function y = fcn(u,v) %#codegen y = 2*u; y = y + v;</pre> <p>Range analysis considers the design ranges specified for u and v for the instance of y in $y = y + v$; because this is the instance of y associated with the output of the block.</p> <p>The analysis does not consider design ranges for the instance of y in $y = 2*u$; because it is an intermediate instance.</p>
Level-2 MATLAB S-Function	Not supported
S-Function	Not supported
S-Function Builder	Not supported
Simulink Function	Simulink Functions with output arguments that are of complex type are not supported.

Limitations of Support for Model Blocks

Simulink Design Verifier supports the Model block with the following limitations. The software cannot analyze a model containing one or more Model blocks if:

- The referenced model is protected. Protected referenced models are encoded to obscure their contents. This allows third parties to use the referenced model without being able to view the intellectual property that makes up the model.

For more information, see “Protected Model” (Simulink).

- The parent model or any of the referenced models returns an error when you set the **Configuration Parameters > Diagnostics > Connectivity > Element name mismatch** parameter to `error`.

You can use the **Element name mismatch** diagnostic along with bus objects so that your model meets the bus element naming requirements imposed by some blocks.

- The Model block uses asynchronous function-call inputs.
- Any of the Model blocks in the model reference hierarchy creates an artificial algebraic loop. If this occurs, take the following steps:
 - 1 On the **Diagnostics** pane of the Configuration Parameters dialog box, set the **Minimize algebraic loop** parameter to `error` so that Simulink reports an algebraic loop error.
 - 2 On the **Model Referencing** Pane of the Configuration Parameters dialog box, select the Minimize algebraic loop occurrences parameter.

Simulink tries to eliminate the artificial algebraic loop during simulation.

- 3 Simulate the model.
- 4 If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by selecting **Simulation > Update Diagram**.
- 5 Eliminate the artificial algebraic loop so that the software can analyze the model. Break the loop with Unit Delay blocks so that the execution order is predictable.

Note: For more information, see “Algebraic Loops” (Simulink).

- The parent model uses the base workspace and the referenced model uses a data dictionary.
- The parent model and the referenced model have mismatched data type override settings. The data type override setting of the parent model and its referenced models must be the same, unless the data type override setting of the parent model is **Use local settings**. You can select the data type override settings for your model in the **Analysis** menu, in the Fixed Point Tool dialog box under the **Settings for selected system** pane.
- The referenced model is a Model Reference block with virtual bus inports, and the signals in the bus do not all have the same sample time at compilation. To make the model compatible with Simulink Design Verifier analysis, convert the port to a nonvirtual bus, or specify an explicit sample time for the port.

Working with the MATLAB Function Block

- “Convert MATLAB Function Block to Fixed Point” on page 41-2
- “Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool” on page 41-11
- “Replace Functions in a MATLAB Function Block with a Lookup Table” on page 41-12
- “Derive Ranges of MATLAB Function Block Variables” on page 41-16
- “Convert a System Object to Fixed Point Using the Fixed-Point Tool” on page 41-19

Convert MATLAB Function Block to Fixed Point

This example shows how to use the Fixed-Point Tool to convert a model containing a MATLAB Function block to fixed point.

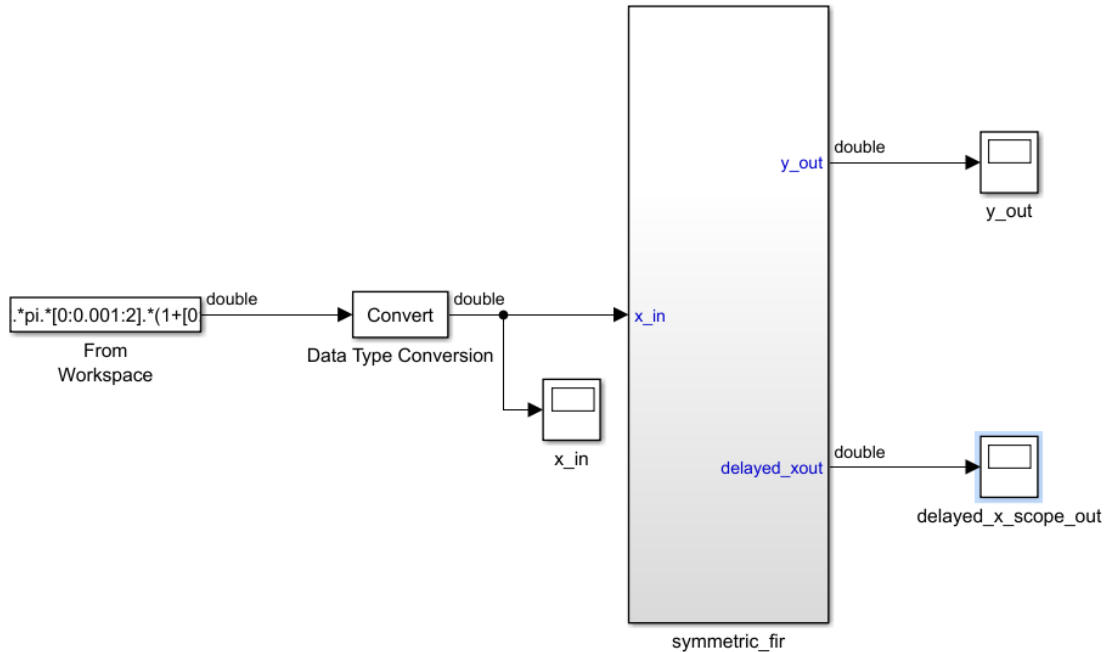
In this section...
“Open the Model” on page 41-2
“Collect Range Information” on page 41-3
“Propose Data Types” on page 41-4
“Inspect Code Using the Code View” on page 41-5
“Apply Proposed Data Types” on page 41-7
“Verify Results” on page 41-8

Open the Model

Change directories to the folder where the model is located. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

Copy the `ex_symmetric_fir.slx` file to a local writable folder and open the model.



The `ex_symmetric_fir` model uses a symmetric FIR filter. Simulate the model and inspect the model output. Inspect the symmetric FIR filter algorithm by double-clicking the MATLAB Function block.

Collect Range Information

Collect idealized ranges to use for data type proposal.

- 1 From the model menu, select **Analysis > Data Type Design > Fixed-Point Tool**
- 2 In the Workflow pane, under **System under design**, click **Change** and select the `symmetric_fir` subsystem, which contains the MATLAB Function block, as the system to convert.

- 3** Under **Configure model settings**, click the **Range collection using double override** button to configure the model to collect ranges using idealized floating-point data types. This configuration also sets the **Run name** to DoubleOverride.
- 4** Under **Range collection**, click **Simulate model**.

Examine the range information of the MATLAB variables in the **Contents** pane.

Propose Data Types

Configure the proposal settings and propose fixed-point data types for the model.

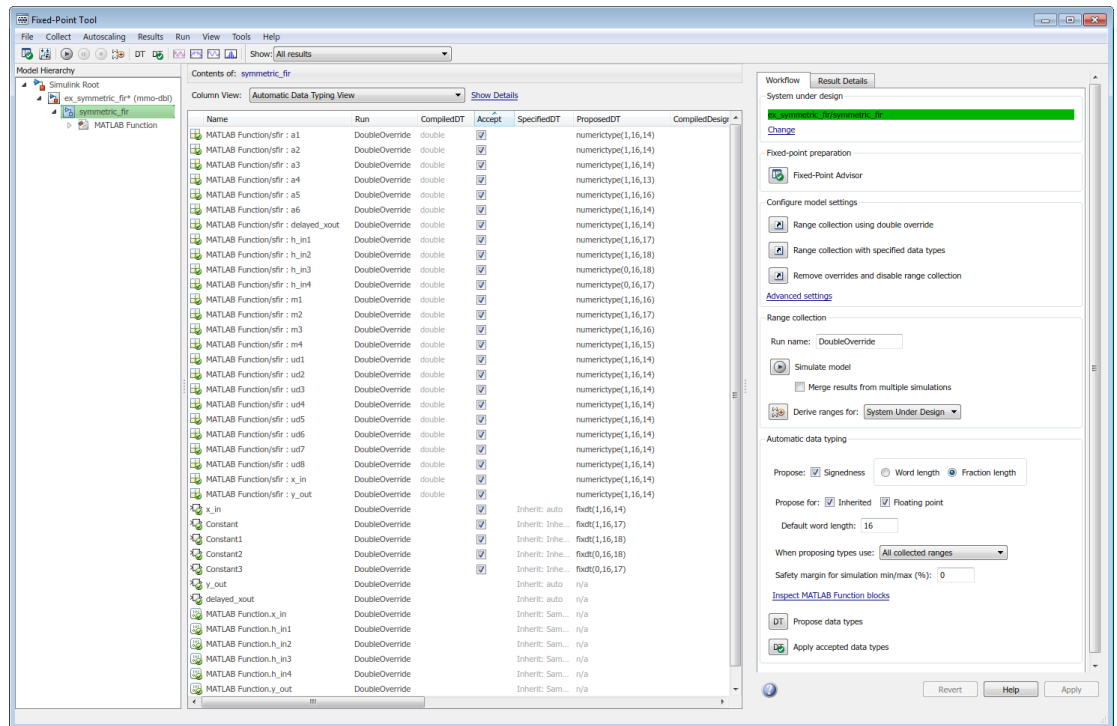
- 1** On the right side of the Fixed-Point Tool, under **Automatic data typing**, you can configure the data type proposal settings for the MATLAB Function block variables.

In this example, use the default proposal settings.

- 2** Click **Propose data types**.

The data type proposals appear in the **ProposedDT** column of the **Contents** pane.

Note: The **SpecifiedDT** column is always blank for MATLAB Function block variables.



Inspect Code Using the Code View

To launch the code view, under **Automatic data typing**, click **Inspect MATLAB Function blocks**.

Using the code view you can:

- View detailed variable and expression information.
- Adjust proposal settings, such as `fimath` settings.
- Edit proposed data types.
- Manage function replacements.

For examples showing how to replace MATLAB functions with a lookup table, see “Replace Functions in a MATLAB Function Block with a Lookup Table” on page 41-12.

- Edit your code.
- Propose fixed-point data types.
- Apply proposed data types to your code.

To view the current proposal settings, click **Settings**. Here you can edit the `fimath` properties for the function. For this example, the default `fimath` properties are sufficient.

The screenshot shows the MATLAB Function Block Converter interface for a system named 'symmetric_fir'. The main window displays MATLAB code for a symmetric FIR filter. Below the code, the 'Function Replacements' tab is active, showing a table of variable types and ranges.

```

21 %%codegen
22 function [y_out, delayed_xout] = sfir(x_in, h_in1, h_in2, h_in3, h_in4)
23 % Symmetric FIR Filter
24
25 % declare and initialize the delay registers
26 persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
27 if isempty(ud1)
28     ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
29 end
30
31 % access the previous value of states/registers
32 a1 = ud1 + ud8; a2 = ud2 + ud7;
33 a3 = ud3 + ud6; a4 = ud4 + ud5;
34
35 % multiplier chain
36 m1 = h_in1 * a1; m2 = h_in2 * a2;
37 m3 = h_in3 * a3; m4 = h_in4 * a4;
38
39 % adder chain
40 a5 = m1 + m2; a6 = m3 + m4;
    
```

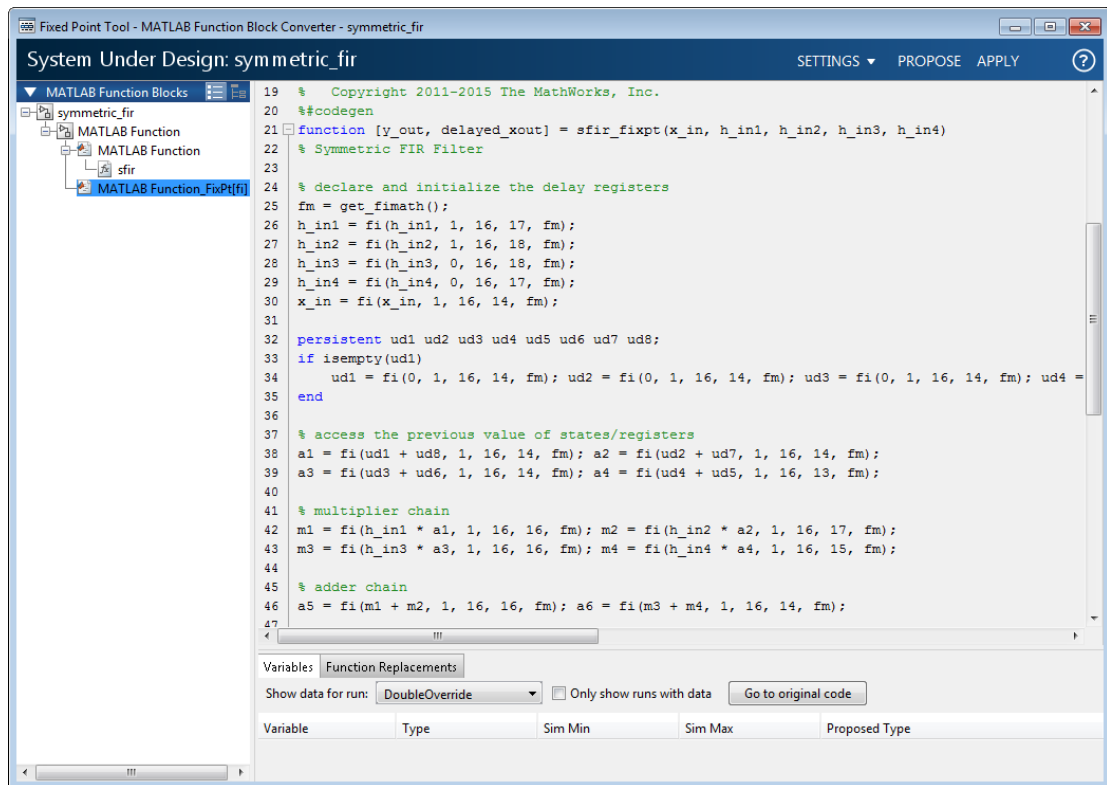
Variable	Type	Sim Min	Sim Max	Proposed Type
Input				
x_in	double	-1	1	numerictype(1, 16, 14)
h_in1	double	-0.13	-0.13	numerictype(1, 16, 17)
h_in2	double	-0.08	-0.08	numerictype(1, 16, 18)
h_in3	double	0.2	0.2	numerictype(0, 16, 18)
h_in4	double	0.41	0.41	numerictype(0, 16, 17)
Output				
y_out	double	-1.22	1.22	numerictype(1, 16, 14)
delayed_xout	double	-1	1	numerictype(1, 16, 14)

Apply Proposed Data Types

When you have finished examining the proposed types, editing proposal settings, and implementing any function replacements, apply the proposed data types to the model. You can apply the data types either from the code view, or from the Fixed-Point Tool.

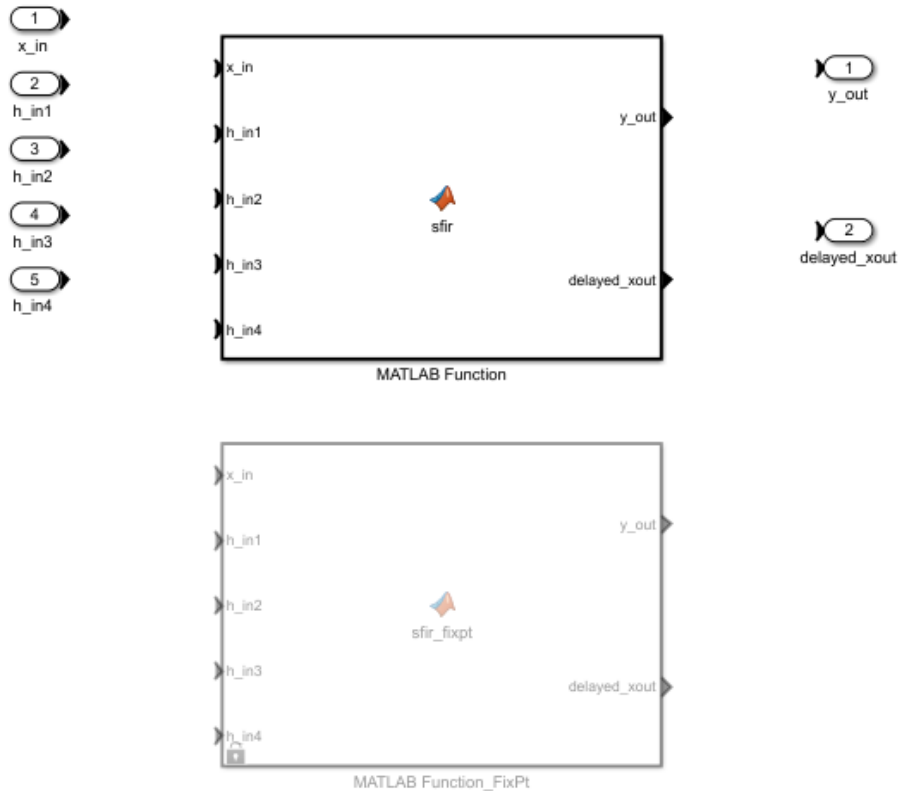
In the code view window, click **Apply**. The left pane displays both the original floating-point MATLAB Function block, as well as a newly generated fixed-point variant MATLAB Function block.

Right-click on the MATLAB Function block node in the left pane. Select **Go to Block** to navigate to the MATLAB Function block in the model.



A variant subsystem is now in the place of the MATLAB Function block. The variant subsystem contains both floating-point and fixed-point versions of the MATLAB Function

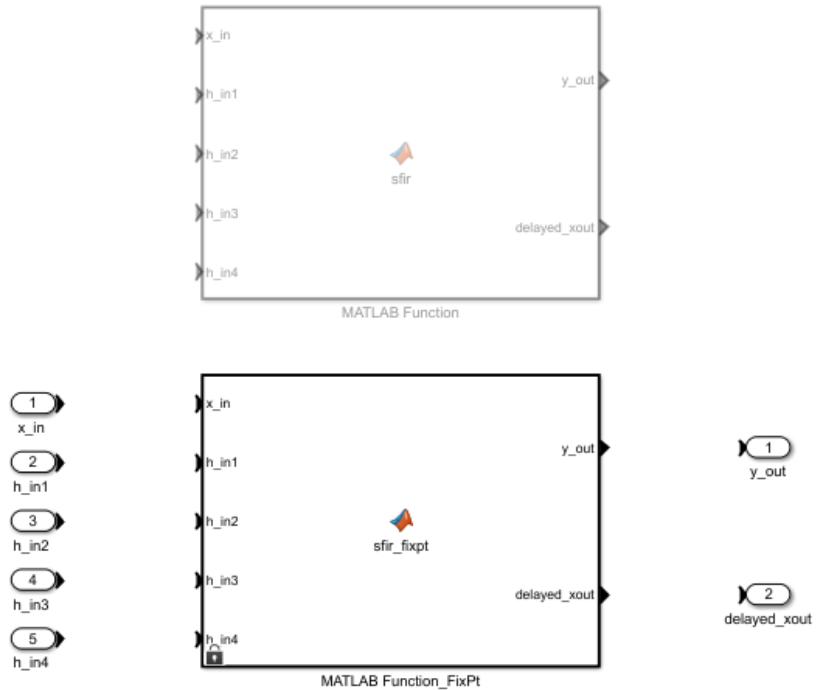
block. The active version is automatically controlled by the Fixed-Point Tool based on the data type override settings of the model. Because double-override is still active on the model, the floating-point version is active.



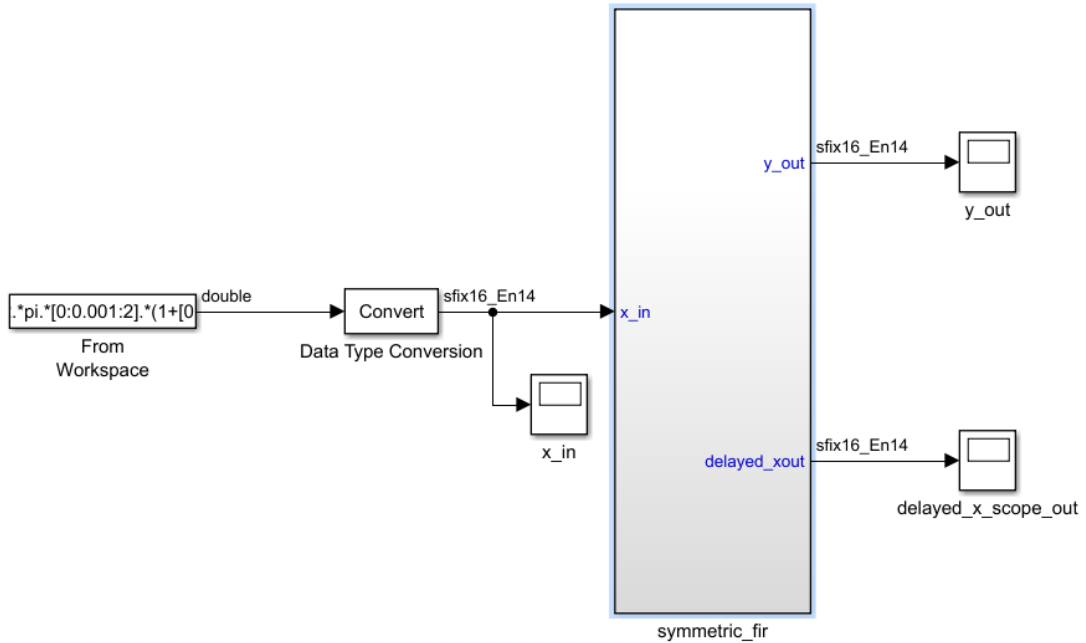
Verify Results

Return to the Fixed-Point Tool to verify the results of the conversion.

Under **Configure model settings**, click the **Range collection with specified data types** button. This setting removes data type override from the model and changes the active variant to the fixed-point MATLAB Function block. This configuration also sets the **Run name** to NoOverride.



Under **Collect ranges**, click **Simulate model** to collect ranges using the newly applied fixed-point data types. The model simulates with the fixed-point variant as the active variant.



Related Examples

- “Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool” on page 41-11
- “Replace Functions in a MATLAB Function Block with a Lookup Table” on page 41-12
- “Derive Ranges of MATLAB Function Block Variables” on page 41-16

Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool

- Do not edit the fixed-point variant of your MATLAB Function block algorithm. Use the code view to edit the floating-point variant of your MATLAB code and repropose and apply data types.
- For a successful conversion, only use modeling constructs supported by the Fixed-Point Converter app. For a list of the supported modeling constructs, see “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 8-42.
- While collecting range information, do not edit the MATLAB code in the MATLAB Function block. Editing the code will cause problems if you try to merge results.
- During the fixed-point conversion process using the Fixed-Point Tool, do not use the “Save as” option to save the MATLAB Function block with a different name. If you do, you might lose existing results for the original block.
- Override the data types in your model with scaled doubles to detect potential overflows within a MATLAB Function block.

Related Examples

- “Convert MATLAB Function Block to Fixed Point” on page 41-2
- “Replace Functions in a MATLAB Function Block with a Lookup Table” on page 41-12
- “Derive Ranges of MATLAB Function Block Variables” on page 41-16

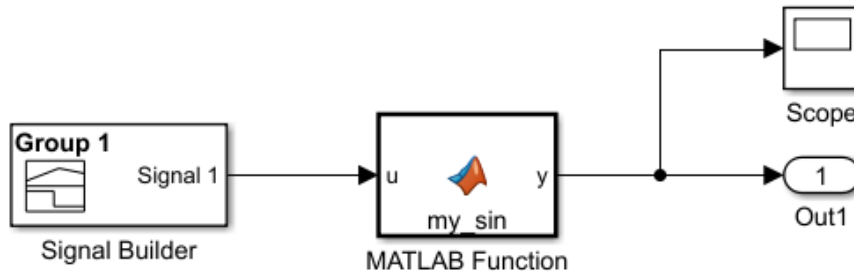
Replace Functions in a MATLAB Function Block with a Lookup Table

This example shows how to replace a function that is used inside a MATLAB Function block, with a more efficient implementation. The following model contains a MATLAB Function block which computes the sine of the input. Use the Code View to replace the built-in `sin` function with a lookup table.

Change directories to the folder where the model is located. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

Copy and save the `ex_mySin.slx` file to a local writable folder and open the model.




```
function y = my_sin(u)
%#codegen
y = sin(u);
```

- 1 From the model menu, select **Analysis > Data Type Design > Fixed-Point Tool**.
- 2 In the **Workflow** pane, under **System under design**, click **Continue** to continue the conversion of the entire model.

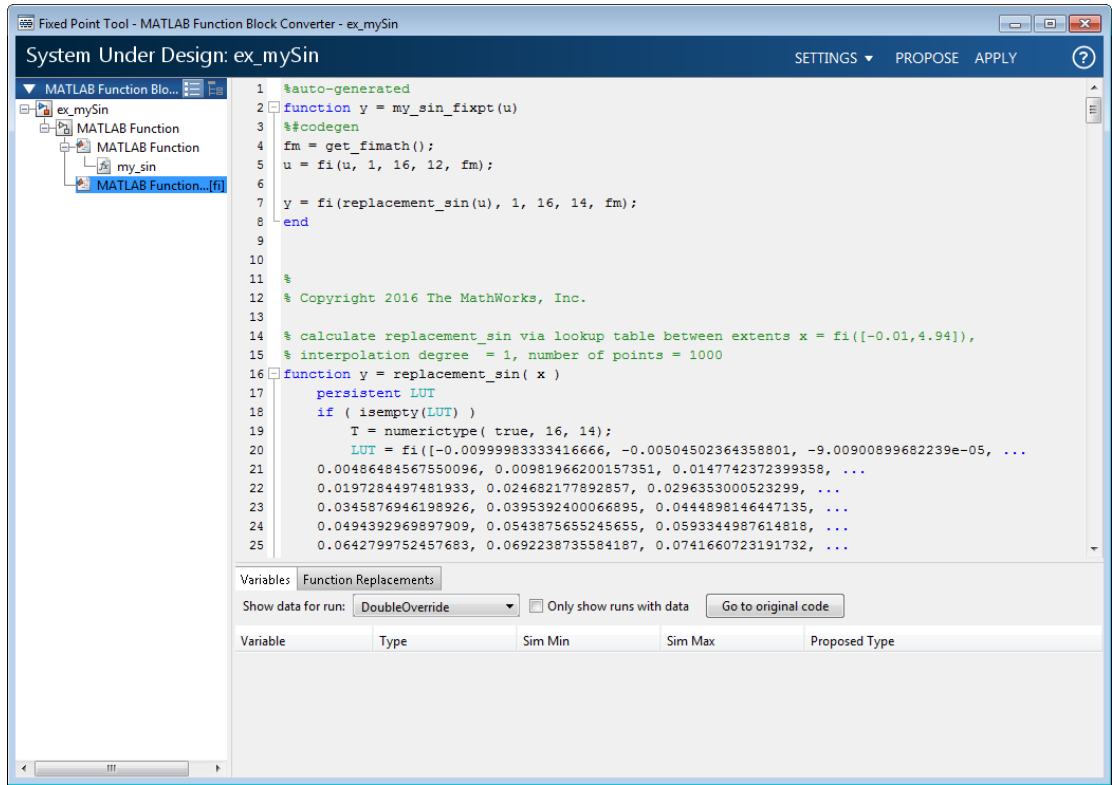
- 3 Under **Configure model settings**, click the **Range collection using double override** button to configure the model to collect ranges using idealized floating-point data types. This configuration also sets the **Run name** to DoubleOverride.
- 4 Under **Collect ranges**, click **Simulate model**.

Examine the range information of the MATLAB variables in the **Contents** pane.

- 5 To launch the code view, under **Automatic data typing**, click **Inspect MATLAB Function blocks**.
- 6 Select the **Function Replacements** tab.
- 7 Enter the name of the function you want to replace. For this example, enter `sin`. Select **Lookup Table**, and then click .

The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

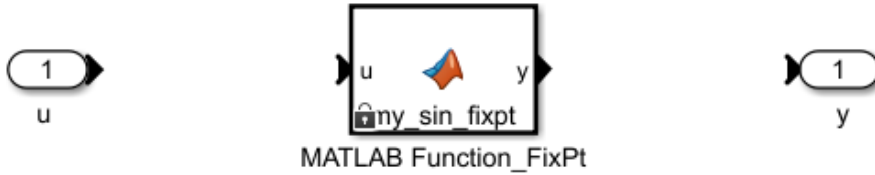
- 8 Click **Propose** to get data type proposals for the variables.
- 9 Click **Apply** to apply the data type proposals and generate a fixed-point lookup table.



If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate the fixed-point code.

- 10 Return to the Fixed-Point Tool. Under **Configure model settings**, click the **Range collection with specified data types** button.
- 11 Under **Collect ranges**, click **Simulate model**.

The model simulates using the generated fixed-point version of the function.



Related Examples

- “Convert MATLAB Function Block to Fixed Point” on page 41-2
- “Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool” on page 41-11

Derive Ranges of MATLAB Function Block Variables

This example shows how to use the Fixed-Point Tool to convert a model containing a MATLAB Function block to fixed point using range analysis.

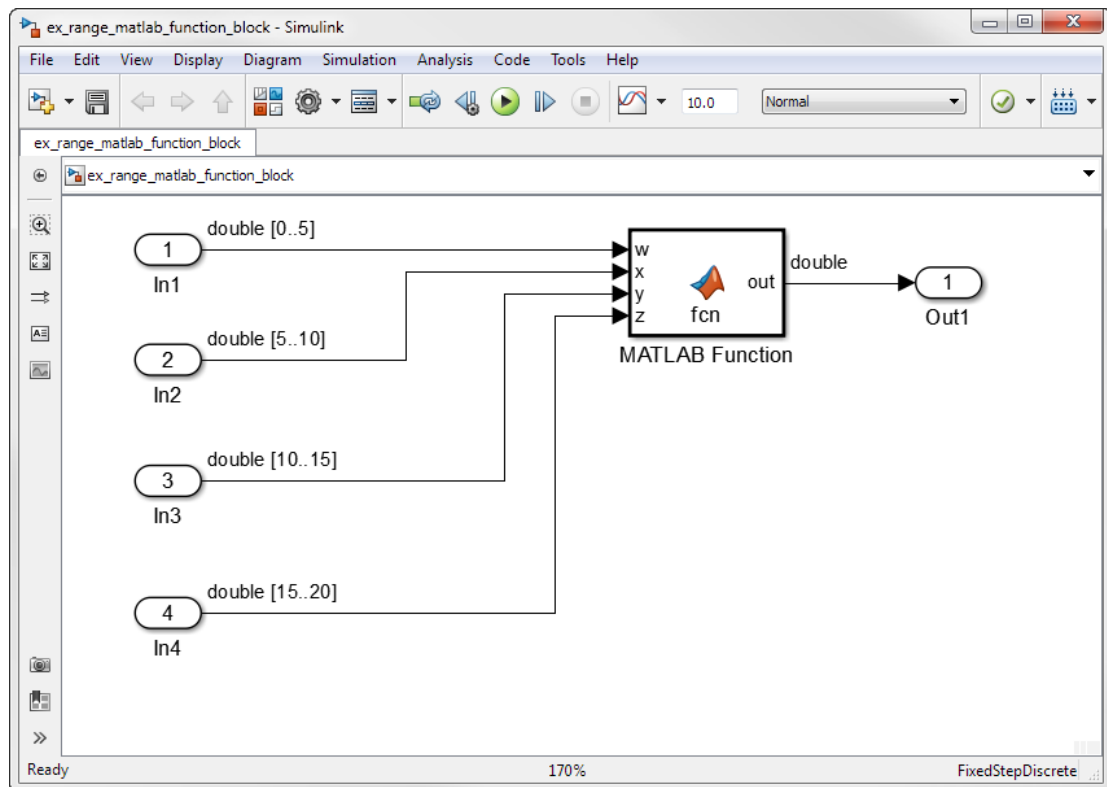
Explore the Model

Open the folder where the model is located. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

Copy the `ex_range_matlab_function_block.slx` file to a writable folder and open the model.

Update the diagram to view design range information for each of the input ports.



Open the MATLAB Function block. The function averages the four inputs.

```
function out = fcn(w,x,y,z)
%#codegen
sum = w+x+y+z;
out = sum/4;
end
```

Derive Ranges

- 1 Open the Fixed-Point Tool. Select **Analysis > Data Type Design > Fixed-Point Tool**.
- 2 In the Workflow pane, under **System under design**, click **Continue** to select `ex_range_matlab_function_block/MATLAB Function` as the system to convert.
- 3 Under **Range collection**, click the **Derive ranges for System Under Design** button to derive ranges for the MATLAB variables.

Examine the derived ranges in the **Contents** pane.

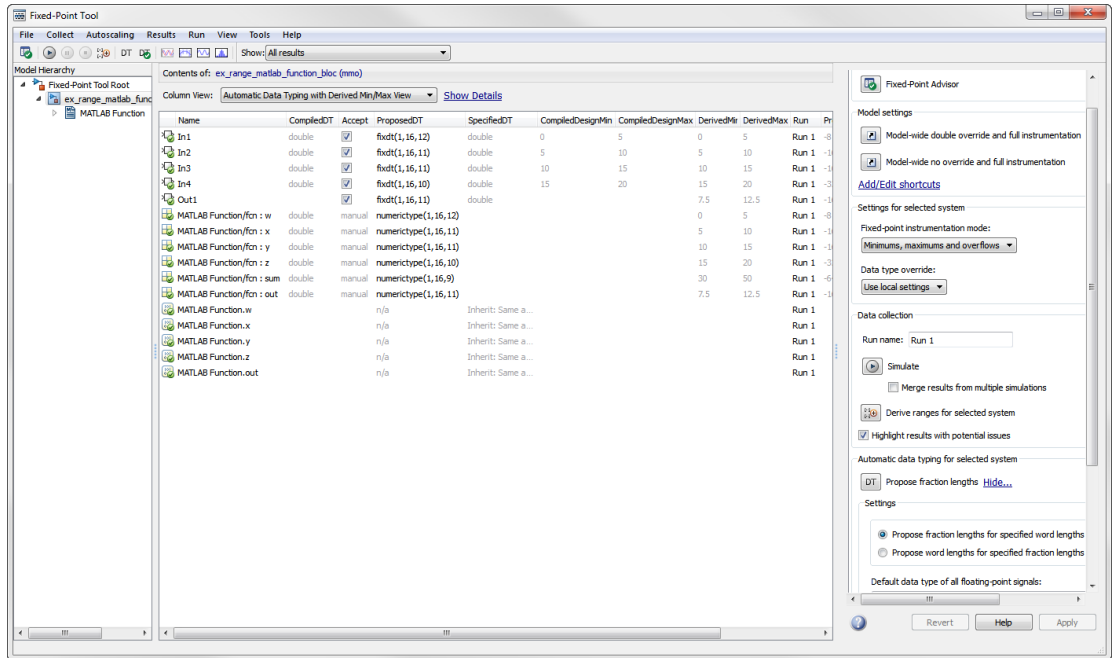
Propose Data Types

Configure the proposal settings and propose fixed-point data types for the model based on derived ranges.

- 1 In the workflow tab, under **Automatic data typing for selected system**, click **Configure**.
- 2 Set **Default data type of all floating point signals** to `int16`.

The Fixed-Point Tool uses the derived range information to propose data types using a 16-bit word length.

- 3 Click the **Propose fraction lengths** button.
- 4 The data type proposals appear in the **ProposedDT** column of the **Contents** pane.



Apply Proposed Data Types

Under Automatic data typing, click **Apply accepted data types**. This action creates a variant subsystem that contains the original floating-point MATLAB Function block, and the automatically generated fixed-point MATLAB Function block.

More About

- “Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool” on page 41-11

Convert a System Object to Fixed Point Using the Fixed-Point Tool

In this section...

“Using the Fixed-Point Tool with System Objects” on page 41-19

“Convert a System Object to Fixed-Point” on page 41-20

Using the Fixed-Point Tool with System Objects

You can use the Fixed-Point Tool to propose and apply fixed-point data types for commonly used System objects. The proposed data types are based on simulation data from the System objects.

Instrumentation and data type proposals are available for these DSP System Toolbox System objects:

- `dsp.ArrayVectorAdder`
- `dsp.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.LUFactor`
- `dsp.UpperTriangularSolver`
- `dsp.VariableFractionalDelay`
- `dsp.Window`

The following limitations apply when working with System objects in the Fixed-Point Tool.

- You cannot propose data types for these System objects based on static range data.
- You must configure the System object to use 'CUSTOM' fixed-point settings.
- The tool treats scaled doubles as fixed point. The scaled doubles workflow for System objects is the same as for regular variables.
- The Fixed-Point Tool ignores the **Default word length** setting under **Automatic data typing**. Data type proposals are based on the settings of the System object.

Convert a System Object to Fixed-Point

This example converts a MATLAB Function block that uses a `dsp.FIRFilter` System object to fixed point using the Fixed-Point Tool.

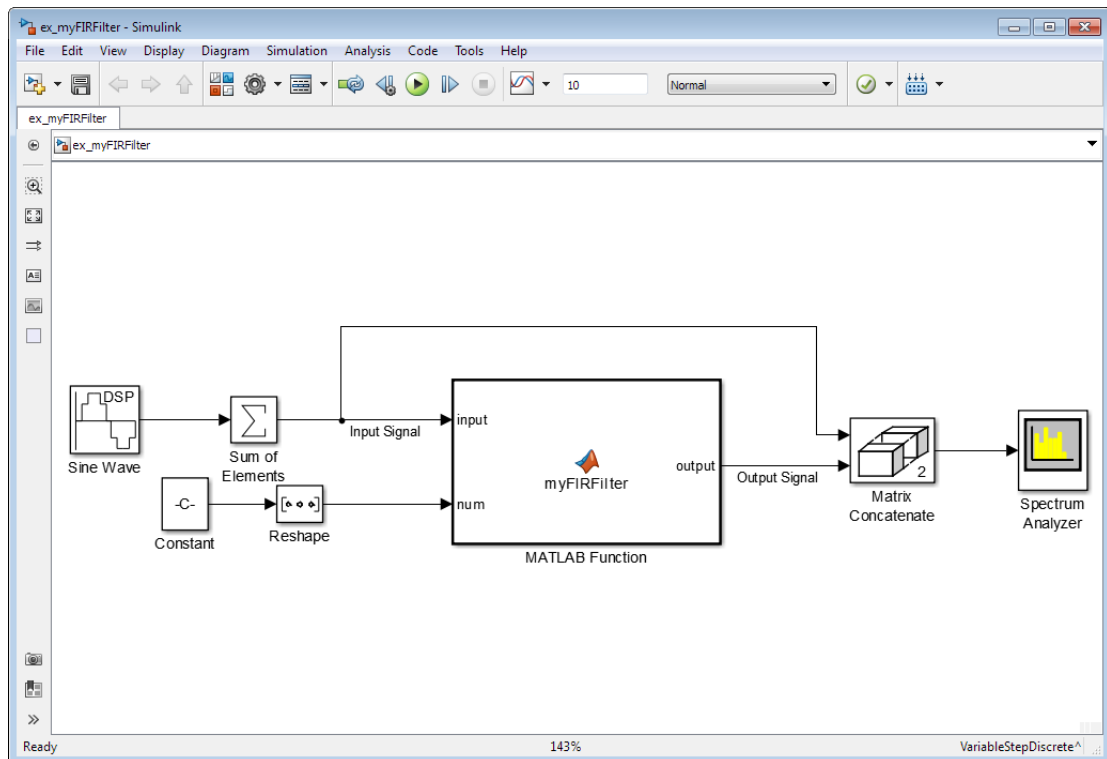
Open Model

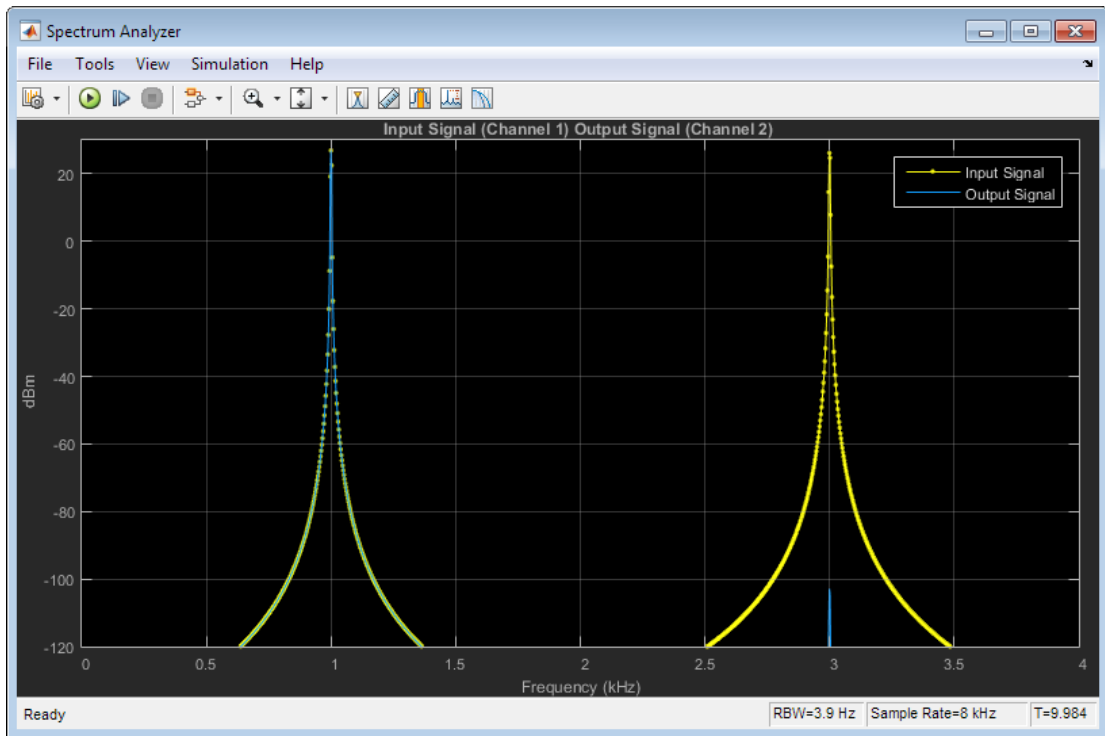
Open the folder where the model is located. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

Copy the `ex_myFIRFilter.slx` file to a writable folder and open the model.

The MATLAB Function block contains a System object that filters a high-frequency sinusoid signal.





Collect Range Information

- 1 Right-click the MATLAB Function block and select **Fixed-Point Tool**.
- 2 In the Workflow pane, under **System under design**, click **Continue** to select `ex_myFIRfilter/MATLAB Function` as the system to convert.
- 3 Under **Configure model settings**, click **Range collection with specified data types**. This shortcut button sets **Run name** under **Range collection** to `NoOverride`.
- 4 Under **Range collection**, click **Simulate model** to store the range information in the `NoOverride` run.

Examine the range information in the **Contents** pane.

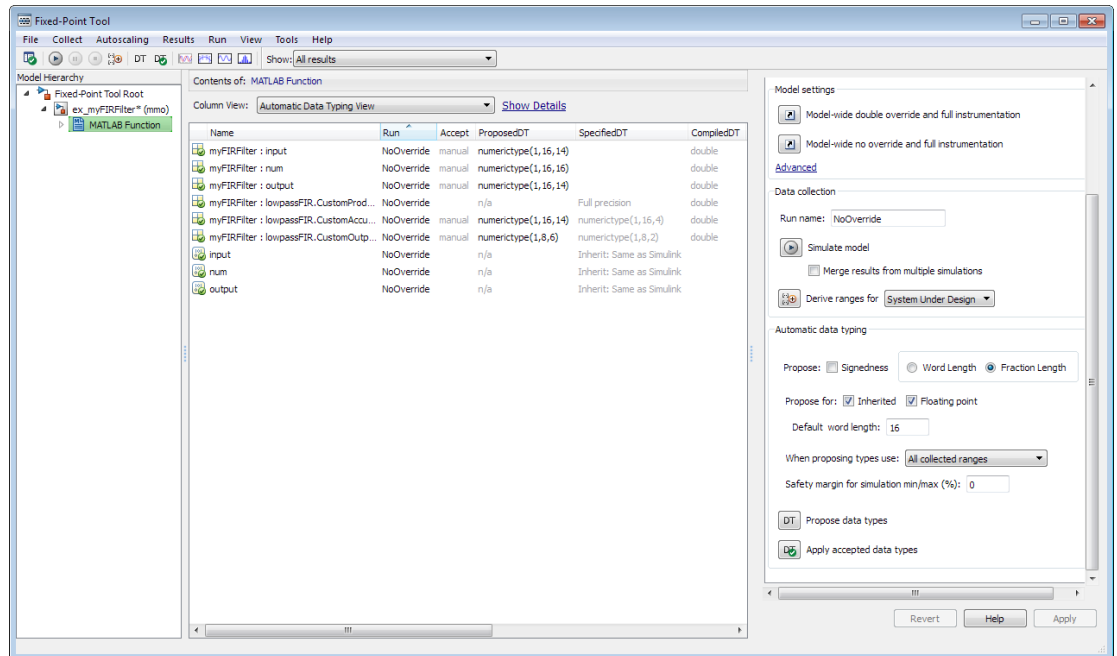
Propose Data Types

Configure proposal settings and propose fixed-point data types for the model.

- 1 Under **Automatic data typing**, configure the data type proposal settings for the System object. In this example, use the default proposal settings to propose data types for the variables in the MATLAB Function block.
- 2 Click **Propose data types**. The data type proposals appear in the **ProposedDT** column of the **Contents** pane.

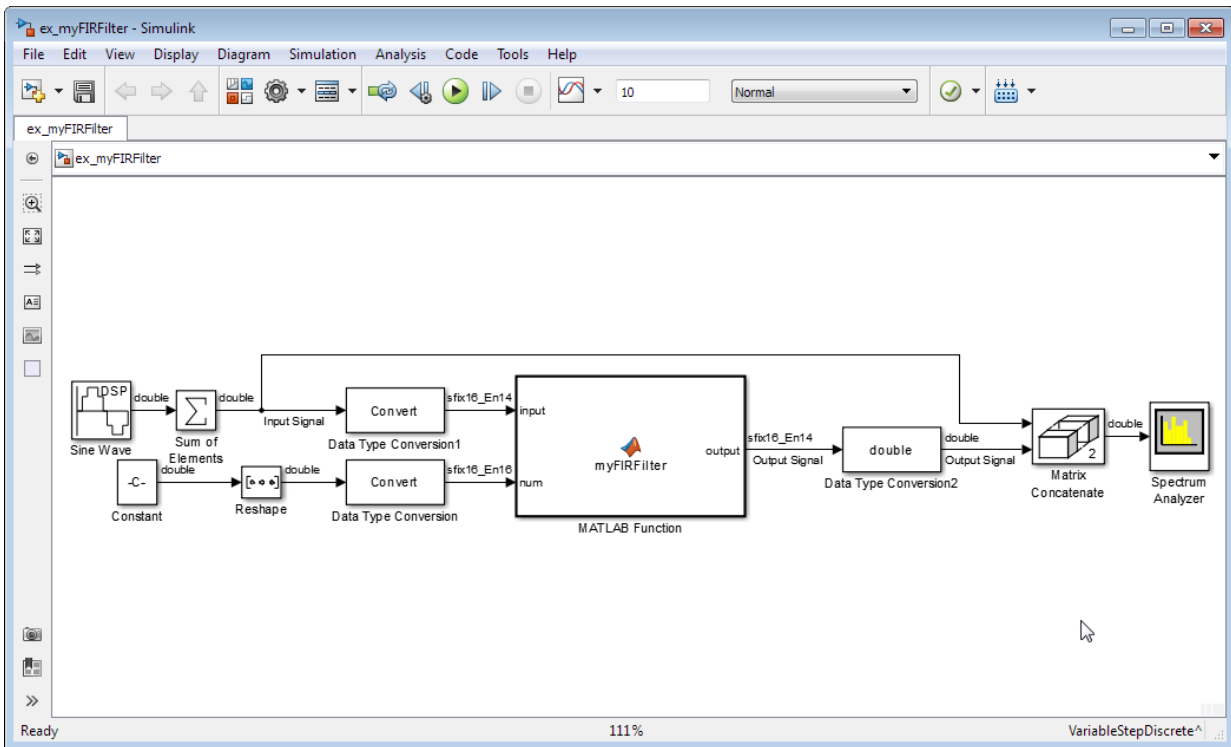
The Fixed-Point Tool did not propose a data type for the `lowpassFIR.CustomProductDataType` property. The Fixed-Point Tool proposes data types only for fields of System objects that are set to 'Custom'.

- 3 To view more information about a proposal, select a result in the **Contents** pane, and select the **Result Details** tab.



Apply Proposed Data Types

Under Automatic data typing, click **Apply accepted data types**. This action creates a variant subsystem that contains the original floating-point MATLAB Function block, and the automatically generated fixed-point MATLAB Function block.



The MATLAB Function block now uses the more precise fixed-point data types proposed by the Fixed-Point Tool.

See Also

“Best Practices for Working with the MATLAB Function Block in the Fixed-Point Tool” on page 41-11

Working with Data Objects in the Fixed-Point Workflow

- “Refine Data Types of a Model with Buses Using Simulation Data” on page 42-2
- “Convert a Model with Buses to Fixed-Point Using Range Analysis” on page 42-11
- “Bus Objects in the Fixed-Point Workflow” on page 42-19
- “Convert Model with Bus Object with Structure Initial Conditions to Fixed-Point” on page 42-23
- “Autoscale Simulink.AliasType Objects” on page 42-27
- “Autoscale Simulink.NumericType Objects” on page 42-30
- “Autoscale Simulink.Parameter Objects” on page 42-33

Refine Data Types of a Model with Buses Using Simulation Data

In this section...

“Open and Simulate the Model” on page 42-2

“Use Data Type Override to Resolve Overflows” on page 42-3

“Propose New Fraction Lengths” on page 42-5

“Examine and Apply Proposed Types” on page 42-6

“Verify New Fixed-Point Settings” on page 42-8

“Save the Model and New Bus Data Types” on page 42-9

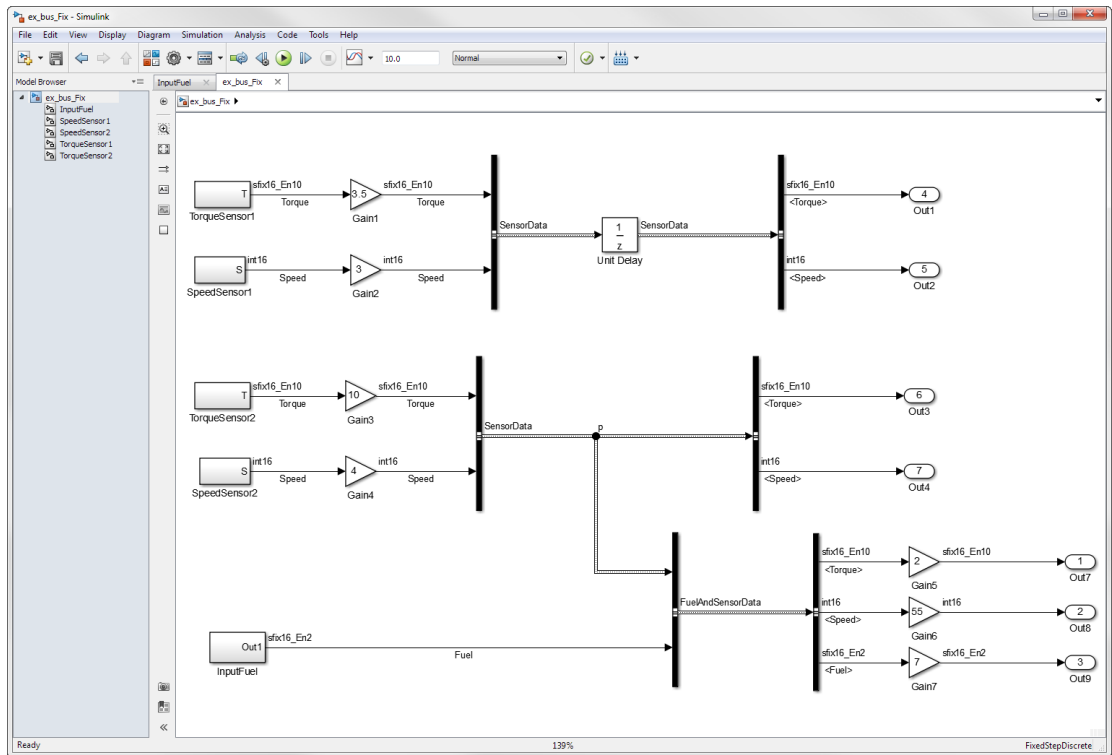
This example shows how to refine the fixed-point types of a model with bus objects based on simulation data. The following video provides an overview of automatic data typing and data type override for bus objects: Automatic Data Typing for Bus Objects.

Open and Simulate the Model

Open the `ex_bus_Fix` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_bus_Fix
```

- 1 Update the diagram. Notice the use of fixed-point data types in the model.



2 Simulate the model.

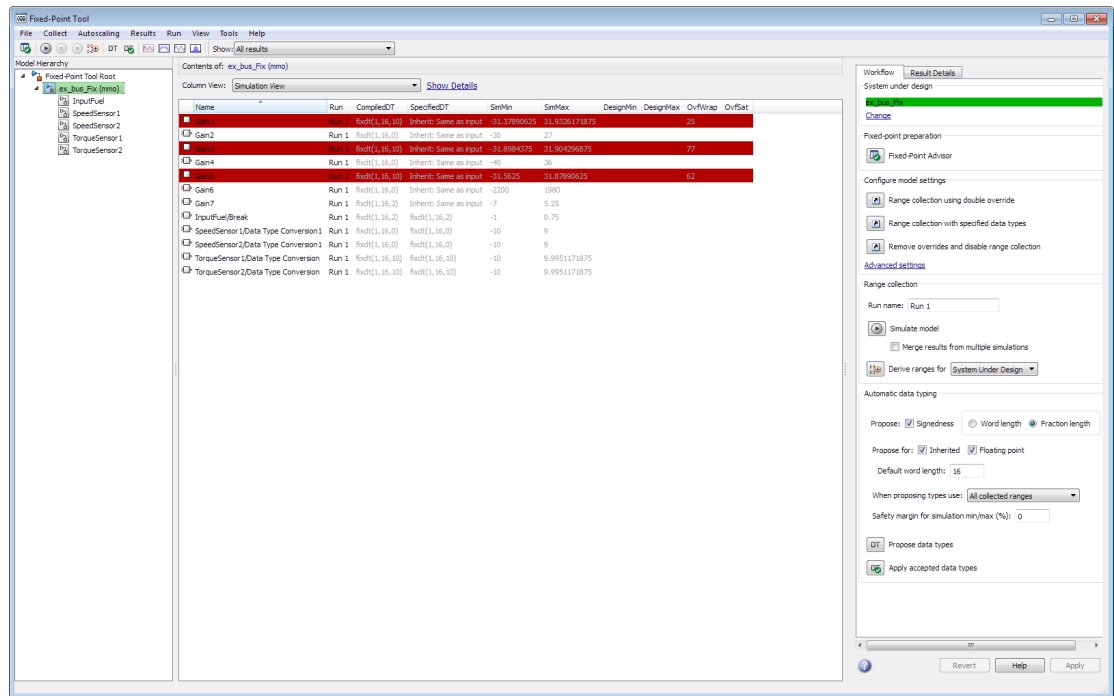
The Diagnostic Viewer warns that several overflows occurred during simulation.

Use Data Type Override to Resolve Overflows

- 1 Open the Fixed-Point Tool from the model menu by selecting **Analysis > Data Type Design > Fixed-Point Tool**.

The rows containing the Gain3, Gain5, and Gain1 blocks are highlighted in red, indicating where the overflows occurred.

- 2 Under **System under design**, click **Continue** to select `ex_bus_Fix` as the system for conversion.



- 3 Obtain the ideal numerical behavior of the model by overriding all of the data types with doubles.

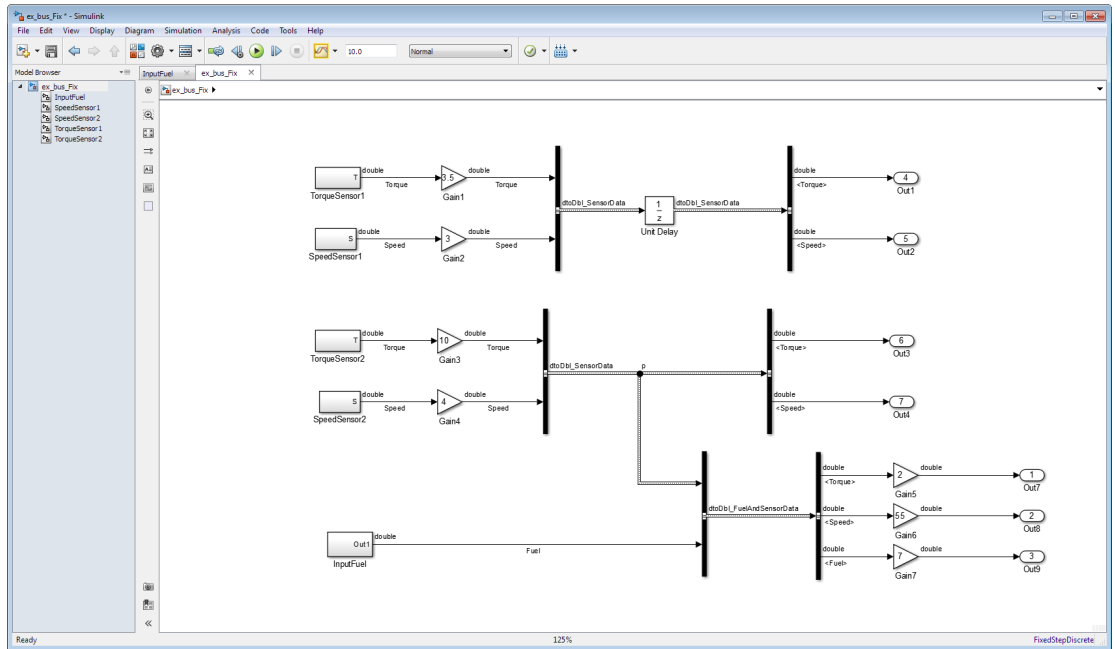
Under **Configure model settings**, click the **Range collection using double override** button. This button overrides all of the data types in the model with doubles and enables instrumentation to collect range information.

Click **Simulate**.

The model is simulated using floating-point doubles, so this run does not contain any overflows. The Fixed-Point Tool stores range information in the run titled **DoubleOverride**.


- 4 Return to the model. A prefix has been added to the names on the bus signals. This prefix indicates that the bus objects are replaced by temporary overridden bus objects specifying double data types. For more information, see “Bus Naming Conventions with Data Type Override” on page 42-21.

Overridden bus objects are not stored in the base workspace, so you cannot see them in the Bus Editor. The Bus Editor continues to display the data types of the bus objects before they were overridden.



Propose New Fraction Lengths

Propose new fraction lengths for your model based on the simulation results from the DoubleOverride run.

- 1 Under **Automatic data typing**, click the **Propose data types** button .
- 2 In the **Propose Data Types** dialog box, select **DoubleOverride** as the run to use for proposing data types. Click **OK**.

The proposed data types appear in the **Contents** pane of the Fixed-Point Tool.

The Fixed-Point Tool uses the default proposal settings to propose new data types for objects in the model.

Examine and Apply Proposed Types

Examine the new data types that the Fixed-Point Tool proposed for `ex_bus_Fix`.

- 1 On the Fixed-Point Tool toolbar, select **Show > Groups that must share the same data type** to better understand the proposed data types.
- 2 In the **Contents** pane, right-click the column labeled **DTGroup** and select **Group by This Column**.

The Fixed-Point Tool organizes the **Contents** pane to group the elements by data type.

The screenshot shows the Fixed-Point Tool interface. The main window displays a table of data type conversions. The table has columns: Name, Run, DTGroup, Inherit, ProposedDT, SpecifiedDT, CompletedDT, SimMin, SimMax, ProposedMin, ProposedMax, and Ovr. The DTGroup column is highlighted, and the 'Show Details' button is visible. The table is grouped by DTGroup, showing various data type conversions for different sensors and data types.

Name	Run	DTGroup	Inherit	ProposedDT	SpecifiedDT	CompletedDT	SimMin	SimMax	ProposedMin	ProposedMax	Ovr
DTGroup G3 (11)											
TorqueSensor2Data Type Conversion	DoubleOverride	G3	✓	fixed(L,16,7)	fixed(L,16,10)	double	-9.999232175641008	9.999736030415055	-256	255.9921875	
can3	DoubleOverride	G3	✓	fixed(L,16,7)	Inherit: Same as input	double	-99.99232175641008	99.95736030415055	-256	255.9921875	
can5	DoubleOverride	G3	✓	fixed(L,16,7)	Inherit: Same as input	double	-199.98465151282016	199.91472060830102	-256	255.9921875	
TorqueSensor1Data Type Conversion	DoubleOverride	G3	✓	fixed(L,16,7)	fixed(L,16,10)	double	-9.999232175641008	9.999736030415055	-256	255.9921875	
can1	DoubleOverride	G3	✓	fixed(L,16,7)	Inherit: Same as input	double	-9.999232175641008	9.999736030415055	-256	255.9921875	
Out7	DoubleOverride	G3	✓	n/a	Inherit: auto						
Out1	DoubleOverride	G3	✓	n/a	Inherit: auto						
Out3	DoubleOverride	G3	✓	n/a	Inherit: auto						
TorqueSensor17	DoubleOverride	G3	✓	n/a	Inherit: auto						
TorqueSensor217	DoubleOverride	G3	✓	n/a	Inherit: auto						
SensorData : Torque	DoubleOverride	G3	✓	fixed(L,16,7)	fixed(L,16,10)				-256	255.9921875	
DTGroup G4 (11)											
SpeedSensor2Data Type Conversion1	DoubleOverride	G4	✓	fixed(L,16,3)	fixed(L,16,6)	double	-9.999232175641008	9.999736030415055	-4096	4095.875	
can4	DoubleOverride	G4	✓	fixed(L,16,3)	Inherit: Same as input	double	-39.999830320206402	39.98294412166022	-4096	4095.875	
can5	DoubleOverride	G4	✓	fixed(L,16,3)	Inherit: Same as input	double	-2199.821366640222	2199.8010366931212	-4096	4095.875	
SpeedSensor1Data Type Conversion1	DoubleOverride	G4	✓	fixed(L,16,3)	fixed(L,16,6)	double	-9.999232175641008	9.999736030415055	-4096	4095.875	
can2	DoubleOverride	G4	✓	fixed(L,16,3)	Inherit: Same as input	double	-29.997697726820322	29.98720809124515	-4096	4095.875	
Out8	DoubleOverride	G4	✓	n/a	Inherit: auto						
Out2	DoubleOverride	G4	✓	n/a	Inherit: auto						
Out4	DoubleOverride	G4	✓	n/a	Inherit: auto						
SpeedSensor15	DoubleOverride	G4	✓	n/a	Inherit: auto						
SpeedSensor25	DoubleOverride	G4	✓	n/a	Inherit: auto						
SensorData : Speed	DoubleOverride	G4	✓	fixed(L,16,3)	fixed(L,16,6)				-4096	4095.875	
DTGroup G5 (8)											
InpuFuelBreak	DoubleOverride	G5	✓	fixed(L,16,12)	fixed(L,16,1)	double	-0.999923275641008	0.9995736030415055	-8	7.99975589375	
can7	DoubleOverride	G5	✓	fixed(L,16,12)	Inherit: Same as input	double	-6.999462802948706	6.9970152211296535	-8	7.99975589375	
Out9	DoubleOverride	G5	✓	n/a	Inherit: auto						
InpuFuel/Out1	DoubleOverride	G5	✓	n/a	Inherit: auto						
Fuel/SensorData : Fuel	DoubleOverride	G5	✓	fixed(L,16,12)	fixed(L,16,1)				-8	7.99975589375	
DTGroup G6 (1)											
Bus Creator 10	DoubleOverride	G6	✓	n/a							

- 3 In the **Contents** pane, select the **SensorData : Speed** bus element (element of the bus object). Click the **Result Details** tab.

In the **Result Details** tab, view the details of the proposed data type for the bus element. The data type proposal for a bus element is found by taking the union of the ranges of all blocks that must share the same data type, which includes all of

the sources driving the bus element. The software then proposes a data type for this range.

Workflow Result Details

SensorData : Speed

Proposed Data Type Summary

- There is a requirement for the data type of this result to match the data type of other results.
 - [Highlight Elements Sharing Same Data Type](#)

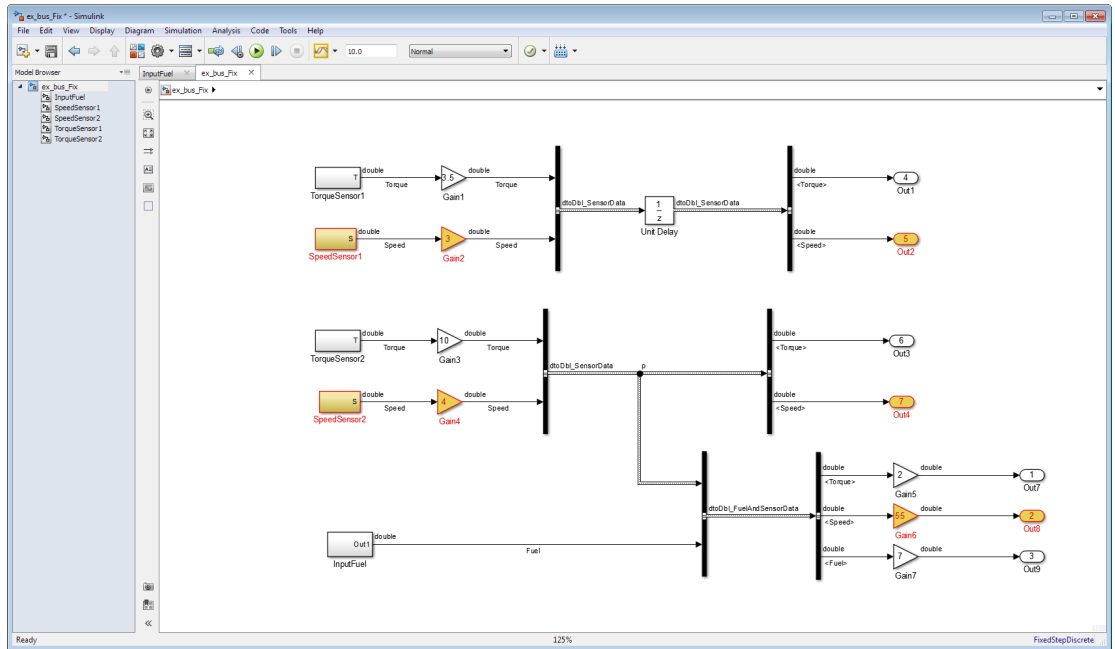
Property	Data Type	Minimum	Maximum	Precision
ProposedDT	fixdt(1,16,3)	-4098	4095.875	0.125
SpecifiedDT	fixdt(1,16,0)	-32768	32767	1


Range Information

Property	Minimum	Maximum
Shared Simulation	-2199.831054	2199.0620117
Shared Initial Value	0	0
Initial Value	0	0

The **Shared Simulation** minimum and maximum values of the **Range Information** table represent the union of the ranges of the sources driving the bus element. The data type proposal for **SensorData : Speed** is based on this range.

Right-click the bus element and select **Highlight Elements Sharing Same Data Type** to see all of the sources driving the bus element.



- 4 Return to the Fixed-Point Tool **Workflow** tab. Click the **Apply accepted data types** button  to write the proposed data types to the model.

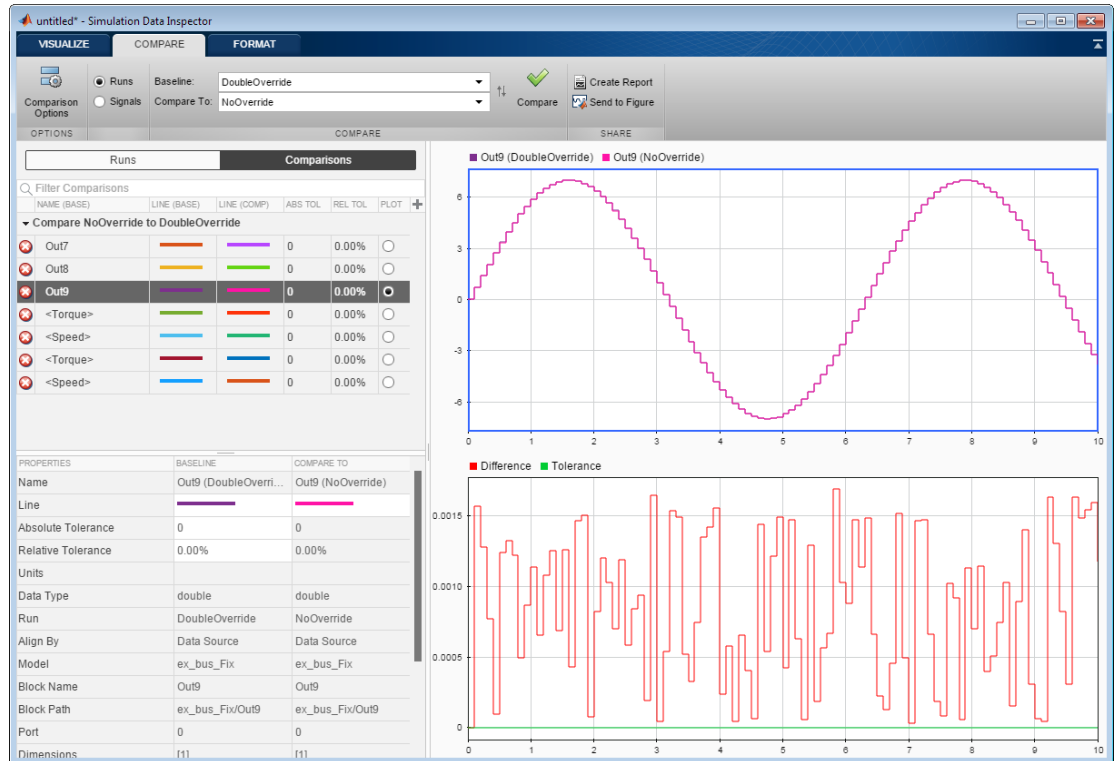
Verify New Fixed-Point Settings

- 1 Under **Configure model settings**, click the **Range collection with specified data types** button to remove data type override. Now, when you simulate the model, it uses the new fixed-point types that you applied.
- 2 Simulate the model. This time, the Fixed-Point Tool stores the results in a new run named NoOverride.
- 3 In the Fixed-Point Tool **Contents** pane, verify that there are no overflows in the NoOverride run.
- 4 Click **Compare Runs**



to launch the Simulation Data Inspector.

In the Simulation Data Inspector, compare the results of NoOverride to the results of the DoubleOverride run.



5 Return to the model. Notice that the bus objects no longer contain the prefixes.

Open the Bus Editor from the model menu by selecting **Edit > Bus Editor**. You can now view the new, refined data types of the bus objects in the Bus Editor.

6 Click the **Remove overrides and disable range collection** button to remove instrumentation and restore model to maximum simulation speed.

Save the Model and New Bus Data Types

Data types for bus elements are stored in the base workspace, not the model. To save the fixed-point data types for the bus elements in `ex_bus_Fix`, export them from the

Bus Editor to a `.mat` file. For instructions, see [Save and Import Bus Objects \(Simulink\)](#). Alternatively, you can save workspace variables to a file at the MATLAB command line using the `save` function .

More About

- [“Bus Objects in the Fixed-Point Workflow”](#) on page 42-19

Convert a Model with Buses to Fixed-Point Using Range Analysis

In this section...

“Open the Model” on page 42-11

“Set Design Minimums and Maximums” on page 42-12

“Open the Fixed-Point Tool and Select System for Conversion” on page 42-14

“Derive Minimum and Maximum Values” on page 42-14

“Propose Fraction Lengths” on page 42-14

“Examine and Apply Proposed Types” on page 42-14

“Save the Model and the New Bus Data Types” on page 42-17

This example shows how to use range analysis to convert a floating-point model that contains buses to a fixed-point model.

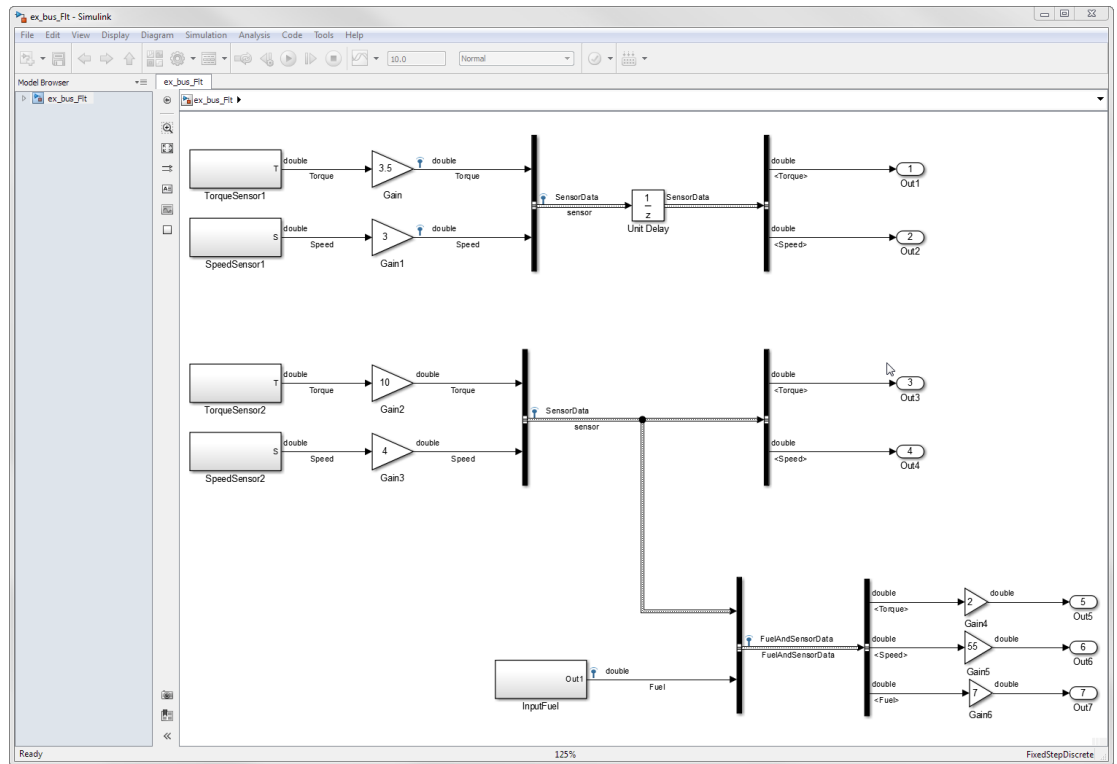
Open the Model

Open the `ex_bus_Flt` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_bus_Flt
```

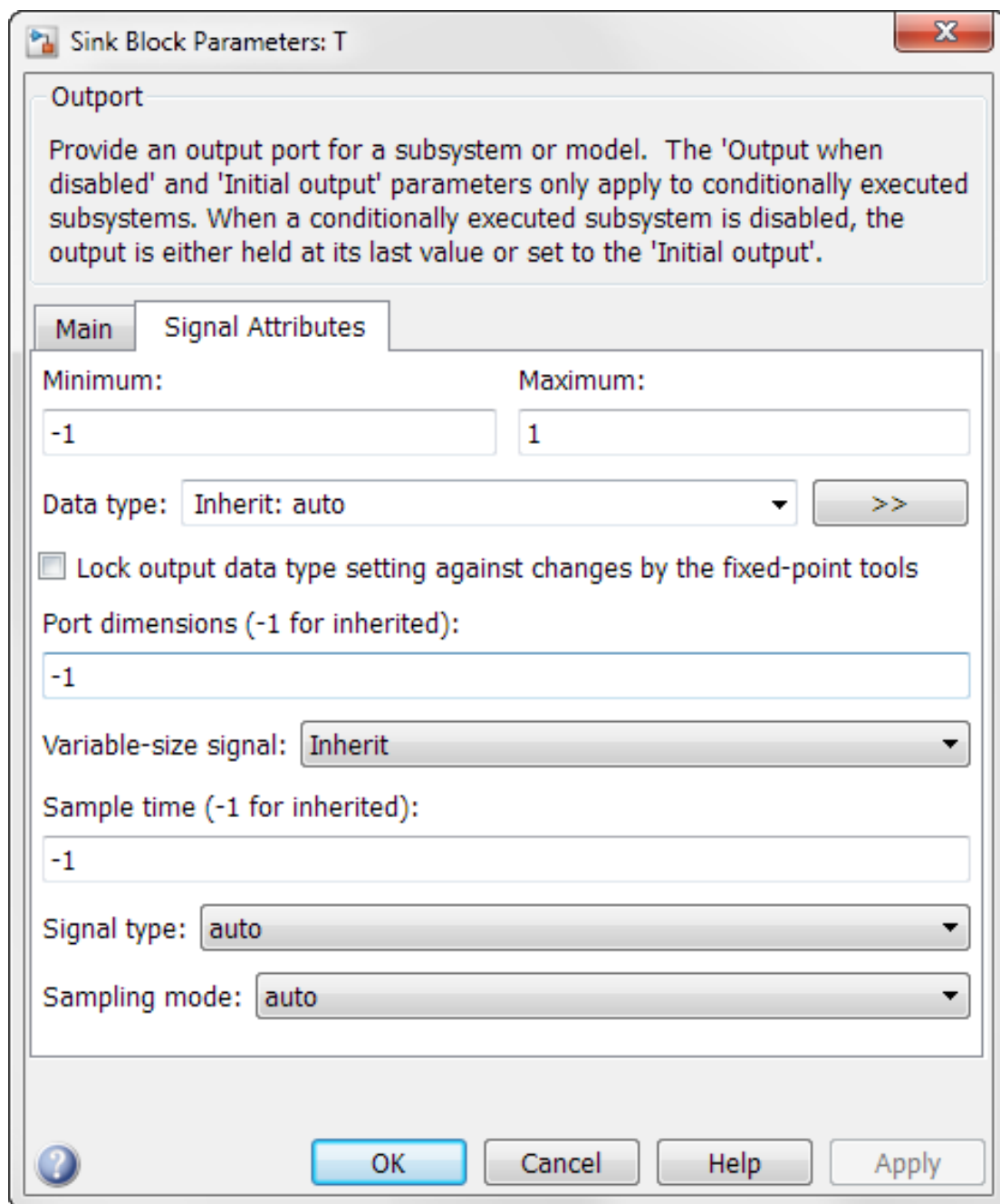
- 1 Press `ctrl+d` to update the diagram and view the data types of each block in the model.

This model uses floating-point doubles for all signals.



Set Design Minimums and Maximums

- 1 Open the TorqueSensor1 subsystem, and double-click the output port, T.
- 2 Under the **Signal Attributes** tab, notice that the design minimum and maximum have been set to -1 and 1, respectively.



- 3 Notice that the TorqueSensor2, SpeedSensor1, SpeedSensor2, and InputFuel subsystems also have their design minimums and maximums set to -1 and 1.

Open the Fixed-Point Tool and Select System for Conversion

- 1 Open the Fixed-Point Tool from the model menu by selecting **Analysis > Data Type Design > Fixed-Point Tool**.
- 2 Under **System under design**, click **Continue** to select `ex_bus_Flt` as the system for conversion.


Derive Minimum and Maximum Values

- 1 Under **Range collection**, click the **Derive ranges for System Under Design** button .

The Fixed-Point Tool **Contents** pane displays the derived minimum and maximum values for each object in the `ex_bus_Flt` model.

Propose Fraction Lengths

Propose fraction lengths for the model using the default proposal settings. The default settings propose a 16-bit word length and best-precision fraction length for the objects in the model.

- 1 Click the **Propose data types** button .

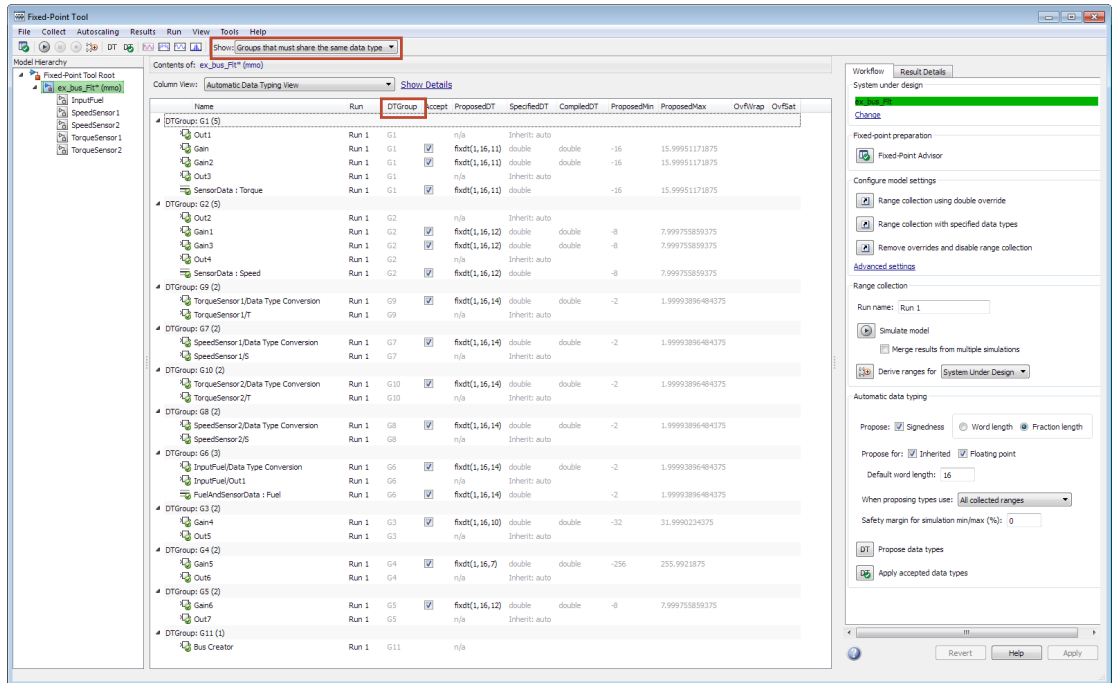
The proposed data types appear in the **Contents** pane of the Fixed-Point Tool.


Examine and Apply Proposed Types

Examine the new data types that the Fixed-Point Tool has proposed for `ex_bus_Flt`.

- 1 On the Fixed-Point Tool toolbar, select **Show > Groups that must share the same data type** to better understand the proposed data types.
- 2 In the **Contents** pane, right-click the column labeled **DTGroup** and select **Group by This Column**.

The Fixed-Point Tool organizes the contents pane to group the elements by data type.



- 3 Select the  **SensorData : Speed** bus element (element of the bus object) and click the **Result Details** tab.

In the **Result Details** tab, view the details of the proposed data type for the bus element. The software finds the data type proposal for a bus element by taking the union of the ranges of all blocks that must share the same data type, which includes all of the sources driving the bus element. It then proposes a data type for this range.

Workflow Result Details

SensorData : Speed

Proposed Data Type Summary

- There is a requirement for the data type of this result to match the data type of other results.
 - [Highlight Elements Sharing Same Data Type](#)

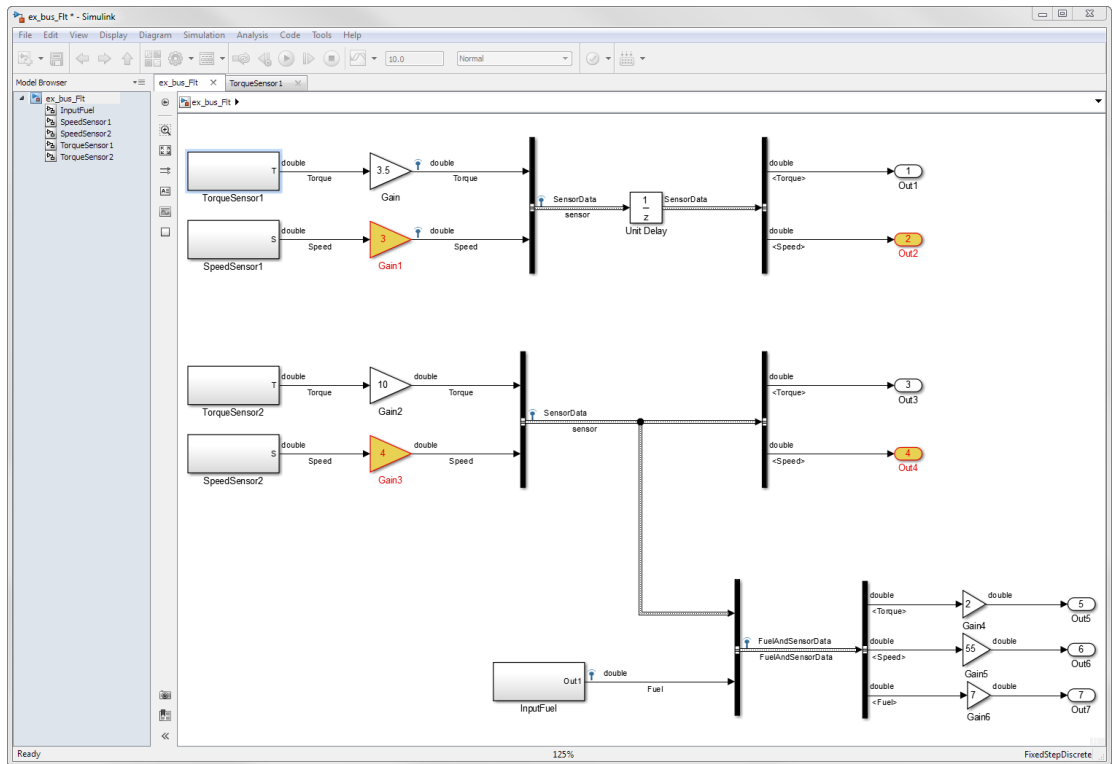
Property	Data Type	Minimum	Maximum	Precision
ProposedDT	fixdt(1,16,12)	-8	7.999755859...	0.000244140...
SpecifiedDT	double	-1.79789313...	1.797893134...	4.940656458...


Range Information

Property	Minimum	Maximum
Shared Initial Value	0	0
Initial Value	0	0
Shared derived	-4	4

The shared derived minimum and maximum in the **Range Information** table represent the union of the ranges of the sources driving the bus element. The data type proposal for **SensorData:Speed** is based on this range.

- Right-click the bus element in the **Contents** pane. Select **Highlight Elements Sharing Same Data Type** to see all of the sources driving the bus element.



- 5 Return to the Fixed-Point Tool **Workflow** tab. Click the **Apply accepted data types** button  to write the proposed data types to the model.
- 6 Open the Bus Editor from the model menu by clicking **Edit > Bus Editor**. You can now view the new fixed-point data types of the bus elements in the Bus Editor.

Save the Model and the New Bus Data Types

Data types for bus elements are stored in the base workspace, not the model. To save the fixed-point data types for the bus elements in `ex_bus_Flt`, export them from the Bus Editor to a `.mat` file. For instructions, see [Export Bus Objects \(Simulink\)](#). Alternatively, you can save workspace variables to a file at the MATLAB command line using the `save` function .

More About

- “Bus Objects in the Fixed-Point Workflow” on page 42-19

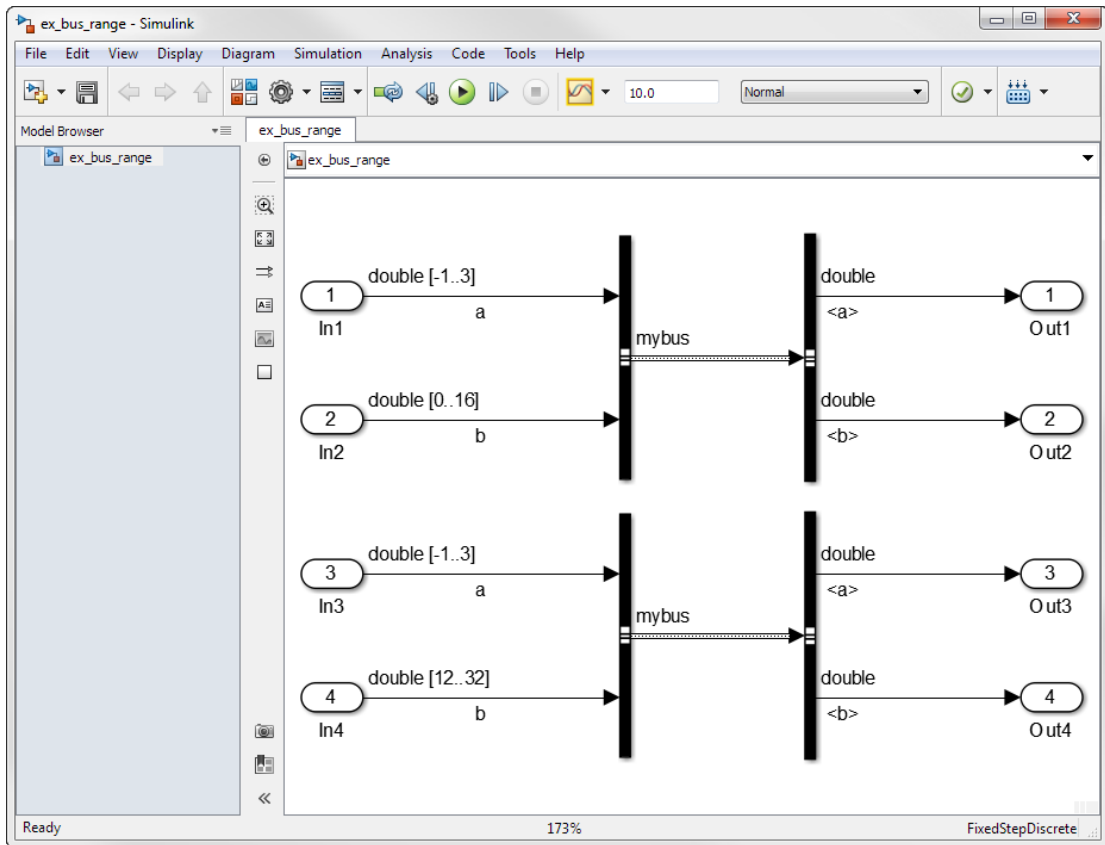
Bus Objects in the Fixed-Point Workflow

In this section...
“How Data Type Proposals Are Determined for Bus Objects” on page 42-19
“Bus Naming Conventions with Data Type Override” on page 42-21
“Limitations of Bus Objects in the Fixed-Point Workflow” on page 42-22

How Data Type Proposals Are Determined for Bus Objects

The data type proposal for a bus object is found by taking the union of the ranges of all sources driving the same bus element, and then proposing a data type for this range. The Fixed-Point Tool does not log minimum and maximum ranges for elements of a bus signal.

The following example shows how the software determines the data types for elements of bus objects.



Each of the four input ports in this model have specified design ranges. The In2 and In4 input ports must share the same data type because they drive the same element of the mybus bus object.

The Fixed-Point Tool proposes a data type based on the union of these two ranges. After proposing data types for the model, in the **Result Details** tab for the mybus : b element of the bus object, notice the row labeled **Shared Design** in the **Range Information** table. The proposed data type is based on this range, which is the union of the design ranges of the In2 and In4 blocks.

Workflow
Result Details

mybus : b

Proposed Data Type Summary

- There is a requirement for the data type of this result to match the data type of other results.
 - [Highlight Elements Sharing Same Data Type](#)

Property	Data Type	Minimum	Maximum	Precision
ProposedDT	fixdt(1,8,1)	-64	63.5	0.5
SpecifiedDT	double	-1.7976e+308	1.79769e+308	4.94065e-324

Range Information

Property	Minimum	Maximum
Shared Design	0	32

Bus Naming Conventions with Data Type Override

When you use Data Type Override on a model that contains buses, the Fixed-Point Tool generates a new bus which uses the overridden data type. To indicate that a model is using an overridden bus, the tool adds a prefix to the name of the original bus object. While a model is in an overridden state, a bus object named `myBUS` is renamed based on the following pattern.

DTO Mode	DTO Applies To		
	All numeric types	Floating point	Fixed point
Scaled Double	dtoScl_myBus	dtoSclFlt_myBus	dtoSclFxp_myBus
Double	dtoDb1_myBus	dtoDb1Flt_myBus	dtoDb1Fxp_myBus
Single	dtoSgl_myBus	dtoSglFlt_myBus	dtoSglFxp_myBus

Note: You cannot see bus objects with an overridden data type within the Bus Editor because they are not stored in the base workspace.

Limitations of Bus Objects in the Fixed-Point Workflow

An update diagram error can occur if any of the following conditions occur.

- Your model is in accelerator mode and has a bus object with an overridden data type at the output port.

To perform data type override, run the model in normal mode.

- The data types in your model are overridden and the model contains Stateflow charts that use MATLAB as the action language.
- Your model contains tunable MATLAB structures assigned to a bus signal (such as Unit Delay blocks with a structure as the initial condition, Stateflow data, and MATLAB structures from the workspace).

Change the structure to a non tunable structure to use the Fixed-Point Tool. Specify the structure fields as doubles to avoid unnecessary quantization effects. For more information on using a structure as an initial condition with bus objects, see “Data Type Mismatch and Structure Initial Conditions” on page 46-31.

- Your model contains a structure parameter specified through the mask of an atomic subsystem.

To use the Fixed-Point Tool, make the system non-atomic.

Related Examples

- “Refine Data Types of a Model with Buses Using Simulation Data” on page 42-2

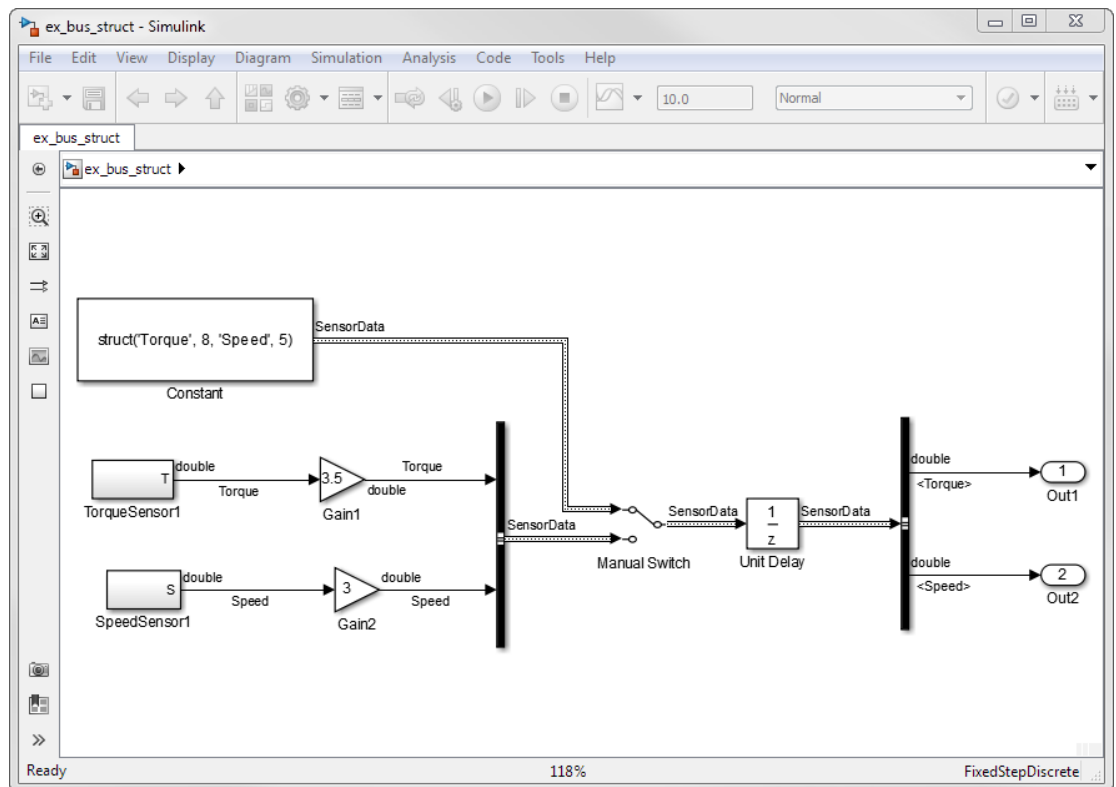
Convert Model with Bus Object with Structure Initial Conditions to Fixed-Point

This example shows how to use the Fixed-Point Tool to convert a floating-point model that contains a bus signal with a structure as the initial condition to fixed-point.

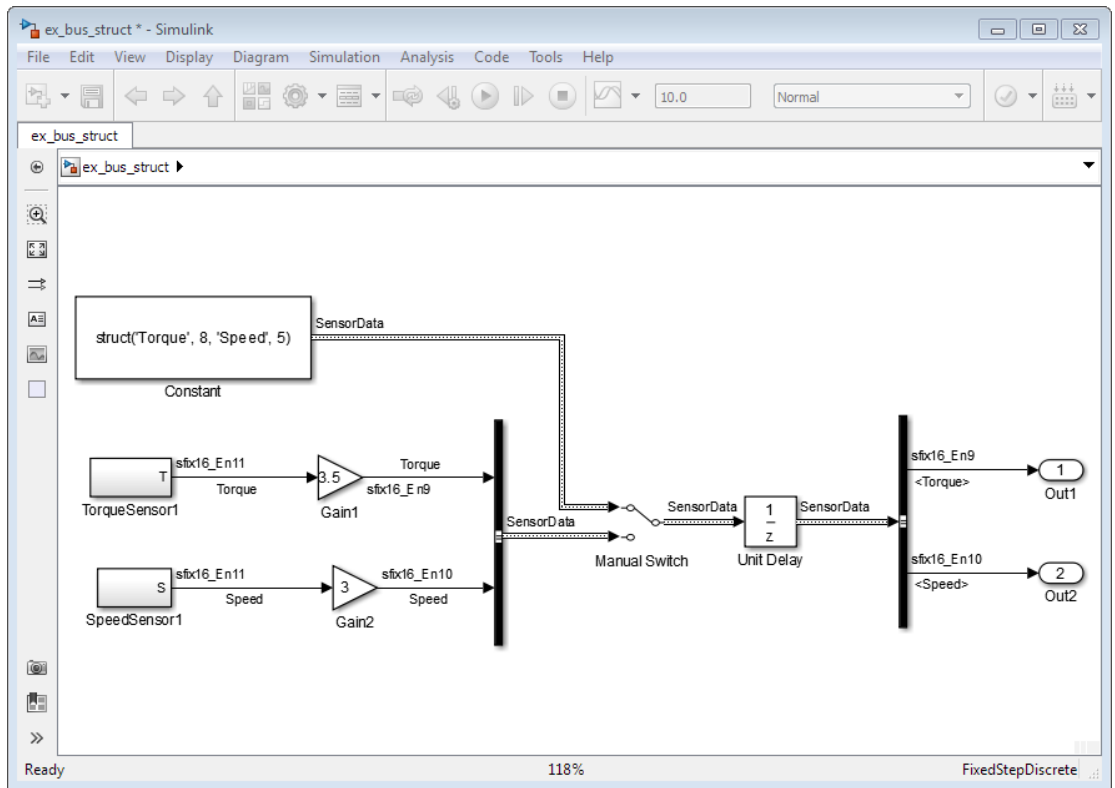
- 1 Open the `ex_bus_struct` model.

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_bus_struct
```

This model uses a structure as the initial condition in the Constant block. The structure fields of the Constant block and the elements of the SensorData bus object are specified as doubles.



- 2 Open the Fixed-Point Tool by selecting **Analysis > Data Type Design > Fixed-Point Tool**.
- 3 In the **Workflow** tab, under **System under design**, click **Continue** to select `ex_bus_struct` as the system that you want to convert.
- 4 Under **Configure model settings**, click **Range collection with specified data types** to enable instrumentation of the model.
- 5 Collect simulation minimum and maximum values. From the **Range collection** pane, simulate the model.
- 6 Under **Automatic data typing**, click the **Propose data types** button to propose data types with 16-bit word length and best-precision fraction length.
- 7 Examine the data type proposals in the **Contents** pane and determine whether to accept the proposals.
- 8 Click the **Apply accepted data types** button to write the proposed data types to the model.
- 9 Under **Range collection**, enter `Fixed_Run` for **Run name** and simulate the model again using the new fixed-point data types.
- 10 In the model window, select **Edit > Bus editor**.



Bus Editor - Manage Bus Objects in the Base Workspace

Name	Data Type	Complexity	Dimensions	Dimensions Mode	Sampling Mode	Sample Time	Min	Max	Doc Units	Description
Torque	fixdt(1,16,9)	real	1	Fixed	Sample based	-1				
Speed	fixdt(1,16,10)	real	1	Fixed	Sample based	-1				

Simulink.Bus: SensorData

Properties

Name: SensorData

Code generation options

Data scope: Auto

Header file:

Alignment: -1

The SensorData bus object now uses fixed-point data types. The initial condition of the Constant block still uses a non tunable structure using double data types. During compilation, an automatic casting of the data type of the structure field occurs so that it matches the data type of the bus signal.

If you generate code for a model that uses a structure as the initial condition for a bus signal, the values appear inline in the generated code.

Related Examples

- Parameter Tunability Loss on page 46-31
- “Refine Data Types of a Model with Buses Using Simulation Data” on page 42-2

More About

- “Bus Objects in the Fixed-Point Workflow” on page 42-19

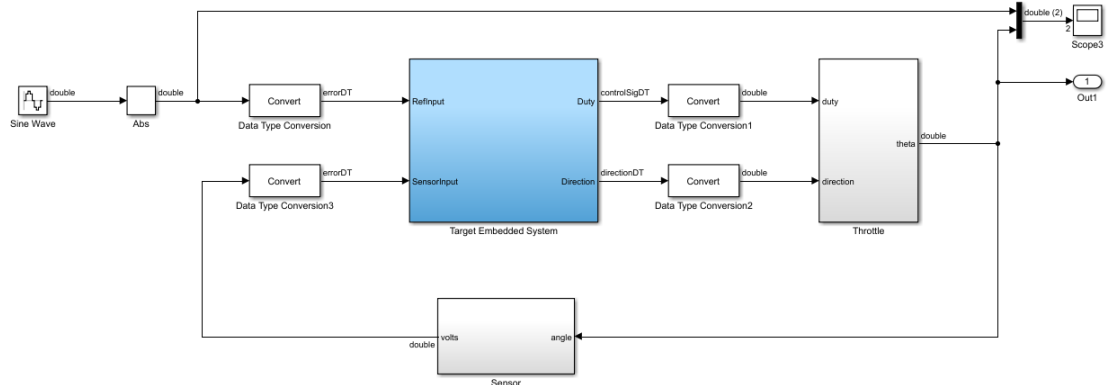
Autoscale Simulink.AliasType Objects

You can propose and apply fixed-point data types for `Simulink.AliasType` objects in your model using the Fixed-Point Tool. The Fixed-Point Tool detects alias type objects in the model and proposes a fixed-point data type based on collected ranges. The tool applies the proposed data type to the alias type object by updating its definition in the model workspace.

This example shows how the Fixed-Point Tool proposes and applies data types for `Simulink.AliasType` objects used in a model.

- 1 Open the `ex_throttle_controller_aliasype` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_throttle_controller_aliasype
```



- 2 Simulate the model. Six `Simulink.AliasType` objects are used in the model and defined in the base workspace. You can view the definition of the alias type objects at the MATLAB command line.

```
errorDT
```

```
errorDT =
```

```
AliasType with properties:
```

```
Description: ''
DataScope: 'Auto'
HeaderFile: ''
BaseType: 'double'
```

- 3 Use the Fixed-Point Tool to propose and apply data types to the model, including the alias type objects. Open the Fixed-Point Tool from the model menu by selecting **Analysis > Data Type Design > Fixed-Point Tool**.
- 4 Under **System under design**, click **Change**. Select the Target Embedded System subsystem as the system for conversion. Click **OK**.
- 5 Under **Configure model settings**, click the **Range collection using double override** button. This button enables instrumentation on the model to collect range information.

Click **Simulate**. The Fixed-Point Tool stores range information in the run titled **DoubleOverride**.

- 6 Propose fixed-point data types for the model. Under **Automatic data typing**, click **Propose data types** to gather data type proposals using the default proposal settings.

The **Contents** pane now displays the alias type objects used in the model. The Fixed-Point Tool proposes data types based on available range data. For alias type objects used in multiple locations in a model, the Fixed-Point Tool bases the data type proposal on a union of the collected ranges.

- 7 Under **Automatic data typing**, click **Apply accepted data types**. The Fixed-Point Tool updates the data type of the alias type objects at their definition.
- 8 View the details of the objects. At the MATLAB command-line, type:

```
errorDT
```

```
errorDT =
```

```
AliasType with properties:
```

```
Description: ''
DataScope: 'Auto'
HeaderFile: ''
BaseType: 'fixdt(1,16,11)'
```

The data type of the **errorDT** object now reflects the data type proposed by the Fixed-Point Tool.

See Also

Simulink.AliasType

More About

- “What Are User-Defined Data Types?” (Embedded Coder)

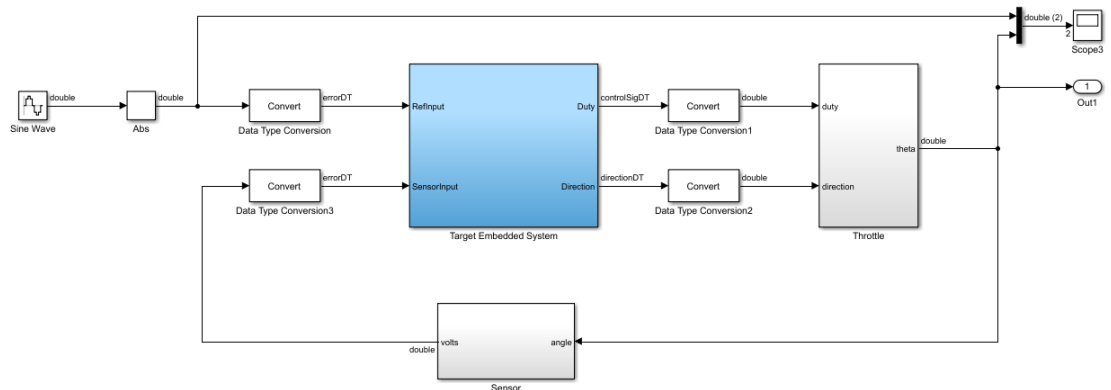
Autoscale Simulink.NumericType Objects

You can propose and apply data types for `Simulink.NumericType` objects in your model using the Fixed-Point Tool. The Fixed-Point Tool detects numeric type objects in the model and proposes a fixed-point data type based on collected ranges. The tool applies the proposed data type to the numeric type object by updating its definition in either the base or model workspace.

This example shows how the Fixed-Point Tool proposes and applies data types for `Simulink.NumericType` objects used in a model.

- 1 Open the `ex_throttle_controller` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_throttle_controller
```



- 2 Simulate the model. Six `Simulink.NumericType` objects are used in the model and defined in the base workspace. You can view the definition of the numeric type objects at the MATLAB command line.

```
errorDT
```

```
errorDT =
```

```
NumericType with properties:
```

```
  DataTypeMode: 'Double'
```

```

        IsAlias: 1
        DataScope: 'Auto'
        HeaderFile: ''
        Description: ''

```

- 3 Use the Fixed-Point Tool to propose and apply data types to the model, including the numeric type objects. Open the Fixed-Point Tool from the model menu by selecting **Analysis > Data Type Design > Fixed-Point Tool**.
- 4 Under **System under design**, click **Change**. Select the Target Embedded System subsystem as the system for conversion. Click **OK**.
- 5 Under **Configure model settings**, click the **Range collection using double override** button. This button enables instrumentation on the model to collect range information.

Click **Simulate**. The Fixed-Point Tool stores range information in the run titled DoubleOverride.

- 6 Propose fixed-point data types for the model. Under **Automatic data typing**, click **Propose data types** to gather data type proposals using the default proposal settings.

The Contents now displays the numeric type objects used in the model. The Fixed-Point Tool proposes data types based on available range data. For numeric type objects used in multiple locations in a model, the Fixed-Point Tool bases the data type proposal on a union of the collected ranges.

- 7 Under **Automatic data typing**, click **Apply accepted data types**. The Fixed-Point Tool updates the data type of the numeric type objects at their definition.
- 8 View the details of the numeric type objects. At the MATLAB command-line, type:

```
errorDT
```

```
errorDT =
```

```
NumericType with properties:
```

```

        DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Signed'
        WordLength: 16
        FractionLength: 11
        IsAlias: 1
        DataScope: 'Auto'
        HeaderFile: ''
        Description: ''

```

The data type of the `errorDT` numeric type object now reflects the data type proposed by the Fixed-Point Tool.

See Also

`Simulink.NumericType`

More About

- “What Are User-Defined Data Types?” (Embedded Coder)

Autoscale Simulink.Parameter Objects

Using the Fixed-Point Tool, you can propose and apply data types for `Simulink.Parameter` objects used in your model. The Fixed-Point Tool detects parameter objects in the model and proposes a data type based on their respective values and ranges. The tool applies the proposed data type to the parameter object by updating its definition in either the base or model workspace.

Whenever possible, it is a best practice to specify design range information on the parameter object. When the data type of the parameter object is set to auto, the Fixed-Point Tool follows the same rules as proposing for inherited data types. The Fixed-Point Tool proposes a data type for a parameter object according to the following rules:

Case		How data type is proposed	
Design ranges are specified on the parameter object		Use design ranges to propose data type	
Design ranges are not specified on parameter object	Non-tunable parameter object	Use parameter value to propose data type	
	Tunable parameter object	Design minimum and maximum specified on client blocks	Use client design range to propose data type
		No design ranges specified on client blocks	Use parameter value to propose data type but issue warning about unavailability of range
	Conflicting design ranges specified on client blocks	Use parameter value to propose data type but issue warning about conflicting range	

When the design ranges specified on a client block do not match the design ranges of the parameter object, there will be extra shifts and typecasts in generated code. The Fixed-Point Tool is not able to detect when a parameter object must be integer only, such as when using a parameter object as a variable for dimensions, variant control, or a boolean

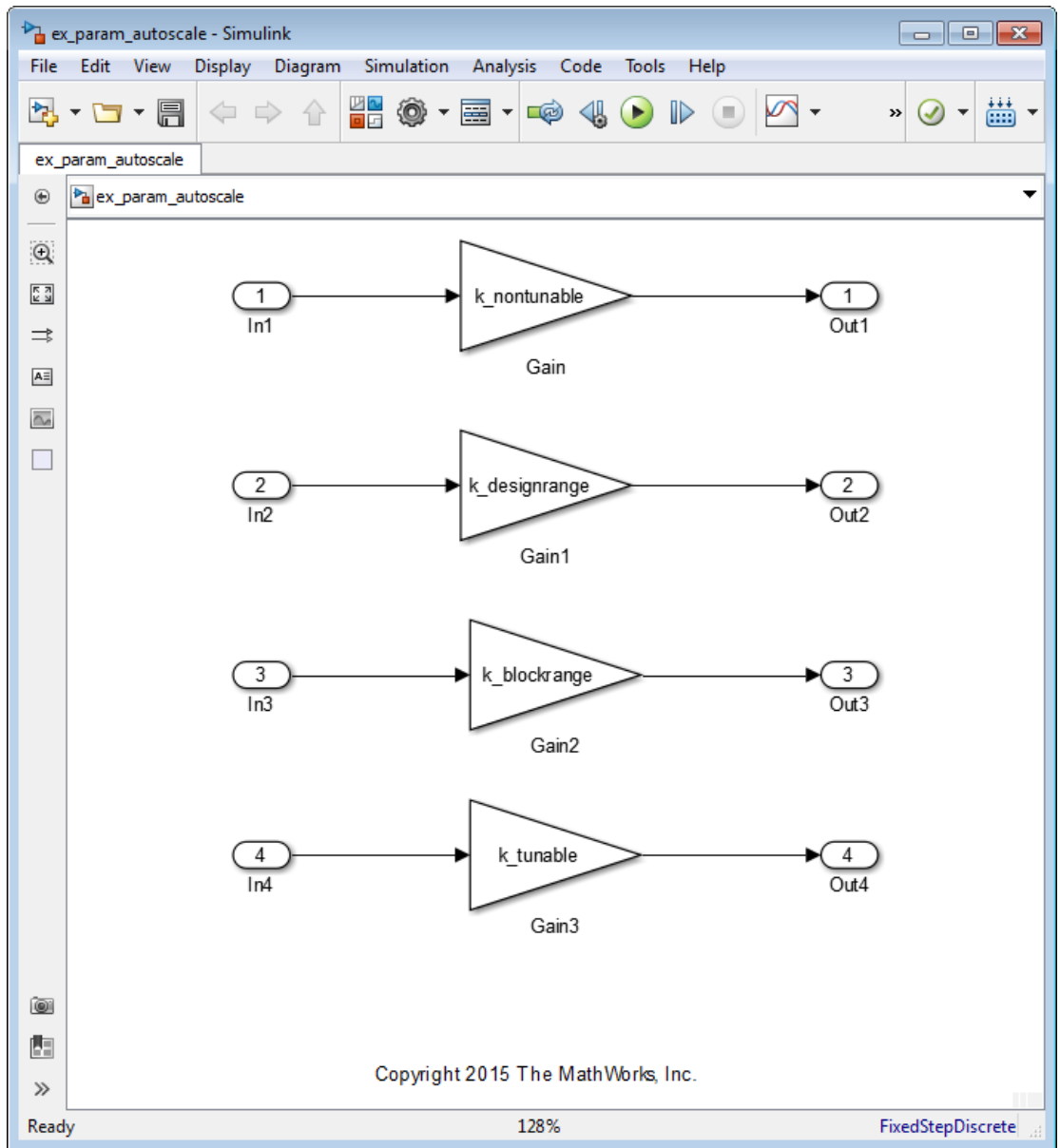
value. In these cases, you must deselect the Accept box before applying data types to your model.

Autoscale a Model Using Parameter Objects

This example shows how the Fixed-Point Tool proposes and applies data types for parameter objects used in a model.

- 1 Open the `ex_param_autoscale` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_param_autoscale
```

- 2 Simulate the model. Three `Simulink.Parameter` objects are defined in the base workspace. A fourth parameter object, `k_nontunable`, is defined in the model workspace.
- 3 Use the Fixed-Point Tool to autoscale the model, including the parameter objects. Open the Fixed-Point Tool from the model menu by selecting **Analysis > Data Type Design > Fixed-Point Tool**.
- 4 Under **System under design**, click **Continue** to select the `ex_param_autoscale` model as the system for conversion.
- 5 Under **Configure model settings**, click the **Range collection using double override** button. This button enables instrumentation on the model to collect range information.

Click **Simulate**. The Fixed-Point Tool stores range information in the run titled `DoubleOverride`.

- 6 Propose fixed-point data types for the model. Under **Automatic data typing**, click **Propose data types** to gather data type proposals using the default proposal settings.

The Contents pane of the Fixed-Point Tool now displays the four parameter objects used in the model. In parentheses, next to the name of each parameter object, is the location where the parameter object is defined.

The Fixed-Point Tool proposes data types based on available range data. The table below provides an explanation for the data type proposal of each parameter object in the model.

Parameter object	Description	Range used in proposal	Proposed data type
<code>k_designrange</code>	Tunable parameter object. Use design ranges specified on parameter object for proposal.	<code>[-100,100]</code>	<code>fixdt(1,16,8)</code>
<code>k_nontunable</code>	Non-tunable parameter object. Use value as min and max for proposal.	<code>[5,5]</code>	<code>fixdt(0,16,13)</code>

Parameter object	Description	Range used in proposal	Proposed data type
k_blockrange	Tunable parameter object with no design ranges. Use design ranges specified on client Gain block for proposal.	[- 1000, 1000]	fixdt(1,16,5)
k_tunable	Tunable parameter object with no design ranges specified on either parameter object or client block. Use value for proposal. Warning issued in this case.	[15, 15]	fixdt(0,16,12)

- 7 Return to the model. Double-click the **Gain3** block to open the block parameters. Change the **Gain** parameter from **k_tunable** to **k_blockrange**. On the **Parameter Attributes** tab, set the **Parameter maximum** to 1500.

Click **OK**.

- 8 Return to the Fixed-Point Tool. Under **Automatic data typing**, click **Propose data types** again. The ranges specified on the client blocks of the **k_blockrange** parameter object are inconsistent, so the Fixed-Point Tool proposes a data type based on the value of the parameter object.

Parameter object	Description	Range used in proposal	Proposed data type
k_blockrange	Design ranges specified on client Gain blocks are inconsistent. Use value of parameter object for proposal. Warning issued in this case.	[15, 15]	fixdt(0,16,12)

- 9 Under **Automatic data typing**, click **Apply accepted data types**. The Fixed-Point Tool updates the data type of the parameter objects at their definition.
- 10 View the details of the parameter objects. At the MATLAB command-line, type:

```
k_designrange
```

```
k_designrange =
```

```
Parameter with properties:
```

```
Value: 10  
CoderInfo: [1x1 Simulink.CoderInfo]  
Description: ''  
DataType: 'fixdt(1,16,8)'  
Min: -100  
Max: 100  
Unit: ''  
Complexity: 'real'  
Dimensions: [1 1]
```

Notice that the data type of the `k_designrange` parameter object now reflects the data type proposed by the Fixed-Point Tool.

See Also

Simulink.Parameter

More About

- “Control Block Parameter Data Types” (Simulink)

Command Line Interface for the Fixed-Point Tool

- “The Command-Line Interface for the Fixed-Point Tool” on page 43-2
- “Convert a Model to Fixed Point Using the Command Line” on page 43-4

The Command-Line Interface for the Fixed-Point Tool

The methods of the `DataTypeWorkflow.Converter` class allow you to collect simulation and derived data, propose and apply data types to the model, and analyze results. The class performs the same fixed-point conversion tasks as the Fixed-Point Tool. The following table summarizes the steps in the workflow and lists the appropriate classes and methods to use at each step.

Step in Workflow	Primary methods and classes for step in workflow
Set up model	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.Converter</code>
Prepare the model for fixed-point conversion	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.Converter.applySettingsFromShortcut</code> • <code>DataTypeWorkflow.Converter.applySettingsFromRun</code>
Gather range information	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.Converter.deriveMinMax</code> • <code>DataTypeWorkflow.Converter.simulateSystem</code>
Propose data types	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.ProposalSettings</code> • <code>DataTypeWorkflow.Converter.proposeDataTypes</code>
Apply proposed data types	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.Converter.applyDataTypes</code>
Verify new fixed-point settings and analyze results	<ul style="list-style-type: none"> • <code>DataTypeWorkflow.Converter.compareResults</code> • <code>DataTypeWorkflow.Converter.compareRuns</code> • <code>DataTypeWorkflow.Converter.results</code> • <code>DataTypeWorkflow.Converter.saturationOverflows</code> • <code>DataTypeWorkflow.Converter.wrapOverflows</code> • <code>DataTypeWorkflow.Converter.proposalIssues</code>

Note: You should not use the Fixed-Point Tool and the command-line interface in the same conversion session.

To decide which workflow is right for you, consult the following table:

Capability	Fixed-Point Tool	Command-Line Interface
Populate runs to dataset	✓	✓
Delete result from dataset	✓	✓

Capability	Fixed-Point Tool	Command-Line Interface
Edit proposed data types	✓	
Selectively apply data type proposals	✓	
Run multiple simulations	✓	✓
Script workflow		✓

Related Examples

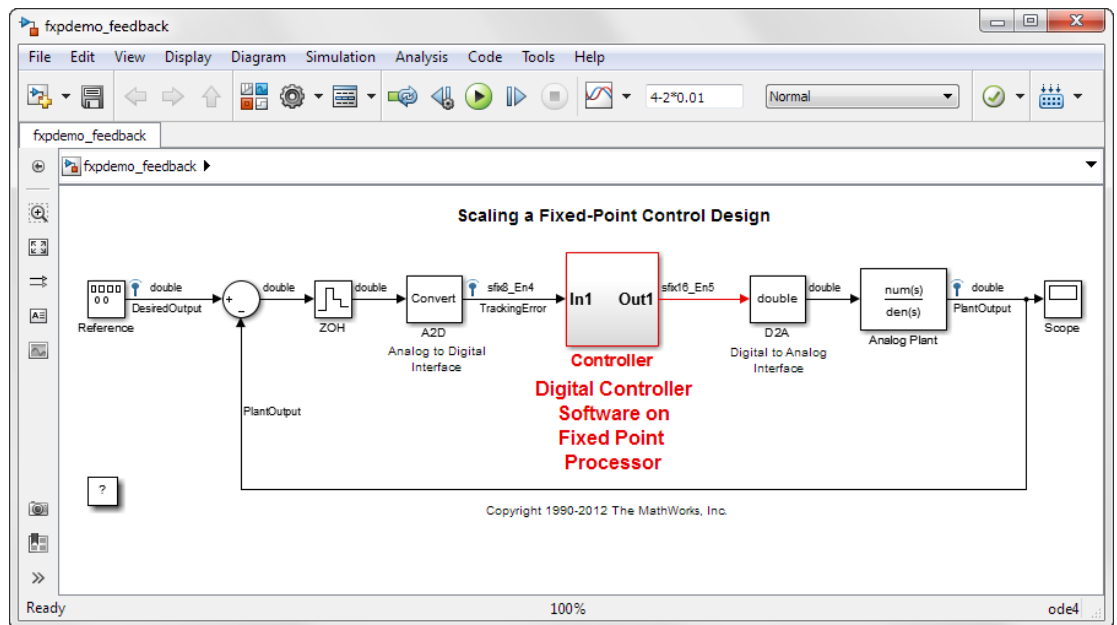
- “Convert a Model to Fixed Point Using the Command Line” on page 43-4

Convert a Model to Fixed Point Using the Command Line

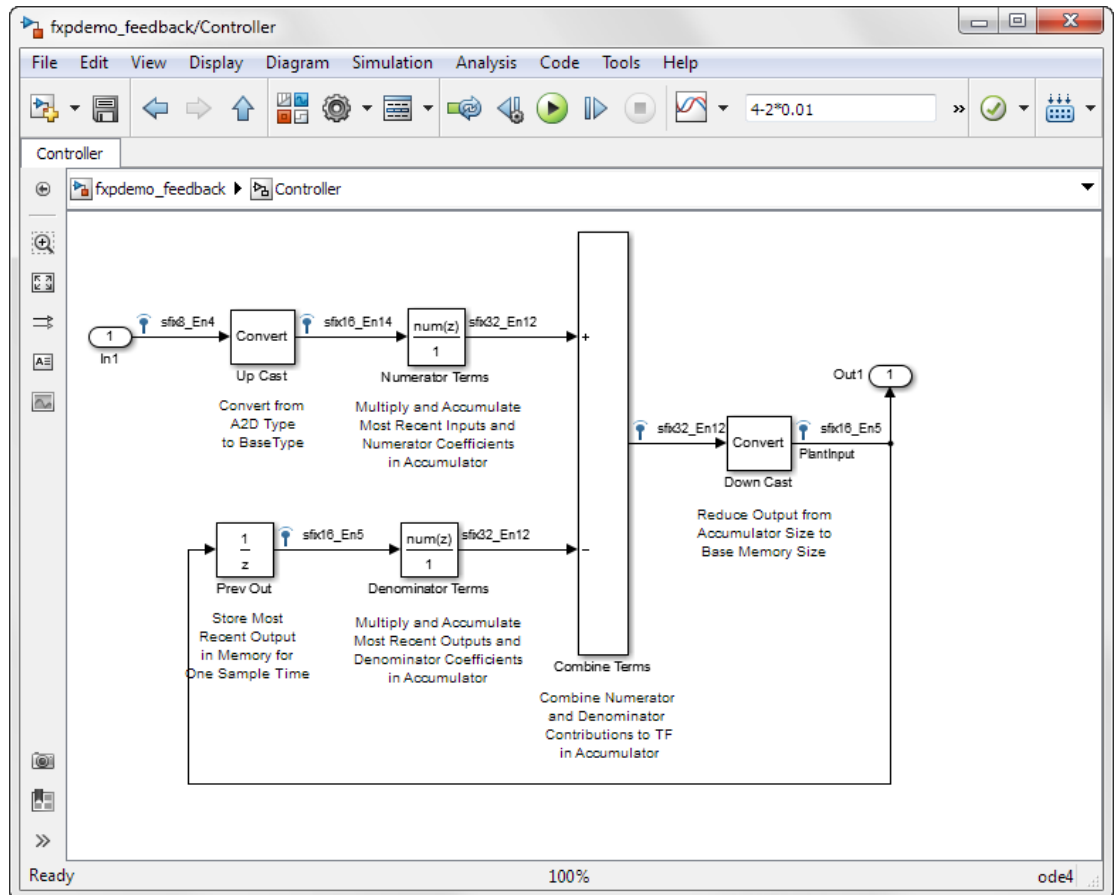
This example shows how to refine the data types of a model using the MATLAB command line.

- 1 Open the `fxpdemo_feedback` model.

```
fxpdemo_feedback
```



The Controller subsystem uses fixed-point data types.



- 2 Create a `DataTypeWorkflow.Converter` object to refine the data types of the `Controller` subsystem of the `fxpdemo_feedback` model.

```
converter = DataTypeWorkflow.Converter('fxpdemo_feedback/Controller');
```

- 3 Simulate the model and store the results in `InitialRun`.

```
converter.CurrentRunName = 'InitialRun';
converter.simulateSystem();
```

- 4 Determine if there were any overflows in `InitialRun`.

```
 saturations = converter.saturationOverflows('InitialRun')
```

```
saturations =  
  
Result with properties:  
  
    ResultName: 'fxpdemo_feedback/Controller/Up Cast'  
SpecifiedDataType: 'fixdt(1,16,14)'  
CompiledDataType: 'fixdt(1,16,14)'  
ProposedDataType: ''  
    Wraps: []  
    Saturations: 23  
    SimMin: -2  
    SimMax: 1.9999  
    DerivedMin: []  
    DerivedMax: []  
    RunName: 'InitialRun'  
    Comments: ''
```

```
wraps = converter.wrapOverflows('InitialRun')
```

```
wraps =
```

```
    []
```

A saturation occurred in the Up Cast block of the **Controller** subsystem during the run. There were no wrapping overflows. Refine the data types of the model so that no saturations occur.

- 5 Configure the model for conversion using a shortcut. Find the shortcuts that are available for the system by accessing the `ShortcutsForSelectedSystem` property of the `converter` object.

```
converter.ShortcutsForSelectedSystem
```

```
ans =
```

```
    'Range collection using double override'  
    'Range collection with specified data types'  
    'Range collection using single override'  
    'Disable range collection'
```

- 6 Gather a floating-point benchmark for the model. Apply the settings from a shortcut to override the entire model with doubles and enable instrumentation.

```
converter.applySettingsFromShortcut('Range collection using double override')
```

This shortcut also updates the `CurrentRunName` property of the converter object.

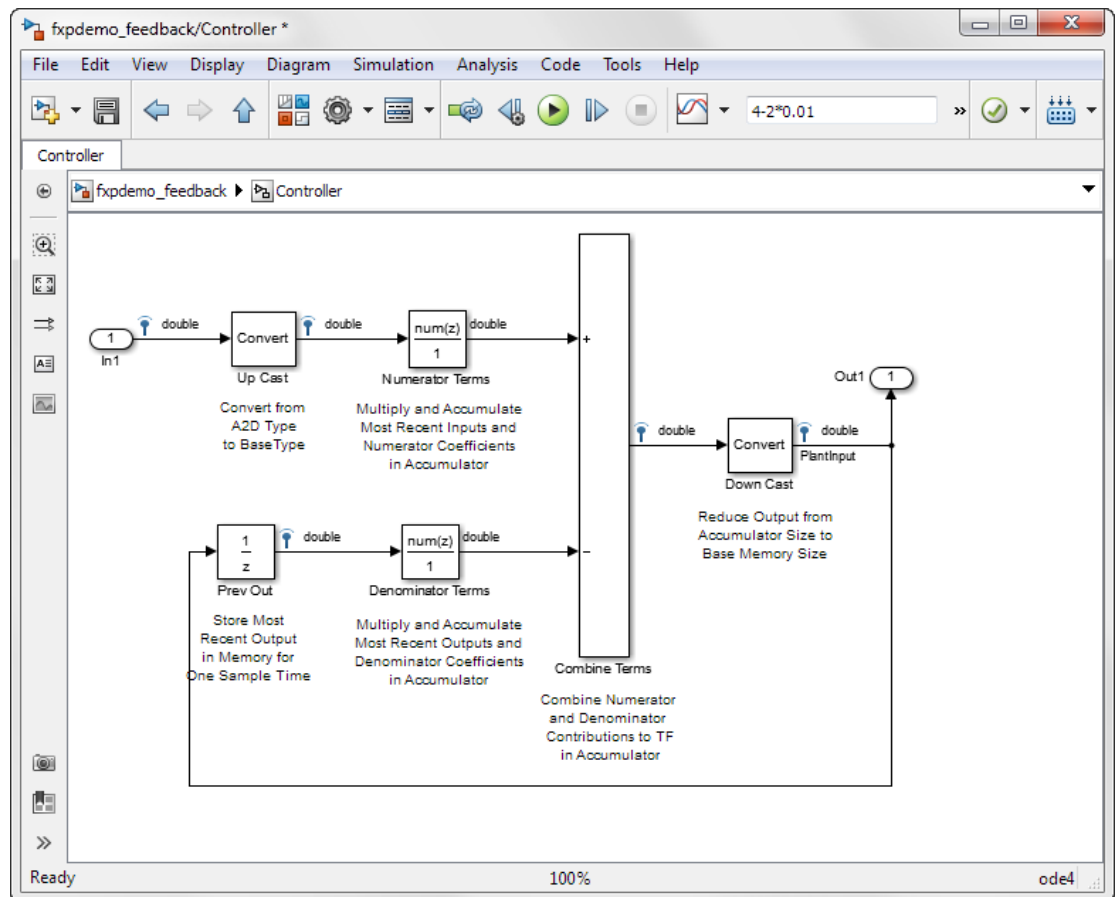
```
converter.CurrentRunName
```

```
ans =
```

```
DoubleOverride
```

- 7 Simulate the model again to gather new range information. Store these results in `DoubleOverride`.

```
converter.simulateSystem();
```



- 8 Create a `ProposalSettings` object to control the data type proposal settings.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

```
propSettings =
```

```
ProposalSettings with properties:
```

```
    ProposeSignedness: true
```

```
    ProposeWordLength: false
```

```
    ProposeFractionLength: true
```

```
    ProposeForInherited: true
```

```
    ProposeForFloatingPoint: true
```

```
        SafetyMargin: 0
```

```
        UseSimMinMax: true
```

```
        UseDerivedMinMax: true
```

```
Defaults for floating-point and inherited types:
```

```
    DefaultWordLength: 16
```

```
    DefaultFractionLength: 4
```

Use the default proposal settings to refine the data types of the `Controller` subsystem.

- 9 Propose data types for the system using the settings specified in the proposal settings object, `propSettings` and the range information gathered in the `'DoubleOverride'` run.

```
converter.proposeDataTypes('DoubleOverride', propSettings);
```

- 10 Apply the data types proposed for the `DoubleOverride` run to the model.

```
converter.applyDataTypes('DoubleOverride')
```

- 11 Turn off data type override by applying a configuration setting to the model.

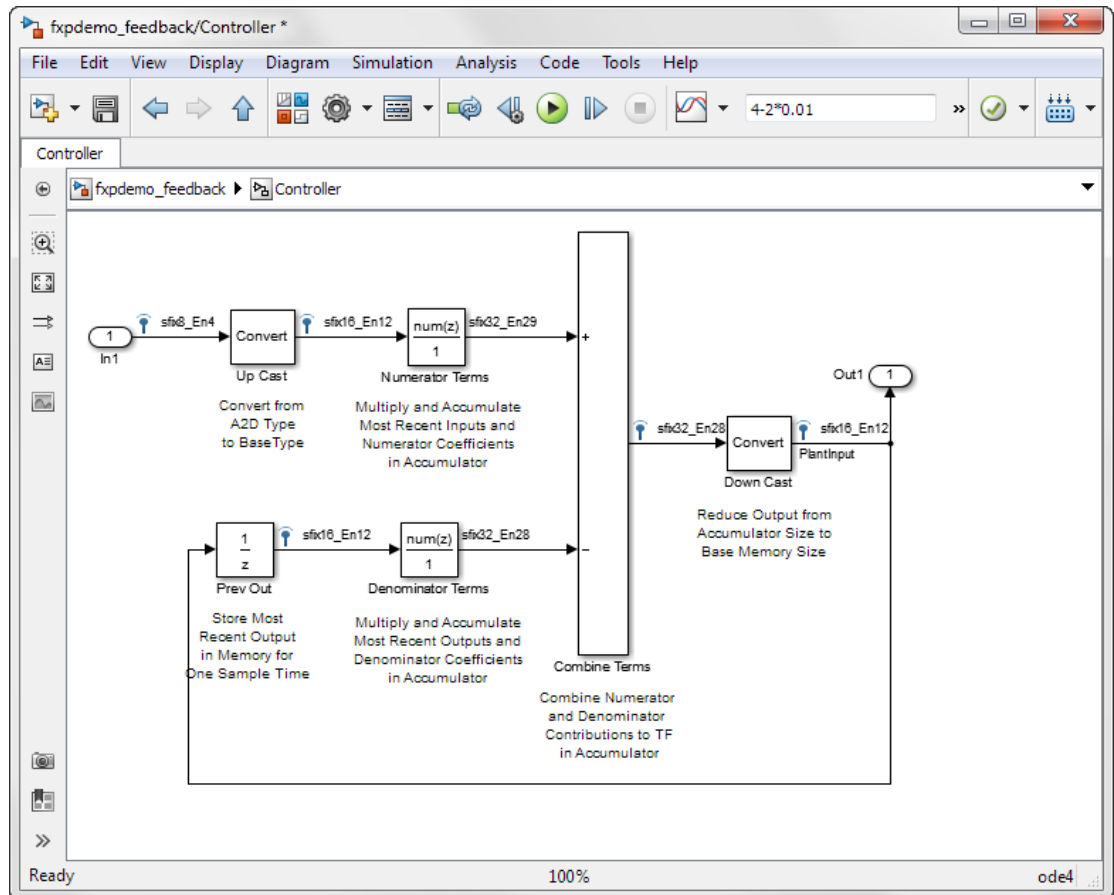
```
converter.applySettingsFromShortcut('Range collection with specified data types')
```

- 12 Set up a new run and simulate the model with the new fixed-point data types.

```
converter.CurrentRunName = 'FixedRun';
```

```
converter.simulateSystem();
```

The model now uses the refined fixed-point types.



- 13 Create two **Result** objects to use to compare the ideal floating-point behavior to the fixed-point behavior.

```
DoubleOverrideResult = converter.results('DoubleOverride',@(r) (strcmp(r.ResultName
```

```
DoubleOverrideResult =
```

```
Result with properties:
```

```
    ResultName: 'fxpdemo_feedback/Controller/Down Cast'
SpecifiedDataType: 'fixdt(1,16,12)'
CompiledDataType: 'double'
```

```
ProposedDataType: 'fixdt(1,16,12)'  
Wraps: []  
Saturation: []  
SimMin: -2.4135  
SimMax: 4.3270  
DerivedMin: []  
DerivedMax: []  
RunName: 'DoubleOverride'  
Comments: {}
```

```
FixedRunResult = converter.results('FixedRun',@(r) (strcmp(r.ResultName, 'fxpdemo_f
```

```
FixedRunResult =
```

```
Result with properties:
```

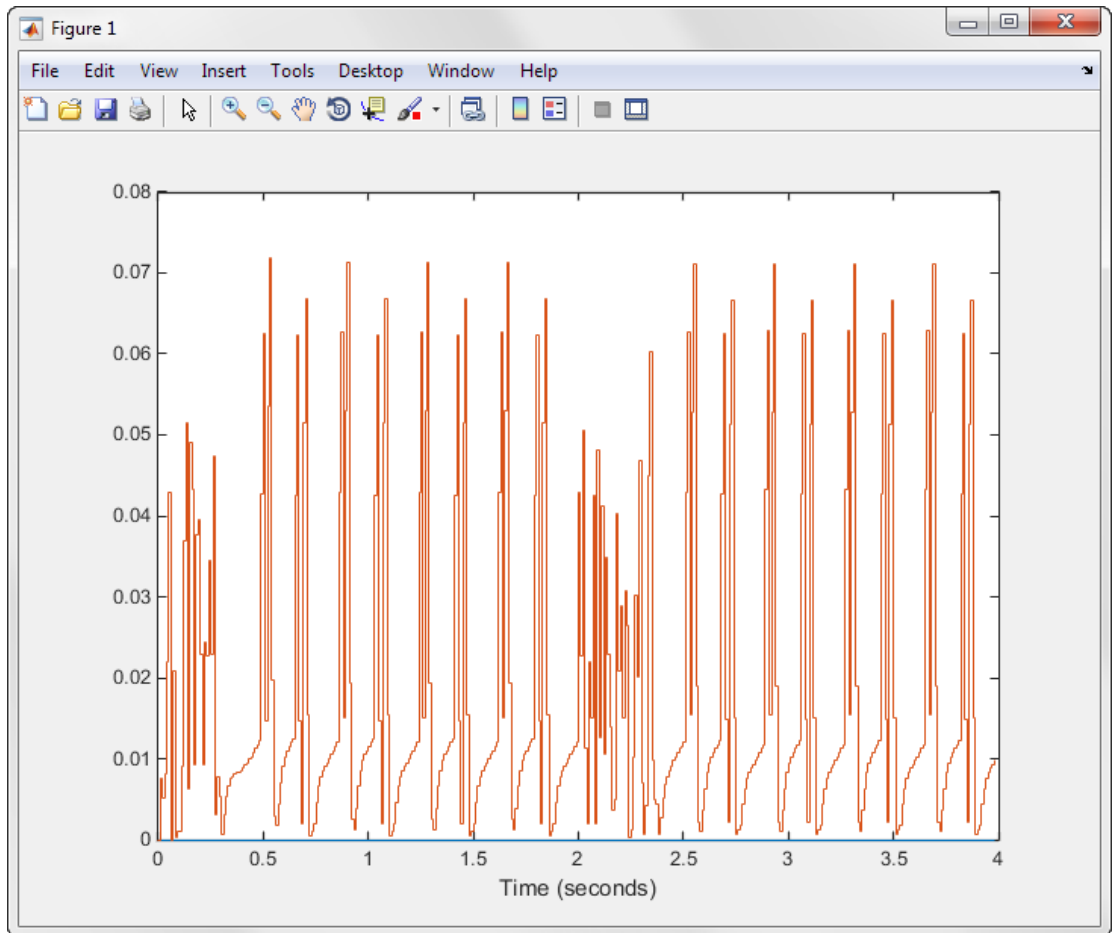
```
ResultName: 'fxpdemo_feedback/Controller/Down Cast'  
SpecifiedDataType: 'fixdt(1,16,12)'  
CompiledDataType: 'fixdt(1,16,12)'  
ProposedDataType: ''  
Wraps: []  
Saturation: []  
SimMin: -2.4211  
SimMax: 4.3496  
DerivedMin: []  
DerivedMax: []  
RunName: 'FixedRun'  
Comments: ''
```

- 14** Compare the `Result` object from the `DoubleOverride` run to the `Result` object from the `FixedRun`.

```
diff = converter.compareResults(DoubleOverrideResult,FixedRunResult);
```

- 15** Plot the difference between the two signals.

```
f = figure;  
plot(diff.tol);  
hold on  
plot(diff.diff);
```



More About

- “The Command-Line Interface for the Fixed-Point Tool” on page 43-2

Code Generation

- “Fixed-Point Code Generation Support” on page 44-2
- “Accelerating Fixed-Point Models” on page 44-4
- “Using External Mode or Rapid Simulation Target” on page 44-6
- “Net Slope Computation” on page 44-8
- “Control the Generation of Fixed-Point Utility Functions” on page 44-22
- “Optimize Generated Code with the Model Advisor” on page 44-28
- “Lookup table optimization” on page 44-36
- “Selecting Data Types for Basic Operations” on page 44-38
- “Use of Shifts by C Code Generation Products” on page 44-41

Fixed-Point Code Generation Support

In this section...

“Introduction” on page 44-2
“Languages” on page 44-2
“Data Types” on page 44-2
“Rounding Modes” on page 44-2
“Overflow Handling” on page 44-2
“Blocks” on page 44-3
“Scaling” on page 44-3

Introduction

All fixed-point blocks support code generation, except particular simulation features. The sections that follow describe the code generation support that the Fixed-Point Designer software provides. You must have a Simulink Coder license to generate C code or a HDL Coder license to generate HDL code.

Languages

C code generation is supported with the use of Simulink Coder. HDL code generation is supported with the use of HDL Coder.

Data Types

Fixed-point code generation supports all integer and fixed-point data types that are supported by simulation. Word sizes of up to 128 bits are supported in simulation. See “Supported Data Types” on page 31-18.

Rounding Modes

All rounding modes—Ceiling, Convergent, Floor, Nearest, Round, Simplest, and Zero — are supported.

Overflow Handling

- Saturation and wrapping are supported.

- Wrapping generates the most efficient code.
- Currently, you cannot choose to exclude saturation code automatically when hardware saturation is available. Select wrapping in order for the Simulink Coder product to exclude saturation code.

Blocks

All blocks generate code for all operations with a few exceptions. The Lookup Table Dynamic block generates code for all lookup methods except **Interpolation-Extrapolation**.

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink block library, including whether they support fixed-point data types and any limitations that apply for C code generation. To view the table, enter the following command at the MATLAB command line:

```
showblockdatatypetable
```

For information on block support for HDL code generation, see “Show Blocks Supported for HDL Code Generation” (HDL Coder). You can also use the HDL Workflow Advisor to check your model for blocks not supported for HDL code generation.

Scaling

Any binary-point-only scaling and [Slope Bias] scaling that is supported in simulation is supported, bit-true, in code generation.

More About

- “Optimize Generated Code with the Model Advisor” on page 44-28

Accelerating Fixed-Point Models

If the model meets the code generation restrictions, you can use Simulink acceleration modes with your fixed-point model. The acceleration modes can drastically increase the speed of some fixed-point models. This is especially true for models that execute a very large number of time steps. The time overhead to generate code for a fixed-point model is generally larger than the time overhead to set up a model for simulation. As the number of time steps increases, the relative importance of this overhead decreases.

Note: Rapid Accelerator mode does not support models with bus objects or 33+ bit fixed-point data types as parameters.

Every Simulink model is configured to have a start time and a stop time in the Configuration Parameters dialog box. Simulink simulations are usually configured for non-real-time execution, which means that the Simulink software tries to simulate the behavior from the specified start time to the stop time as quickly as possible. The time it takes to complete a simulation consists of two parts: overhead time and core simulation time, which is spent calculating changes from one time step to the next. For any model, the time it takes to simulate if the stop time is the same as the start time can be regarded as the overhead time. If the stop time is increased, the simulation takes longer. This additional time represents the core simulation time. Using an acceleration mode to simulate a model has an initially larger overhead time that is spent generating and compiling code. For any model, if the simulation stop time is sufficiently close to the start time, then Normal mode simulation is faster than an acceleration mode. But an acceleration mode can eliminate the overhead of code generation for subsequent simulations if structural changes to the model have not occurred.

In Normal mode, the Simulink software runs general code that can handle various situations. In an acceleration mode, code is generated that is tailored to the current usage. For fixed-point use, the tailored code is much leaner than the simulation code and executes much faster. The tailored code allows an acceleration mode to be much faster in the core simulation time. For any model, when the stop time is close to the start time, overhead dominates the overall simulation time. As the stop time is increased, there is a point at which the core simulation time dominates overall simulation time. Normal mode has less overhead compared to an acceleration mode when fresh code generation is necessary. Acceleration modes are faster in the core simulation portion. For any model, there is a stop time for which Normal mode and acceleration mode with fresh code generation have the same overall simulation time. If the stop time is decreased,

then Normal mode is faster. If the stop time is increased, then an acceleration mode has an increasing speed advantage. Eventually, the acceleration mode speed advantage is drastic.

Normal mode generally uses more tailored code for floating-point calculations compared to fixed-point calculations. Normal mode is therefore generally much faster for floating-point models than for similar fixed-point models. For acceleration modes, the situation often reverses and fixed point becomes significantly faster than floating point. As noted above, the fixed-point code goes from being general to highly tailored and efficient. Depending on the hardware, the integer-based fixed-point code can gain speed advantages over similar floating-point code. Many processors can do integer calculations much faster than similar floating-point operations. In addition, if the data bus is narrow, there can also be speed advantages to moving around 1-, 2-, or 4-byte integer signals compared to 4- or 8-byte floating-point signals.

Using External Mode or Rapid Simulation Target

In this section...

“Introduction” on page 44-6

“External Mode” on page 44-6

“Rapid Simulation Target” on page 44-7

Introduction

If you are using the Simulink Coder external mode or rapid simulation (RSim) target, there are situations where you might get unexpected errors when tuning block parameters. These errors can arise when you specify the **Best precision** scaling option for blocks that support constant scaling for best precision. See “Constant Scaling for Best Precision” on page 32-15 for a description of the constant scaling feature.

The sections that follow provide further details about the errors you might encounter. To avoid these errors, specify a scaling value instead of using the **Best precision** scaling option.

External Mode

If you change a parameter such that the binary point moves during an external mode simulation or during graphical editing, and you reconnect to the target, a checksum error occurs and you must rebuild the code. When you use **Best Precision** scaling, the binary point is automatically placed based on the value of a parameter. Each power of two roughly marks the boundary where a parameter value maps to a different binary point. For example, a parameter value of 1 to 2 maps to a particular binary point position. If you change the parameter to a value of 2 to 4, the binary point moves one place to the right, while if you change the parameter to a value of 0.5 to 1, it moves one place to the left.

For example, suppose a block has a parameter value of -2. You then build the code and connect in external mode. While connected, you change the parameter to -4. If the simulation is stopped and then restarted, this parameter change causes a binary point change. In external mode, the binary point is kept fixed. If you keep the parameter value of -4 and disconnect from the target, then when you reconnect, a checksum error occurs and you must rebuild the code.

Rapid Simulation Target

If a parameter change is great enough, and you are using the best precision mode for constant scaling, then you cannot use the RSim target.

If you change a block parameter by a sufficient amount (approximately a factor of two), the best precision mode changes the location of the binary point. Any change in the binary point location requires the code to be rebuilt because the model checksum is changed. This means that if best precision parameters are changed over a great enough range, you cannot use the rapid simulation target and a checksum error message occurs when you initialize the RSim executable.

Net Slope Computation

In this section...

“Handle Net Slope Computation” on page 44-8

“Use Division to Handle Net Slope Computation” on page 44-9

“Improve Numerical Accuracy of Simulation Results with Rational Approximations to Handle Net Slope” on page 44-10

“Improve Efficiency of Generated Code with Rational Approximations to Handle Net Slope” on page 44-15

“Use Integer Division to Handle Net Slope Computation” on page 44-21

Handle Net Slope Computation

The Fixed-Point Designer software provides an optimization parameter, **Use division for fixed-point net slope computation**, that controls how the software handles net slope computation. To learn how to enable this optimization, see “Use Integer Division to Handle Net Slope Computation” on page 44-21.

When a change of fixed-point slope is not a power of two, net slope computation is necessary. Normally, net slope computation is implemented using an integer multiplication followed by shifts. Under certain conditions, net slope computation can be implemented using integer division or a rational approximation of the net slope. One of the conditions is that the net slope can be accurately represented as a rational fraction or as the reciprocal of an integer. Under this condition, the division implementation gives more accurate numerical behavior. Depending on your compiler and embedded hardware, a division implementation might be more desirable than the multiplication and shifts implementation. The generated code for the rational approximation and/or integer division implementation might require less ROM or improve model execution time.

When to Use Division for Fixed-Point Net Slope Computation

This optimization works if:

- The net slope can be approximated with a fraction or is the reciprocal of an integer.
- Division is more efficient than multiplication followed by shifts on the target hardware.

Note: The Fixed-Point Designer software is not aware of the target hardware. Before selecting this option, verify that division is more efficient than multiplication followed by shifts on your target hardware.

When Not to Use Division to Handle Net Slope Computation

This optimization does not work if:

- The software cannot perform the division using the production target `long` data type and therefore must use multiword operations.

Using multiword division does not produce code suitable for embedded targets. Therefore, do not use division to handle net slope computation in models that use multiword operations. If your model contains blocks that use multiword operations, change the word length of these blocks to avoid these operations.

- Net slope is a power of 2 or a rational approximation of the net slope contains division by a power of 2.

Binary-point-only scaling, where the net slope is a power of 2, involves moving the binary point within the fixed-point word. This scaling mode already minimizes the number of processor arithmetic operations.

Use Division to Handle Net Slope Computation

To enable this optimization:

- 1 In the **Configuration Parameters** dialog box, set **Optimization > Use division for fixed-point net slope computation** to `On`, or `Use division for reciprocals of integers only`

For more information, see “Use division for fixed-point net slope computation” (Simulink).

- 2 On the **Hardware Implementation > Production hardware** pane, set the **Signed integer division rounds to** configuration parameter to `FLOOR` or `ZERO`, as appropriate for your target hardware. The optimization does not occur if the **Signed integer division rounds to** parameter is `Undefined`.

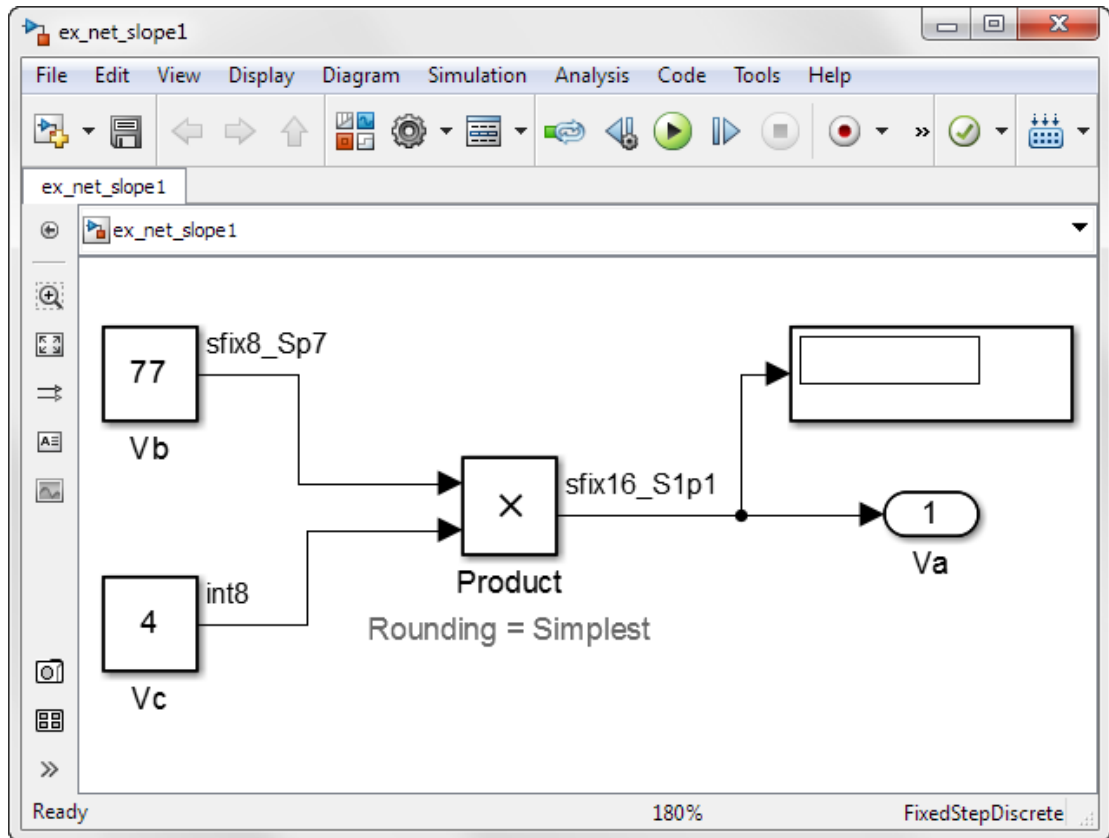
Note: You must set this parameter to a value that is appropriate for the target hardware. Failure to do so might result in division operations that comply with the definition on the **Hardware Implementation** pane, but are inappropriate for the target hardware.

- 3** Set the **Integer rounding mode** of the blocks that require net slope computation (for example, Product, Gain, and Data Type Conversion) to **Simplest** or match the rounding mode of your target hardware.

Note: You can use the Model Advisor to alert you if you have not configured your model correctly for this optimization. Open the Model Advisor and run the **Identify questionable fixed-point operations** check. For more information, see “Identify blocks that will invoke net slope computation” on page 44-30 .

Improve Numerical Accuracy of Simulation Results with Rational Approximations to Handle Net Slope

This example illustrates how setting the **Optimization > Use division for fixed-point net slope computation** parameter to **On** improves numerical accuracy. It uses the following model.



For the Product block in this model,

$$V_a = V_b \times V_c$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$$S_a Q_a = S_b Q_b \cdot S_c Q_c, \text{ or}$$

$$Q_a = \frac{S_b S_c}{S_a} Q_b Q_c$$

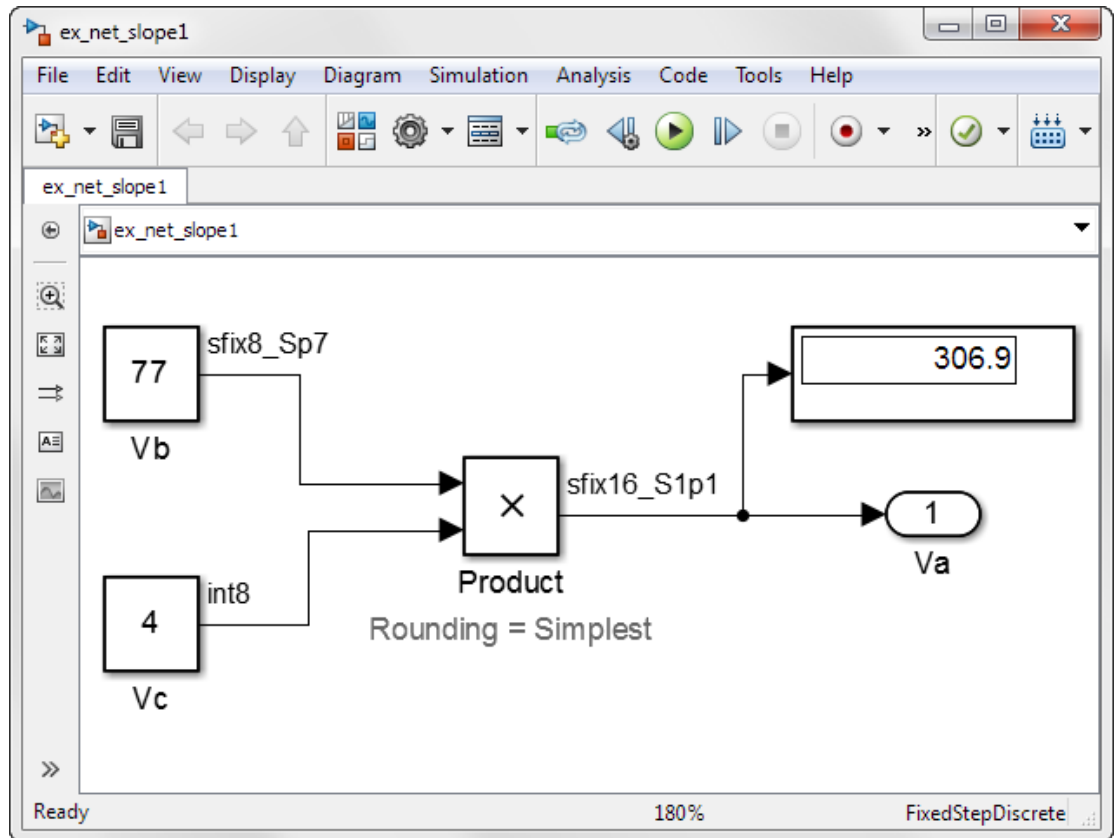
where the net slope is:

$$\frac{S_b S_c}{S_a}$$

The net slope for the Product block is 7/11. Because the net slope can be represented as a fractional value consisting of small integers, you can use the **On** setting of the **Use division for fixed-point net slope computation** optimization parameter if your model and hardware configuration are suitable. For more information, see “When to Use Division for Fixed-Point Net Slope Computation” on page 44-8.

To set up the model and run the simulation:

- 1 For the Constant block Vb, set the **Output data type** to `fixdt(1, 8, 0.7, 0)`. For the Constant block Vc, set the **Output data type** to `fixdt(1, 8, 0)`.
- 2 For the Product block, set the **Output data type** to `fixdt(1, 16, 1.1, 0)`. Set the **Integer rounding mode** to `Simplest`.
- 3 Set the **Hardware Implementation > Production hardware > Signed integer division rounds to** configuration parameter to `Zero`.
- 4 Set the **Optimization > Use division for fixed-point net slope computation** to `Off`.
- 5 In your Simulink model window, select **Simulation > Run**.

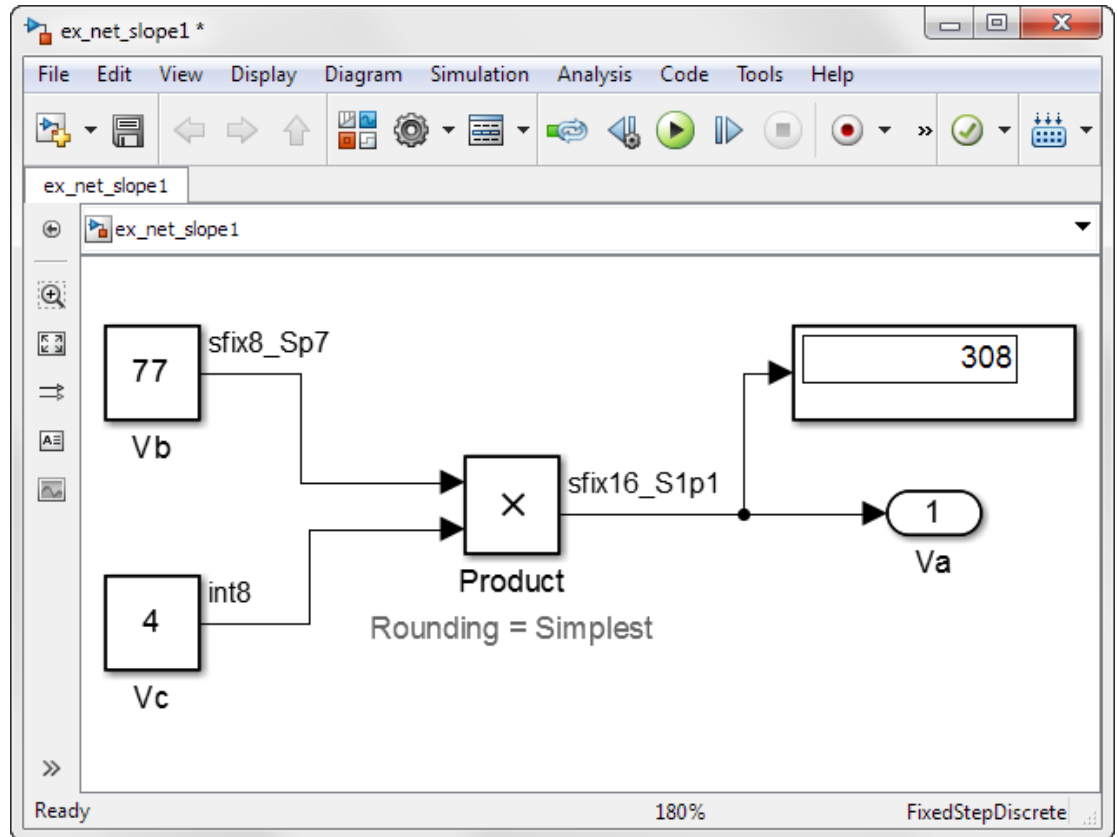


Because the simulation uses multiplication followed by shifts to handle the net slope computation, net slope precision loss occurs. This precision loss results in numerical inaccuracy: the calculated product is 306.9, not 308, as you expect.

Note: You can set up the Fixed-Point Designer software to provide alerts when precision loss occurs in fixed-point constants. For more information, see “Net Slope and Net Bias Precision” on page 33-26.

- 6 Set the **Optimization > Use division for fixed-point net slope computation** to On.

Save your model, and simulate again.



The software implements the net slope computation using a rational approximation instead of multiplication followed by shifts. The calculated product is 308, as you expect.

The optimization works for this model because:

- The net slope is representable as a fraction with small integers in the numerator and denominator.
- The **Hardware Implementation > Production hardware > Signed integer division rounds** to configuration parameter is set to Zero.

Note: This setting must match your target hardware rounding mode.

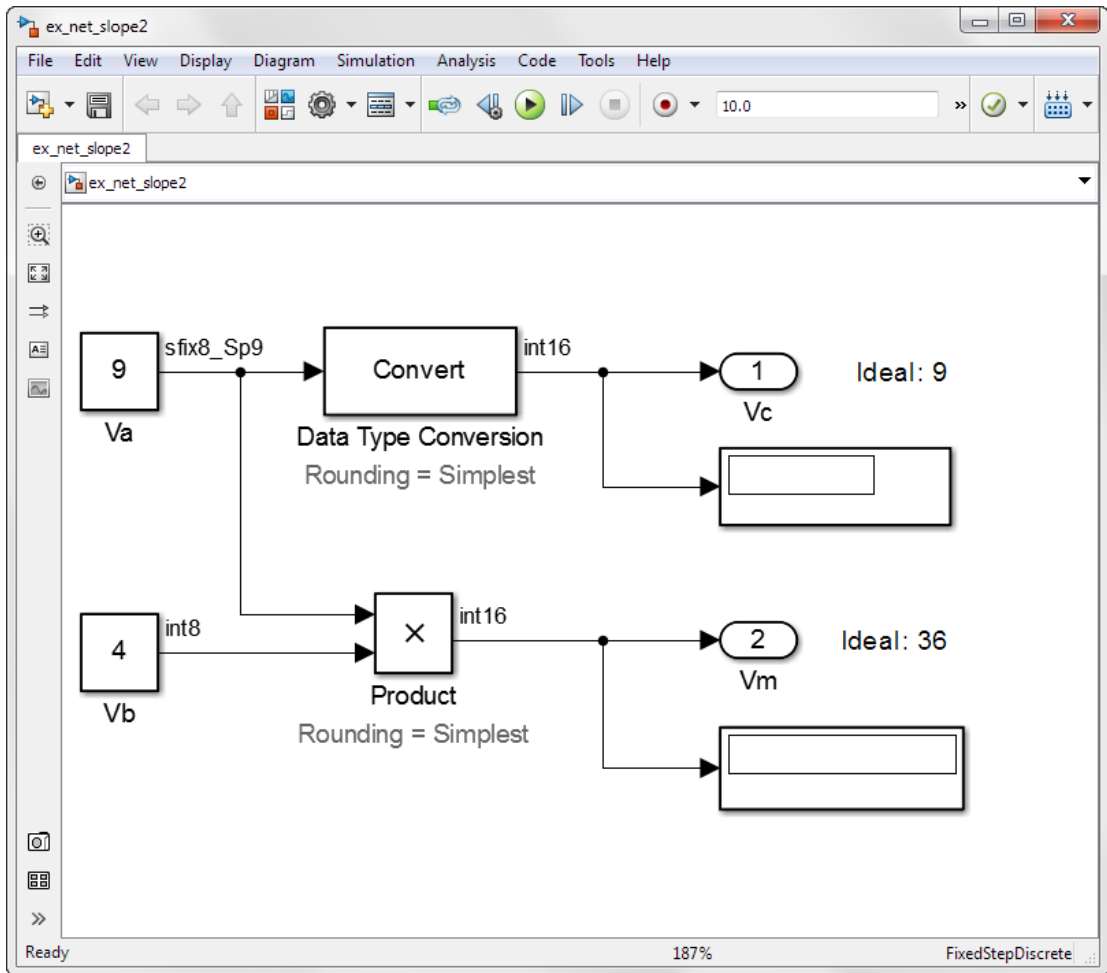
- The **Integer rounding mode** of the Product block in the model is set to `Simplest`.
- The model does not use multiword operations.

Improve Efficiency of Generated Code with Rational Approximations to Handle Net Slope

This example shows how setting the optimization parameter **Optimization > Use division for fixed-point net slope computation** to `On` improves the efficiency of generated code.

Note: The generated code is more efficient only if division is more efficient than multiplication followed by shifts on your target hardware.

This example uses the following model.



For the Product block in this model,

$$V_m = V_a \times V_b$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 32-6: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$$S_m Q_m = S_a Q_a + S_b Q_b$$

, or

$$Q_m = \frac{S_a S_b}{S_m} Q_a Q_b$$

where the net slope is:

$$\frac{S_a S_b}{S_m}$$

The net slope for the Product block is **9/10**.

Similarly, for the Data Type Conversion block in this model,

$$S_a Q_a + B_a = S_b Q_b + B_b$$

There is no bias. Therefore, the net slope is $\frac{S_b}{S_a}$. The net slope for this block is also **9/10**.

Because the net slope can be represented as a fraction, you can set the **Optimization > Use division for fixed-point net slope computation** optimization parameter to **On** if your model and hardware configuration are suitable. For more information, see “When to Use Division for Fixed-Point Net Slope Computation” on page 44-8.

To set up the model and generate code:

- 1** For the Inport block Va, set the **Data type** to `fixdt(1, 8, 9/10, 0)`; for the Inport block Vb, set the **Data type** to `int8`.
- 2** For the Data Type Conversion block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `int16`.

- 3 For the Product block, set the **Integer rounding mode** to Simplest. Set the **Output data type** to int16.
- 4 Set the **Hardware Implementation > Production hardware > Signed integer division rounds to configuration parameter** to Zero.
- 5 Set the **Optimization > Use division for fixed-point net slope computation** to Off.
- 6 From the Simulink model menu, select **Code > C/C++ Code > Build Model**.

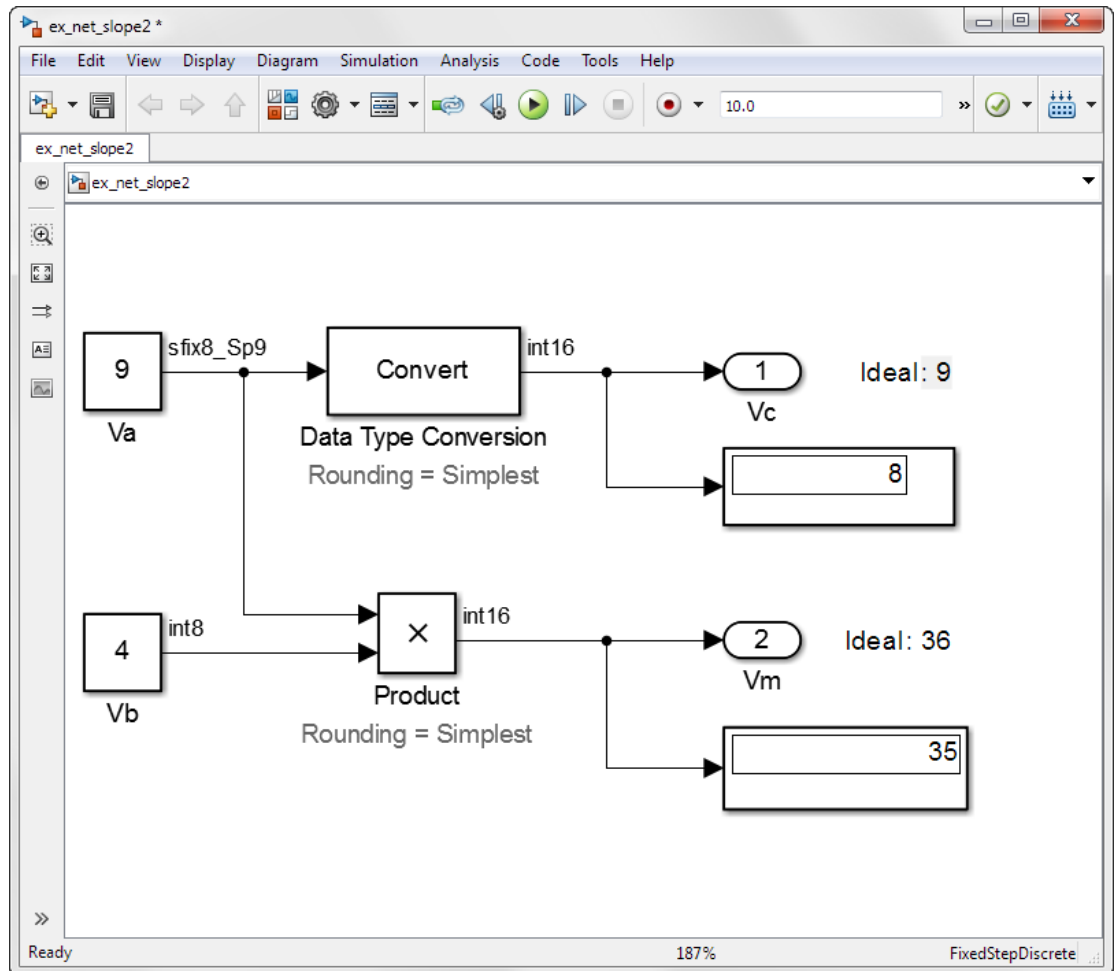
Conceptually, the net slope computation is 9/10 or 0.9:

```
Vc = 0.9 * Va;  
Vm = 0.9 * Va * Vb;
```

The generated code uses multiplication with shifts:

```
% For the conversion  
Vc = (int16_T)(Va * 115 >> 7);  
% For the multiplication  
Vm = (int16_T)((Va * Vb >> 1) * 29491 >> 14);
```

The ideal value of the net slope computation is 0.9. In the generated code, the approximate value of the net slope computation is $29491 \gg 15 = 29491/2^{15} = 0.899993896484375$. This approximation introduces numerical inaccuracy. For example, using the same model with constant inputs produces the following results.



- 7 In the original model with inputs V_a and V_b , set the **Optimization > Use division for fixed-point net slope computation** parameter to On, update the diagram, and generate code again.

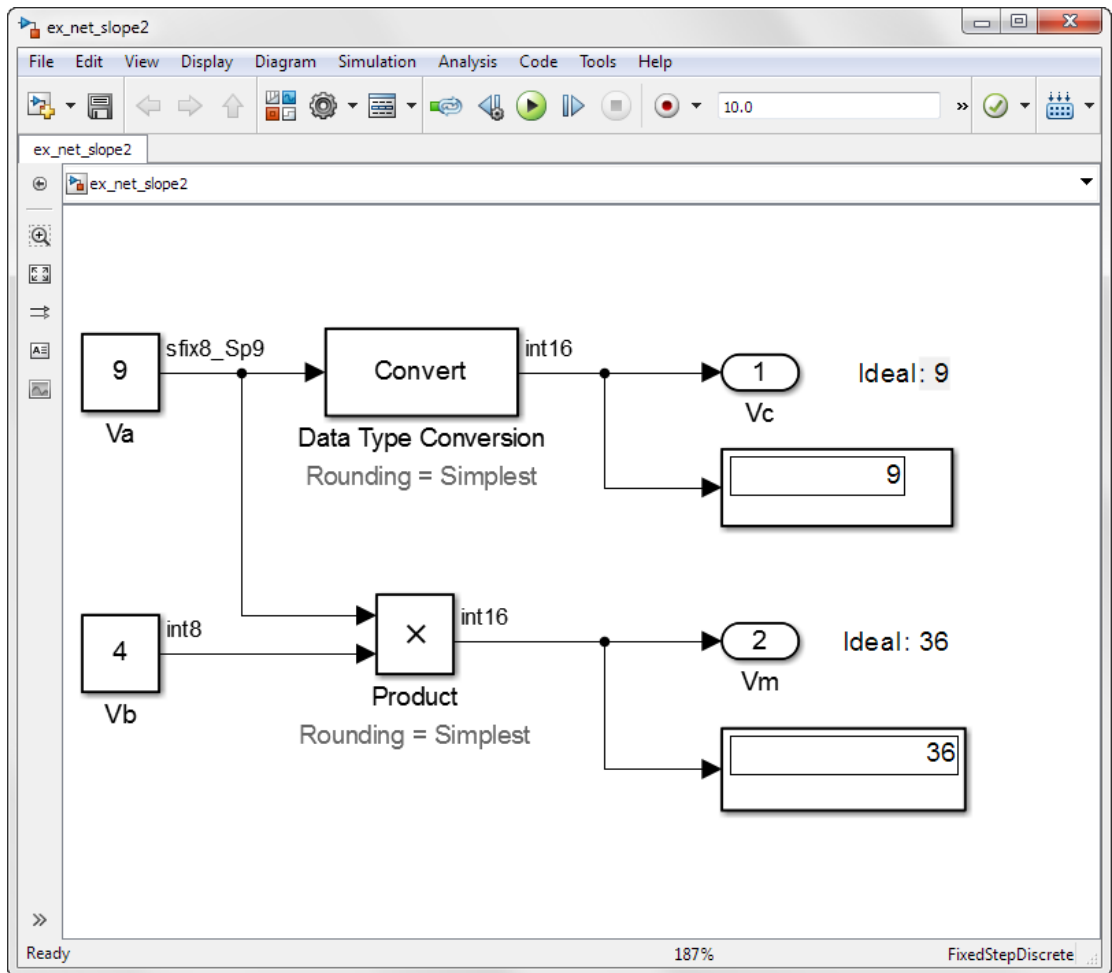
The generated code now uses integer division instead of multiplication followed by shifts:

```
% For the conversion
Vc = (int16_T)(Va * 9/10);
```

```
% For the multiplication
Vm = (int16_T)(Va * Vb * 9/10);
```

- 8 In the generated code, the value of the net slope computation is now the ideal value of 0.9. Using division, the results are numerically accurate.

In the model with constant inputs, set the **Optimization > Use division for fixed-point net slope computation** parameter to On and simulate the model.



The optimization works for this model because the:

- Net slope is representable as a fraction with small integers in the numerator and denominator.
- **Hardware Implementation > Production hardware > Signed integer division rounds to** configuration parameter is set to **Zero**.

Note: This setting must match your target hardware rounding mode.

- For the Product and Data Type Conversion blocks in the model, the **Integer rounding mode** is set to **Simplest**.
- Model does not use multiword operations.

Use Integer Division to Handle Net Slope Computation

Setting the **Optimization > Use division for fixed-point net slope computation** parameter to **Use division for reciprocals of integers only** triggers the optimization only in cases where the net slope is the reciprocal of an integer. This setting results in a single integer division to handle net slope computations.

Control the Generation of Fixed-Point Utility Functions

In this section...

“Optimize Generated Code Using Specified Minimum and Maximum Values” on page 44-22

“Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values” on page 44-24

Optimize Generated Code Using Specified Minimum and Maximum Values

The Fixed-Point Designer software uses representable minimum and maximum values and constant values to determine if it is possible to optimize the generated code, for example, by eliminating unnecessary utility functions and saturation code from the generated code.

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

When you select the **Optimize using specified minimum and maximum values** configuration parameter, the software takes into account input range information, also known as *design minimum and maximum*, that you specify for signals and parameters in your model. It uses these minimum and maximum values to derive range information for downstream signals in the model and then uses this derived range information to simplify mathematical operations in the generated code whenever possible.

Prerequisites

The **Optimize using specified minimum and maximum values** parameter appears for ERT-based targets only and requires an Embedded Coder license when generating code.

How to Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:

- Inport and Outport blocks
 - Block outputs
 - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
 - Simulink.Signal objects
- Before generating code, test the minimum and maximum values for signals and parameters. Otherwise, optimization might result in numerical mismatch with simulation. You can simulate your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

How to Enable Simulation Range Checking

- 1 In your model, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
 - 2 In the Configuration Parameters dialog box, select **Diagnostics > Data Validity**.
 - 3 On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to warning or error.
- Use fixed-point data types with binary-point-only (power-of-two) scaling.
 - Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream. For more information, see “Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values” on page 44-24.

How to Enable Optimization

- 1 In the Configuration Parameters dialog box, set the **Code Generation > System target file** to select an Embedded Real-Time (ERT) target (requires an Embedded Coder license).
- 2 Specify design minimum and maximum values for signals and parameters in your model using the tips in “How to Configure Your Model” on page 44-22.
- 3 Select the **Optimization > Optimize using specified minimum and maximum values** configuration parameter.

For more information, see “Optimize using the specified minimum and maximum values” (Simulink).

Limitations

- This optimization does not occur for:
 - Multiword operations
 - Fixed-point data types with slope and bias scaling
 - Addition unless the fraction length is zero
- This optimization does not take into account minimum and maximum values for:
 - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
 - Bus elements.
 - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Output block.

Output blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.

- There are limitations on precision because you specify the minimum and maximum values as double-precision values. If the true value of a minimum or maximum value cannot be represented as a double, ensure that you round the minimum and maximum values correctly so that they cover the true design range.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different specified minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, the Simulink Coder software generates code once for the subsystem and shares this code among the multiple instances of the subsystem.

Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values

This example shows how the Fixed-Point Designer software uses the input range for a division operation to determine whether it can eliminate unnecessary utility functions from the generated code. It uses the `fxpdemo_min_max_optimization` model. First, you generate code without using the specified minimum and maximum values to see that the generated code contains utility functions to ensure that division by zero

does not occur. You then turn on the optimization, and generate code again. With the optimization, the generated code does not contain the utility function because it is not necessary for the input range.

Generate Code Without Using Minimum and Maximum Values

First, generate code without taking into account the design minimum and maximum values for the first input of the division operation to show the code without the optimization. In this case, the software uses the representable ranges for the two inputs, which are both `uint16`. With these input ranges, it is not possible to implement the division with the specified precision using shifts, so the generated code includes a division utility function.

- 1 Run the example. At the MATLAB command line, enter:

```
fxpdemo_min_max_optimization
```

- 2 In the example window, double-click the **View Optimization Configuration** button.

The Optimization pane of the Configuration Parameters dialog box appears.

Note that the **Optimize using specified minimum and maximum values** parameter is not selected.

- 3 Double-click the **Generate Code** button.

The code generation report appears.

- 4 In the model, right-click the **Division with increased fraction length output type** block.

The context menu appears.

- 5 From the context menu, select **C/C++ Code > Navigate To C/C++ Code**.

The code generation report highlights the code generated for this block. The generated code includes a call to the `div_repeat_u32` utility function.

```
rtY.Out3 = div_repeat_u32((uint32_T)rtU.In5 << 16,  
    (uint32_T)rtU.In6, 1U);
```

- 6 Click the `div_repeat_u32` link to view the utility function, which contains code for handling division by zero.

Generate Code Using Minimum and Maximum Values

Next, generate code for the same division operation, this time taking into account the design minimum and maximum values for the first input of the Product block. These minimum and maximum values are specified on the Inport block directly upstream of the Product block. With these input ranges, the generated code implements the division by simply using a shift. It does not need to generate a division utility function, reducing both memory usage and execution time.

- 1 Double-click the Inport block labelled 5 to open the block parameters dialog box.
- 2 On the block parameters dialog box, select the **Signal Attributes** pane and note that:
 - The **Minimum** value for this signal is 1.
 - The **Maximum** value for this signal is 100.
- 3 Click **OK** to close the dialog box.
- 4 Double-click the **View Optimization Configuration** button.

The Optimization pane of the Configuration Parameters dialog box appears.

- 5 On this pane, select the **Optimize using specified minimum and maximum values** parameter and click **Apply**.
- 6 Double-click the **Generate Code** button.

The code generation report appears.

- 7 In the model, right-click the Division with increased fraction length output type block.

The context menu appears.

- 8 From the context menu, select **C/C++ Code > Navigate To C/C++ Code**.

The code generation report highlights the code generated for this block. This time the generated code implements the division with a shift operation and there is no division utility function.

```
tmp = rtU.In6;
rtY.Out3 = (uint32_T)tmp ==
  (uint32_T)0 ? MAX_uint32_T : ((uint32_T)rtU.In5 << 17) /
  (uint32_T)tmp;
```

Modify the Specified Minimum and Maximum Values

Finally, modify the minimum and maximum values for the first input to the division operation so that its input range is too large to guarantee that the value does not overflow when shifted. Here, you cannot shift a 16-bit number 17 bits to the right without overflowing the 32-bit container. Generate code for the division operation, again taking into account the minimum and maximum values. With these input ranges, the generated code includes a division utility function to ensure that no overflow occurs.

- 1 Double-click the Inport block labelled **5** to open the block parameters dialog box.
- 2 On the block parameters dialog box, select the **Signal Attributes** pane and set the **Maximum** value to **40000**, then click **OK** to close the dialog box.
- 3 Double-click the **Generate Code** button.

The code generation report appears.

- 4 In the model, right-click the **Division with increased fraction length output type** block.

The context menu appears.

- 5 From the context menu, select **C/C++ Code > Navigate To C/C++ Code**.

The code generation report highlights the code generated for this block. The generated code includes a call to the `div_repeat_32` utility function.

```
rtY.Out3 = div_repeat_u32((uint32_T)rtU.In5 << 16,  
    (uint32_T)rtU.In6, 1U);
```

Optimize Generated Code with the Model Advisor

In this section...

“Identify Blocks that Generate Expensive Fixed-Point and Saturation Code” on page 44-28


“Identify Questionable Fixed-Point Operations” on page 44-31

“Identify Blocks that Generate Expensive Rounding Code” on page 44-34

You can use the Simulink Model Advisor to help you configure your fixed-point models to achieve a more efficient design and optimize your generated code. To use the Model Advisor to check your fixed-point models:

- 1 From the **Analysis** menu of the model you want to analyze, select **Model Advisor**.
- 2 In the System Selector, select the system to analyze.
- 3 In the Model Advisor left pane, expand the **By Product** node and then the **Embedded Coder** node.
- 4 For fixed-point code generation, the most important check boxes to select are **Identify blocks that generate expensive fixed-point and saturation code**, **Identify questionable fixed-point operations**, **Identify blocks that generate expensive rounding code**, and **Check the hardware implementation**.

To enable all Model Advisor checks associated with the selected node, from the Model Advisor **Edit** menu, select **Select All**.

- 5 Click **Run Selected Checks** . Any tips for improving the efficiency of your fixed-point model appear in the Model Advisor window.

The sections that follow discuss fixed-point related checks and sub-checks found in the Model Advisor. The sections explain the checks, discuss their importance in fixed-point code generation, and offer suggestions for tweaking your model to optimize your generated code.

Identify Blocks that Generate Expensive Fixed-Point and Saturation Code

Identify Sum blocks for questionable fixed-point operations

- When the input range of a Sum block exceeds the output range, a range error occurs. Users can get any addition or subtraction their application requires by inserting data type conversion blocks before and/or after the sum block.

- When a Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output, the mismatch requires the Sum block to perform a multiply operation each time the input is converted to the output's data type and scaling. The mismatch can be removed by changing the scaling of the output or the input.
- When the net sum of the Sum block input biases does not equal the bias of the output, the generated code includes one extra addition or subtraction instruction to correctly account for the net bias adjustment. Changing the bias of the output scaling can make the net bias adjustment zero and eliminate the need for the extra operation.

Identify Min Max blocks for questionable fixed-point operations

- When the input and output of the MinMax block have different data types, a conversion operation is required every time the block is executed. The model is more efficient with the same data types.
- When the data type and scaling of the input of the MinMax block does not match the data type and scaling of the output, a conversion is required before performing a relational operation. This could result in a range error when casting, or a precision loss each time a conversion is performed. Change the scaling of either the input or output to generate more efficient code.
- When the input of the MinMax block has a different slope adjustment factor than the output, the MinMax block requires a multiply operation each time the block is executed to convert the input to the data type and scaling of the output. You can correct the mismatch by changing the scaling of either the input or output.

Identify Discrete Integrator blocks for questionable fixed-point operations

- When the initial condition for the Discrete-Time Integrator blocks is used to initialize the state and output, the output equation generates excessive code and an extra global variable is required. It is recommended that you set the **Function Block Parameters > Initial condition setting** parameter to `State` (most efficient).

Identify Compare to Constant blocks for questionable fixed-point operations

- If the input data type of the Compare to Zero block cannot represent zero exactly, the input signal is compared to the closest representable value of zero, resulting in parameter overflow. To avoid this parameter overflow, select an input data type that can represent zero.
- If the Compare to Constant block's **Constant value** is outside the range that the input data type can represent, the input signal is compared to the closest

representable value of the constant. This results in parameter overflow. To avoid this parameter overflow, select an input data type that can represent the **Constant value**, or change the **Constant value** to a value that can be accommodated by the input data type.

Identify Lookup Table blocks for questionable fixed-point operations

Efficiency trade-offs related to lookup table data are described in “Effects of Spacing on Speed, Error, and Memory Usage” on page 38-21. Based on these trade-offs, the Model Advisor identifies blocks where there is potential for efficiency improvements, such as:

- Lookup table input data is not evenly spaced.
- Lookup table input data is *not* evenly spaced when quantized, but it is very close to being evenly spaced.
- Lookup table input data is evenly spaced, but the spacing is not a power of two.

For more information on lookup table optimization, see “Lookup table optimization” on page 44-36.

Check optimization and hardware implementation settings

- Integer division generated code contains protection against arithmetic exceptions such as division by zero, INT_MIN/-1, and LONG_MIN/-1. If you construct models making it impossible for exception triggering input combinations to reach a division operation, the protection code generated as part of the division operation is redundant.
- The index search method **Evenly-spaced points** requires a division operation, which can be computationally expensive.

Identify blocks that will invoke net slope computation

When a change of fixed-point slope is not a power of two, net slope computation is necessary. Normally, net slope computation is implemented using an integer multiplication followed by shifts. Under some conditions, an alternate implementation requires just an integer division by a constant. One of the conditions is that the net slope can be very accurately represented as the reciprocal of an integer. When this condition is met, the division implementation produces more accurate numerical behavior. Depending on your compiler and embedded hardware, the division implementation might be more desirable than the multiplication and shifts implementation. The generated code might be more efficient in either ROM size or model execution size.

The Model Advisor alerts you when:

- You set the **Use division for fixed-point net slope computation** optimization parameter to 'On', but your model configuration is not compatible with this selection.
- Your model configuration is suitable for using division to handle net slope computation, but you do not set the **Use division for fixed-point net slope computation** to 'On' optimization parameter.

For more information, see “Net Slope Computation” on page 44-8.

Identify product blocks that are less efficient

The number of multiplications and divisions that a block performs can have a big impact on accuracy and efficiency. The Model Advisor detects some, but not all, situations where rearranging the operations can improve accuracy, efficiency, or both.

One such situation is when a calculation using more than one division operation is computed. A general guideline from the field of numerical analysis is to multiply all the denominator terms together first, then do one and only one division. This improves accuracy and often speed in floating-point and especially fixed-point. This can be accomplished in Simulink by cascading Product blocks. Note that multiple divisions spread over a series of blocks are not detected by the Model Advisor.

Another situation is when a single Product block is configured to do more than one multiplication or division operation. This is supported, but if the output data type is integer or fixed-point, then better results are likely if this operation is split across several blocks each doing one multiplication or one division. Using several blocks allows the user to control the data type and scaling used for intermediate calculations. The choice of data types for intermediate calculations affects precision, range errors, and efficiency.

Check for expensive saturation code

Setting the **Saturate on integer overflow** parameter can produce condition checking code that your application might not require.

Check whether your application requires setting **Function Block Parameters > Signal Attributes > Saturate on integer overflow**. Otherwise, clear this parameter for the most efficient implementation of the block in the generated code.

Identify Questionable Fixed-Point Operations

This check identifies blocks that generate multiword operations, cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

Check for multiword operations

When an operation results in a data type larger than the largest word size of your processor, the generated code contains multiword operations. Multiword operations can be inefficient on hardware. To prevent multiword operations, adjust the word lengths of inputs to operations so that they do not exceed the largest word size of your processor. For more information on controlling multiword operations in generated code, see “Fixed-Point Multiword Operations In Generated Code”.

Check for expensive multiplication code

- “Targeting an Embedded Processor” on page 34-4 discusses the capabilities and limitations of embedded processors. “Design Rules” on page 34-5 recommends that inputs to a multiply operation should not have word lengths larger than the base integer type of your processor. Multiplication with larger word lengths can always be handled in software, but that approach requires much more code and is much slower. The Model Advisor identifies blocks where undesirable software multiplications are required. Visual inspection of the generated code, including the generated multiplication utility function, will make the cost of these operations clear. It is strongly recommended that you adjust the model to avoid these operations.
- “Rules for Arithmetic Operations” on page 33-56 discusses the implementation details of fixed-point multiplication and division. That section shows the significant increase in complexity that occurs when signals with nonzero biases are involved in multiplication and division. It is strongly recommended that you make changes to eliminate the need for these complicated operations. Extra steps are required to implement the multiplication. Inserting a Data Type Conversion block before and after the block doing the multiplication allows the biases to be removed and allows the user to control data type and scaling for intermediate calculations. In many cases the Data Type Conversion blocks can be moved to the “edges” of a (sub)system. The conversion is only done once and all blocks can benefit from simpler bias-free math.

Check for expensive division code

The rounding behavior of signed integer division is not fully specified by C language standards. Therefore, the generated code for division is too large to provide bit-true agreement between simulation and code generation. To avoid integer division generated code that is too large, in the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the **Signed integer division rounds to** (Simulink) parameter to the recommended value.

Identify lookup blocks with uneven breakpoint spacing

Efficiency trade-offs related to lookup table data are described in “Effects of Spacing on Speed, Error, and Memory Usage” on page 38-21. Based on these trade-offs, the Model Advisor identifies blocks where there is potential for efficiency improvements, and issues a warning when:

- Lookup table input data is not evenly spaced.
- Lookup table input data is *not* evenly spaced when quantized, but it is very close to being evenly spaced.
- Lookup table input data is evenly spaced, but the spacing is not a power of two.

Check for expensive pre-lookup division

For a Prelookup or n-D Lookup Table block, **Index search method** is **Evenly spaced points**. Breakpoint data does not have power of 2 spacing.

If breakpoint data is nontunable, it is recommended that you adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different **Index search method** to avoid the computation-intensive division operation.

Check for expensive data type conversions

When a block is configured such that it would generate inefficient code for a data type conversion, the Model Advisor generates a warning and makes suggestions on how to make your model more efficient.

Check for fixed-point comparisons with predetermined results

When you select `isNan`, `isFinite`, or `isInf` as the operation for the Relational Operator block, the block switches to one-input mode. In this mode, if the input data type is fixed point, boolean, or a built-in integer, the output is `FALSE` for `isInf` and `isNan`, `TRUE` for `isFinite`. This might result in dead code which will be eliminated by Simulink Coder.

Check for expensive binary comparison operations

- When the input data types of a Relational Operator block are not the same, a conversion operation is required every time the block is executed. If one of the inputs is invariant, then changing the data type and scaling of the invariant input to match the other input improves the efficiency of the model.

- When the inputs of a Relational Operator block have different ranges, there will be a range error when casting, and a precision loss each time a conversion is performed. You can insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type that has enough range and precision to represent each input.
- When the inputs of a Relational Operator block have different slope adjustment factors, the Relational Operator block is required to perform a multiply operation each time the input with lesser positive range is converted to the data type and scaling of the input with greater positive range. The extra multiplication requires extra code, slows down the speed of execution, and usually introduces additional precision loss. By adjusting the scaling of the inputs, you can eliminate mismatched slopes.

Check for expensive comparison code

When your model is configured such that the generated code contains expensive comparison code, the Model Advisor generates a warning.

Identify Blocks that Generate Expensive Rounding Code

This check alerts you when rounding optimizations are available. To check for blocks that generate expensive rounding code, the Model Advisor performs the following sub-checks:

- Check for expensive rounding operations in multiplication and division
- Check optimization and Hardware Implementation settings (Lookup Blocks)
- Check for expensive rounding in a data type conversion
- Check for expensive rounding modes in the model

Traditional handwritten code, especially for control applications, almost always uses “no effort” rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the simplest rounding mode. In general the simplest mode provides the minimum cost solution with no overflows. If the simplest mode is not available, round to floor.

The primary exception to this rule is the rounding behavior of signed integer division. The C standard leaves this rounding behavior unspecified, but for most production targets the “no effort” mode is to round to zero. For unsigned division, everything is nonnegative, so rounding to floor and rounding to zero are identical. To improve rounding efficiency, set **Model Configuration Parameters > Hardware Implementation >**

Production hardware > Signed integer division rounds to using the mode that your production target uses.

Use the **Integer rounding mode** parameter on your model's blocks to simulate the rounding behavior of the C compiler that you use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product block. To obtain the most efficient generated code, change the **Integer rounding mode** parameter of the block to the recommended setting.

For more information on properties to consider when choosing a rounding mode, see “Choosing a Rounding Method” on page 1-10.

Lookup table optimization

A function lookup table is a method by which you can approximate a function using a table with a finite number of points (X , Y). The X values of the lookup table are called the breakpoints. You approximate the value of the ideal function at a point by interpolating between the two breakpoints closest to the point. Because table lookups and simple estimations can be faster than mathematical function evaluations, using lookup table blocks often result in speed gains when simulating a model.

To optimize lookup tables in your model:

- Limit uneven lookup tables.

Unevenly-spaced breakpoints require a general-purpose algorithm such as a binary search to determine where the input lies in relation to the breakpoints. This additional computation increases ROM and execution time.

- Prevent evenly spaced lookup tables from being treated as unevenly spaced.

The position search in evenly spaced lookup tables is much faster. In addition, the interpolation requires a simple division.

Sometimes, when a lookup table is converted to fixed-point, a quantization error results. A lookup table that is evenly spaced in floating-point, could be unevenly spaced in the generated fixed-point code. Use the `fixpt_evenspace_cleanup` function to convert the data into an evenly spaced lookup table again.

- Use power of two spaced breakpoints in lookup tables.

In power of two spaced lookup tables, a bit shift replaces the position search, and a bit mask replaces the interpolation making this construct the most efficient regardless of your target language and hardware.

The following table summarizes the effects of lookup table breakpoint spacing.

Parameter	Even Power of 2 Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
Execution speed	The execution speed is the fastest. The position search and interpolation are the same as for evenly-	The execution speed is faster than that for unevenly-spaced data, because the position search is faster and the	The execution speed is the slowest of the different spacings because the position search is slower, and

Parameter	Even Power of 2 Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
	spaced data. However, to increase the speed more, a bit shift replaces the position search, and a bit mask replaces the interpolation.	interpolation requires a simple division.	the interpolation requires more operations.
Error	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be smaller because approximating a function with nonuniform curvature requires fewer points to achieve the same accuracy.
ROM usage	Uses less command ROM, but more data ROM.	Uses less command ROM, but more data ROM.	Uses more command ROM, but less data ROM.
RAM usage	Not significant.	Not significant.	Not significant.

Use the Model Advisor “Identify Questionable Fixed-Point Operations” on page 44-31 check to identify lookup table blocks where there is potential for efficiency improvements.

More About

- “Effects of Spacing on Speed, Error, and Memory Usage” on page 38-21
- “Create Lookup Tables for a Sine Function” on page 38-5

Selecting Data Types for Basic Operations

In this section...

“Restrict Data Type Word Lengths” on page 44-38

“Avoid Fixed-Point Scalings with Bias” on page 44-38

“Wrap and Round to Floor or Simplest” on page 44-39

“Limit the Use of Custom Storage Classes” on page 44-40

Restrict Data Type Word Lengths

If possible, restrict the fixed-point data type word lengths in your model so that they are equal to or less than the integer size of your target microcontroller. This results in fewer mathematical instructions in the microcontroller, and reduces ROM and execution time.

This recommendation strongly applies to global variables that consume global RAM. For example, Unit Delay blocks have discrete states that have the same word lengths as their input and output signals. These discrete states are global variables that consume global RAM, which is a scarce resource on many embedded systems.

For temporary variables that only occupy a CPU register or stack location briefly, the space consumed by a `long` is less critical. However, depending on the operation, the use of `long` variables in math operations can be expensive. Addition and subtraction of long integers generally requires the same effort as adding and subtracting regular integers, so that operation is not a concern. In contrast, multiplication and division with long integers can require significantly larger and slower code.

Avoid Fixed-Point Scalings with Bias

Whenever possible, avoid using fixed-point numbers with bias. In certain cases, if you choose biases carefully, you can avoid significant increases in ROM and execution time. Refer to “Recommendations for Arithmetic and Scaling” on page 33-40 for more information on how to choose appropriate biases in cases where it is necessary; for example if you are interfacing with a hardware device that has a built-in bias. In general, however, it is safer to avoid using fixed-point numbers with bias altogether.

Inputs to lookup tables are an important exception to this recommendation. If a lookup table input and the associated input data use the same bias, then there is no penalty associated with nonzero bias for that operation.

Wrap and Round to Floor or Simplest

For most fixed-point and integer operations, the Simulink software provides you with options on how overflows are handled and how calculations are rounded. Traditional handwritten code, especially for control applications, almost always uses the “no effort” rounding mode. For example, to reduce the precision of a variable, that variable is shifted right. For unsigned integers and two's complement signed integers, shifting right is equivalent to rounding to floor. To get results comparable to or better than what you expect from traditional handwritten code, you should round to floor in most cases.

The primary exception to this rule is the rounding behavior of signed integer division. The C language leaves this rounding behavior unspecified, but for most targets the “no effort” mode is round to zero. For unsigned division, everything is nonnegative, so rounding to floor and rounding to zero are identical.

You can improve code efficiency by setting the value of the **Model Configuration Parameters > Hardware Implementation > Production hardware > Signed integer division rounds to** parameter to describe how your production target handles rounding for signed division. For Product blocks that are doing only division, setting the **Integer rounding mode** parameter to the rounding mode of your production target gives the best results. You can also use the **Simplest** rounding mode on blocks where it is available. For more information, refer to “Rounding Mode: Simplest” on page 33-16.

The options for overflow handling also have a big impact on the efficiency of your generated code. Using software to detect overflow situations and saturate the results requires the code to be much bigger and slower compared to simply ignoring the overflows. When overflows are ignored for unsigned integers and two's complement signed integers, the results usually wrap around modulo 2^N , where N is the number of bits. Unhandled overflows that wrap around are highly undesirable for many situations.

However, because of code size and speed needs, traditional handwritten code contains very little software saturation. Typically, the fixed-point scaling is very carefully set so that overflow does not occur in most calculations. The code for these calculations safely ignores overflow. To get results comparable to or better than what you would expect from traditional handwritten code, the **Saturate on integer overflow** parameter should not be selected for Simulink blocks doing those calculations.

In a design, there might be a few places where overflow can occur and saturation protection is needed. Traditional handwritten code includes software saturation for these few places where it is needed. To get comparable generated code, the **Saturate on**

integer overflow parameter should only be selected for the few Simulink blocks that correspond to these at-risk calculations.

A secondary benefit of using the most efficient options for overflow handling and rounding is that calculations often reduce from multiple statements requiring several lines of C code to small expressions that can be folded into downstream calculations. Expression folding is a code optimization technique that produces benefits such as minimizing the need to store intermediate computations in temporary buffers or variables. This can reduce stack size and make it more likely that calculations can be efficiently handled using only CPU registers. An automatic code generator can carefully apply expression folding across parts of a model and often see optimizations that might not be obvious. Automatic optimizations of this type often allow generated code to exceed the efficiency of typical examples of handwritten code.

Limit the Use of Custom Storage Classes

In addition to the tip mentioned in “Wrap and Round to Floor or Simplest” on page 44-39, to obtain the maximum benefits of expression folding you also need to make sure that the **Storage class** field in the Signal Properties dialog box is set to **Auto** for each signal. When you choose a setting other than **Auto**, you need to name the signal, and a separate statement is created in the generated code. Therefore, only use a setting other than **Auto** when it is necessary for global variables.

You can access the Signal Properties dialog box by selecting any connection between blocks in your model, and then selecting **Signal Properties** from the Simulink **Edit** menu.

Use of Shifts by C Code Generation Products

Required products: Embedded Coder, Fixed-Point Designer, Simulink.

Introduction to Shifts by Code Generation Products

MATLAB Coder, Simulink Coder, and Embedded Coder generate C code that uses the C language's shift left << and shift right >> operators. Modern C compilers provide consistent behavior for shift operators. However, some behaviors of the shift operators are not fully defined by some C standards. When you work with The MathWorks code generation products, you need to know how to manage the use of C shifts.

Two's Complement

Two's complement is a way to interpret a binary number. Most modern processors represent integers using two's complement. MathWorks code generation products require C and C++ compilers to represent signed integers using two's complement. MathWorks toolboxes and documentation use two's complement representation exclusively.

Arithmetic and Logical Shifts

The primary difference between an arithmetic shift and a logical shift is intent. Arithmetic shifts have a mathematical meaning. The intent of logical shifts is to move bits around, making them useful only for unsigned integers being used as collections of bit flags.

The C language does not distinguish between arithmetic and logical shifts and provides only one set of shift operators. When MathWorks code generation products use shifts on signed integers in generated code, the intent is always an arithmetic shift. For unsigned integers, there is no detectable difference in behavior between logical and arithmetic shifts.

Arithmetic Left-Shifts

An arithmetic left-shift represents multiplication by a power of 2.

$$a \ll b = a * 2^b$$

If the value produced by multiplying by 2^b is too big, then an overflow occurs. In case of an overflow, the ideal answer wraps around modulo 2^n to fit in the data type. The C90 standard specifies left-shift behavior. At the bit level, b of the bits are shifted off the

left end and discarded. At the right end, b bits of value 0 are shifted in. The standard does not specify a difference between unsigned and signed. For both unsigned and two's complement signed, the bit level behavior provides the intended arithmetic left-shift behavior.

The C99 standard describes the arithmetic interpretation. It also states that for signed types, the behavior is undefined for any negative value or for a positive value that would overflow. A compiler vendor might exploit the C99 standard undefined behavior clause to optimize code in a way that changes the behavior intended by the coder. If your compiler is C99-compliant but not C90-compliant, then turn off the option “Replace multiplications by powers of two with signed bitwise shifts” (Embedded Coder) (Embedded Coder). Older C++ standards follow the C90 standard with regard to shift left. Newer C++ standards are similar to the C99 standard.

Arithmetic Right-Shifts

An arithmetic right-shift represents division by a power of 2, where the ideal quotient rounds to floor.

```
a >> b = a/2^b
```

When a is nonnegative, the C standards state that right-shift must provide this arithmetic behavior. If a is signed and negative, then the standard states that the implementation defines the behavior. The C standard requires that compilers document their implementation behavior. Nearly all compilers implement signed shift right as an arithmetic shift that rounds to floor. This is the simplest and most efficient behavior for the compiler vendor to provide. If you have a compiler that does not provide arithmetic right-shift, or your coding standards do not allow you to use signed right-shift, then you can select options that avoid signed shift right. For example, “Allow right shifts on signed integers” (Embedded Coder) (Embedded Coder) replaces signed right shifts with a function call.

Out-of-Range Shifts

In C, when shifting an integer of word length n , use a shift amount between 0 and $n - 1$, inclusive. The C standard does not define shifting by other amounts, such as:

- Shifting by a negative constant.
- Shifting by an amount greater than word length.

When the shift amount is constant, the products do not generate out-of-range shifts. The risk of out-of-range shifts comes from explicitly modeled shifts where the shift amount is

a non constant variable. When modeling shifts with variable shift amounts, make sure that the shift amount is always in range.

Modeling Sources of Shifts

There are explicit and implicit sources of shifts in models and algorithms.

Explicit

- MATLAB bit-shift functions: `bitsll`, `bitsra`, `bitsrl`, `bitshift` (Fixed-Point Designer)
- Simulink Shift Arithmetic block
- Stateflow bitwise operations

Implicit

- Fixed-point operations that involve a scaling change

When converting fixed-point scaling, if the net slope change is not an exact power of two, then a multiplication followed by a shift approximates the ideal net slope. For more information on net slope computation, see “Handle Net Slope Computation” on page 44-8 (Fixed-Point Designer).

- Underlying higher-level algorithms (for example, FFT algorithms)

Controlling Shifts in Generated Code

Several configuration parameters have an effect on the number and style of shifts that appear in generated code.

- “Signed integer division rounds to” (Simulink) (Simulink)

Set this parameter to `Floor` or `Zero` to avoid extra generated code.

- “Use division for fixed-point net slope computation” (Simulink) (Simulink)

When enabled, this parameter uses division in place of multiplication followed by shifts to perform fixed-point net slope computation.

- “Replace multiplications by powers of two with signed bitwise shifts” (Embedded Coder) (Embedded Coder)

When this parameter is enabled, multiplications by powers of two are replaced with signed bitwise shifts. Clearing this option supports MISRA C compliance.

- “Allow right shifts on signed integers” (Embedded Coder) (Embedded Coder)

When this parameter is enabled, generated code can contain right bitwise shifts on signed integers. To prevent right bitwise shifts on signed integers, clear this option.

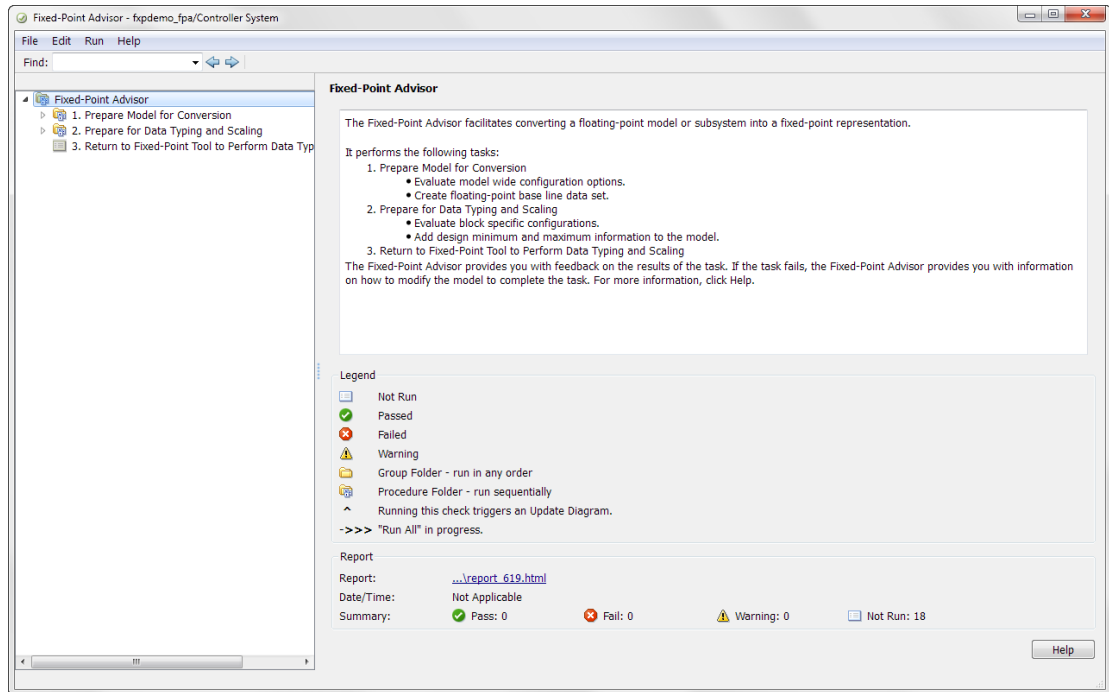
- “Shift right on a signed integer as arithmetic shift” (Simulink) (Simulink)

Select this parameter if the C compiler implements a signed integer right shift as an arithmetic right shift.

Fixed-Point Advisor Reference

- “Fixed-Point Advisor” on page 45-2
- “Preparing Model for Conversion” on page 45-6
- “Preparing for Data Typing and Scaling” on page 45-23
- “Return to the Fixed-Point Tool to Perform Data Typing and Scaling” on page 45-28

Fixed-Point Advisor



Fixed-Point Advisor Overview

The Fixed-Point Advisor is a tool you can use to prepare your model for conversion from floating-point data types to fixed-point data types. The Fixed-Point Advisor also makes recommendations for a model, such as model-level diagnostic settings. It configures the model for autoscaling by the Fixed-Point Tool. Therefore, even if your model uses only fixed-point data types, it can be useful to run the Fixed-Point Advisor on the model prior to autoscaling.

The Fixed-Point Advisor performs checks on the entire model reference hierarchy. It checks the top model and referenced models against fixed-point guidelines and reports results for each referenced model.

Description

Use the Fixed-Point Advisor to:

- Set model-wide configuration options.
- Set block-specific dialog parameters.
- Check for unsupported blocks.

Procedures

Automatically Run Tasks

The following steps list how you can automatically run all tasks within a folder.

- 1 Click the **Run All** button. The tasks run in order until a task fails.
- 2 Fix the failure:
 - Manually fix the problem using the **Explore Result** button, if present.
 - Manually fix the problem by modifying the model as instructed in the Analysis Result box.
 - Automatically fix the problem using the **Modify All** button, if available.
- 3 Continue to run the tasks by selecting **Run > Continue**.

Run Individual Tasks

The following steps list how you can run an individual task.

- 1 Specify **Input Parameters**, if present.

- 2 Run the task by clicking **Run This Task**.
- 3 Review Results. The possible results are:
 - Pass:** Move on to the next task.
 - Warning:** Review results, decide whether to move on or fix.
 - Fail:** Review results, do not move on without fixing.
- 4 If Status is **Warning** or **Fail**, you can:
 - Manually fix the problem using the **Explore Result** button, if present.
 - Manually fix the problem by modifying the model.
 - Automatically fix the problem using the **Modify All** button, if available.
- 5 Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run This Task**.

Run to Selected Task

If you know that a particular task causes a failure, you might want to run all the tasks prior to this task and save a restore point before continuing the run. For more information about restore points, see “Save a Restore Point” on page 35-7. To run all tasks up to and including the currently selected task:

- 1 Select the last task that you want to run.
- 2 Right click this task to open the context menu.
- 3 From the context menu, select **Run to Selected Task** to run all tasks up to and including the selected task.

Note: If a task before the selected task fails, the Fixed-Point Advisor stops the run at the failed task.

Rerun a Task

You might want to rerun a task to see if changes you make result in a different answer. To rerun a task that you have run before:

- 1 Select the task that you want to rerun.
- 2 Specify input parameters, if present.
- 3 Run the task by clicking **Run This Task**.

The task reruns.

Caution: All downstream tasks are reset to **Not Run** if:

- The task fails.
 - You click the **Modify All** button.
-

View a Run Summary

To view a complete run summary of **Pass**, **Failed**, **Warning**, and **Not Run** tasks:

- 1** Select the **Fixed-Point Advisor** folder.
- 2** Click the path link listed for Report. A report containing a summary of all tasks is displayed.

See Also

- “Best Practices” on page 35-2
- “Preparation for Fixed-Point Conversion” on page 35-2

Preparing Model for Conversion

In this section...

“Prepare Model for Conversion Overview” on page 45-7

“Verify model simulation settings” on page 45-8

“Verify update diagram status” on page 45-11

“Address unsupported blocks” on page 45-12

“Set up signal logging” on page 45-14

“Create simulation reference data” on page 45-15

“Verify Fixed-Point Conversion Guidelines Overview” on page 45-17

“Check model configuration data validity diagnostic parameters settings” on page 45-18

“Implement logic signals as Boolean data” on page 45-19

“Simulation range checking” on page 45-20

“Check for implicit signal resolution” on page 45-21

Prepare Model for Conversion Overview

This folder contains tasks for configuring and setting up the model for data logging.

Description

Validate model-wide settings and create simulation reference data for downstream tasks.

See Also

- “Preparation for Fixed-Point Conversion” on page 35-2
- “Convert Floating-Point Model to Fixed Point” on page 37-8

Verify model simulation settings

Validate that model simulation settings allow signal logging and disable data type override to facilitate conversion to fixed point. Logged signals are used for analysis and comparison in later tasks.

Description

Ensures that fixed-point data can be logged in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
The Fixed-Point Tool Data type override setting is not set to the correct value.	Set Data type override to Use local settings.
The Model Configuration Parameters Data Import/Export > Signal logging check box is set to off.	Set to on
The fipref <code>DataTypeOverride</code> property is not set to 'ForceOff'.	Set <code>DataTypeOverride</code> to 'ForceOff'
A Model Reference block is not in Normal mode.	Configure all referenced models that are in non-Normal mode to use Normal mode.
Note: If your model contains Model Reference blocks that are not in Normal mode, it does not make recommendations for them in subsequent tasks.	Use Modify All or manually configure each Model Reference block to use Normal mode: <ol style="list-style-type: none"> 1 To identify which referenced model instances are not in Normal mode, click the Model Dependency Viewer link to open the Model Dependency Viewer. The viewer displays all instances and shows which mode they are in. 2 To change the mode, select the referenced model, right-click, and then select Block Parameters (ModelReference). In the Block Parameters dialog box, set Simulation mode to Normal.

Conditions	Recommended Action
<p>A model in the model hierarchy is using a referenced configuration set instead of an active configuration set.</p>	<p>Tip: Ignore this warning for protected models or any other models that you do not want the Fixed-Point Advisor to check.</p> <hr/> <p>Use Modify All or manually configure all models in the model hierarchy to use the active configuration set. See “Manage a Configuration Set” (Simulink).</p> <p>If you use Modify All, the Fixed-Point Advisor creates an active configuration set by copying the referenced configuration set and configuring the model to use this new configuration set as the active set.</p> <p>If there are multiple reference sets, the Fixed-Point Advisor creates multiple new configuration sets.</p> <hr/> <p>Note: When a model uses a referenced configuration set, its configuration settings are read-only. Downstream tasks in the Fixed-Point Advisor make recommendations to tune configuration settings. If you do not update models to use an active configuration set, these tasks might generate warnings.</p>

Action Results

Clicking **Modify All**:

- Configures the model for recommended simulation settings and `fipref` objects. A table displays the current and previous block settings.
- Configures all referenced model instances to use Normal mode.
- Updates models in the model hierarchy that use referenced configuration sets to use an active configuration set.

See Also

- “Data Type Override” on page 39-44
- “Signal logging” (Simulink)
- “Data Type Override Preferences Using fipref” on page 5-12

Verify update diagram status

Verify update diagram succeeds.

Description

A model update diagram action is necessary for most down stream tasks.

Results and Recommended Actions

Conditions	Recommended Action
The model diagram does not update.	Fix the model. Verify that all required mat files are loaded.

See Also

“Update Diagram and Run Simulation” (Simulink) in the Simulink documentation

Address unsupported blocks

Identify blocks that do not support fixed-point data types.

Description

Blocks that do not support fixed-point data types cannot be converted.

Results and Recommended Actions

Conditions	Recommended Action
<p>Blocks that do not support fixed-point data types and cannot be converted exist in model.</p>	<ul style="list-style-type: none"> • Replace the block with the block specified in the Result pane by right-clicking the block and selecting the replacement from the context menu. <hr/> <p>Note: The Fixed-Point Advisor provides a preview of the replacement block. To view the replacement and verify its settings, click the Preview link. If the settings on the replacement block differ from the settings on the original block, set up the replacement block to match the original block.</p> <hr/> <ul style="list-style-type: none"> • Set the Approximation method of Trigonometric Function blocks to CORDIC. • Isolate the block by right-clicking the block and selecting Insert Data Type Conversion > All Ports. <p>The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, replace or isolate unsupported blocks in each referenced model.</p>

Tips

- Before inserting a replacement block, use the Preview link to view the replacement block. If necessary, update the settings on the replacement block to match the settings on the original block.

- If the Fixed-Point Advisor does not recommend a corresponding fixed-point block, replace the unsupported block with a number of lower-level blocks to provide the same functionality.
- The goal is to replace all blocks that do not support fixed-point data types. Using Data Type Conversion blocks to isolate blocks at this stage enables you to continue running through the conversion process. However, this will cause the **Summarize data type** task to fail downstream. To fix this failure, you must replace the block that does not support fixed-point data types.

See Also

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Fixed-Point Designer block libraries, including whether or not they support fixed-point data types. To view this table, enter the following command at the MATLAB command line:

```
showblockdatatypetable
```

Set up signal logging

Specify at least one signal of interest to log during simulation. Logged signals are used for analysis and comparison in other tasks. Suggested signals to log are model inports and outports.

Description

The Fixed-Point Advisor uses logged signals to compare the initial data type to the fixed-point data type.

Analysis Result and Recommended Actions

Conditions	Recommended Action
No signals are logged.	<p>If you are using simulation minimum and maximum values, specify at least one signal to be logged. Otherwise, ignore this warning.</p> <p>The Fixed-Point Advisor provides separate results for each referenced model. Specify at least one signal to be logged for each referenced model.</p>
A model in the model hierarchy is using a referenced configuration set and signal logging is disabled in this configuration set.	<p>Enable signal logging in the referenced configuration set.</p> <p>Return to the Verify model simulation settings task and configure all models to use an active configuration set.</p>

Tip

Log inports and outports of the system under conversion.

Create simulation reference data

Simulate the model using the current solver settings, and create reference data to use for comparison and analysis. If necessary, you can stop the simulation by selecting the waitbar and then pressing Ctrl+C. To set **Fixed-point instrumentation mode** to Minimums, maximums and overflows, click the **Modify All** button.

Description

Simulate the model using the current solver settings, create and archive reference signal data to use for comparison and analysis in downstream tasks.

Input Parameters

Merge results from multiple simulations

Merges new simulation minimum and maximum results with existing simulation results in the active run. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

Results and Recommended Actions

Conditions	Recommended Action
Simulation does not run.	Fix errors so simulation will run.
Fixed-point instrumentation mode is not set to Minimums, maximums and overflows	<p>If you are using simulation minimum and maximum values, set Fixed-point instrumentation mode to Minimums, maximums and overflows. Otherwise, ignore this warning.</p> <p>The Fixed-Point Advisor provides separate results for each referenced model. Use Modify All or manually configure Fixed-point instrumentation mode for each referenced model. Then the Fixed-Point Advisor collects simulation reference data for the entire model reference hierarchy.</p> <p>If your model is using a referenced configuration set and signal logging is disabled in this configuration set, either enable signal logging</p>

Conditions	Recommended Action
	in the referenced configuration set or return to the Verify model simulation settings task and configure all models to use an active configuration set.

Action Results

Clicking **Modify All** sets **Fixed-point instrumentation mode** to **Minimums, maximums and overflows**. A table displays the current and previous block settings.

If your model contains referenced models, sets **Fixed-point instrumentation mode** to **Minimums, maximums and overflows** on all referenced models.

If your model is using a referenced configuration set and signal logging is disabled in this configuration set, **Modify All** is disabled.

Tips

- If the simulation is set up to have a long simulation time, after starting the run of this task you can stop the simulation by selecting the waitbar and then pressing **Ctrl+C**. This allows you to change the simulation time and continue without having to wait for the long simulation to complete.
- Specifying small simulation run times reduces task processing times. You can change the simulation run time in the Configuration Parameters dialog box. See “Start time” (Simulink) and “Stop time” (Simulink) in the Simulink reference for more information.

Verify Fixed-Point Conversion Guidelines Overview

Verify modeling guidelines related to fixed-point conversion goals.

Description

Validate model-wide settings.

See Also

- “Preparation for Fixed-Point Conversion” on page 35-2
- “Convert Floating-Point Model to Fixed Point” on page 37-8

Check model configuration data validity diagnostic parameters settings

Verify that **Model Configuration Parameters > Diagnostic > Data Validity** parameters are not set to error.

Description

If the **Model Configuration Parameters > Diagnostic > Data Validity** parameters are set to error, the model update diagram action fails in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
Detect downcast is set to error.	Set all Model Configuration Parameters > Diagnostics > Data Validity > Parameters options to warning. The Fixed-Point Advisor provides separate results for each referenced model. Use Modify All or manually configure these options for each referenced model.
Detect overflow is set to error.	
Detect underflow is set to error.	
Detect precision loss is set to error.	
Detect loss of tunability is set to error.	

Action Results

Clicking **Modify All** sets all **Model Configuration Parameters > Diagnostics > Data Validity > Parameters** options to warning. A table displays the current and previous settings.

If your model contains referenced models, modifies these settings on all referenced models.

If your model is using a referenced configuration set, because the referenced configuration set is read-only, **Modify All** cannot update this parameter. Either manually modify the referenced configuration set or return to the **Verify model simulation settings** task and configure all models to use an active configuration set.

Implement logic signals as Boolean data

Confirm that Simulink simulations are configured to treat logic signals as Boolean data.

Description

Configuring logic signals as Boolean data optimizes the code generated in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
Implement logic signals as Boolean data is set to off.	<p>Set Model Configuration Parameters > All Parameters > Implement logic signals as Boolean data to on.</p> <p>The Fixed-Point Advisor provides separate results for each referenced model. Use Modify All or manually configure this option for each referenced model.</p>

Action Results

Clicking **Modify All** selects the model **Model Configuration Parameters > All Parameters > Implement logic signals as Boolean data** check box. A table displays the current and previous parameter settings.

If your model contains referenced models, modifies this setting on all referenced models.

If your model is using a referenced configuration set, because the referenced configuration set is read-only, **Modify All** cannot update this parameter. Either manually modify the referenced configuration set or return to the **Verify model simulation settings** task and configure all models to use an active configuration set.

Simulation range checking

Verify that **Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** is not set to none.

Description

If **Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** is set to none, the simulation does not generate any range checking warnings.

Results and Recommended Actions

Conditions	Recommended Action
Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking is set to none.	<p>Set Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking to warning.</p> <p>The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, use Modify All or manually configure this option for each referenced model.</p>

Action Results

Clicking **Modify All** sets **Model Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** to warning.

If your model contains referenced models, modifies this setting on all referenced models.

If your model is using a referenced configuration set, because the referenced configuration set is read-only, **Modify All** cannot update this parameter. Either manually modify the referenced configuration set or return to the **Verify model simulation settings** task and configure all models to use an active configuration set.

Check for implicit signal resolution

Check if model uses implicit signal resolution.

Description

Models with implicit signal resolution attempt to resolve all named signals and states to Simulink signal objects, which is inefficient and slows incremental code generation and model reference. This task identifies those signals and states for which you may turn off implicit signal resolution and enforce resolution.

Results and Recommended Actions

Conditions	Recommended Action
Model uses implicit signal resolution.	<ul style="list-style-type: none"> • Set Model Configuration Parameters > Diagnostics > Data Validity > Signal resolution to Explicit only. • Enforce resolution for each of the signals and states in the model by selecting Signal name must resolve to Simulink signal object. <p>The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, use Modify All or manually configure these options for each referenced model.</p>

Action Results

Clicking **Modify All** sets **Model Configuration Parameters > Diagnostics > Data Validity > Signal resolution** to **Explicit only** and enforces resolution for each of the signals and states in the model. Tables display the current and previous settings.

If your model contains referenced models, modifies these settings on all referenced models.

If your model is using a referenced configuration set, because the referenced configuration set is read-only, **Modify All** cannot update this parameter. Either manually modify the referenced configuration set or return to the **Verify model simulation settings** task and configure all models to use an active configuration set.

See Also

“Resolve Signal Objects for Output Data” (Simulink) in the Simulink documentation

Preparing for Data Typing and Scaling

In this section...
“Prepare for Data Typing and Scaling Overview” on page 45-24
“Review locked data type settings” on page 45-25
“Verify Stateflow charts have strong data typing with Simulink” on page 45-26
“Specify block minimum and maximum values” on page 45-27

Prepare for Data Typing and Scaling Overview

Configure blocks with constraints to avoid data type propagation errors.

Description

The block settings from this folder simplify the initial data typing and scaling. The optimal block configuration is achieved in later stages. The tasks in this folder are preparation for automatic data typing.

Tips

Block output and parameter minimum and maximum values can be specified in this step.

See Also

- “Preparation for Fixed-Point Conversion” on page 35-2
- “Convert Floating-Point Model to Fixed Point” on page 37-8

Review locked data type settings

Review blocks that currently have their data types locked down and will be excluded from automatic data typing.

Description

When blocks have their data types locked, the Fixed-Point Advisor excludes them from automatic data typing. This task identifies blocks that have locked data types so that you can unlock them.

Results and Recommended Actions

Conditions	Recommended Action
Blocks have locked data types.	Unlock data types on blocks that currently have locked data types. The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, use Modify All or unlock data types on blocks for all referenced models.

Action Results

Clicking **Modify All** unlocks data types on blocks that currently have locked data types.

If your model contains referenced models, unlocks data types on blocks for all referenced models.

Verify Stateflow charts have strong data typing with Simulink

Verify all Stateflow charts are configured to have strong data typing with Simulink I/O.

Description

Identify mismatches between input or output fixed-point data in Stateflow charts and their counterparts in Simulink models.

Note: This task is preparation for automatic data typing, not actual automatic data typing.

Results and Recommended Actions

Conditions	Recommended Action
Stateflow charts do not have strong data typing with Simulink I/O.	<p>Select the Use Strong Data Typing with Simulink I/O check box in the chart properties dialog.</p> <p>The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, use Modify All or manually configure this setting in each referenced model.</p>

Action Results

Clicking **Modify All** configures all Stateflow charts to have strong data typing with Simulink I/O.

If your model contains referenced models, configures this setting for all referenced models.

Specify block minimum and maximum values

Specify block output and parameter minimum and maximum values.

Description

Block output and parameter minimum and maximum values are used for fixed-point data typing in other tasks. Typically, they are determined during the design process based on the system you are creating.

Note: This task is preparation for automatic data typing, not actual automatic data typing.

Results and Recommended Actions

Conditions	Recommended Action
Minimum and maximum values are not specified for Inport blocks.	Specify minimum and maximum values for Inport blocks.
Warning if no simulation minimum or maximum for any signals.	If you are using simulation minimum and maximum data, return to “Create simulation reference data” to set up signal logging.

Tips

- In this task, you can specify minimum and maximum values for any block.
- You can promote simulation minimum and maximum values to output minimum and maximum values using the Model Advisor Result Explorer, launched by clicking the **Explore Result** button. In the center pane of the Model Advisor Result Explorer, use the check boxes in the **PromoteSimMinMax** column to promote values.
- If you do not specify block minimum and maximum values, the **Propose data types** task might fail later in the conversion.

See Also

“Batch-Fix Warnings or Failures” (Simulink) in the Simulink documentation.

Return to the Fixed-Point Tool to Perform Data Typing and Scaling

Close the Fixed-Point Advisor and return to the Fixed-Point Tool to autoscale your model.

See Also

- “Preparation for Fixed-Point Conversion” on page 35-2

Troubleshooting

- “What Is the Difference Between Fixed-Point and Built-In Integer Types?” on page 46-3
- “Negative Fraction Length” on page 46-4
- “Fraction Length Greater Than Word Length” on page 46-6
- “fi Constructor Does Not Follow globalfimath Rules” on page 46-8
- “Decide Which Workflow Is Right For Your Application” on page 46-9
- “Tips for Making Generated Code More Efficient” on page 46-11
- “Know When a Function is Supported for Instrumentation and Acceleration” on page 46-13
- “What to Do If a Function Is Not Supported for Fixed-Point Conversion” on page 46-14
- “fi*non-fi Errors” on page 46-16
- “Data Type Mismatch Errors” on page 46-17
- “Mismatched fimath Errors” on page 46-18
- “Why Does the Fixed-Point Converter App Not Propose Data Types for System Objects?” on page 46-19
- “Slow Operations in the Fixed-Point Converter App” on page 46-20
- “Using Blocks that Don’t Support Fixed-Point Data Types” on page 46-21
- “Prevent The Fixed-Point Tool From Overriding Integer Data Types” on page 46-23
- “Why Did The Fixed-Point Tool Not Propose Data Types?” on page 46-24
- “Frequently Asked Questions About Fixed-Point Numbers” on page 46-25
- “Why am I missing data type proposals for MATLAB Function block variables?” on page 46-26
- “Data Type Propagation Errors After Applying Proposed Data Types” on page 46-27
- “Resolve Range Analysis Issues” on page 46-30

- “Data Type Mismatch and Structure Initial Conditions” on page 46-31
- “Compile-Time Recursion Limit Reached” on page 46-33
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 46-38
- “Unable to Determine That Every Element of Cell Array Is Assigned” on page 46-41
- “Nonconstant Index into varargin or varargin in a for-Loop” on page 46-45

What Is the Difference Between Fixed-Point and Built-In Integer Types?

There are several distinct differences between fixed-point data types and the built-in integer types in MATLAB. The most notable difference, is that the built-in integer data types can only represent whole numbers, while the fixed-point data types also contain information on the position of the binary point, or the scaling of the number. This scaling allows the fixed-point data types to represent both integers and non-integers. There are also slight differences in how math is performed with these types. Fixed-point types allow you to specify rules for math using the `fimath` object, including overflow and rounding modes. However, the built-in types have their own internal rules for arithmetic operations. See “Integers” (MATLAB) for more information on how math is performed using built-in types.

See Also

`fi` | `fimath`

Negative Fraction Length

A negative fraction length occurs when the input value of a `fi` object contains trailing zeros before the decimal point. For example,

```
x = fi(16000,1,8)
```

produces a fixed-point number with the specified signedness and word length, and best precision fraction length.

```
x =
```

```
16000
```

```
    DataTypeMode: Fixed-point: binary point scaling
```

```
    Signedness: Signed
```

```
    WordLength: 8
```

```
    FractionLength: -7
```

View the binary representation of `x`.

```
disp(bin(x))
```

```
01111101
```

There are seven implicit zeros at the end of this number before the binary point because the fraction length of `x` is `-7`.

Convert from binary to decimal the binary representation of `x` with seven zero bits appended to the end.

```
bin2dec('01111101000000')
```

```
ans =
```

```
16000
```

The result is the real world value of `x`.

You can also find the real world value using the equation

$$\text{Real World Value} = \text{Stored Integer Value} \times 2^{-\text{Fraction Length}}$$

Start by finding the stored integer of `x`.

```
Q = storedInteger(x)
```

```
Q =
```

```
125
```

Use the stored integer to find the real world value of x.

```
real_world_value = double(Q) * 2^-x.FractionLength
```

```
real_world_value =
```

```
16000
```

See Also

fi

Fraction Length Greater Than Word Length

A fraction length greater than the word length of a fixed-point number occurs when the number has an absolute value less than one and contains leading zeros.

```
x = fi(.0234,1,8)
```

```
x =
```

```
0.0234
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 8  
      FractionLength: 12
```

View the binary representation of x .

```
disp(bin(x))
```

```
01100000
```

There are four implied leading zeros after the binary point and before the binary representation of the stored integer because the fraction length of x is four greater than the word length.

Convert from binary to decimal the binary representation of x with four leading zeros, and scale this value according to the fraction length.

```
bin2dec('000001100000')*2^(-12)
```

```
ans =
```

```
0.0234
```

The result is the real world value of x .

You can also find the real world value using the equation

$$\text{Real World Value} = \text{Stored Integer Value} \times 2^{-\text{Fraction Length}}$$

Start by finding the stored integer of x .

```
Q = storedInteger(x)
```

Q =

96

Use the stored integer to find the real world value of x.

```
real_world_value = double(Q) * 2^-x.FractionLength
```

```
real_world_value =
```

0.0234

See Also

fi

fi Constructor Does Not Follow globalfimath Rules

If no `fimath` properties are used in the argument of the `fi` constructor, then it always uses nearest rounding and saturates on overflow for the creation of the `fi` object, regardless of any `globalfimath` settings.

If this behavior is undesirable for your application, you can do one of the following:

- Use the `cast` function to create a `fi` object using the `globalfimath` rules.

```
G = globalfimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap');  
cast(x, 'like', fi([],1,16,10))
```

- Specify `fimath` properties in the `fi` constructor.

```
fi(x,1,16,10,'RoundingMethod','Floor','OverflowAction','Wrap');
```

Note: When you create a `fi` object using the `cast` function, the resulting `fi` object does not have a local `fimath`. When you create a `fi` object with `fimath` properties in the constructor, the `fi` object does have a local `fimath`.

See Also

`fi` | `fimath` | `globalfimath`

Decide Which Workflow Is Right For Your Application

There are two primary workflows available for converting MATLAB code to fixed-point code.

- **Manual Workflow**

The manual workflow provides the most control to optimize the fixed-point types, but requires a greater understanding of fixed-point concepts.

For more information, see “Manual Fixed-Point Conversion Best Practices” on page 12-4.

- **Automated Workflow**

The Fixed-Point Converter app enables you to convert your MATLAB code to fixed-point code without requiring extensive preexisting knowledge of fixed-point concepts. However, this workflow provides less control over your data types.

For more information, see “Automated Fixed-Point Conversion Best Practices” on page 8-54.

	Manual Workflow	Automated Workflow
Fully automated conversion		✓
Less fixed-point expertise required		✓
Quick turnaround time		✓
Simulation range analysis	✓	✓
Static range analysis		✓
Iterative workflow	✓	
Portable design	✓	✓
Greatest control and optimization of data types	✓	
Data type proposal	✓	✓
Histogram logging	✓	✓
Code coverage		✓

	Manual Workflow	Automated Workflow
Automatic plotting of output variables for comparison		✓

More About

- “Automated Conversion”
- “Manual Conversion”

Tips for Making Generated Code More Efficient

In this section...

“fimath Settings for Efficient Code” on page 46-11

“Replace Functions With More Efficient Fixed-Point Implementations” on page 46-11

fimath Settings for Efficient Code

The default settings of the `fimath` object are:

- `RoundingMethod`: `Nearest`
- `OverflowAction`: `Saturate`
- `ProductMode`: `FullPrecision`
- `SumMode`: `FullPrecision`

These settings offer the smallest rounding error and prevent overflows. However, they can result in extra logic in generated code. For leaner code, it is recommended that you match the `fimath` settings to the settings of your processor.

- The `KeepLSB` setting for `ProductMode` and `SumMode` models the behavior of integer operations in the C language. `KeepMSB` for `ProductMode` models the behavior of many DSP devices.
- Different rounding methods require different amounts of overhead code. Setting the `RoundingMethod` property to `FLOOR`, which is equivalent to two’s complement truncation, provides the most efficient rounding implementation for most operations. For the divide function, the most efficient `RoundingMethod` is `Zero`
- The standard method for handling overflows is to wrap using modulo arithmetic. Other overflow handling methods create costly logic. It is recommended that you set the `OverflowAction` property to `Wrap` when possible.

Replace Functions With More Efficient Fixed-Point Implementations

CORDIC

The CORDIC-based algorithms are among the most hardware friendly because they require only iterative shift-add operations. Replacing functions with one of the CORDIC

implementations can make your generated code more efficient. For a list of the CORDIC functions, and examples of them being implemented, see “CORDIC”.

Lookup tables

You can implement some functions more efficiently by using a lookup table approach. For an example, see “Implement Fixed-Point Log2 Using Lookup Table”.

Division

Division is often not supported by hardware. When possible, it is best to avoid division operations.

When the denominator is a power of two, you can rewrite the division as a bitshift operation.

$x/8$

can be rewritten as

`bitsra(x,3)`

Other times it is more efficient to implement division as a multiplication by a reciprocal.

$x/5$

can be rewritten as

$x*0.2$

Know When a Function is Supported for Instrumentation and Acceleration

There are several steps you can take to identify the features which could result in errors during conversion.

- `%#codegen` and `coder . screener`

Add the `%#codegen` pragma to the top of the MATLAB file that is being converted to fixed point. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during when you try to accelerate or instrument your code.

The `coder . screener` function takes your function as its input argument and warns you of anything in your code that is not supported for codegen. Codegen support is essential for minimum and maximum logging and data type proposals.

- Consult the table of supported functions

See “Language Support” for a table of features supported for code generation and fixed-point conversion.

For information on what to do if a function is not supported for fixed-point conversion, see “What to Do If a Function Is Not Supported for Fixed-Point Conversion” on page 46-14.

What to Do If a Function Is Not Supported for Fixed-Point Conversion

In this section...

“Isolate the Unsupported Functions” on page 46-14

“Create a Replacement Function” on page 46-14

Isolate the Unsupported Functions

When you encounter a function that is not supported for conversion, you can temporarily leave that part of the algorithm in floating point.

The following code returns an error because the `log` function is not supported for fixed-point inputs.

```
x = fi(rand(3),1,16,15);
y = log(x)
```

Cast the input, `x`, to a double, and then cast the output back to a fixed-point data type.

```
y = fi(log(double(x)),1,16)
y =
    -0.2050    -0.0906    -1.2783
    -0.0990    -0.4583    -0.6035
    -2.0637    -2.3275    -0.0435
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13
```

This casting allows you to continue with your conversion until you can find a replacement.

Create a Replacement Function

Lookup Table Approximation

You can replace many functions that are not supported for fixed-point conversion with a lookup table. For an example, see “Implement Fixed-Point Log2 Using Lookup Table”.

Polynomial Approximation

You can approximate the results of a function that is not supported for fixed-point with a polynomial approximation. For an example, see “Calculate Fixed-Point Arctangent”.

User-Authored Function

You can also write your own function that supports fixed-point inputs. For example, using the `mod` function, which does support fixed-point inputs, you can write your own version of the `rem` function, which does not support fixed-point inputs.

fi*non-fi Errors

Original Algorithm		New Algorithm
<pre>function y = myProduct(x) y = 1; for n = 1:length(x) y(:) = y*x(n); end end</pre>	<p><i>Issue:</i></p> <p>When multiplying a fixed-point variable by a non-fixed-point variable, the variable that does not have a fixed-point type can only be a constant.</p> <p><i>Fix:</i></p> <p>Before instrumenting your code, cast the non-fi variable to an acceptable fixed-point type.</p>	<pre>function y = myProduct(x) y = ones(1,1, 'like', x(1)*x(1)); for n = 1:length(x) y(:) = y*x(n); end end</pre>

Data Type Mismatch Errors

Original Algorithm		New Algorithm
<p><i>Function:</i></p> <pre>function y = mysum(x,T) %#C y = zeros(size(x), 'like', T); for n = 1:length(x) y = y + x(n); end end</pre>	<p><i>Issue:</i></p> <p><code>y</code> uses the default <code>fimath</code> setting <code>FullPrecision</code> for the <code>SumMode</code> property. At each iteration of the for-loop in the function <code>mysum</code>, the word length of <code>y</code> grows by one bit. During simulation in MATLAB, there is no issue because data types can easily change in MATLAB. However, a type mismatch error occurs at build time because data types must remain static in C.</p>	<p><i>Function:</i></p> <pre>function y = mysum(x,T) %#codegen y = zeros(size(x), 'like', T.y); for n = 1:length(x) y(:) = y + x(n); end end</pre>
<p><i>Types Table:</i></p> <pre>function T = mytypes(dt) switch(dt) case 'fixed' F = fimath('RoundingMethod', 'F'); T.x = fi([],1,16,11, F); T.y = fi([],1,16,6, F); end end</pre>	<p><i>Fix:</i></p> <p>Rewrite the function to use subscripted assignment within the for-loop.</p> <p>By rewriting <code>y = y + x(n)</code> as <code>y(:) = y + x(n)</code>, the value on the right is assigned in to the data type of <code>y</code>. This assignment preserves the <code>numericType</code> of <code>y</code> and avoids the type mismatch error.</p>	<p><i>Types Table:</i></p> <pre>function T = mytypes(dt) switch(dt) case 'fixed' F = fimath('RoundingMethod', 'F'); T.x = fi([],1,16,11, F); T.y = fi([],1,16,6, F); end end</pre>

Mismatched fimath Errors

Original Algorithm		New Algorithm
<p><i>Function:</i></p> <pre>function y = mysum(x,T) %#C y = zeros(size(x), 'like', T); for n = 1:length(x) y(:) = y + x(n); end end</pre>	<p><i>Issue:</i></p> <p>If two fi object operands have an attached fimath, the fimaths must be equal.</p> <p><i>Fix:</i></p>	<p><i>Function:</i></p> <pre>function y = mysum(x,T) %#codegen y = zeros(size(x), 'like', T.y); for n = 1:length(x) y(:) = removefimath(y) + x(n); end end</pre>
<p><i>Types Table:</i></p> <pre>function T = mytypes(dt) switch(dt) case 'fixed' T.x = fi([],1,16,0, 'OverflowAction', 'Wrap'); T.y = fi([],1,16,0, 'RoundingMethod', 'RoundNearest'); end end</pre>	<p>Use the <code>removefimath</code> function to remove the <code>fimath</code> of one of the variables in just one instance. By removing the <code>fimath</code>, you avoid the “mismatched <code>fimath</code>” error without permanently changing the <code>fimath</code> of the variable.</p>	<p><i>Types Table:</i></p> <pre>function T = mytypes(dt) switch(dt) case 'fixed' T.x = fi([],1,16,0, 'RoundingMethod', 'RoundNearest'); T.y = fi([],1,16,0, 'RoundingMethod', 'RoundNearest'); end end</pre>

Why Does the Fixed-Point Converter App Not Propose Data Types for System Objects?

The Fixed-Point Converter app might not display simulation range data or data type proposals for a System object because:

- The app displays range information for a subset of DSP System Toolbox System objects only. For a list of supported System objects, see [Converting System Objects to Fixed-Point Using the Fixed-Point Converter App](#) on page 8-84.
- The System object is not configured to use custom fixed-point settings.

If the system object is not configured correctly, the proposed data type column appears dimmed and displays `Full precision` or `Same as...` to show the current property setting.

Related Examples

- [“Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App”](#) on page 8-85

Slow Operations in the Fixed-Point Converter App

By default, the Fixed-Point Converter app screens your entry-point functions for code generation readiness. For some large entry-point functions, or functions with many calls, screening can take a long time. If the screening takes a long time, certain app or MATLAB operations can be slower than expected or appear to be unresponsive.

To determine if slow operations are due to the code generation readiness screening, disable the screening.

Using Blocks that Don't Support Fixed-Point Data Types

Find Blocks That Support Fixed-Point Data Types

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink block library, including whether or not they support fixed-point data types.

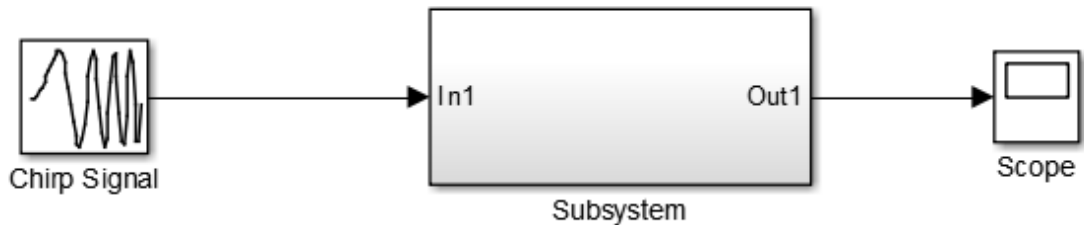
To view the table, enter the following command at the MATLAB command line:

```
showblockdatatypetable
```

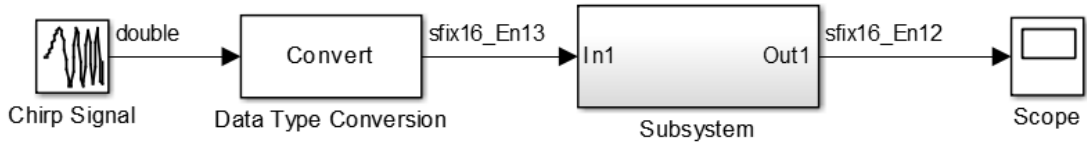
What to Do When a Block Is Not Supported For Fixed-Point Conversion

Isolate the Block

If you encounter a block that is not supported for fixed-point conversion, you can isolate the block by decoupling it with a Data Type Conversion block. This workaround is useful when you do not intend to use the unsupported block on an embedded processor. One example of this is using the Chirp Signal block, which does not support fixed-point outputs, to generate a signal for simulation data.



The subsystem shown is designed for use on an embedded processor and must be converted to fixed point. The Chirp Signal block creates simulation data. The Chirp Signal block supports only floating-point double outputs. However, if you decouple the Chirp Signal from the rest of the model by inserting a data type conversion block after the Chirp Signal block, you can use the Fixed-Point Tool to continue converting the subsystem to fixed point.



Lookup Table Block Implementation


Many blocks that are not supported by the Fixed-Point Tool can be approximated with a lookup table block. Design an efficient fixed-point implementation of an unsupported block by using the `fixpt_look1_func_approx` and `fixpt_look1_func_plot` functions. For an example, see “Approximate the Square Root Function” on page 38-3.

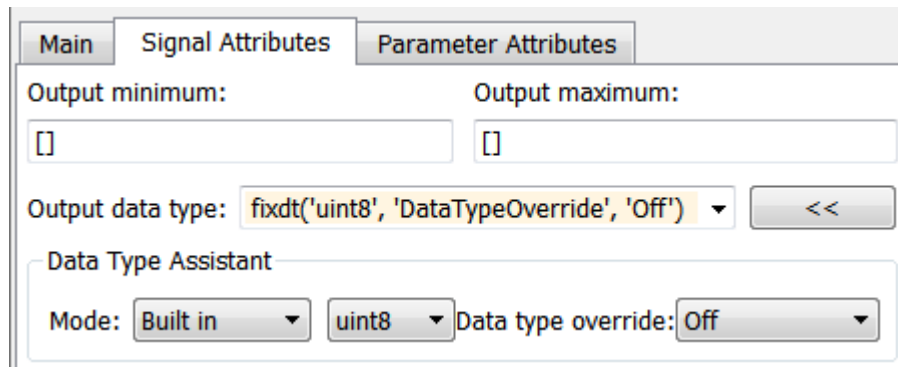
User-Authored Blocks

You can create your own block which is supported by the Fixed-Point Tool from one of the blocks in the User-Defined Functions Library.

Prevent The Fixed-Point Tool From Overriding Integer Data Types

When performing data type override (DTO) on a selected system, the Fixed-Point Tool overrides the output data types of each block in the system. The only blocks that are never affected by DTO are blocks with `boolean` or enumerated output data types, or blocks that are untouched by DTO by design (for example, lookup table blocks). Depending on your application, you might want certain signals to preserve their data type (for example, blocks that represent indices). To prevent the Fixed-Point Tool from overriding the data type of a specific block, set the `Data type override` setting of the numeric type of the block to `Off`.

- 1 Open the Block Parameters dialog box by double-clicking the block.
- 2 Under the **Signal Attributes** tab, open the Data Type Assistant by clicking .
- 3 Set **Data type override** to `Off`.



You can set this override to off at the command line by changing the Data Type Override setting of a signal's `numericType`. In this example, the output data type of this block remains a built-in `uint8` even after performing data type override.

Why Did The Fixed-Point Tool Not Propose Data Types?

In this section...
“Inadequate Range Information” on page 46-24
“Inherited Output Data Types” on page 46-24

Inadequate Range Information

The Fixed-Point Tool bases its data type proposition on range information collected through simulation, derivation, and design ranges that you provide. Before proposing data types, you must collect range information which the Fixed-Point Tool uses to propose data types.

To collect range information, simulate the system with range collection enabled in the Fixed-Point Tool, or derive minimum and maximum values. The Fixed-Point Advisor check **Create simulation reference data** also creates reference data to use for comparison and analysis.

Inherited Output Data Types

Blocks with inherited output data types use internal block rules to determine the output data type of the block. The Fixed-Point Tool proposes data types for these blocks only if the **Inherited** checkbox is selected in the proposal parameters under **Automatic data typing**. If this checkbox is not selected, the Fixed-Point Tool marks the proposal for these blocks as N/A.

Frequently Asked Questions About Fixed-Point Numbers

In this section...


“Fraction Length Greater Than Word Length” on page 46-25

“Negative Fraction Length” on page 46-25

Fraction Length Greater Than Word Length

A fraction length greater than the word length of a fixed-point number occurs when the number has an absolute value less than one and contains leading zeros.



In the following example, the fixed-point tool proposed a data type with a fraction length that is four greater than the word length. A binary representation of this number consists of the binary point, four implied leading zeros, followed by the binary representation of the stored integer: . X X X X 0 1 1 0 0 0 0, where the x’s represent the implied zeros.

Name	Run	CompiledDT	Accept	ProposedDT	SimMin	SimMax
 Gain	Run1	double	<input checked="" type="checkbox"/>	fixdt(1,8,12)	0.0234	0.0234

Negative Fraction Length

A negative fraction length occurs when the number contains trailing zeros before the decimal point.

In the following example, the fixed-point tool proposed a data type with a negative fraction length. A binary representation of this number consists of the binary representation of the stored integer, followed by seven implied zeros, and then the binary point: 0 1 1 1 1 0 1 X X X X X X ., where the x’s represent the implied zeros.

Name	Run	CompiledDT	Accept	ProposedDT	SimMin	SimMax
 Gain	Run 1	double	<input checked="" type="checkbox"/>	fixdt(1,8,-7)	16000	16000
 Out1	Run 1		<input type="checkbox"/>	n/a		

Why am I missing data type proposals for MATLAB Function block variables?

Fixed-Point Tool will not propose data types for variables in code inside a MATLAB Function block that is not executed during simulation. If your MATLAB Function block contains dead code, the variables will not appear in the Fixed-Point Tool.

- Update your input source so that all sections of your code are executed during simulation.
- This section of code may not be necessary. Delete the portion of code that is not exercised during simulation.

Data Type Propagation Errors After Applying Proposed Data Types


Under certain conditions, the Fixed-Point Tool may propose a data type that is not compatible with the model. The following topic outlines model configurations that may cause this issue, and how you can resolve the issue.

Tip: Before attempting to autoscale a model, always ensure that you can update diagram successfully without data type override turned on.

Shared Data Type Groups

View Shared Data Type Groups

Organizing Fixed-Point Tool results into groups that must share the same data type can aid in the debugging process.

- 1 In the Fixed-Point Tool, next to **Column View**, click **Show Details**.
- 2 On the left select the DTGroup property. To add the column to the current view, click . When you finish, click **Hide Details**.
- 3 Right-click the **DTGroup** column header and select **Group by this column**.

Results are now organized into groups that must share the same data type.

- 4 To limit your view to only results that have conflicts, below the menu bar, select **Show > Conflicts with proposed data types**.

Locked data type in a shared group

When an object is locked against changes by the Fixed-Point Tool, the Fixed-Point Tool does not propose a new data type for the object. If one of the results in a group of results that must share the same data type is locked, the Fixed-Point Tool proposes data types for all other objects in the group except for the locked object. If the data type proposed for the group is not compatible with the locked data type, a propagation error results.

To avoid incompatible data type proposals, perform one of the following.

- Lock all objects in the group against changes by the Fixed-Point Tool.
- Unlock the object in the group with the locked data type.

The **ProposedDT** column of the Fixed-Point Tool displays **Locked** for all results that are locked against changes by the Fixed-Point Tool.

Part of a Shared Data Type Group is Out of Scope

When results that are in a shared data type group share a data type from outside the scope of the system under design, the Fixed-Point Tool is not able to propose a data type.

To get a data type proposal, perform one of the following.

- Ensure that objects inside the system under design do not share their output data type with an object outside the selected system. One way to ensure that objects inside your system under design do not share their data type with objects outside the system, is by inserting Data Type Conversion blocks at the system boundaries.
- Ensure that all objects that must share a data type are within the scope of the system under design.

Model Reference Blocks

Systems that share data types across model reference boundaries may get data type propagation errors.

To avoid data type propagation errors, consider the following.

- Do not use the same signal object across model reference boundaries.
- Insert Data Type Conversion blocks at model reference boundaries.

Block Constraints

Certain blocks have constraints on which data types it can support. For example, the Merge block requires that all inputs use the same data type.

- Certain blocks in the Communications System Toolbox, DSP System Toolbox, and Computer Vision System Toolbox libraries have data type constraints. The Fixed-Point Tool is not aware of this requirement and does not use it for automatic data typing. Therefore, the tool might propose a data type that is inconsistent with the block requirements. In this case, manually edit the proposed data type such that it complies with block constraints.

Visit the individual block reference pages for more information on these constraints.

Internal Block Rules

Sum Blocks

Sum blocks have both an output data type as well as an accumulator data type. Under certain conditions, when the accumulator data type is set to `Inherit: Inherit via internal rule`, a data type propagation error can result.

To get a compatible data type proposal, perform one of the following.

- Change the accumulator data type to something other than `Inherit: Inherit via internal rule` and repropose data types for your model to get compatible data type proposals.
- Lock the block against changes by the fixed-point tools.

More About

- “Models That Might Cause Data Type Propagation Errors” on page 39-10

Resolve Range Analysis Issues

The following table shows the different types of range analysis issues and the steps to resolve them.

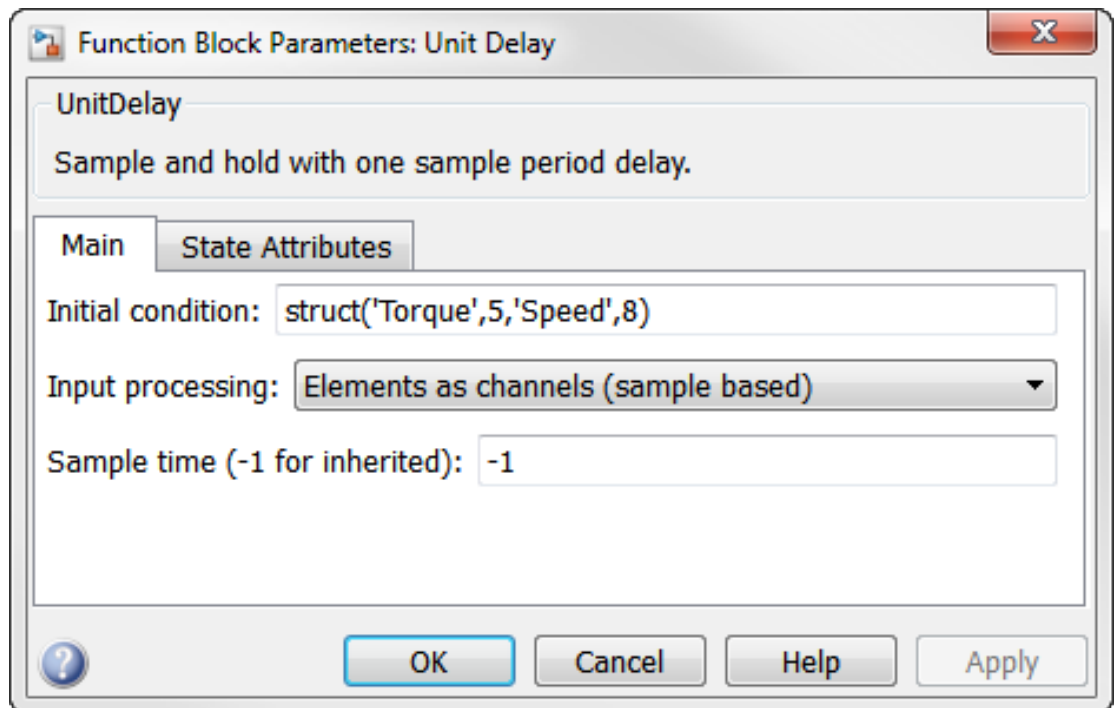
Analysis Results	Next Steps	For More Information
The analysis fails because the model contains blocks that it does not support. The Fixed-Point Tool generates an error.	Review the error message information and replace the unsupported blocks.	“Model Compatibility with Range Analysis” on page 40-6
The analysis cannot derive range data because the model contains conflicting design range information. The Fixed-Point Tool generates an error.	Examine the design ranges specified in the model to identify inconsistent design specifications and modify them to make them consistent.	“Fixing Design Range Conflicts” on page 40-31
The analysis cannot derive range data for an object because there is insufficient design range information specified on the model. The Fixed-Point Tool highlights the results for the object.	Examine the model to determine which design range information is missing.	“Troubleshoot Range Analysis of System Objects” on page 40-25

Data Type Mismatch and Structure Initial Conditions

Specify Bus Signal Initial Conditions Using Simulink.Parameter Objects

This example shows how to replace a structure initial condition with a `Simulink.Parameter` object. This approach allows the structure to maintain its tunability.

- 1 Double-click the Unit Delay block to view the block parameters. The Unit Delay block uses a structure initial condition.



- 2 Define a `Simulink.Parameter` object at the MATLAB command line. Set the data type of the parameter object to the bus object `SensorData`. Set the value of the parameter object to the specified structure. To maintain tunability, set the `StorageClass` property to `ExportedGlobal`.

```
P = Simulink.Parameter;
```

```
P.DataType = 'Bus: SensorData';  
P.Value = struct('Torque',5,'Speed',8);  
P.StorageClass = 'ExportedGlobal';
```

- 3 In the Unit Delay block dialog box, set **Initial condition** to P, the `Simulink.Parameter` object you defined. The structure defined in the `Simulink.Parameter` object remains tunable.

For more information on generating code for bus signals that use tunable initial condition structures, see “Control Signal and State Initialization in the Generated Code” (Simulink Coder).

Data Type Mismatch and Masked Atomic Subsystems

A data type mismatch occurs when a structure initial condition drives a bus signal that you specified using a masked atomic subsystem.

Change the subsystem to non atomic, or specify the structure parameter using a `Simulink.Parameter` object (as described in “Specify Bus Signal Initial Conditions Using `Simulink.Parameter` Objects” on page 46-31) to avoid the data type mismatch error.

Related Examples

- “Convert Model with Bus Object with Structure Initial Conditions to Fixed-Point” on page 42-23
- “Bus Objects in the Fixed-Point Workflow” on page 42-19

Compile-Time Recursion Limit Reached

Issue

You see a message such as:

```
Compile-time recursion limit reached. Size or type of input #1 of function 'foo'
may change at every call.
```

```
Compile-time recursion limit reached. Size of input #1 of function 'foo' may
change at every call.
```

```
Compile-time recursion limit reached. Value of input #1 of function 'foo' may
change at every call.
```

Cause

With compile-time recursion, the code generator produces multiple versions of the recursive function instead of producing a recursive function in the generated code. These versions are known as function specializations. The code generator is unable to use compile-time recursion for a recursive function in your MATLAB code because the number of function specializations exceeds the limit.

Solutions

To address the issue, try one of these solutions:

- “Force Run-Time Recursion” on page 46-33
- “Increase the Compile-Time Recursion Limit” on page 46-36

Force Run-Time Recursion

- For this message:

```
Compile-time recursion limit reached. Value of input #1 of function 'foo' may
change at every call.
```

Use this solution:

“Force Run-Time Recursion by Treating the Input Value as Nonconstant” on page 46-34.

- For this message:

```
Compile-time recursion limit reached. Size of input #1 of function 'foo' may
change at every call.
```

Use this solution:

“Force Run-Time Recursion by Making the Input Variable-Size” on page 46-35.

- For this message:

```
Compile-time recursion limit reached. Size or type of input #1 of function 'foo'
may change at every call.
```

In the , look at the function specializations. If you can see that the size of an argument is changing for each function specialization, then try this solution:

“Force Run-Time Recursion by Making the Input Variable-Size” on page 46-35.

Force Run-Time Recursion by Treating the Input Value as Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 100;
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

The second input to `recfcn` has the constant value 100. The code generator determines that the number of recursive calls is finite and tries to produce 100 copies of `recfcn`. This number of specializations exceeds the compile-time recursion limit. To force run-time recursion, instruct the code generator to treat the second input as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
```

```

A = ones(1,n);
x = coder.ignoreConst(100);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end

```

If the code generator cannot determine that the number of recursive calls is finite, it produces a run-time recursive function.

Force Run-Time Recursion by Making the Input Variable-Size

Consider this function:

```

function z = call_mysum(A)
%#codegen
z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+mysum(A(2:end));
end
end

```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. If `A` is large enough, the number of function specializations exceeds the compile-time limit. To cause the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```

function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);

```

```
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Increase the Compile-Time Recursion Limit

The default compile-time recursion limit of 50 is large enough for most recursive functions that require compile-time recursion. Usually, increasing the limit does not fix the issue. However, if you can determine the number of recursive calls and you want compile-time recursion, increase the limit. For example, consider this function:

```
function z = call_mysum()
%#codegen
B = 1:125;
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

You can determine that the code generator produces 125 copies of the `mysum` function. In this case, if you want compile-time recursion, increase the compile-time recursion limit to 125.

To increase the limit, in a code acceleration configuration object, increase the value of the `CompileTimeRecursionLimit` configuration parameter.

More About

- “Code Generation for Recursive Functions” on page 16-18

- “Set Up C Code Compilation Options” on page 14-25

Output Variable Must Be Assigned Before Run-Time Recursive Call

Issue

You see this error message:

```
All outputs must be assigned before any run-time recursive call. Output 'y' is not assigned here.
```

Cause

Run-time recursion produces a recursive function in the generated code. The code generator is unable to use run-time recursion for a recursive function in your MATLAB code because an output is not assigned before the first recursive call.

Solution

Rewrite the code so that it assigns the output before the recursive call.

Direct Recursion Example

In the following code, the statement `y = A(1)` assigns a value to the output `y`. This statement occurs after the recursive call `y = A(1) + mysum(A(2:end))`.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) > 1
    y = A(1) + mysum(A(2:end));

else
    y = A(1);
end
end
```

Rewrite the code so that assignment `y = A(1)` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');

if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Alternatively, before the if block, add an assignment, for example, $y = 0$.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
y = 0;
if size(A,2) > 1
    y = A(1)+ mysum(A(2:end));

else
    y = A(1);
end
end
```

Indirect Recursion Example

In the following code, `rec1` calls `rec2` before the assignment $y = 0$.

```
function y = rec1(x)
%#codegen

if x >= 0
    y = rec2(x-1)+1;
else
```

```
        y = 0;
    end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

Rewrite this code so that in `rec1`, the assignment `y = 0` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function y = rec1(x)
%#codegen

if x < 0
    y = 0;
else
    y = rec2(x-1)+1;
end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

More About

- “Code Generation for Recursive Functions” on page 16-18

Unable to Determine That Every Element of Cell Array Is Assigned

Issue

You see one of these messages:

```
Unable to determine that every element of 'y' is assigned before this line.
```

```
Unable to determine that every element of 'y' is assigned before exiting the function.
```

```
Unable to determine that every element of 'y' is assigned before exiting the recursively called function.
```

Cause

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array.

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. The code generator detects that all elements are assigned when the code follows this pattern:

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

Here is the pattern for a multidimensional cell array:

```
function z = mycell(m,n,p)
%#codegen
x = cell(m,n,p);
for i = 1:m
    for j =1:n
```

```
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

If the code generator detects that some elements are not assigned, code generation fails. Sometimes, even though your code assigns all elements of the cell array, code generation fails because the analysis does not detect that all elements are assigned.

Here are examples where the code generator is unable to detect that elements are assigned:

- Elements are assigned in different loops

```
...
x = cell(1,n)
for i = 1:5
    x{i} = 5;
end
for i = 6:n
    x{i} = 7;
end
...
```

- The variable that defines the loop end value is not the same as the variable that defines the cell dimension.

```
...
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
...
```

For more information, see “Definition of Variable-Size Cell Array by Using cell” on page 28-10.

Solution

Try one of these solutions:

- “Use recognized pattern for assigning elements” on page 46-43
- “Use repmat” on page 46-43
- “Use coder.nullcopy” on page 46-44

Use recognized pattern for assigning elements

If possible, rewrite your code to follow this pattern:

```
...
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
...
```

Use repmat

Sometimes, you can use `repmat` to define the variable-size cell array.

Consider this code that defines a variable-size cell array. It assigns the value 1 to odd elements and the value 2 to even elements.

```
function z = mycell12(n, j)
%#codegen
c = cell(1,n);
for i = 1:2:n-1
    c{i} = 1;
end
for i = 2:2:n
    c{i} = 2;
end
z = c{j};
```

Code generation does not allow this code because:

- More than one loop assigns the elements.
- The loop counter does not increment by 1.

Rewrite the code to first use `cell` to create a 1-by-2 cell array whose first element is 1 and whose second element is 2. Then, use `repmat` to create a variable-size cell array whose element values alternate between 1 and 2.

```
function z = mycell2(n, j)
%#codegen
c = cell(1,2);
c{1} = 1;
c{2} = 2;
c1= repmat(c,1,n);
z = c1{j};
end
```

Use `coder.nullcopy`

As a last resort, you can use `coder.nullcopy` to indicate that the code generator can allocate the memory for your cell array without initializing the memory. For example:

```
function z = mycell3(n, j)
%#codegen
c =cell(1,n);
c1 = coder.nullcopy(c);
for i = 1:4
    c1{i} = 1;
end
for i = 5:n
    c1{i} = 2;
end
z = c1{j};
end
```

Use `coder.nullcopy` with caution. If you access uninitialized memory, results are unpredictable.

See Also

`cell` | `coder.nullcopy` | `repmat`

More About

- “Cell Array Limitations for Code Generation” on page 28-9

Nonconstant Index into varargin or varargout in a for-Loop

Issue

Your MATLAB code contains a `for`-loop that indexes into `varargin` or `varargout`. When you generate code, you see this error message:

```
Non-constant expression or empty matrix. This expression must be constant because its value determines the size or class of some expression.
```

Cause

At code generation time, the code generator must be able to determine the value of an index into `varargin` or `varargout`. When `varargin` or `varargout` are indexed in a `for`-loop, the code generator determines the index value for each loop iteration by unrolling the loop. Loop unrolling makes a copy of the loop body for each loop iteration. In each iteration, the code generator determines the value of the index from the loop counter.

The code generator is unable to determine the value of an index into `varargin` or `varargout` when:

- The number of copies of the loop body exceeds the limit for loop unrolling.
- Heuristics fail to identify that loop unrolling is warranted for a particular `for`-loop. For example, consider the following function:

```
function [x,y,z] = fcn(a,b,c)
    %#codegen

    [x,y,z] = subfcn(a,b,c);

    function varargout = subfcn(varargin)
        j = 0;
        for i = 1:nargin
            j = j+1;
            varargout{j} = varargin{j};
        end
```

The heuristics do not detect the relationship between the index `j` and the loop counter `i`. Therefore, the code generator does not unroll the `for`-loop.

Solution

Use one of these solutions:

- “Force Loop Unrolling” on page 46-46
- “Rewrite the Code” on page 46-46

Force Loop Unrolling

Force loop unrolling by using `coder.unroll`. For example:

```
function [x,y,z] = fcn(a,b,c)
    %#codegen
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    j = 0;

    coder.unroll();
    for i = 1:nargin
        j = j + 1;
        varargout{j} = varargin{j};
    end
```

Rewrite the Code

Rewrite the code so that the code generator can detect the relationship between the index and the loop counter. For example:

```
function [x,y,z] = fcn(a,b,c)
    %#codegen
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    for i = 1:nargin
        varargout{i} = varargin{i};
    end
```

See Also

`coder.unroll`

More About

- “Code Generation for Variable Length Argument Lists” on page 19-2

- “Unroll `for-Loops`” on page 24-33

Single-Precision Conversion in Simulink

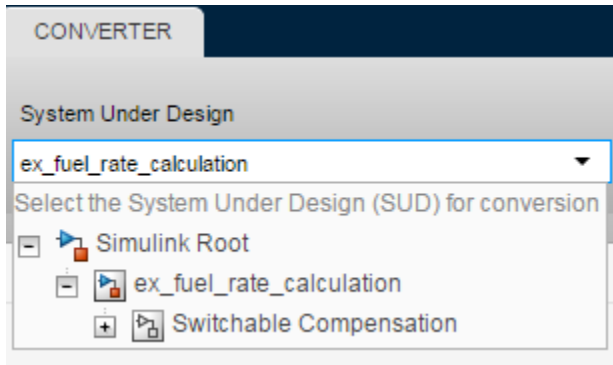
- “Getting Started with Single Precision Converter” on page 47-2
- “Convert a System to Single Precision” on page 47-5

Getting Started with Single Precision Converter

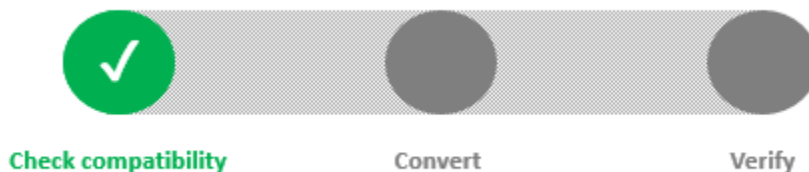
The Single Precision Converter converts your model or a system in your model from double precision to single precision. To open the Single Precision Converter, from the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**.

Select System Under Design

To begin, hover over the **System under design** drop-down list and select the system to convert to single precision.



Check Compatibility



The Single Precision Converter performs these checks:

- Check that all blocks in the selected system support single precision.

The Single Precision Converter displays a list of blocks that do not support single precision or are locked against changes by the fixed-point tools. To restart the conversion, replace the blocks that support only double precision and unlock the blocks that are locked against changes by the Fixed-Point Tools. Then click **Convert to Single**.

- Check that the system uses a library standard that supports single-precision designs.

To convert a system to single precision, the standard math library must be set to C99 (ISO). If the specified standard math library is not set to C99, the Single Precision Converter changes the math library.

- Check that the solver settings are set to fixed step.

Convert



Following the compatibility check, the Single Precision Converter converts the system to single-precision. The Converter makes these changes:

- Conversion of user-specified double-precision data types to single-precision data types (applies to block settings, Stateflow chart settings, signal objects, and bus objects).
- Output signals and intermediate settings using inherited data types that compile to double-precision change to single-precision data types.

The converter does not change Boolean, built-in integer, or user-specified fixed-point data types. When the conversion is finished, the converter displays a table summarizing the compiled and proposed data types of the objects in the system under design.

Verify



Finally, the Single Precision Converter verifies that the model containing the converted system can successfully update the diagram. If the model is not able to update the diagram due to data type mismatch errors at the system boundaries, the Single Precision Converter displays a message.

To resolve the data type mismatch, insert Data Type Conversion blocks at the system boundaries. You can also resolve the data type mismatch errors by changing the output data type of the blocks feeding into the system to single or `Inherit`: `Inherit via back propagation`.

Related Examples

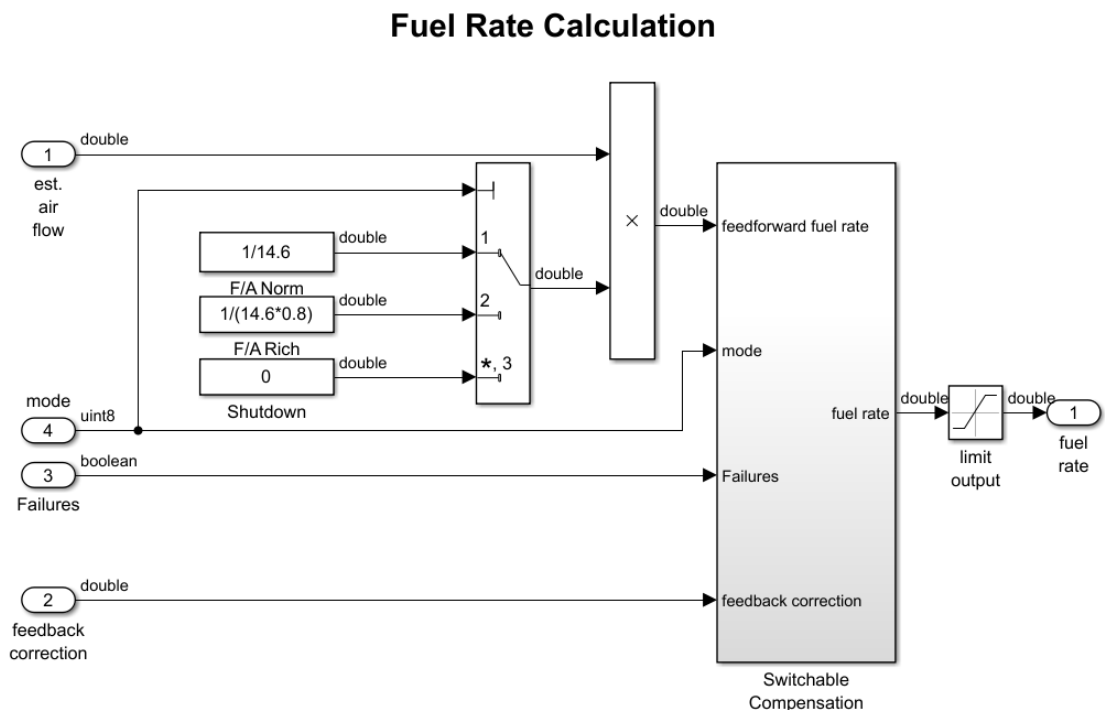
- “Convert a System to Single Precision” on page 47-5
- “Specify Single-Precision Data Type for Embedded Application” (Simulink Coder)

Convert a System to Single Precision

This example shows how to convert a system to single precision using the Single Precision Converter. This example converts all blocks in the model using double-precision data types to single-precision. To convert just a subsystem in a model to single precision, surround the subsystem under design with Data Type Conversion blocks before opening the Single Precision Converter.

- 1 Open the model. At the command line, enter

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
ex_fuel_rate_calculation
```

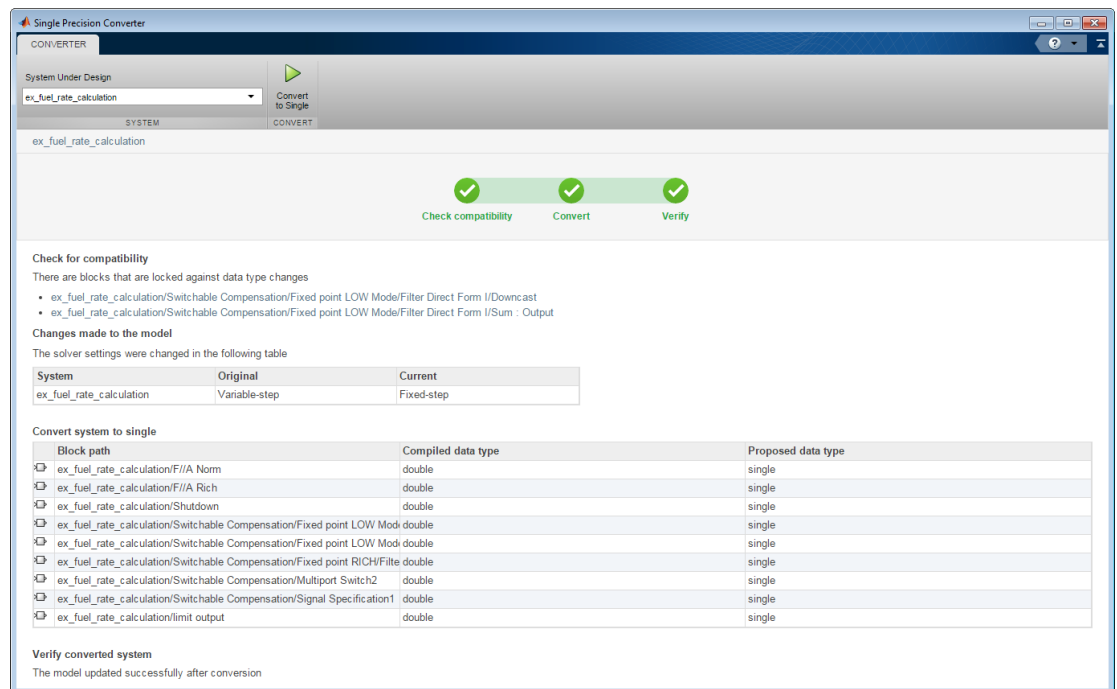


The model uses a combination of double-precision, Boolean, and built-in integer data types.

- 2 Open the Single Precision Converter. From the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**.
- 3 Under **System Under Design**, select the system or subsystem to convert to single precision. For this example, select the top-level model, `ex_fuel_rate_calculation`. Click **Convert to Single**.

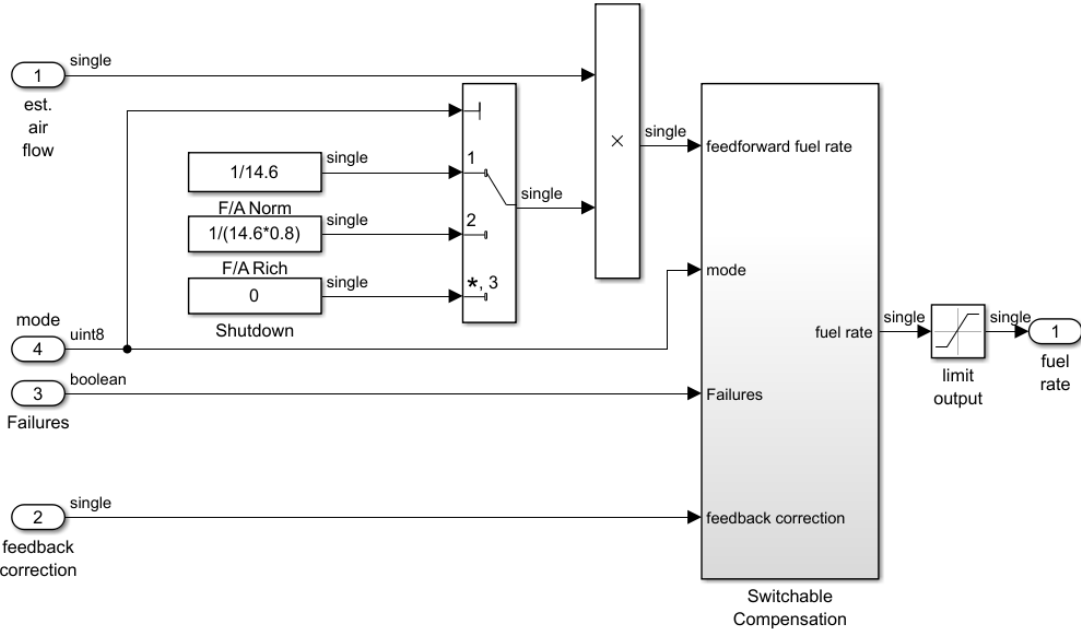
The converter first checks the system for compatibility with the conversion and changes any model settings that are incompatible. The standard math library of the model is now set to C99 (ISO), and the model now uses a fixed-step solver.

The converter then converts the system and lists all converted data types. The converter changes only double-precision data types. It does not convert Boolean, fixed-point, or built-in integer types to single precision. In the final stage of the conversion, the Converter verifies that the conversion was successful by updating the model.



- 4 Return to the model and update the diagram. The model no longer contains double-precision data types.

Fuel Rate Calculation



More About

- “Getting Started with Single Precision Converter” on page 47-2
- “Specify Single-Precision Data Type for Embedded Application” (Simulink Coder)

Writing Fixed-Point S-Functions

This appendix discusses the API for user-written fixed-point S-functions, which enables you to write Simulink C S-functions that directly handle fixed-point data types. Note that the API also provides support for standard floating-point and integer data types. You can find the files and examples associated with this API in the following locations:

- `matlabroot/simulink/include/`
- `matlabroot/toolbox/simulink/fixedandfloat/fixpdemos/`

Data Type Support

In this section...
“Supported Data Types” on page A-2
“The Treatment of Integers” on page A-3
“Data Type Override” on page A-3

Supported Data Types

The API for user-written fixed-point S-functions provides support for a variety of Simulink and Fixed-Point Designer data types, including

- Built-in Simulink data types
 - `single`
 - `double`
 - `uint8`
 - `int8`
 - `uint16`
 - `int16`
 - `uint32`
 - `int32`
- Fixed-point Simulink data types, such as
 - `sfix16_En15`
 - `ufix32_En16`
 - `ufix128`
 - `sfix37_S3_B5`
- Data types resulting from a data type override with `Scaled double`, such as
 - `flts16`
 - `flts16_En15`
 - `fltu32_S3_B5`

For more information, see “Fixed-Point Data Type and Scaling Notation” on page 32-17.

The Treatment of Integers

The API treats integers as fixed-point numbers with trivial scaling. In [Slope Bias] representation, fixed-point numbers are represented as $real\text{-}world\ value = (slope \times integer) + bias$.

In the trivial case, $slope = 1$ and $bias = 0$.

In terms of binary-point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real\text{-}world\ value = integer \times 2^{-fraction\ length} = integer \times 2^0.$$

In either case, trivial scaling means that the real-world value is equal to the stored integer value:

$$real\text{-}world\ value = integer.$$

All integers, including Simulink built-in integers such as `uint8`, are treated as fixed-point numbers with trivial scaling by this API. However, Simulink built-in integers are different in that their use does not cause a Fixed-Point Designer software license to be checked out.

Data Type Override

The Fixed-Point Tool enables you to perform various data type overrides on fixed-point signals in your simulations. This API can handle signals whose data types have been overridden in this way:

- A signal that has been overridden with `Single` is treated as a Simulink built-in `single`.
- A signal that has been overridden with `Double` is treated as a Simulink built-in `double`.
- A signal that has been overridden with `Scaled double` is treated as being of data type `ScaledDouble`.

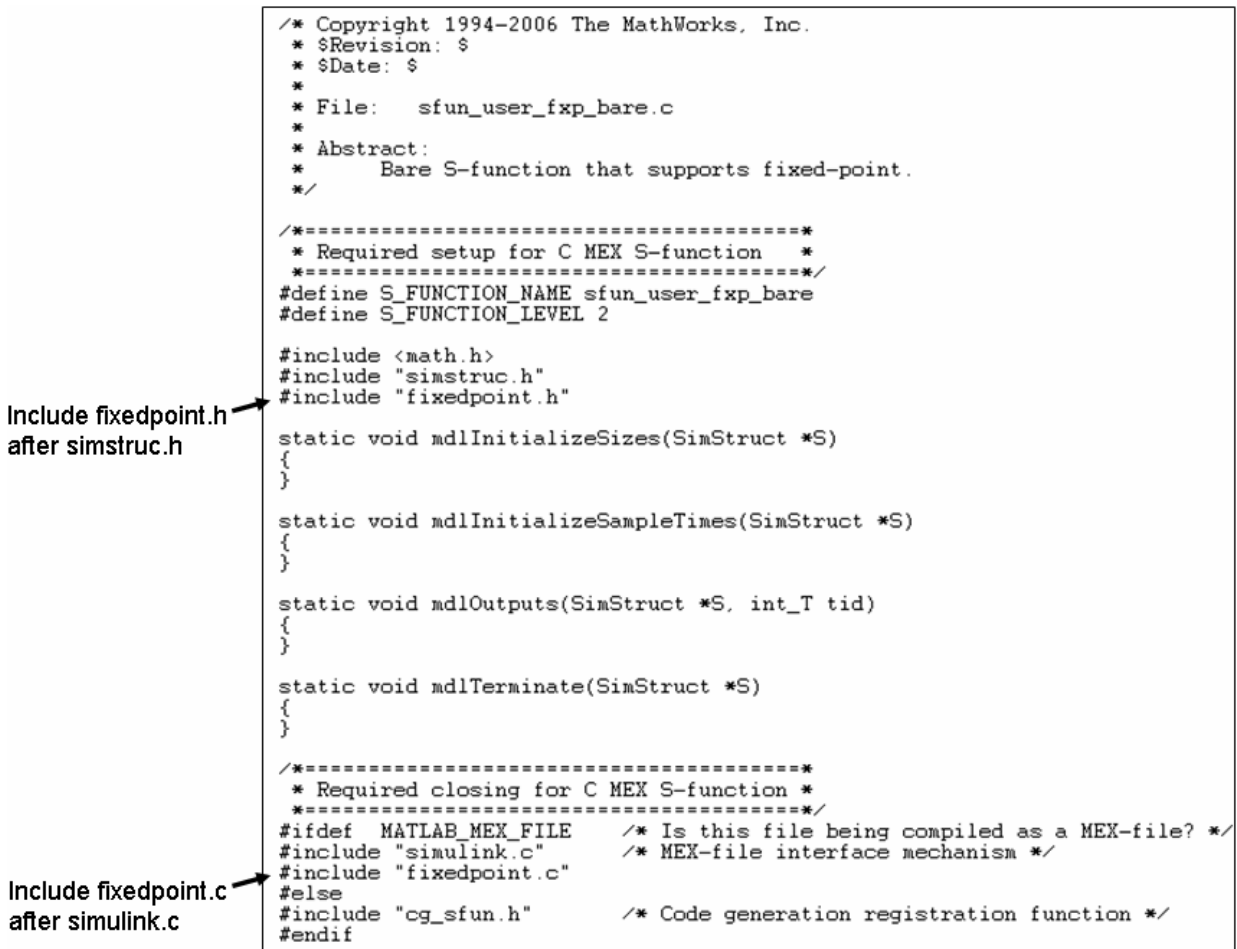
`ScaledDouble` signals are a hybrid between floating-point and fixed-point signals, in that they are stored as `doubles` with the scaling, sign, and word length information retained. The value is stored as a floating-point `double`, but as with a fixed-point number, the distinction between the stored integer value and the real-world value remains. The scaling information is applied to the stored integer `double` to obtain the real-world value. By storing the value in a `double`, overflow and precision issues are

almost always eliminated. Refer to any individual API function reference page at the end of this appendix to learn how that function treats `ScaledDouble` signals.

For more information about the Fixed-Point Tool and data type override, see “Fixed-Point Tool” on page 36-2 and the `fxptdlg` reference page.

Structure of the S-Function

The following diagram shows the basic structure of an S-function that directly handles fixed-point data types.



The callouts in the diagram alert you to the fact that you must include `fixedpoint.h` and `fixedpoint.c` at the appropriate places in the S-function. The other elements of the S-function displayed in the diagram follow the standard requirements for S-functions.

To learn how to create a MEX-file for your user-written fixed-point S-function, see “Create MEX-Files” on page A-20.

Storage Containers

In this section...

“Introduction” on page A-7

“Storage Containers in Simulation” on page A-7

“Storage Containers in Code Generation” on page A-10

Introduction

While coding with the API for user-written fixed-point S-functions, it is important to keep in mind the difference between storage container size, storage container word length, and signal word length. The sections that follow discuss the containers used by the API to store signals in simulation and code generation.

Storage Containers in Simulation

In simulation, signals are stored in one of several types of containers of a specific size.

Storage Container Categories

During simulation, fixed-point signals are held in one of the types of storage containers, as shown in the following table. In many cases, signals are represented in containers with more bits than their specified word length.

Fixed-Point Storage Containers

Container Category	Signal Word Length	Container Word Length	Container Size
FXP_STORAGE_INT8 (signed) FXP_STORAGE_UINT8 (unsigned)	1 to 8 bits	8 bits	1 byte
FXP_STORAGE_INT16 (signed) FXP_STORAGE_UINT16 (unsigned)	9 to 16 bits	16 bits	2 bytes
FXP_STORAGE_INT32 (signed) FXP_STORAGE_UINT32 (unsigned)	17 to 32 bits	32 bits	4 bytes
FXP_STORAGE_OTHER_SINGLE_WORD	33 to word length of long data type	Length of long data type	Length of long data type

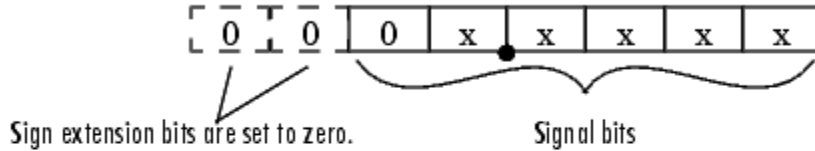
Container Category	Signal Word Length	Container Word Length	Container Size
FXP_STORAGE_MULTIWORD	Greater than the word length of long data type to 128 bits	Multiples of length of long data type to 128 bits	Multiples of length of long data type to 128 bits

When the number of bits in the signal word length is less than the size of the container, the word length bits are always stored in the least significant bits of the container. The remaining container bits must be sign extended:

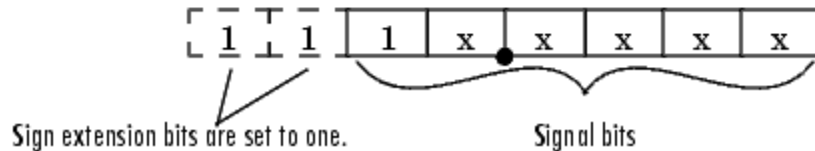
- If the data type is unsigned, the sign extension bits must be cleared to zero.
- If the data type is signed, the sign extension bits must be set to one for strictly negative numbers, and cleared to zero otherwise.

For example, a signal of data type `sfix6_En4` is held in a `FXP_STORAGE_INT8` container. The signal is held in the six least significant bits. The remaining two bits are set to zero when the signal is positive or zero, and to one when it is negative.

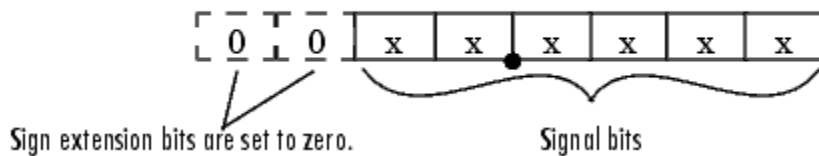
8-bit container for a signed, 6-bit signal that is positive or zero



8-bit container for a signed, 6-bit signal that is negative



A signal of data type `ufix6_En4` is held in a `FXP_STORAGE_UINT8` container. The signal is held in the six least significant bits. The remaining two bits are always cleared to zero.

8-bit container for an unsigned, 6-bit signal

The signal and storage container word lengths are returned by the `ssGetDataFxpWordLength` and `ssGetDataFxpContainWordLen` functions, respectively. The storage container size is returned by the `ssGetDataStorageContainerSize` function. The container category is returned by the `ssGetDataStorageContainCat` function, which in addition to those in the table above, can also return the following values.

Other Storage Containers

Container Category	Description
<code>FXP_STORAGE_UNKNOWN</code>	Returned if the storage container category is unknown
<code>FXP_STORAGE_SINGLE</code>	The container type for a Simulink single
<code>FXP_STORAGE_DOUBLE</code>	The container type for a Simulink double
<code>FXP_STORAGE_SCALEDDOUBLE</code>	The container type for a data type that has been overridden with Scaled double

Storage Containers in Simulation Example

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataStorageContainCat` returns `FXP_STORAGE_INT32`.
- `ssGetDataStorageContainerSize` or `sizeof()` returns 4, which is the storage container size in bytes.
- `ssGetDataFxpContainWordLen` returns 32, which is the storage container word length in bits.
- `ssGetDataFxpWordLength` returns 24, which is the data type word length in bits.

Storage Containers in Code Generation

The storage containers used by this API for code generation are not always the same as those used for simulation. During code generation, a native C data type is always used. Floating-point data types are held in C `double` or `float`. Fixed-point data types are held in C signed and unsigned `char`, `short`, `int`, or `long`.

Emulation

Because it is valuable for rapid prototyping and hardware-in-the-loop testing, the emulation of smaller signals inside larger containers is supported in code generation. For example, a 29-bit signal is supported in code generation if there is a C data type available that has at least 32 bits. The rules for placing a smaller signal into a larger container, and for dealing with the extra container bits, are the same in code generation as for simulation.

If a smaller signal is emulated inside a larger storage container in simulation, it is not necessarily emulated in code generation. For example, a 24-bit signal is emulated in a 32-bit storage container in simulation. However, some DSP chips have native support for 24-bit quantities. On such a target, the C compiler can define an `int` or a `long` to be exactly 24 bits. In this case, the 24-bit signal is held in a 32-bit container in simulation, and in a 24-bit container in code generation.

Conversely, a signal that was not emulated in simulation might need to be emulated in code generation. For example, some DSP chips have minimal support for integers. On such chips, `char`, `short`, `int`, and `long` might all be defined to 32 bits. In that case, it is necessary to emulate 8- and 16-bit fixed-point data types in code generation.

Storage Container TLC Functions

Since the mapping of storage containers in simulation to storage containers in code generation is not one-to-one, the Target Language Compiler (TLC) functions for storage containers are different from those in simulation:

- `FixPt_DataTypeNativeType`
- `FixPt_DataTypeStorageDouble`
- `FixPt_DataTypeStorageSingle`
- `FixPt_DataTypeStorageScaledDouble`
- `FixPt_DataTypeStorageSInt`
- `FixPt_DataTypeStorageUInt`

- `FixPt_DataTypeStorageSLong`
- `FixPt_DataTypeStorageULong`
- `FixPt_DataTypeStorageSShort`
- `FixPt_DataTypeStorageUShort`
- `FixPt_DataTypeStorageMultiword`

The first of these TLC functions, `FixPt_DataTypeNativeType`, is the closest analogue to `ssGetDataTypeStorageContainCat` in simulation. `FixPt_DataTypeNativeType` returns a TLC string that specifies the type of the storage container, and the Simulink Coder product automatically inserts a `typedef` that maps the string to a native C data type in the generated code.

For example, consider a fixed-data type that is held in `FXP_STORAGE_INT8` in simulation. `FixPt_DataTypeNativeType` will return `int8_T`. The `int8_T` will be `typedef'd` to a `char`, `short`, `int`, or `long` in the generated code, depending upon what is appropriate for the target compiler.

The remaining TLC functions listed above return `TRUE` or `FALSE` depending on whether a particular standard C data type is used to hold a given API-registered data type. Note that these functions do not necessarily give mutually exclusive answers for a given registered data type, due to the fact that C data types can potentially overlap in size. In C,

$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}.$$

One or more of these C data types can be, and very often are, the same size.

Data Type IDs

In this section...

“The Assignment of Data Type IDs” on page A-12

“Registering Data Types” on page A-13

“Setting and Getting Data Types” on page A-15

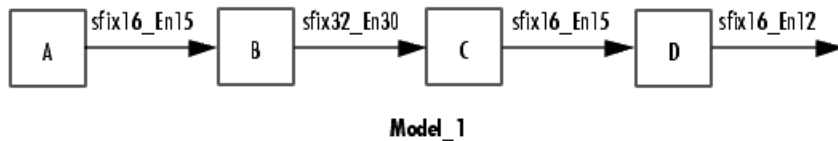
“Getting Information About Data Types” on page A-15

“Converting Data Types” on page A-17

The Assignment of Data Type IDs

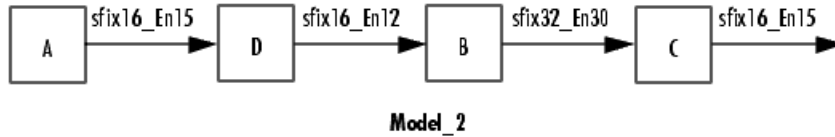
Each data type used in your S-function is assigned a data type ID. You should always use data type IDs to get and set information about data types in your S-function.

In general, the Simulink software assigns data type IDs during model initialization on a “first come, first served” basis. For example, consider the generalized schema of a block diagram below.



The Simulink software assigns a data type ID for each output data type in the diagram in the order it is requested. For simplicity, assume that the order of request occurs from left to right. Therefore, the output of block A may be assigned data type ID 13, and the output of block B may be assigned data type ID 14. The output data type of block C is the same as that of block A, so the data type ID assigned to the output of block C is also 13. The output of block D is assigned data type ID 15.

Now if the blocks in the model are rearranged,



The Simulink software still assigns the data type IDs in the order in which they are used. Therefore each data type might end up with a different data type ID. The output of block A is still assigned data type ID 13. The output of block D is now next in line and is assigned data type ID 14. The output of block B is assigned data type ID 15. The output data type of block C is still the same as that of block A, so it is also assigned data type ID 13.

This table summarizes the two cases described above.

Block	Data Type ID in Model_1	Data Type ID in Model_2
A	13	13
B	14	15
C	13	13
D	15	14

This example illustrates that there is no strict relationship between the attributes of a data type and the value of its data type ID. In other words, the data type ID is not assigned based on the characteristics of the data type it is representing, but rather on when that data type is first needed.

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function.

Registering Data Types

The functions in the following table are available in the API for user-written fixed-point S-functions for registering data types in simulation. Each of these functions will return a data type ID. To see an example of a function being used, go to the file and line indicated in the table.

Data Type Registration Functions

Function	Description	Example of Use
<code>ssRegisterDataTypeFxpBinaryPoint</code>	Register a fixed-point data type with binary-point-only scaling and return its data type ID	<code>sfun_user_fxp_asr.c</code> Line 252
<code>ssRegisterDataTypeFxpFSlopeFixExpBias</code>	Register a fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID	Not Available
<code>ssRegisterDataTypeFxpScaledDouble</code>	Register a scaled double data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID	Not Available
<code>ssRegisterDataTypeFxpSlopeBias</code>	Register a data type with [Slope Bias] scaling and return its data type ID	<code>sfun_user_fxp_dtprop.c</code> Line 319

Preassigned Data Type IDs

The Simulink software registers its built-in data types, and those data types always have preassigned data type IDs. The built-in data type IDs are given by the following tokens:

- `SS_DOUBLE`
- `SS_SINGLE`
- `SS_INT8`
- `SS_UINT8`
- `SS_INT16`
- `SS_UINT16`
- `SS_INT32`

- SS_UINT32
- SS_BOOLEAN

You do not need to register these data types. If you attempt to register a built-in data type, the registration function simply returns the preassigned data type ID.

Setting and Getting Data Types

Data type IDs are used to specify the data types of input and output ports, run-time parameters, and DWork states. To set fixed-point data types for quantities in your S-function, the procedure is as follows:

- 1 Register a data type using one of the functions listed in the table Data Type Registration Functions. A data type ID is returned to you.

Alternately, you can use one of the preassigned data type IDs of the Simulink built-in data types.

- 2 Use the data type ID to set the data type for an input or output port, run-time parameter, or DWork state using one of the following functions:

- ssSetInputPortDataType
- ssSetOutputPortDataType
- ssSetRunTimeParamInfo
- ssSetDWorkDataType

To get the data type ID of an input or output port, run-time parameter, or DWork state, use one of the following functions:

- ssGetInputPortDataType
- ssGetOutputPortDataType
- ssGetSFcnParamDataType or ssGetRunTimeParamInfo
- ssGetDWorkDataType

Getting Information About Data Types

You can use data type IDs with functions to get information about the built-in and registered data types in your S-function. The functions in the following tables are available in the API for extracting information about registered data types. To see an example of a function being used, go to the file and line indicated in the table. Note

that data type IDs can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Storage Container Information Functions

Function	Description	Example of Use
<code>ssGetDataTypeFxpContainWordLen</code>	Return the word length of the storage container of a registered data type	<code>sfun_user_fxp_ContainWordLenProbe.c</code> Line 181
<code>ssGetDataTypeStorageContainCat</code>	Return the storage container category of a registered data type	<code>sfun_user_fxp_asr.c</code> Line 294
<code>ssGetDataTypeStorageContainerSize</code>	Return the storage container size of a registered data type	<code>sfun_user_fxp_StorageContainSizeProbe.c</code> Line 171

Signal Data Type Information Functions

Function	Description	Example of Use
<code>ssGetDataTypeFxpIsSigned</code>	Determine whether a fixed-point registered data type is signed or unsigned	<code>sfun_user_fxp_asr.c</code> Line 254
<code>ssGetDataTypeFxpWordLength</code>	Return the word length of a fixed-point registered data type	<code>sfun_user_fxp_asr.c</code> Line 255
<code>ssGetDataTypeIsFixedPoint</code>	Determine whether a registered data type is a fixed-point data type	<code>sfun_user_fxp_const.c</code> Line 127
<code>ssGetDataTypeIsFloatingPoint</code>	Determine whether a registered data type is a floating-point data type	<code>sfun_user_fxp_IsFloatingPointProbe.c</code> Line 176
<code>ssGetDataTypeIsFxpFltApiComp</code>	Determine whether a registered data type is supported by the API for user-written fixed-point S-functions	<code>sfun_user_fxp_asr.c</code> Line 184
<code>ssGetDataTypeIsScalingPow2</code>	Determine whether a registered data type has power-of-two scaling	<code>sfun_user_fxp_asr.c</code> Line 203

Function	Description	Example of Use
<code>ssGetDataTypeInfoScalingTrivial</code>	Determine whether the scaling of a registered data type is slope = 1, bias = 0	<code>sfun_user_fxp_IsScalingTrivialProbe.c</code> Line 171

Signal Scaling Information Functions

Function	Description	Example of Use
<code>ssGetDataTypeInfoBias</code>	Return the bias of a registered data type	<code>sfun_user_fxp_dtprop.c</code> Line 243
<code>ssGetDataTypeInfoFixedExponent</code>	Return the exponent of the slope of a registered data type	<code>sfun_user_fxp_dtprop.c</code> Line 237
<code>ssGetDataTypeInfoFracSlope</code>	Return the fractional slope of a registered data type	<code>sfun_user_fxp_dtprop.c</code> Line 234
<code>ssGetDataTypeInfoFractionLength</code>	Return the fraction length of a registered data type with power-of-two scaling	<code>sfun_user_fxp_asr.c</code> Line 256
<code>ssGetDataTypeInfoTotalSlope</code>	Return the total slope of the scaling of a registered data type	<code>sfun_user_fxp_dtprop.c</code> Line 240

Converting Data Types

The functions in the following table allow you to convert values between registered data types in your fixed-point S-function.

Data Type Conversion Functions

Function	Description	Example of Use
<code>ssFxpConvert</code>	Convert a value from one data type to another data type.	Not Available
<code>ssFxpConvertFromRealWorldValue</code>	Convert a value of data type <code>double</code> to another data type.	Not Available
<code>ssFxpConvertToRealWorldValue</code>	Convert a value of any data type to a <code>double</code> .	Not Available

Overflow Handling and Rounding Methods

In this section...

“Tokens for Overflow Handling and Rounding Methods” on page A-18

“Overflow Logging Structure” on page A-19

Tokens for Overflow Handling and Rounding Methods

The API for user-written fixed-point S-functions provides functions for some mathematical operations, such as conversions. When these operations are performed, a loss of precision or overflow may occur. The tokens in the following tables allow you to control the way an API function handles precision loss and overflow. The data type of the overflow handling methods is `fxpModeOverflow`. The data type of the rounding modes is `fxpModeRounding`.

Overflow Handling Tokens

Token	Description
FXP_OVERFLOW_SATURATE	Saturate overflows
FXP_OVERFLOW_WRAP	Wrap overflows

Rounding Method Tokens

Token	Description
FXP_ROUND_CEIL	Round to the closest representable number in the direction of positive infinity
FXP_ROUND_CONVERGENT	Round toward nearest integer with ties rounding to nearest even integer
FXP_ROUND_FLOOR	Round to the closest representable number in the direction of negative infinity
FXP_ROUND_NEAR	Round to the closest representable number, with the exact midpoint rounded in the direction of positive infinity
FXP_ROUND_NEAR_ML	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers

Token	Description
FXP_ROUND_SIMPLEST	Automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible
FXP_ROUND_ZERO	Round to the closest representable number in the direction of zero

Overflow Logging Structure

Math functions of the API, such as `ssFxpConvert`, can encounter overflows when carrying out an operation. These functions provide a mechanism to log the occurrence of overflows and to report that log back to the caller.

You can use a fixed-point overflow logging structure in your S-function by defining a variable of data type `fxpOverflowLogs`. Some API functions, such as `ssFxpConvert`, accept a pointer to this structure as an argument. The function initializes the logging structure and maintains a count of each the following events that occur while the function is being performed:

- Overflows
- Saturations
- Divide-by-zeros

When a function that accepts a pointer to the logging structure is invoked, the function initializes the event counts of the structure to zero. The requested math operations are then carried out. Each time an event is detected, the appropriate event count is incremented by one.

The following fields contain the event-count information of the structure:

- `OverflowOccurred`
- `SaturationOccurred`
- `DivisionByZeroOccurred`

Create MEX-Files

To create a MEX-file for a user-written fixed-point C S-function on either Windows or UNIX[®] systems:

- In your S-function, include `fixedpoint.c` and `fixedpoint.h`. For more information, see “Structure of the S-Function” on page A-5.
- Pass an extra argument, `-lfixedpoint`, to the `mex` command. For example,

```
mex('sfun_user_fxp_asr.c', '-lfixedpoint')
```

Fixed-Point S-Function Examples

The following files in *matlabroot/toolbox/simulink/fixedandfloat/fixpdemos/* are examples of S-functions written with the API for user-written fixed-point S-functions:

- `sfun_user_fxp_asr.c`
- `sfun_user_fxp_BiasProbe.c`
- `sfun_user_fxp_const.c`
- `sfun_user_fxp_ContainWordLenProbe.c`
- `sfun_user_fxp_dtprop.c`
- `sfun_user_fxp_FixedExponentProbe.c`
- `sfun_user_fxp_FracLengthProbe.c`
- `sfun_user_fxp_FracSlopeProbe.c`
- `sfun_user_fxp_IsFixedPointProbe.c`
- `sfun_user_fxp_IsFloatingPointProbe.c`
- `sfun_user_fxp_IsFxpFltApiCompatProbe.c`
- `sfun_user_fxp_IsScalingPow2Probe.c`
- `sfun_user_fxp_IsScalingTrivialProbe.c`
- `sfun_user_fxp_IsSignedProbe.c`
- `sfun_user_fxp_prodsum.c`
- `sfun_user_fxp_StorageContainCatProbe.c`
- `sfun_user_fxp_StorageContainSizeProbe.c`
- `sfun_user_fxp_TotalSlopeProbe.c`
- `sfun_user_fxp_U32BitRegion.c`
- `sfun_user_fxp_WordLengthProbe.c`

Related Examples

- “Get the Input Port Data Type” on page A-23
- “Set the Output Port Data Type” on page A-25
- “Interpret an Input Value” on page A-26
- “Write an Output Value” on page A-28

- “Determine Output Type Using the Input Type” on page A-30

Get the Input Port Data Type

Within your S-function, you might need to know the data types of different ports, runtime parameters, and DWorks. In each case, you will need to get the data type ID of the data type, and then use functions from this API to extract information about the data type.

For example, suppose you need to know the data type of your input port. To do this,

- 1 Use `ssGetInputPortDataType`. The data type ID of the input port is returned.
- 2 Use API functions to extract information about the data type.

The following lines of example code are from `sfun_user_fxp_dtprop.c`.

In lines 191 and 192, `ssGetInputPortDataType` is used to get the data type ID for the two input ports of the S-function:

```
dataTypeIdU0 = ssGetInputPortDataType( S, 0 );
dataTypeIdU1 = ssGetInputPortDataType( S, 1 );
```

Further on in the file, the data type IDs are used with API functions to get information about the input port data types. In lines 205 through 226, a check is made to see whether the input port data types are `single` or `double`:

```
storageContainerU0 = ssGetDataTypeInfoStorageContainer( S,
dataTypeIdU0 );
storageContainerU1 = ssGetDataTypeInfoStorageContainer( S,
dataTypeIdU1 );
if ( storageContainerU0 == FXP_STORAGE_DOUBLE ||
storageContainerU1 == FXP_STORAGE_DOUBLE )
{
/* Doubles take priority over all other rules.
* If either of first two inputs is double,
* then third input is set to double.
*/
dataTypeIdU2Desired = SS_DOUBLE;
}
else if ( storageContainerU0 == FXP_STORAGE_SINGLE ||
storageContainerU1 == FXP_STORAGE_SINGLE )
{
/* Singles take priority over all other rules,
* except doubles.
* If either of first two inputs is single
```

```
* then third input is set to single.
*/
dataTypeIdU2Desired = SS_SINGLE;
}
else
```

In lines 227 through 244, additional API functions are used to get information about the data types if they are neither `single` nor `double`:

```
{
    isSignedU0 = ssGetDataTypeIdSigned( S, dataTypeIdU0 );
    isSignedU1 = ssGetDataTypeIdSigned( S, dataTypeIdU1 );

    wordLengthU0 = ssGetDataTypeIdWordLength( S, dataTypeIdU0 );
    wordLengthU1 = ssGetDataTypeIdWordLength( S, dataTypeIdU1 );

    fracSlopeU0 = ssGetDataTypeIdFracSlope( S, dataTypeIdU0 );
    fracSlopeU1 = ssGetDataTypeIdFracSlope( S, dataTypeIdU1 );

    fixedExponentU0 = ssGetDataTypeIdFixedExponent( S, dataTypeIdU0 );
    fixedExponentU1 = ssGetDataTypeIdFixedExponent( S, dataTypeIdU1 );

    totalSlopeU0 = ssGetDataTypeIdTotalSlope( S, dataTypeIdU0 );
    totalSlopeU1 = ssGetDataTypeIdTotalSlope( S, dataTypeIdU1 );

    biasU0 = ssGetDataTypeIdBias( S, dataTypeIdU0 );
    biasU1 = ssGetDataTypeIdBias( S, dataTypeIdU1 );
}
```

The functions used above return whether the data types are signed or unsigned, as well as their word lengths, fractional slopes, exponents, total slopes, and biases. Together, these quantities give full information about the fixed-point data types of the input ports.

Related Examples

- “Set the Output Port Data Type” on page A-25

Set the Output Port Data Type

You may want to set the data type of various ports, run-time parameters, or DWorks in your S-function.

For example, suppose you want to set the output port data type of your S-function. To do this,

- 1 Register a data type by using one of the functions listed in the table Data Type Registration Functions. A data type ID is returned.

Alternately, you can use one of the predefined data type IDs of the Simulink built-in data types.

- 2 Use `ssSetOutputPortDataType` with the data type ID from Step 1 to set the output port to the desired data type.

In the example below from lines 336 - 352 of `sfun_user_fxp_const.c`, `ssRegisterDataTypeFxpBinaryPoint` is used to register the data type. `ssSetOutputPortDataType` then sets the output data type either to the given data type ID, or to be dynamically typed:

```

/* Register data type
 */
if ( notSizesOnlyCall )
{
    DataTypeId DataTypeId = ssRegisterDataTypeFxpBinaryPoint(
        S,
        V_ISSIGNED,
        V_WORDLENGTH,
        V_FRACTIONLENGTH,
        1 /* true means obey data type override setting for
this subsystem */ );

    ssSetOutputPortDataType( S, 0, DataTypeId );
}
else
{
    ssSetOutputPortDataType( S, 0, DYNAMICALLY_TYPED );
}

```

Related Examples

- “Interpret an Input Value” on page A-26

Interpret an Input Value

Suppose you need to get the value of the signal on your input port to use in your S-function. You should write your code so that the pointer to the input value is properly typed, so that the values read from the input port are interpreted correctly. To do this, you can use these steps, which are shown in the example code below:

- 1** Create a void pointer to the value of the input signal.
- 2** Get the data type ID of the input port using `ssGetInputPortDataType`.
- 3** Use the data type ID to get the storage container type of the input.
- 4** Have a case for each input storage container type you want to handle. Within each case, you will need to perform the following in some way:
 - Create a pointer of the correct type according to the storage container, and cast the original void pointer into the new fully typed pointer (see **a** and **c**).
 - You can now store and use the value by dereferencing the new, fully typed pointer (see **b** and **d**).

For example,

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    const void *pVoidIn =
        (const void *)ssGetInputPortSignal( S, 0 ); (1)

    DTypeId dataTypeIdU0 = ssGetInputPortDataType( S, 0 ); (2)

    fxpStorageContainerCategory storageContainerU0 =
        ssGetDataTypeIdStorageContainerCat( S, dataTypeIdU0 ); (3)

    switch ( storageContainerU0 )
    {
        case FXP_STORAGE_UINT8: (4)
        {
            const uint8_T *pU8_Properly_Typed_Pointer_To_U0; (a)

            uint8_T u8_Stored_Integer_U0; (b)

            pU8_Properly_Typed_Pointer_To_U0 =
                (const uint8_T *)pVoidIn; (c)
        }
    }
}
```

```
        u8_Stored_Integer_U0 =
        *pU8_Properly_Typed_Pointer_To_U0; (d)

        <snip: code that uses input when it's in a uint8_T>
    }
    break;

case FXP_STORAGE_INT8: (4)
{
    const int8_T *pS8_Properly_Typed_Pointer_To_U0; (a)

    int8_T s8_Stored_Integer_U0; (b)

    pS8_Properly_Typed_Pointer_To_U0 =
    (const int8_T *)pVoidIn; (c)

    s8_Stored_Integer_U0 =
    *pS8_Properly_Typed_Pointer_To_U0; (d)

    <snip: code that uses input when it's in a int8_T>
}
break;
```

Related Examples

- “Write an Output Value” on page A-28

Write an Output Value

Suppose you need to write the value of the output signal to the output port in your S-function. You should write your code so that the pointer to the output value is properly typed. To do this, you can use these steps, which are followed in the example code below:

- 1 Create a void pointer to the value of the output signal.
- 2 Get the data type ID of the output port using `ssGetOutputPortDataType`.
- 3 Use the data type ID to get the storage container type of the output.
- 4 Have a case for each output storage container type you want to handle. Within each case, you will need to perform the following in some way:
 - Create a pointer of the correct type according to the storage container, and cast the original void pointer into the new fully typed pointer (see **a** and **c**).
 - You can now write the value by dereferencing the new, fully typed pointer (see **b** and **d**).

For example,

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    <snip>

    void *pVoidOut = ssGetOutputPortSignal( S, 0 ); (1)

    DTypeId dataTypeIdY0 = ssGetOutputPortDataType( S, 0 ); (2)

    fxpStorageContainerCategory storageContainerY0 =
        ssGetDataTypeStorageContainCat( S,
        dataTypeIdY0 ); (3)

    switch ( storageContainerY0 )
    {
        case FXP_STORAGE_UINT8: (4)
        {
            const uint8_T *pU8_Properly_Typed_Pointer_To_Y0; (a)

            uint8_T u8_Stored_Integer_Y0; (b)

            <snip: code that puts the desired output stored integer
            value in to temporary variable u8_Stored_Integer_Y0>

            pU8_Properly_Typed_Pointer_To_Y0 =
            (const uint8_T *)pVoidOut; (c)

            *pU8_Properly_Typed_Pointer_To_Y0 =
            u8_Stored_Integer_Y0; (d)
        }
    }
}
```

```
    }  
    break;  
  
    case FXP_STORAGE_INT8: (4)  
    {  
        const int8_T *pS8_Properly_Typed_Pointer_To_Y0; (a)  
  
        int8_T s8_Stored_Integer_Y0; (b)  
  
        <snip: code that puts the desired output stored integer  
value in to temporary variable s8_Stored_Integer_Y0>  
  
        pS8_Properly_Typed_Pointer_To_Y0 =  
(const int8_T *)pVoidY0; (c)  
  
        *pS8_Properly_Typed_Pointer_To_Y0 =  
s8_Stored_Integer_Y0; (d)  
  
    }  
    break;  
  
<snip>
```

Related Examples

- “Determine Output Type Using the Input Type” on page A-30

Determine Output Type Using the Input Type

The following sample code from lines 243 through 261 of `sfun_user_fxp_asr.c` gives an example of using the data type of the input to your S-function to calculate the output data type. Notice that in this code

- The output is signed or unsigned to match the input **(a)**.
- The output is the same word length as the input **(b)**.
- The fraction length of the output depends on the input fraction length and the number of shifts **(c)**.

```
#define MDL_SET_INPUT_PORT_DATA_TYPE
static void mdlSetInputPortDataType(SimStruct *S, int port,
    DTypeId dataTypeIdInput)
{
    if ( isDataTypeSupported( S, dataTypeIdInput ) )
    {
        DTypeId dataTypeIdOutput;

        ssSetInputPortDataType( S, port, dataTypeIdInput );

        dataTypeIdOutput = ssRegisterDataTypeFxpBinaryPoint(
            S,
            ssGetDataTypeFxpIsSigned( S, dataTypeIdInput ), (a)
            ssGetDataTypeFxpWordLength( S, dataTypeIdInput ), (b)
            ssGetDataTypeFractionLength( S, dataTypeIdInput )
            - V_NUM_BITS_TO_SHIFT_RGHT, (c)
            0 /* false means do NOT obey data type override
               setting for this subsystem */ );

        ssSetOutputPortDataType( S, 0, dataTypeIdOutput );
    }
}
```

API Function Reference

ssFxpConvert

Convert value from one data type to another

Syntax

```
extern void ssFxpConvert (SimStruct *S,  
                          void *pVoidDest,  
                          size_t sizeofDest,  
                          DTypeId dataTypeIdDest,  
                          const void *pVoidSrc,  
                          size_t sizeofSrc,  
                          DTypeId dataTypeIdSrc,  
                          fxpModeRounding roundMode,  
                          fxpModeOverflow overflowMode,  
                          fxpOverflowLogs *pFxpOverflowLogs)
```

Arguments

S

SimStruct representing an S-function block.

pVoidDest

Pointer to the converted value.

sizeofDest

Size in memory of the converted value.

dataTypeIdDest

Data type ID of the converted value.

pVoidSrc

Pointer to the value you want to convert.

sizeofSrc

Size in memory of the value you want to convert.

dataTypeIdSrc

Data type ID of the value you want to convert.

roundMode

Rounding mode you want to use if a loss of precision is necessary during the conversion. Possible values are `FXP_ROUND_CEIL`, `FXP_ROUND_CONVERGENT`, `FXP_ROUND_FLOOR`, `FXP_ROUND_NEAR`, `FXP_ROUND_NEAR_ML`, `FXP_ROUND_SIMPLEST` and `FXP_ROUND_ZERO`.

overflowMode

Overflow mode you want to use if overflow occurs during the conversion. Possible values are `FXP_OVERFLOW_SATURATE` and `FXP_OVERFLOW_WRAP`.

pFxpOverflowLogs

Pointer to the fixed-point overflow logging structure.

Description

This function converts a value of any registered built-in or fixed-point data type to any other registered built-in or fixed-point data type.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvertFromRealWorldValue`, `ssFxpConvertToRealWorldValue`

Introduced before R2006a

ssFxpConvertFromRealWorldValue

Convert value of data type `double` to another data type

Syntax

```
extern void ssFxpConvertFromRealWorldValue
    (SimStruct *S,
     void *pVoidDest,
     size_t sizeofDest,
     DTypeId dataTypeIdDest,
     double dblRealWorldValue,
     fxpModeRounding roundMode,
     fxpModeOverflow overflowMode,
     fxpOverflowLogs *pFxpOverflowLogs)
```

Arguments

`S`

SimStruct representing an S-function block.

`pVoidDest`

Pointer to the converted value.

`sizeofDest`

Size in memory of the converted value.

`dataTypeIdDest`

Data type ID of the converted value.

`dblRealWorldValue`

Double value you want to convert.

`roundMode`

Rounding mode you want to use if a loss of precision is necessary during the conversion. Possible values are `FXP_ROUND_CEIL`, `FXP_ROUND_CONVERGENT`, `FXP_ROUND_FLOOR`, `FXP_ROUND_NEAR`, `FXP_ROUND_NEAR_ML`, `FXP_ROUND_SIMPLEST` and `FXP_ROUND_ZERO`.

`overflowMode`

Overflow mode you want to use if overflow occurs during the conversion. Possible values are `FXP_OVERFLOW_SATURATE` and `FXP_OVERFLOW_WRAP`.

`pFxpOverflowLogs`

Pointer to the fixed-point overflow logging structure.

Description

This function converts a `double` value to any registered built-in or fixed-point data type.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertToRealWorldValue`

Introduced before R2006a

ssFxpConvertToRealWorldValue

Convert value of any data type to double

Syntax

```
extern double ssFxpConvertToRealWorldValue (SimStruct *S,  
                                             const void *pVoidSrc,  
                                             size_t sizeofSrc,  
                                             DTypeId dataTypeIdSrc)
```

Arguments

S

SimStruct representing an S-function block.

pVoidSrc

Pointer to the value you want to convert.

sizeofSrc

Size in memory of the value you want to convert.

dataTypeIdSrc

Data type ID of the value you want to convert.

Description

This function converts a value of any registered built-in or fixed-point data type to a double.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertFromRealWorldValue`

Introduced before R2006a

ssFxpGetU32BitRegion

Return stored integer value for 32-bit region of real, scalar signal element

Syntax

```
extern uint32 ssFxpGetU32BitRegion(SimStruct *S,  
                                   const void *pVoid  
                                   DTypeId dataTypeId  
                                   unsigned int regionIndex)
```

Arguments

S

SimStruct representing an S-function block.

pVoid

Pointer to the storage container of the real, scalar signal element in which the 32-bit region of interest resides.

dataTypeId

Data type ID of the registered data type corresponding to the signal.

regionIndex

Index of the 32-bit region whose stored integer value you want to retrieve, where 0 accesses the least significant 32-bit region.

Description

This function returns the stored integer value in the 32-bit region specified by **regionIndex**, associated with the fixed-point data type designated by **dataTypeId**. You can use this function with any fixed-point data type, including those with word sizes less than 32 bits. If the fixed-point word size is less than 32 bits, the remaining bits are sign extended.

This function generates an error if **dataTypeId** represents a floating-point data type.

To view an example model whose S-functions use the `ssFxpGetU32BitRegion` function, at the MATLAB prompt, enter `fxpdemo_sfun_user_U32BitRegion`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

See Also

`ssFxpSetU32BitRegion`

Introduced in R2007b

ssFxpGetU32BitRegionCompliant

Determine whether S-function is compliant with the U32 bit region interface

Syntax

```
extern ssFxpSGetU32BitRegionCompliant(SimStruct *S,  
                                     int *result)
```

Arguments

S

SimStruct representing an S-function block.

result

- 1 if S-function calls `ssFxpSetU32BitRegionCompliant` to declare compliance with memory footprint for fixed-point data types with 33 or more bits
- 0 if S-function does not call `ssFxpSetU32BitRegionCompliant`

Description

This function checks whether the S-function calls `ssFxpSetU32BitRegionCompliant` to declare compliance with the memory footprint for fixed-point data types with 33 or more bits. Before calling any other Fixed-Point Designer API function on data with 33 or more bits, you must call `ssFxpSetU32BitRegionCompliant` as follows:

```
ssFxpSetU32BitRegionCompliant(S,1);
```

Note: The Fixed-Point Designer software assumes that S-functions that use fixed-point data types with 33 or more bits without calling `ssFxpSetU32BitRegionCompliant` are using the obsolete memory footprint that existed until R2007b. Either redesign these S-functions or isolate them using the library `fixpt_legacy_sfun_support`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

See Also

`ssFxpSetU32BitRegionCompliant`

Introduced in R2009a

ssFxpSetU32BitRegion

Set stored integer value for 32-bit region of real, scalar signal element

Syntax

```
extern ssFxpSetU32BitRegion(SimStruct *S,  
                             void *pVoid  
                             DTypeId dataTypeId  
                             uint32 regionValue  
                             unsigned int regionIndex)
```

Arguments

S

SimStruct representing an S-function block.

pVoid

Pointer to the storage container of the real, scalar signal element in which the 32-bit region of interest resides.

dataTypeId

Data type ID of the registered data type corresponding to the signal.

regionValue

Stored integer value that you want to assign to a 32-bit region.

regionIndex

Index of the 32-bit region whose stored integer value you want to set, where 0 accesses the least significant 32-bit region.

Description

This function sets `regionValue` as the stored integer value of the 32-bit region specified by `regionIndex`, associated with the fixed-point data type designated by `dataTypeId`. You can use this function with any fixed-point data type, including those with word sizes less than 32 bits. If the fixed-point word size is less than 32 bits, ensure that the remaining bits are sign extended.

This function generates an error if `dataTypeId` represents a floating-point data type, or if the stored integer value that you set is invalid.

To view an example model whose S-functions use the `ssFxpSetU32BitRegion` function, at the MATLAB prompt, enter `fxpdemo_sfun_user_U32BitRegion`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

See Also

`ssFxpGetU32BitRegion`

Introduced in R2007b

ssFxpSetU32BitRegionCompliant

Declare compliance with the U32 bit region interface for fixed-point data types with 33 or more bits

Syntax

```
extern ssFxpSetU32BitRegionCompliant(SimStruct *S,  
                                     int Value)
```

Arguments

S

SimStruct representing an S-function block.

Value

- 1 declare compliance with memory footprint for fixed-point data types with 33 or more bits.

Description

This function declares compliance with the Fixed-Point Designer bit region interface for data types with 33 or more bits. The memory footprint for data types with 33 or more bits varies between MATLAB host platforms and might change between software releases. To make an S-function robust to memory footprint changes, use the U32 bit region interface. You can use identical source code on different MATLAB host platforms and with any software release from R2008b. If the memory footprint changes between releases, you do not have to recompile U32 bit region compliant S-functions.

To make an S-function U32 bit region compliant, before calling any other Fixed-Point Designer API function on data with 33 or more bits, you must call this function as follows:

```
ssFxpSetU32BitRegionCompliant(S,1);
```

If an S-function block contains a fixed-point data type with 33 or more bits, call this function in mdlInitializeSizes().

Note: The Fixed-Point Designer software assumes that S-functions that use fixed-point data types with 33 or more bits without calling `ssFxpSetU32BitRegionCompliant` are using the obsolete memory footprint that existed until R2007b. Either redesign these S-functions or isolate them using the library `fixpt_legacy_sfun_support`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

See Also

`ssFxpGetU32BitRegionCompliant`

Introduced in R2009a

ssGetDataTypeBias

Return bias of registered data type

Syntax

```
extern double ssGetDataTypeBias(SimStruct *S, DTypeId  
                               dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the bias.

Description

Fixed-point numbers can be represented as
real-world value = (slope × integer) + bias.

This function returns the bias of a registered data type:

- For both trivial scaling and power-of-two scaling, 0 is returned.
- If the registered data type is `ScaledDouble`, the bias returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeInfo`

See Also

`ssGetDataTypeInfoFixedExponent`, `ssGetDataTypeInfoFracSlope`,
`ssGetDataTypeInfoTotalSlope`

Introduced before R2006a

ssGetDataTypeIdFixedExponent

Return exponent of slope of registered data type

Syntax

```
extern int ssGetDataTypeIdFixedExponent (SimStruct *S, DTypeId
                                         dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the exponent.

Description

Fixed-point numbers can be represented as
real-world value = (slope × integer) + bias,

where the slope can be expressed as

slope = fractional slope × 2^{exponent}.

This function returns the exponent of a registered fixed-point data type:

- For power-of-two scaling, the exponent is the negative of the fraction length.
- If the data type has trivial scaling, including for data types `single` and `double`, the exponent is 0.
- If the registered data type is `ScaledDouble`, the exponent returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIdIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeInfoFixedExponent`

See Also

`ssGetDataTypeInfoBias`, `ssGetDataTypeInfoFracSlope`, `ssGetDataTypeInfoTotalSlope`

Introduced before R2006a

ssGetDataTypeFracSlope

Return fractional slope of registered data type

Syntax

```
extern double ssGetDataTypeFracSlope(SimStruct *S, DTypeId
                                     dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the fractional slope.

Description

Fixed-point numbers can be represented as
real-world value = (slope × integer) + bias,

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}.$$

This function returns the fractional slope of a registered fixed-point data type. To get the total slope, use `ssGetDataTypeTotalSlope`:

- For power-of-two scaling, the fractional slope is 1.
- If the data type has trivial scaling, including data types `single` and `double`, the fractional slope is 1.
- If the registered data type is `ScaledDouble`, the fractional slope returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeFracSlope`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFixedExponent`, `ssGetDataTypeTotalSlope`

Introduced before R2006a

ssGetDataTypeInfoLength

Return fraction length of registered data type with power-of-two scaling

Syntax

```
extern int ssGetDataTypeInfoLength (SimStruct *S, DTypeInfo  
dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the fraction length.

Description

This function returns the fraction length, or the number of bits to the right of the binary point, of the data type designated by `dataTypeId`.

This function errors out when `ssGetDataTypeInfoIsScalingPow2` returns `FALSE`.

This function also errors out when `ssGetDataTypeInfoIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeInfoFractionLength`

See Also

`ssGetDataTypeInfoExpWordLength`

Introduced before R2006a

ssGetDataTypeFxpContainWordLen

Return word length of storage container of registered data type

Syntax

```
extern int ssGetDataTypeFxpContainWordLen (SimStruct *S,  
                                           DTypeId dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the container word length.

Description

This function returns the word length, in bits, of the storage container of the fixed-point data type designated by `dataTypeId`. This function does not return the size of the storage container or the word length of the data type. To get the storage container size, use `ssGetDataTypeStorageContainerSize`. To get the data type word length, use `ssGetDataTypeFxpWordLength`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeFxpContainWordLen` returns 32, which is the storage container word length in bits.
- `ssGetDataTypeFxpWordLength` returns 24, which is the data type word length in bits.
- `ssGetDataTypeStorageContainerSize` or `sizeof()` returns 4, which is the storage container size in bytes.

TLC Functions

`FixPt_DataTypeFxpContainWordLen`

See Also

`ssGetDataTypeFxpWordLength`, `ssGetDataTypeStorageContainCat`,
`ssGetDataTypeStorageContainerSize`

Introduced before R2006a

ssGetDataTypeFxpIsSigned

Determine whether fixed-point registered data type is signed or unsigned

Syntax

```
extern int ssGetDataTypeFxpIsSigned (SimStruct *S, DTypeId  
                                     dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered fixed-point data type for which you want to know whether it is signed.

Description

This function determines whether a registered fixed-point data type is signed:

- If the fixed-point data type is signed, the function returns **TRUE**. If the fixed-point data type is unsigned, the function returns **FALSE**.
- If the registered data type is **ScaledDouble**, the function returns **TRUE** or **FALSE** according to the signedness of the nonoverridden data type.
- If the registered data type is **single** or **double**, this function errors out.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns **FALSE**.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

FixPt_DataTypeInfoIsSigned

Introduced before R2006a

ssGetDataTypeIdWordLength

Return word length of fixed-point registered data type

Syntax

```
extern int ssGetDataTypeIdWordLength (SimStruct *S, DTypeId  
                                     dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered fixed-point data type for which you want to know the word length.

Description

This function returns the word length of the fixed-point data type designated by `dataTypeId`. This function does not return the word length of the container of the data type. To get the container word length, use `ssGetDataTypeIdContainWordLen`:

- If the registered data type is fixed point, this function returns the total word length including any sign bits, integer bits, and fractional bits.
- If the registered data type is `ScaledDouble`, this function returns the word length of the nonoverridden data type.
- If registered data type is `single` or `double`, this function errors out.

This function errors out when `ssGetDataTypeIdIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

Examples

An `sfixed24_en10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataFxpWordLength` returns 24, which is the data type word length in bits.
- `ssGetDataFxpContainWordLen` returns 32, which is the storage container word length in bits.
- `ssGetDataStorageContainerSize` or `sizeof()` returns 4, which is the storage container size in bytes.

TLC Functions

`FixPt_DataTypeFxpWordLength`

See Also

`ssGetDataFxpContainWordLen`, `ssGetDataFractionLength`,
`ssGetDataStorageContainerSize`

Introduced before R2006a

ssGetDataTypesFixedPoint

Determine whether registered data type is fixed-point data type

Syntax

```
extern int ssGetDataTypeIdIsFixedPoint(SimStruct *S, DTypeId  
                                       dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know whether it is fixed-point.

Description

This function determines whether a registered data type is a fixed-point data type:

- This function returns **TRUE** if the registered data type is fixed-point, and **FALSE** otherwise.
- If the registered data type is a pure Simulink integer, such as `int8`, this function returns **TRUE**.
- If the registered data type is `ScaledDouble`, this function returns **FALSE**.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsFixedPoint`

See Also

`ssGetDataTypeIsFloatingPoint`

Introduced before R2006a

ssGetDataTypesFloatingPoint

Determine whether registered data type is floating-point data type

Syntax

```
extern int ssGetDataTypeIsFloatingPoint (SimStruct *S, DTypeId  
                                         dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know whether it is floating-point.

Description

This function determines whether a registered data type is **single** or **double**:

- If the registered data type is either **single** or **double**, this function returns **TRUE**, and **FALSE** is returned otherwise.
- If the registered data type is **ScaledDouble**, this function returns **FALSE**.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsFloatingPoint`

See Also

`ssGetDataTypeIsFixedPoint`

Introduced before R2006a

ssGetDataTypesFxpFltApiCompat

Determine whether registered data type is supported by API for user-written fixed-point S-functions

Syntax

```
extern int ssGetDataTypeIsFxpFltApiCompat(SimStruct *S, DTypeId  
                                         dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to determine compatibility with the API for user-written fixed-point S-functions.

Description

This function determines whether the registered data type is supported by the API for user-written fixed-point S-functions. The supported data types are all standard Simulink data types, all fixed-point data types, and data type override data types.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None. Checking for API-compatible data types is done in simulation. Checking for API-compatible data types is not supported in TLC.

Introduced before R2006a

ssGetDataTypesScalingPow2

Determine whether registered data type has power-of-two scaling

Syntax

```
extern int ssGetDataTypeIdScalingPow2 (SimStruct *S, DTypeId
                                       dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know whether the scaling is strictly power-of-two.

Description

This function determines whether the registered data type is scaled strictly by a power of two. Fixed-point numbers can be represented as *real-world value* = (*slope* × *integer*) + *bias*,

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}.$$

When *bias* = 0 and *fractional slope* = 1, the only scaling factor that remains is a power of two:

$$\text{real-world value} = (2^{\text{exponent}} \times \text{integer}) = (2^{\text{fraction length}} \times \text{integer}).$$

Trivial scaling is considered a case of power-of-two scaling, with the exponent being equal to zero.

Note Many fixed-point algorithms are designed to accept only power-of-two scaling. For these algorithms, you can call `ssGetDataTypeIdScalingPow2` in

`mdlSetInputPortDataType` and `mdlSetOutputPortDataType`, to prevent unsupported data types from being accepted.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsScalingPow2`

See Also

`ssGetDataTypeIsScalingTrivial`

Introduced before R2006a

ssGetDataTypesScalingTrivial

Determine whether scaling of registered data type is slope = 1, bias = 0

Syntax

```
extern int ssGetDataTypeIsScalingTrivial (SimStruct *S, DTypeId
                                         dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know whether the scaling is trivial.

Description

This function determines whether the scaling of a registered data type is trivial. In [Slope Bias] representation, fixed-point numbers can be represented as *real-world value = (slope × integer) + bias*.

In the trivial case, *slope = 1* and *bias = 0*.

In terms of binary-point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$\text{real-world value} = \text{integer} \times 2^{-\text{fraction length}} = \text{integer} \times 2^0.$$

In either case, trivial scaling means that the real-world value is simply equal to the stored integer value:

$$\text{real-world value} = \text{integer}.$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsScalingTrivial`

See Also

`ssGetDataTypeIsScalingPow2`

Introduced before R2006a

ssGetDataTypeIdNumberOfChunks

Return number of chunks in multiword storage container of registered data type

Syntax

```
extern int ssGetDataTypeIdNumberOfChunks(SimStruct *S,  
                                         DTypeId dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the number of chunks in its multiword storage container.

Description

This function returns the number of chunks in the multiword storage container of the fixed-point data type designated by `dataTypeId`. This function is valid only for a registered data type whose storage container uses a multiword representation. You can use the `ssGetDataTypeIdStorageContainCat` function to identify the storage container category; for multiword storage containers, the function returns the category `FXP_STORAGE_MULTIWORD`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

See Also

`ssGetDataTypeNameOfChunks`

Introduced in R2007b

ssGetDataTypeIdStorageContainCat

Return storage container category of registered data type

Syntax

```
extern fxpStorageContainerCategory
ssGetDataTypeIdStorageContainCat(SimStruct *S, DTypeId dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the container category.

Description

This function returns the storage container category of the data type designated by `dataTypeId`. The container category returned by this function is used to store input and output signals, run-time parameters, and DWorks during Simulink simulations.

During simulation, fixed-point signals are held in one of the types of containers shown in the following table. Therefore in many cases, signals are represented in containers with more bits than their actual word length.

Fixed-Point Storage Containers

Container Category	Signal Word Length	Container Word Length	Container Size
FXP_STORAGE_INT8 (signed) FXP_STORAGE_UINT8 (unsigned)	1 to 8 bits	8 bits	1 byte
FXP_STORAGE_INT16 (signed) FXP_STORAGE_UINT16 (unsigned)	9 to 16 bits	16 bits	2 bytes

Container Category	Signal Word Length	Container Word Length	Container Size
FXP_STORAGE_INT32 (signed) FXP_STORAGE_UINT32 (unsigned)	17 to 32 bits	32 bits	4 bytes
FXP_STORAGE_OTHER_SINGLE_WORD	33 to word length of long data type	Length of long data type	Length of long data type
FXP_STORAGE_MULTIWORD	Greater than the word length of long data type to 128 bits	Multiples of length of long data type to 128 bits	Multiples of length of long data type to 128 bits

When the number of bits in the signal word length is less than the size of the container, the word length bits are always stored in the least significant bits of the container. The remaining container bits must be sign extended to fit the bits of the container:

- If the data type is unsigned, then the sign-extended bits must be cleared to zero.
- If the data type is signed, then the sign-extended bits must be set to one for strictly negative numbers, and cleared to zero otherwise.

The `ssGetDataTypeInfoStorageContainCat` function can also return the following values.

Other Storage Containers

Container Category	Description
FXP_STORAGE_UNKNOWN	Returned if the storage container category is unknown
FXP_STORAGE_SINGLE	Container type for a Simulink single
FXP_STORAGE_DOUBLE	Container type for a Simulink double
FXP_STORAGE_SCALEDDOUBLE	Container type for a data type that has been overridden with Scaled double

This function errors out when `ssGetDataTypeInfoIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

Because the mapping of storage containers in simulation to storage containers in code generation is not one-to-one, the TLC functions for storage containers in TLC are different from those in simulation. Refer to “Storage Container TLC Functions” on page A-10 for more information:

- `FixPt_DataTypeInfoNativeType`
- `FixPt_DataTypeInfoStorageDouble`
- `FixPt_DataTypeInfoStorageSingle`
- `FixPt_DataTypeInfoStorageScaledDouble`
- `FixPt_DataTypeInfoStorageSInt`
- `FixPt_DataTypeInfoStorageUInt`
- `FixPt_DataTypeInfoStorageSLong`
- `FixPt_DataTypeInfoStorageULong`
- `FixPt_DataTypeInfoStorageSShort`
- `FixPt_DataTypeInfoStorageUShort`

See Also

`ssGetDataTypeInfoStorageContainerSize`

Introduced before R2006a

ssGetDataTypeInfoStorageContainerSize

Return storage container size of registered data type

Syntax

```
extern size_t ssGetDataTypeInfoStorageContainerSize  
              (SimStruct *S, DTypeId  
              dataTypeId)
```

Arguments

S

SimStruct representing an S-function block.

dataTypeId

Data type ID of the registered data type for which you want to know the container size.

Description

This function returns the storage container size of the data type designated by `dataTypeId`. This function returns the same value as would the `sizeof()` function; it does not return the word length of either the storage container or the data type. To get the word length of the storage container, use `ssGetDataTypeInfoFxpContainWordLen`. To get the word length of the data type, use `ssGetDataTypeInfoFxpWordLength`.

The container of the size returned by this function stores input and output signals, runtime parameters, and DWorks during Simulink simulations. It is also the appropriate size measurement to pass to functions like `memcpy()`.

This function errors out when `ssGetDataTypeInfoIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

Examples

An `sfixed24_en10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeInfoStorageContainerSize` or `sizeof()` returns 4, which is the storage container size in bytes.
- `ssGetDataTypeInfoContainWordLen` returns 32, which is the storage container word length in bits.
- `ssGetDataTypeInfoWordLength` returns 24, which is the data type word length in bits.

TLC Functions

`FixPt_GetDataTypeInfoStorageContainerSize`

See Also

`ssGetDataTypeInfoContainWordLen`, `ssGetDataTypeInfoWordLength`,
`ssGetDataTypeInfoStorageContainerCat`

Introduced before R2006a

`ssGetDataTypeIdTotalSlope`

Return total slope of scaling of registered data type

Syntax

```
extern double ssGetDataTypeIdTotalSlope (SimStruct *S, DTypeId  
                                         dataTypeId)
```

Arguments

`S`

SimStruct representing an S-function block.

`dataTypeId`

Data type ID of the registered data type for which you want to know the total slope.

Description

Fixed-point numbers can be represented as
real-world value = (slope × integer) + bias,

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}.$$

This function returns the total slope, rather than the fractional slope, of the data type designated by `dataTypeId`. To get the fractional slope, use `ssGetDataTypeIdFracSlope`:

- If the registered data type has trivial scaling, including `double` and `single` data types, the function returns a total slope of 1.
- If the registered data type is `ScaledDouble`, the function returns the total slope of the nonoverridden data type. Refer to the examples below.

This function errors out when `ssGetDataTypeIdIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

Examples

The data type `sfix32_En4` becomes `flts32_En4` with data type override. The total slope returned by this function in either case is $0.0625 (2^{-4})$.

The data type `ufix16_s7p98` becomes `fltu16_s7p98` with data type override. The total slope returned by this function in either case is 7.98 .

TLC Functions

`FixPt_DataTypeTotalSlope`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFixedExponent`, `ssGetDataTypeFracSlope`

Introduced before R2006a

ssLogFixptInstrumentation

Record information collected during simulation

Syntax

```
extern void ssLogFixptInstrumentation
    (SimStruct *S,
     double minValue,
     double maxValue,
     int countOverflows,
     int countSaturation,
     int countDivisionsByZero,
     char *pStrName)
```

Arguments

S

SimStruct representing an S-function block.

minValue

Minimum output value that occurred during simulation.

maxValue

Maximum output value that occurred during simulation.

countOverflows

Number of overflows that occurred during simulation.

countSaturation

Number of saturations that occurred during simulation.

countDivisionsByZero

Number of divisions by zero that occurred during simulation.

***pStrName**

The string argument is currently unused.

Description

ssLogFixptInstrumentation records information collected during a simulation, such as output maximum and minimum, any overflows, saturations, and divisions by zero that occurred. The Fixed-Point Tool displays this information after a simulation.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

Introduced in R2008b

`ssRegisterDataTypeFxpBinaryPoint`

Register fixed-point data type with binary-point-only scaling and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpBinaryPoint
    (SimStruct *S,
     int isSigned,
     int wordLength,
     int fractionLength,
     int obeyDataTypeOverride)
```

Arguments

`S`

SimStruct representing an S-function block.

`isSigned`

TRUE if the data type is signed.

FALSE if the data type is unsigned.

`wordLength`

Total number of bits in the data type, including any sign bit.

`fractionLength`

Number of bits in the data type to the right of the binary point.

`obeyDataTypeOverride`

TRUE indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`, `Scaled double`, or the fixed-point data type specified by the other arguments of the function.

FALSE indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type with binary-point-only scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled double.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer software license is checked out. To prevent a Fixed-Point Designer software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )  
    ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to “Data Type IDs” on page A-12.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpFSlopeFixExpBias`,
`ssRegisterDataTypeFxpScaledDouble`, `ssRegisterDataTypeFxpSlopeBias`

Introduced before R2006a

ssRegisterDataTypeFxpFSlopeFixExpBias

Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpFSlopeFixExpBias
    (SimStruct *S,
     int isSigned,
     int wordLength,
     double fractionalSlope,
     int fixedExponent,
     double bias,
     int obeyDataTypeOverride)
```

Arguments

S

SimStruct representing an S-function block.

isSigned

TRUE if the data type is signed.

FALSE if the data type is unsigned.

wordLength

Total number of bits in the data type, including any sign bit.

fractionalSlope

Fractional slope of the data type.

fixedExponent

Exponent of the slope of the data type.

bias

Bias of the scaling of the data type.

obeyDataTypeOverride

TRUE indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be **Double**, **Single**, **Scaled double**, or the fixed-point data type specified by the other arguments of the function.

FALSE indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type by specifying the word length, fractional slope, fixed exponent, and bias. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled double.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer software license is checked out. To prevent a Fixed-Point Designer software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )  
    ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to “Data Type IDs” on page A-12.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpBinaryPoint`, `ssRegisterDataTypeFxpScaledDouble`,
`ssRegisterDataTypeFxpSlopeBias`

Introduced before R2006a

ssRegisterDataTypeFxpScaledDouble

Register scaled double data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpScaledDouble
    (SimStruct *S,
     int isSigned,
     int wordLength,
     double fractionalSlope,
     int fixedExponent,
     double bias,
     int obeyDataTypeOverride)
```

Arguments

S

SimStruct representing an S-function block.

isSigned

TRUE if the data type is signed.

FALSE if the data type is unsigned.

wordLength

Total number of bits in the data type, including any sign bit.

fractionalSlope

Fractional slope of the data type.

fixedExponent

Exponent of the slope of the data type.

bias

Bias of the scaling of the data type.

obeyDataTypeOverride

TRUE indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be **Double**, **Single**, **Scaled double**, or the fixed-point data type specified by the other arguments of the function.

FALSE indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a scaled double data type. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer software license is checked out. To prevent a Fixed-Point Designer software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
    ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to “Data Type IDs” on page A-12.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpBinaryPoint`,
`ssRegisterDataTypeFxpFSlopeFixExpBias`, `ssRegisterDataTypeFxpSlopeBias`

Introduced before R2006a

ssRegisterDataTypeFxpSlopeBias

Register data type with [Slope Bias] scaling and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpSlopeBias
    (SimStruct *S,
     int isSigned,
     int wordLength,
     double totalSlope,
     double bias,
     int obeyDataTypeOverride)
```

Arguments

S

SimStruct representing an S-function block.

isSigned

TRUE if the data type is signed.

FALSE if the data type is unsigned.

wordLength

Total number of bits in the data type, including any sign bit.

totalSlope

Total slope of the scaling of the data type.

bias

Bias of the scaling of the data type.

obeyDataTypeOverride

TRUE indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be **Double**, **Single**, **Scaled double**, or the fixed-point data type specified by the other arguments of the function.

FALSE indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type with [Slope Bias] scaling. Alternately, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled double.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer software license is checked out. To prevent a Fixed-Point Designer software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
    ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to “Data Type IDs” on page A-12.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see “Structure of the S-Function” on page A-5.

Languages

C

TLC Functions

None.

See Also

ssRegisterDataTypeFxpBinaryPoint,
ssRegisterDataTypeFxpFSlopeFixExpBias,
ssRegisterDataTypeFxpScaledDouble

Introduced before R2006a

