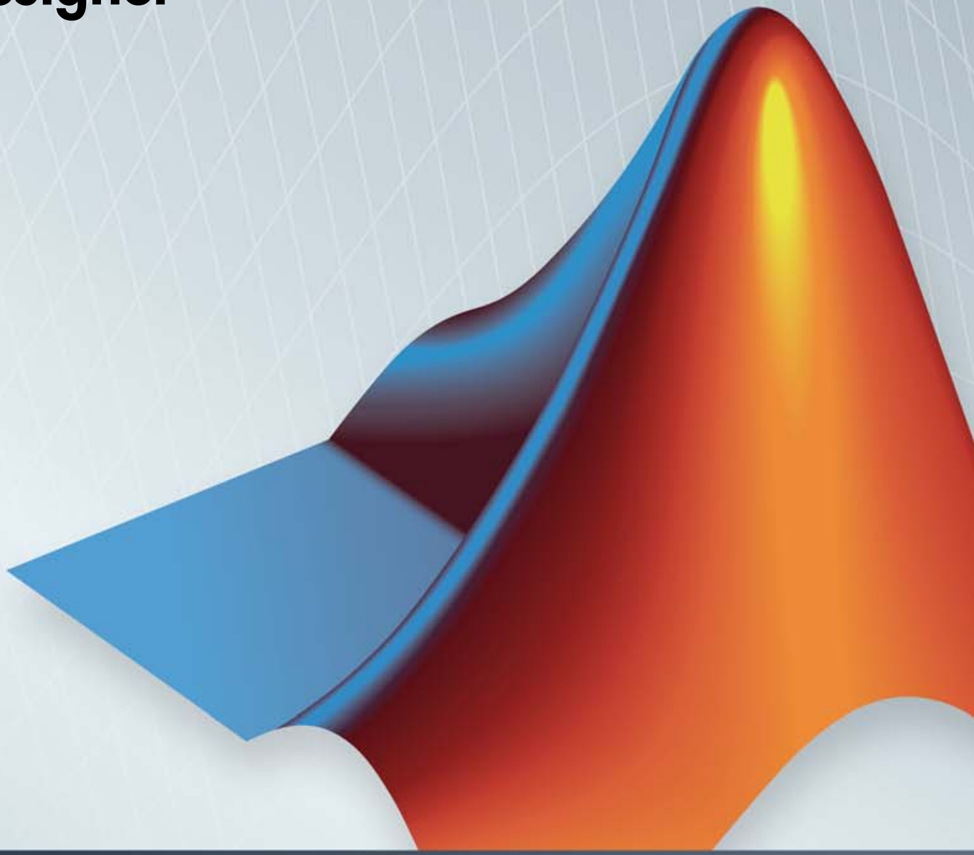
Fixed-Point Designer™

User's Guide

R2013**b**



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

☎ 508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Fixed-Point Designer™ User's Guide

Trademarks

Patents

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (R2013b)

Contents

Fixed-Point Designer™ for MATLAB® Code

1 Fixed-Point Concepts

Fixed-Point Data Types **1-2**

Scaling **1-4**

Precision and Range **1-5**
Range **1-5**
Precision **1-6**

Arithmetic Operations **1-10**
Modulo Arithmetic **1-10**
Two’s Complement **1-11**
Addition and Subtraction **1-12**
Multiplication **1-13**
Casts **1-19**

fi Objects and C Integer Data Types **1-22**
Integer Data Types **1-22**
Unary Conversions **1-24**
Binary Conversions **1-25**
Overflow Handling **1-28**

2 Working with fi Objects

Ways to Construct fi Objects **2-2**
Types of fi Constructors **2-2**

Examples of Constructing fi Objects 2-3

Cast fi Objects 2-12
Overwriting by Assignment 2-12
Ways to Cast with MATLAB Software 2-12

fi Object Properties 2-17
Data Properties 2-17
fimath Properties 2-17
numerictype Properties 2-19
Setting fi Object Properties 2-20

fi Object Functions 2-23

3 Fixed-Point Topics

Set Up Fixed-Point Objects 3-2
Create Fixed-Point Data 3-2

View Fixed-Point Number Circles 3-18

Perform Binary-Point Scaling 3-33

Develop Fixed-Point Algorithms 3-39

Calculate Fixed-Point Sine and Cosine 3-51

Calculate Fixed-Point Arctangent 3-76

Compute Sine and Cosine Using CORDIC Rotation Kernel 3-105

Perform QR Factorization Using CORDIC 3-111

Compute Square Root Using CORDIC Hyperbolic Kernel 3-151

Convert Cartesian to Polar Using CORDIC Vectoring Kernel 3-161

Set Data Types Using Min/Max Instrumentation 3-167

Convert Fast Fourier Transform (FFT) to Fixed Point 3-181

Detect Limit Cycles in Fixed-Point State-Space Systems 3-194

Compute Quantization Error 3-208

Normalize Data for Lookup Tables 3-218

Implement Fixed-Point Log2 Using Lookup Table 3-224

Implement Fixed-Point Square Root Using Lookup Table 3-229

Set Fixed-Point Math Attributes 3-235

4 Working with fimath Objects

fimath Object Construction 4-2
- fimath Object Syntaxes 4-2
- Building fimath Object Constructors in a GUI 4-4

fimath Object Properties 4-6
- Math, Rounding, and Overflow Properties 4-6
- Setting fimath Object Properties 4-7

fimath Properties Usage for Fixed-Point Arithmetic . . **4-11**
fimath Rules for Fixed-Point Arithmetic **4-11**
Binary-Point Arithmetic **4-13**
[Slope Bias] Arithmetic **4-17**

fimath for Rounding and Overflow Modes **4-20**

fimath for Sharing Arithmetic Rules **4-22**
Default fimath Usage to Share Arithmetic Rules **4-22**
Local fimath Usage to Share Arithmetic Rules **4-22**

fimath ProductMode and SumMode **4-25**
Example Setup **4-25**
FullPrecision **4-26**
KeepLSB **4-27**
KeepMSB **4-28**
SpecifyPrecision **4-30**

5 Working with fipref Objects

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**

6 Working with numerictype Objects

numerictype Object Construction **6-2**
numerictype Object Syntaxes **6-2**
Example: Construct a numerictype Object with Property Name and Property Value Pairs **6-3**
Example: Copy a numerictype Object **6-4**
Example: Build numerictype Object Constructors in a GUI **6-5**

numerictype Object Properties **6-7**
Data Type and Scaling Properties **6-7**
Set numerictype Object Properties **6-8**

numerictype Structure of Fixed-Point Objects **6-11**
Valid Values for numerictype Structure Properties **6-11**
Properties That Affect the Slope **6-13**
Stored Integer Value and Real World Value **6-13**

numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects **6-14**
Example 1 .... **6-14**
Example 2 .... **6-15**

7 Working with quantizer Objects

Constructing quantizer Objects **7-2**

quantizer Object Properties **7-3**

Quantizing Data with quantizer Objects **7-4**

Transformations for Quantized Data **7-6**

8 Fixed-Point Conversion

Fixed-Point Conversion Workflows **8-2**
Command-Line Workflow **8-2**
Automated Project Workflow **8-2**

Code Coverage **8-3**

Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros **8-7**

Propose Data Types Based on Simulation Ranges **8-37**

Propose Data Types Based on Derived Ranges **8-55**

Command-Line Fixed-Point Conversion Workflow **8-70**

Command-Line Fixed-Point Conversion Best Practices **8-72**
Separate data types from algorithm **8-72**
Create a Test File **8-72**
Entry-Point Function **8-73**

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

Code Acceleration and Code Generation from MATLAB **9-3**

Requirements for Generating Complied C Code Files **9-4**

Functions Supported for Code Acceleration or C Code Generation **9-5**

Fixed-Point Code Acceleration and Generation Workflow 9-15

Set Up C Compiler 9-16

Accelerate Code Using fiaccel 9-17
Speeding Up Fixed-Point Execution with fiaccel 9-17
Running fiaccel 9-17
Generated Files and Locations 9-18
Data Type Override Using fiaccel 9-21

File Infrastructure and Paths Setup 9-22
Compile Path Search Order 9-22
Naming Conventions 9-22

Detect and Debug Code Generation Errors 9-25
Debugging Strategies 9-25
Error Detection at Design Time 9-26
Error Detection at Compile Time 9-26

Set Up C Code Compilation Options 9-28
C Code Compiler Configuration Object 9-28
Compilation Options Modification at the Command Line Using Dot Notation 9-28
How fiaccel Resolves Conflicting Options 9-29

MEX Configuration Dialog Box Options 9-30
See Also 9-35

Specify Primary Function Input Properties 9-36
Why You Must Specify Input Properties 9-36
Properties to Specify 9-36
Rules for Specifying Properties of Primary Inputs 9-39
Methods for Defining Properties of Primary Inputs 9-39
Input Properties Definition by Example at the Command Line 9-40

Best Practices for Accelerating Fixed-Point Code 9-47
Recommended Compilation Options for fiaccel 9-47
Build Scripts 9-48

Check Code Interactively Using MATLAB Code Analyzer 9-49
Separating Your Test Bench from Your Function Code ... 9-50
Preserving Your Code 9-50
File Naming Conventions 9-50

Use Fixed-Point Code Generation Reports **9-51**
Code Generation Report Creation 9-51
Code Generation Report Opening 9-52
Viewing Your MATLAB Code 9-52
Viewing Variables in the Variables Tab 9-55
See Also 9-56

Generate C Code from Code Containing Global Data .. **9-57**
Workflow Overview 9-57
Declaring Global Variables 9-57
Defining Global Data 9-58
Synchronizing Global Data with MATLAB 9-59
Limitations of Using Global Data 9-62

Define Input Properties Programmatically in MATLAB File **9-63**
How to Use `assert` 9-63
Rules for Using assert Function 9-67
Example: Specifying Properties of Primary Fixed-Point Inputs 9-68
Example: Specifying Class and Size of Scalar Structure .. 9-69
Example: Specifying Class and Size of Structure Array .. 9-70

Control Run-Time Checks **9-71**
Types of Run-Time Checks 9-71
When to Disable Run-Time Checks 9-72
How to Disable Run-Time Checks 9-72

Fix Run-Time Stack Overflows **9-74**

Code Generation with MATLAB Coder **9-75**

Code Generation with MATLAB Function Block **9-76**
Composing a MATLAB Language Function in a Simulink Model 9-76

MATLAB Function Block with Data Type Override 9-76
Fixed-Point Data Types with MATLAB Function Block ... 9-78

Generate Fixed-Point FIR Code Using MATLAB Function Block **9-85**
Program the MATLAB Function Block 9-85
Prepare the Inputs 9-86
Create the Model 9-86
Define the fimath Object Using the Model Explorer 9-88
Run the Simulation 9-89

Fixed-Point FIR Code Example Parameter Values **9-90**

Accelerate Code for Variable-Size Data **9-93**
Disable Support for Variable-Size Data 9-93
Control Dynamic Memory Allocation 9-94
Accelerate Code for MATLAB Functions with Variable-Size Data 9-95
Accelerate Code for a MATLAB Function That Expands a Vector in a Loop 9-97

Accelerate Fixed-Point Simulation **9-104**

Propose Fixed-Point Data Types Using an Instrumented Mex Function **9-107**

Apply Fixed-Point Data Types Using an Instrumented Mex Function **9-117**

Code Generation Readiness Tool **9-123**
What Information Does the Code Generation Readiness Tool Provide? 9-123
Summary Tab 9-124
Code Structure Tab 9-126
See Also 9-129

Check Code Using the Code Generation Readiness Tool **9-130**
Run Code Generation Readiness Tool at the Command Line 9-130

Run the Code Generation Readiness Tool From the Current Folder Browser 9-130
See Also 9-130

Check Code Using the MATLAB Code Analyzer 9-131

Fix Errors Detected at Code Generation Time 9-132
See Also 9-132

Avoid Multiword Operations in Generated Code 9-133

10 Interoperability with Other Products

fi Objects with Simulink 10-2
Reading Fixed-Point Data from the Workspace 10-2
Writing Fixed-Point Data to the Workspace 10-2
Setting the Value and Data Type of Block Parameters . . . 10-6
Logging Fixed-Point Signals 10-6
Accessing Fixed-Point Block Data During Simulation 10-7

fi Objects with DSP System Toolbox 10-8
Reading Fixed-Point Signals from the Workspace 10-8
Writing Fixed-Point Signals to the Workspace 10-8
fi Objects with dfilt Objects 10-12

Ways to Generate Code 10-13

11 Calling Functions for Code Generation

Resolution of Function Calls in MATLAB Generated Code 11-2
Key Points About Resolving Function Calls 11-4

Compile Path Search Order 11-4
When to Use the Code Generation Path 11-5

Resolution of File Types on Code Generation Path ... 11-6

Compilation Directive %#codegen 11-8

Call Local Functions 11-9

Call Supported Toolbox Functions 11-10

Call MATLAB Functions 11-11
Declaring MATLAB Functions as Extrinsic Functions ... 11-12
Calling MATLAB Functions Using feval 11-16
How MATLAB Resolves Extrinsic Functions During Simulation 11-16
Working with mxArrays 11-17
Restrictions on Extrinsic Functions for Code Generation .. 11-19
Limit on Function Arguments 11-19

12 Code Generation for MATLAB Classes

MATLAB Classes Definition for Code Generation 12-2
Language Limitations 12-2
Code Generation Features Not Compatible with Classes .. 12-3
Defining Class Properties for Code Generation 12-4
Calls to Base Class Constructor 12-6

Classes That Support Code Generation 12-8

Generate Code for MATLAB Value Classes 12-9

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

MATLAB Classes in Code Generation Reports **12-18**
What Reports Tell You About Classes **12-18**
How Classes Appear in Code Generation Reports **12-18**
How to Generate a Code Generation Report **12-20**

Troubleshooting Issues with MATLAB Classes **12-21**
Class `class` does not have a property with name `name` ... **12-21**

13 Defining Data for Code Generation

Data Definition for Code Generation **13-2**

Code Generation for Complex Data **13-4**
Restrictions When Defining Complex Variables **13-4**
Expressions With Complex Operands Yield Complex Results **13-5**

Code Generation for Characters **13-7**

14 Defining Functions for Code Generation

Specify Variable Numbers of Arguments **14-2**

Supported Index Expressions **14-3**

Apply Operations to a Variable Number of Arguments **14-4**
When to Force Loop Unrolling **14-4**
Using Variable Numbers of Arguments in a for-Loop **14-5**

Implement Wrapper Functions **14-7**

Passing Variable Numbers of Arguments from One Function to Another 14-7

Pass Property/Value Pairs 14-8

Variable Length Argument Lists for Code Generation 14-10

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

Variables Definition for Code Generation 15-2

Best Practices for Defining Variables for C/C++ Code Generation 15-3
Define Variables By Assignment Before Using Them 15-3
Use Caution When Reassigning Variables 15-6
Use Type Cast Operators in Variable Definitions 15-6
Define Matrices Before Assigning Indexed Variables 15-6

Eliminate Redundant Copies of Variables in Generated Code 15-7
When Redundant Copies Occur 15-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables 15-7
Defining Uninitialized Variables 15-8

Reassignment of Variable Properties 15-9

Define and Initialize Persistent Variables 15-10

Reuse the Same Variable with Different Properties 15-11
When You Can Reuse the Same Variable with Different Properties 15-11
When You Cannot Reuse Variables 15-12
Limitations of Variable Reuse 15-14

Avoid Overflows in for-Loops **15-16**

Supported Variable Types **15-18**

16 Design Considerations for C/C++ Code Generation

When to Generate Code from MATLAB Algorithms ... **16-2**
When Not to Generate Code from MATLAB Algorithms .. **16-2**

Which Code Generation Feature to Use **16-4**

Prerequisites for C/C++ Code Generation from MATLAB **16-6**

MATLAB Code Design Considerations for Code Generation **16-7**
See Also **16-8**

Expected Differences in Behavior After Compiling MATLAB Code **16-9**
Why Are There Differences? **16-9**
Character Size **16-9**
Order of Evaluation in Expressions **16-9**
Termination Behavior **16-10**
Size of Variable-Size N-D Arrays **16-10**
Size of Empty Arrays **16-11**
Floating-Point Numerical Results **16-11**
NaN and Infinity Patterns **16-12**
Code Generation Target **16-12**
MATLAB Class Initial Values **16-12**
Variable-Size Support for Code Generation **16-12**

MATLAB Language Features Supported for C/C++ Code Generation **16-13**
MATLAB Language Features Not Supported for C/C++ Code Generation **16-14**

17 Code Generation for Enumerated Data

Enumerated Data Definition for Code Generation 17-2

Enumerated Types Supported for Code Generation ... 17-3
Enumerated Type Based on int32 .... **17-3**

When to Use Enumerated Data for Code Generation .. 17-5

Generate Code for Enumerated Data from MATLAB Algorithms 17-6
How to Generate Code for Enumerated Data **17-6**

Define Enumerated Data for Code Generation 17-8
Naming Enumerated Types for Code Generation **17-9**

Instantiate Enumerated Types for Code Generation .. 17-10

Operations on Enumerated Data Allowed for Code Generation 17-11
Assignment Operator, = ... **17-11**
Relational Operators, < > <= >= == ~= **17-11**
Cast Operation **17-12**
Indexing Operation **17-12**
Control Flow Statements: if, switch, while **17-13**

Include Enumerated Data in Control Flow Statements 17-14
`if` Statement with Enumerated Data Types **17-14**
`switch` Statement with Enumerated Data Types **17-15**
`while` Statement with Enumerated Data Types **17-18**

Customize Enumerated Types Based on int32 17-20
About Customizing Enumerated Types **17-20**
Specify a Default Enumerated Value **17-22**
Specify a Header File **17-23**

Control Names of Enumerated Type Values in Generated Code 17-26

Change and Reload Enumerated Data Types 17-28

Restrictions on Use of Enumerated Data in `for`-Loops 17-29

Toolbox Functions That Support Enumerated Types for Code Generation 17-30

18 Code Generation for Function Handles

Function Handle Definition for Code Generation 18-2

Define and Pass Function Handles for Code Acceleration 18-3

Function Handle Limitations for Code Generation . . . 18-5

19 Generate Efficient and Reusable Code

Optimization Strategies 19-3

Modularize MATLAB Code 19-6

Eliminate Redundant Copies of Function Inputs 19-7

Inline Code 19-9
- Prevent Function Inlining 19-9
- Use Inlining in Control Flow Statements 19-9

Control Inlining Using Configuration Object **19-11**
Control Size of Functions Inlined **19-11**
Control Size of Functions After Inlining **19-12**
Control Stack Size Limit on Inlined Functions **19-12**

Fold Function Calls into Constants **19-14**

Control Stack Space Usage **19-16**

Stack Allocation and Performance **19-17**

Rewrite Logical Array Indexing as a Loop **19-18**

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

Minimize Dynamic Memory Allocation **19-21**

Provide Maximum Size for Variable-size Arrays **19-22**

Disable Dynamic Memory Allocation During Code Generation **19-29**

Set Dynamic Memory Allocation Threshold **19-30**
Set Dynamic Memory Allocation Threshold Using Project Interface **19-30**
Set Dynamic Memory Allocation Threshold from Command Line **19-32**

Excluding Unused Paths from Generated Code **19-33**

Prevent Code Generation for Unused Execution Paths **19-34**
Prevent Code Generation When Local Variable Controls Flow **19-34**
Prevent Code Generation When Input Variable Controls Flow **19-35**

Generate Code with Parallel for-loops (parfor) **19-37**

Minimize Redundant Operations in Loops **19-39**

Unroll for-Loops **19-41**
Limit Copying the `for`-loop Body in Generated Code **19-41**

Support for Integer Overflow and Non-Finites **19-44**
Disable Support for Integer Overflow **19-44**
Disable Support for Non-Finites **19-45**

Integrate Custom Code **19-46**

MATLAB Coder Optimizations in Generated Code **19-52**
Constant Folding **19-52**
Loop Fusion **19-53**
Successive Matrix Operations Combined **19-54**
Unreachable Code Elimination **19-54**

Generate Reusable Code **19-56**

20 Code Generation for MATLAB Structures

Structure Definition for Code Generation **20-2**

Structure Operations Allowed for Code Generation ... **20-3**

Define Scalar Structures for Code Generation **20-4**
Restriction When Using struct **20-4**
Restrictions When Defining Scalar Structures by Assignment **20-4**
Adding Fields in Consistent Order on Each Control Flow Path **20-4**
Restriction on Adding New Fields After First Use **20-5**

Define Arrays of Structures for Code Generation **20-7**
Ensuring Consistency of Fields **20-7**
Using repmat to Define an Array of Structures with Consistent Field Properties **20-7**
Defining an Array of Structures Using Concatenation **20-8**

Make Structures Persistent **20-9**

Index Substructures and Fields **20-10**

Assign Values to Structures and Fields **20-12**

Pass Large Structures as Input Parameters **20-14**

21 Functions Supported for Code Generation

Functions Supported for C/C++ Code Generation — Alphabetical List **21-2**

Functions Supported for C/C++ Code Generation — Categorical List **21-88**
Aerospace Toolbox Functions **21-89**
Arithmetic Operator Functions **21-90**
Bit-Wise Operation Functions **21-90**
Casting Functions **21-91**
Communications System Toolbox Functions **21-91**
Complex Number Functions **21-91**
Computer Vision System Toolbox Functions **21-92**
Data and File Management Functions **21-94**
Data Type Functions **21-94**
Derivative and Integral Functions **21-95**
Discrete Math Functions **21-95**
Error Handling Functions **21-96**
Exponential Functions **21-96**
Filtering and Convolution Functions **21-97**
Fixed-Point Designer Functions **21-97**
Histogram Functions **21-106**

Image Processing Toolbox Functions 21-106
Input and Output Functions 21-108
Interpolation and Computational Geometry Functions . . . 21-109
Linear Algebra 21-109
Logical Operator Functions 21-109
MATLAB Compiler Functions 21-110
MATLAB Desktop Environment Functions 21-110
Matrix and Array Functions 21-110
Nonlinear Numerical Methods 21-114
Phased Array System Toolbox Functions 21-115
Polynomial Functions 21-118
Programming Utilities 21-118
Relational Operator Functions 21-118
Rounding and Remainder Functions 21-119
Set Functions 21-119
Signal Processing Functions in MATLAB 21-120
Signal Processing Toolbox Functions 21-120
Special Values 21-125
Specialized Math 21-125
Statistical Functions 21-126
Statistics Toolbox Functions 21-126
String Functions 21-132
Structure Functions 21-133
Trigonometric Functions 21-133

22 Code Generation for Variable-Size Data

What Is Variable-Size Data? **22-2**

Variable-Size Data Definition for Code Generation . . . **22-3**

Bounded Versus Unbounded Variable-Size Data **22-4**

Control Memory Allocation of Variable-Size Data **22-5**

Specify Variable-Size Data Without Dynamic Memory Allocation **22-6**
Fixing Upper Bounds Errors **22-6**

Specifying Upper Bounds for Variable-Size Data **22-6**

Variable-Size Data in Code Generation Reports 22-10
What Reports Tell You About Size **22-10**
How Size Appears in Code Generation Reports **22-11**
How to Generate a Code Generation Report **22-11**

Define Variable-Size Data for Code Generation 22-12
When to Define Variable-Size Data Explicitly **22-12**
Using a Matrix Constructor with Nonconstant Dimensions **22-13**
Inferring Variable Size from Multiple Assignments **22-13**
Defining Variable-Size Data Explicitly Using coder.varsize **22-15**

C Code Interface for Arrays 22-19
C Code Interface for Statically Allocated Arrays **22-19**
C Code Interface for Dynamically Allocated Arrays **22-20**
Utility Functions for Creating emxArray Data Structures **22-21**

Diagnose and Fix Variable-Size Data Errors 22-23
Diagnosing and Fixing Size Mismatch Errors **22-23**
Diagnosing and Fixing Errors in Detecting Upper Bounds **22-25**

Incompatibilities with MATLAB in Variable-Size Support for Code Generation 22-27
Incompatibility with MATLAB for Scalar Expansion **22-27**
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays **22-29**
Incompatibility with MATLAB in Determining Size of Empty Arrays **22-30**
Incompatibility with MATLAB in Determining Class of Empty Arrays **22-31**
Incompatibility with MATLAB in Vector-Vector Indexing **22-32**
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation **22-33**
Incompatibility with MATLAB in Concatenating Variable-Size Matrices **22-34**

Dynamic Memory Allocation Not Supported for MATLAB Function Blocks 22-35

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation **22-36**
Common Restrictions 22-36
Toolbox Functions with Variable Sizing Restrictions 22-37

23 Primary Functions

Primary Function Input Specification **23-2**
Why You Must Specify Input Properties 23-2
Properties to Specify 23-2
Rules for Specifying Properties of Primary Inputs 23-4
Methods for Defining Properties of Primary Inputs 23-5

Define Input Properties Programmatically in the MATLAB File **23-6**
How to Use assert with MATLAB Coder 23-6
Rules for Using assert Function 23-13
Specifying General Properties of Primary Inputs 23-14
Specifying Properties of Primary Fixed-Point Inputs 23-15
Specifying Class and Size of Scalar Structure 23-15
Specifying Class and Size of Structure Array 23-16

24 System Objects Supported for Code Generation

System Objects Supported for C/C++ Code Generation **24-2**
Code Generation for System Objects 24-2
Computer Vision System Toolbox System Objects 24-2
Communications System Toolbox System Objects 24-7
DSP System Toolbox System Objects 24-13
Phased Array System Toolbox System Objects 24-19

Image Acquisition Toolbox System Objects 24-23

25 System Objects

Create System Objects **25-2**
Create a System object **25-2**
Change a System object Property **25-3**
Check if a System object Property Has Changed **25-3**
Run a System object **25-3**
Display Available System Objects **25-3**

Set Up System Objects **25-4**
Create a New System object **25-4**
Retrieve System object Property Values **25-4**
Set System object Property Values **25-4**

Process Data Using System Objects **25-7**
What are System object Methods? **25-7**
The Step Method **25-7**
Common Methods **25-7**
Advantages of Using Methods **25-9**

Tuning System object Properties in MATLAB **25-10**
Understand System object Modes **25-10**
Change Properties While Running System Objects **25-11**
Change System object Input Complexity or Dimensions .. **25-11**

Find Help and Examples for System Objects **25-12**

Use System Objects in MATLAB Code Generation **25-13**
Considerations for Using System Objects in Generated Code **25-13**
Use System Objects with codegen **25-17**
Use System Objects with the MATLAB Function Block ... **25-17**
Use System Objects with MATLAB Compiler **25-17**

Fixed-Point Designer™ for Simulink® Models

26 Getting Started

Product Description **26-2**
Key Features **26-2**

What You Need to Get Started **26-3**
Installation **26-3**
Sharing Fixed-Point Models **26-3**

Physical Quantities and Measurement Scales **26-5**
Introduction **26-5**
Selecting a Measurement Scale **26-6**
Select a Measurement Scale for Temperature **26-8**

Why Use Fixed-Point Hardware? **26-14**

Why Use the Fixed-Point Designer Software? **26-16**

Developing and Testing Fixed-Point Systems in Simulink **26-17**

Supported Data Types **26-20**

Configure Blocks with Fixed-Point Output **26-21**
Specify the Output Data Type and Scaling **26-22**
Specify Fixed-Point Data Types with the Data Type Assistant **26-24**
Rounding **26-27**
Overflow Handling **26-28**
Lock the Output Data Type Setting **26-28**
Real-World Values Versus Stored Integer Values **26-29**

Configure Blocks with Fixed-Point Parameters **26-31**
Specify Fixed-Point Values Directly **26-31**

Specify Fixed-Point Values Via Parameter Objects 26-32

Pass Fixed-Point Data Between Simulink Models and the MATLAB Software 26-35
Read Fixed-Point Data from the Workspace 26-35
Write Fixed-Point Data to the Workspace 26-35
Log Fixed-Point Signals 26-38
Access Fixed-Point Block Data During Simulation 26-38

Cast from Doubles to Fixed Point 26-40
About This Example 26-40
Block Descriptions 26-41
Simulations 26-42

27 Data Types and Scaling

Data Types and Scaling in Digital Hardware 27-2

Fixed-Point Numbers 27-3
Fixed-Point Numbers 27-3
Signed Fixed-Point Numbers 27-4
Binary Point Interpretation 27-4
Scaling 27-6
Quantization 27-8
Range and Precision 27-10

Fixed-Point Numbers in Simulink 27-15
Constant Scaling for Best Precision 27-15
Fixed-Point Data Type and Scaling Notation 27-18
Scaled Doubles 27-20
Use Scaled Doubles to Avoid Precision Loss 27-22
Display Port Data Types 27-25

Floating-Point Numbers 27-27
Floating-Point Numbers 27-27
Scientific Notation 27-28
The IEEE Format 27-28

Range and Precision 27-30
Exceptional Arithmetic 27-33

28 Arithmetic Operations

Fixed-Point Arithmetic Operations 28-2

Precision 28-3
Limitations on Precision 28-3
Rounding 28-4
Choose a Rounding Mode 28-4
Rounding Modes for Fixed-Point Simulink Blocks 28-5
Rounding Mode: Ceiling 28-7
Rounding Mode: Convergent 28-8
Rounding Mode: Floor 28-10
Rounding Mode: Nearest 28-11
Rounding Mode: Round 28-12
Rounding Mode: Simplest 28-14
Rounding Mode: Zero 28-17
Pad with Trailing Zeros 28-19
Limitations on Precision and Errors 28-20
Maximize Precision 28-21
Net Slope and Net Bias Precision 28-21
Detect Net Slope and Bias Precision Issues 28-24
Detect Fixed-Point Constant Precision Loss 28-25

Range 28-27
Limitations on Range 28-27
What Are Saturation and Wrapping? 28-28
Saturation and Wrapping 28-28
Guard Bits 28-31
Determine the Range of Fixed-Point Numbers 28-31

Recommendations for Arithmetic and Scaling 28-33
Arithmetic Operations and Fixed-Point Scaling 28-33
Addition 28-34
Accumulation 28-37
Multiplication 28-37

Gain 28-39
Division 28-41
Summary 28-43

Parameter and Signal Conversions **28-44**
Introduction 28-44
Parameter Conversions 28-45
Signal Conversions 28-46

Rules for Arithmetic Operations **28-49**
Introduction 28-49
Computational Units 28-49
Addition and Subtraction 28-50
The Summation Process 28-52
Multiplication 28-55
The Multiplication Process 28-60
Division 28-62
The Division Process 28-63
Shifts 28-64

Conversions and Arithmetic Operations **28-67**

29 Realization Structures

Realizing Fixed-Point Digital Filters **29-2**
Introduction 29-2
Realizations and Data Types 29-2

Targeting an Embedded Processor **29-4**
Introduction 29-4
Size Assumptions 29-4
Operation Assumptions 29-4
Design Rules 29-5

Canonical Forms **29-7**
Canonical Forms 29-7
Direct Form II 29-8

Series Cascade Form **29-13**
Parallel Form **29-15**

30 Fixed-Point Advisor

Preparation for Fixed-Point Conversion **30-2**
Introduction **30-2**
Best Practices **30-2**
Run the Fixed-Point Advisor **30-4**
Fix a Task Failure **30-5**
Manually Fixing Failures **30-6**
Automatically Fixing Failures **30-6**
Batch Fixing Failures **30-7**
Restore Points **30-7**
Save a Restore Point **30-8**
Load a Restore Point **30-9**

Converting a Model from Floating- to Fixed-Point Using Simulation Data **30-11**
About This Example **30-11**
Starting the Preparation **30-11**
Preparing Model for Conversion **30-12**
Prepare for Data Typing and Scaling **30-17**
Propose Data Types Based on the Simulation Reference Run **30-20**
Apply the New Fixed-Point Data Types **30-20**
Simulate the Model Using New Fixed-Point Settings **30-21**

31 Fixed-Point Tool

Fixed-Point Tool **31-2**
Introduction to the Fixed-Point Tool **31-2**
Using the Fixed-Point Tool **31-2**

Run Management **31-5**
Run Management **31-5**
Why Use Shortcuts to Manage Runs **31-7**
When to Use Shortcuts to Manage Runs **31-7**
Add Shortcuts **31-8**
Edit Shortcuts **31-9**
Delete Shortcuts **31-10**
Capture Model Settings Using the Shortcut Editor **31-10**

Debug a Fixed-Point Model **31-12**
Simulating the Model to See the Initial Behavior **31-12**
Debugging the Model **31-14**
Simulating the Model Using a Different Input Stimulus . . **31-16**
Debugging the Model with the New Input **31-17**
Proposing Fraction Lengths for Math2 Based on Simulation Results **31-18**
Verifying the New Settings **31-18**

Logging Simulation Ranges for Referenced Models . . . **31-20**
Viewing Simulation Ranges for Referenced Models **31-20**
Fixed-Point Instrumentation and Data Type Override Settings **31-22**
See Also **31-23**

Log Simulation Ranges for Referenced Models **31-24**

Propose Data Types for a Referenced Model **31-30**

Logging Simulation Ranges for MATLAB Function Block **31-34**
See Also **31-34**

Log Simulation Ranges for MATLAB Function Block . . **31-35**

View Signal Names in Fixed-Point Tool **31-39**

32 Convert Floating-Point Model to Fixed Point

Learning Objectives **32-2**

Model Description **32-4**
Model Overview **32-4**
Model Set Up **32-5**

Before You Begin **32-7**

Convert Floating-Point Model to Fixed Point **32-9**
Open the Model **32-9**
Prepare Floating-Point Model for Conversion to Fixed Point **32-9**
Propose Data Types **32-18**
Apply Fixed-Point Data Types **32-19**
Verify Fixed-Point Settings **32-19**
Test Fixed-Point Settings With New Input Data **32-22**
Gather a Floating-Point Benchmark **32-23**
Propose Data Types for the New Input **32-24**
Apply the New Fixed-Point Data Types **32-25**
Verify New Fixed-Point Settings **32-25**
Prepare for Code Generation **32-26**

Key Points to Remember **32-28**

Where to Learn More **32-29**

33 Producing Lookup Table Data

Producing Lookup Table Data **33-2**

Worst-Case Error for a Lookup Table **33-3**
What Is the Worst-Case Error? **33-3**
Approximate the Square Root Function **33-3**

Create Lookup Tables for a Sine Function **33-6**
Introduction **33-6**
Parameters for fixpt_look1_func_approx **33-6**
Setting Function Parameters for the Lookup Table **33-8**
Using errmax with Unrestricted Spacing **33-8**
Using nptsmax with Unrestricted Spacing **33-11**
Using errmax with Even Spacing **33-14**
Using nptsmax with Even Spacing **33-16**
Using errmax with Power of Two Spacing **33-17**
Using nptsmax with Power of Two Spacing **33-19**
Specifying Both errmax and nptsmax **33-21**
Comparison of Example Results **33-22**

Use Lookup Table Approximation Functions **33-24**

Effects of Spacing on Speed, Error, and Memory Usage **33-25**
Criteria for Comparing Types of Breakpoint Spacing **33-25**
Model That Illustrates Effects of Breakpoint Spacing **33-25**
Data ROM Required for Each Lookup Table **33-26**
Determination of Out-of-Range Inputs **33-27**
How the Lookup Tables Determine Input Location **33-27**
Interpolation for Each Lookup Table **33-29**
Summary of the Effects of Breakpoint Spacing **33-32**

34 Automatic Data Typing

About Automatic Data Typing **34-2**

Before Proposing Data Types for Your Model **34-3**

Best Practices for Proposing Data Types **34-5**
Use a Known Working Simulink Model **34-5**
Back Up Your Simulink Model **34-5**
Capture the Current Data Type Override Settings **34-5**
Convert Individual Subsystems **34-5**
Isolate the System Under Conversion **34-5**
Use Lock Output Data Type Setting **34-6**

Save Simulink Signal Objects 34-6
Test Update Diagram Failure 34-6

Models That Might Cause Data Type Propagation Errors **34-7**

Automatic Data Typing Using Simulation Data **34-11**
Workflow for Automatic Data Typing Using Simulation Data 34-11
Set Up the Model 34-12
Prepare the Model for Conversion 34-13
Gather a Floating-Point Benchmark 34-14
Proposing Data Types 34-15
Propose Data Types 34-17
Examine Results to Resolve Conflicts 34-17
Apply Proposed Data Types 34-22
Verify New Settings 34-22
Automatic Data Typing of Simulink Signal Objects 34-23

Automatic Data Typing Using Derived Ranges **34-25**
Prerequisites for Autoscaling Using Derived Ranges 34-25
Workflow for Autoscaling Using Derived Data 34-25
Set Up the Model 34-26
Prepare Model for Autoscaling Using Derived Data 34-27
Derive Minimum and Maximum Values 34-28
Resolve Range Analysis Issues 34-30
Proposing Data Types 34-30
Propose Data Types 34-32
Examine Results to Resolve Conflicts 34-33
Apply Proposed Data Types 34-37
Update Diagram 34-38

Propose Fraction Lengths **34-39**
Propose Fraction Lengths 34-39
About the Feedback Controller Example Model 34-40
Propose Fraction Lengths Using Simulation Range Data 34-46

Propose Word Lengths **34-57**
How the Fixed-Point Tool Proposes Word Lengths 34-57
Propose Word Lengths 34-59
Propose Word Lengths Based on Simulation Data 34-60

Propose Data Types Using Multiple Simulations **34-67**
About This Example **34-67**
Running the Simulation **34-70**

View Simulation Results **34-73**
Compare Runs **34-73**
Compare Signals **34-74**
Inspect Signals **34-76**
Histogram Plot of Signal **34-77**
See Also **34-78**

Viewing Results With the Simulation Data Inspector . . **34-79**
Why Use the Simulation Data Inspector **34-79**
When to Use the Simulation Data Inspector **34-79**
What You Can Inspect in the Simulation Data Inspector . . **34-79**
See Also **34-80**

35 Range Analysis

How Range Analysis Works **35-2**
Analyzing a Model with Range Analysis **35-2**
Automatic Stubbing **35-5**
Model Compatibility with Range Analysis **35-6**

Derive Ranges **35-7**

Derive Ranges at the Subsystem Level **35-10**
Deriving Ranges at the Subsystem Level **35-10**
Derive Ranges at the Subsystem Level **35-11**

View Derived Ranges in the Fixed-Point Tool **35-12**

Range Analysis Examples **35-14**
Derive Ranges Using Design Minimum and Maximum Values **35-14**
Derive Ranges Using Block Initial Conditions **35-16**

Derive Ranges Using Design Range Information for Simulink.Parameter Objects 35-19
Insufficient Design Range Information 35-22
Providing More Design Range Information 35-25
Fixing Design Range Conflicts 35-28

Derive Ranges for a Referenced Model 35-31

Propose Data Types for a Referenced Model 35-37

Deriving Ranges for a Referenced Model 35-39
Viewing Derived Minimum and Maximum Values for Referenced Models 35-39
Data Type Override Settings 35-40
See Also 35-40

Unsupported Simulink Software Features 35-41

Supported and Unsupported Simulink Blocks 35-43
Overview of Simulink Block Support 35-43
Limitations of Support for Model Blocks 35-53

36 Code Generation

Generating and Deploying Production Code 36-2

Code Generation Support 36-3
Introduction 36-3
Languages 36-3
Data Types 36-3
Rounding Modes 36-3
Overflow Handling 36-3
Blocks 36-4
Scaling 36-4

Accelerating Fixed-Point Models 36-5

Using External Mode or Rapid Simulation Target **36-7**
Introduction **36-7**
External Mode **36-7**
Rapid Simulation Target **36-8**

Optimize Your Generated Code **36-9**
Reducing ROM Consumption or Model Execution Time .. **36-9**
Restrict Data Type Word Lengths **36-10**
Avoid Fixed-Point Scalings with Bias **36-11**
Wrap and Round to Floor or Simplest **36-11**
Limit the Use of Custom Storage Classes **36-13**
Limit the Use of Unevenly Spaced Lookup Tables **36-13**
Minimize the Variety of Similar Fixed-Point Utility Functions **36-13**
Handle Net Slope Correction **36-14**
Use Integer Division to Handle Net Slope Correction **36-15**
Improve Numerical Accuracy of Simulation Results with Integer Division to Handle Net Slope **36-16**
Improve Efficiency of Generated Code with Integer Division to Handle Net Slope **36-21**
Optimize Generated Code Using Specified Minimum and Maximum Values **36-27**
Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values **36-30**

Optimizing Your Generated Code with the Model Advisor **36-34**
Optimize Generated Code with Model Advisor **36-34**
Identify Blocks that Generate Expensive Fixed-Point and Saturation Code **36-35**
Identify Questionable Fixed-Point Operations **36-38**
Identify Blocks that Generate Expensive Rounding Code **36-41**
Use the Model Advisor to Optimize Fixed-Point Operations in Generated Code **36-42**

37 Fixed-Point Advisor Reference

Fixed-Point Advisor **37-2**

Fixed-Point Advisor Overview 37-3

Preparing Model for Conversion 37-6
Prepare Model for Conversion Overview 37-7
Verify model simulation settings 37-8
Verify update diagram status 37-10
Address unsupported blocks 37-11
Set up signal logging 37-13
Create simulation reference data 37-14
Verify Fixed-Point Conversion Guidelines Overview 37-16
Check model configuration data validity diagnostic parameters settings 37-17
Implement logic signals as Boolean data 37-18
Check bus usage 37-19
Simulation range checking 37-20
Check for implicit signal resolution 37-21

Preparing for Data Typing and Scaling 37-23
Prepare for Data Typing and Scaling Overview 37-24
Review locked data type settings 37-25
Remove output data type inheritance 37-26
Relax input data type settings 37-29
Verify Stateflow charts have strong data typing with Simulink 37-31
Remove redundant specification between signal objects and blocks 37-32
Verify hardware selection 37-34
Specify block minimum and maximum values 37-36

Return to the Fixed-Point Tool to Perform Data Typing and Scaling 37-38
See Also 37-38

A Writing Fixed-Point S-Functions

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-13
The Assignment of Data Type IDs A-13
Registering Data Types A-14
Setting and Getting Data Types A-16
Getting Information About Data Types A-17
Converting Data Types A-19

Overflow Handling and Rounding Methods A-20
Tokens for Overflow Handling and Rounding Methods . . . A-20
Overflow Logging Structure A-21

Create MEX-Files A-23

Fixed-Point S-Function Examples A-24
List of Fixed-Point S-Function Examples A-24
Get the Input Port Data Type A-25
Set the Output Port Data Type A-27
Interpret an Input Value A-28
Write an Output Value A-30
Determine Output Type Using the Input Type A-31

API Function Reference A-33

Index

Fixed-Point Designer for MATLAB Code

Fixed-Point Concepts

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

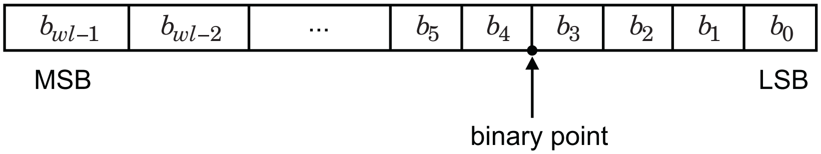
Fixed-Point Data Types

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

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

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

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



where

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

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

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

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

For information on how to create fixed-point data with Fixed-Point Designer software, see "Create Fixed-Point Data in MATLAB®" and "Configure Blocks with Fixed-Point Output" on page 26-21.

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

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

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.

Precision and Range

In this section...

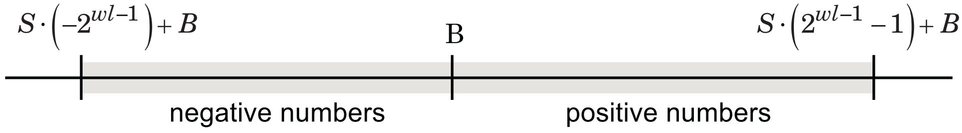
"Range" on page 1-5

"Precision" on page 1-6

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

Range

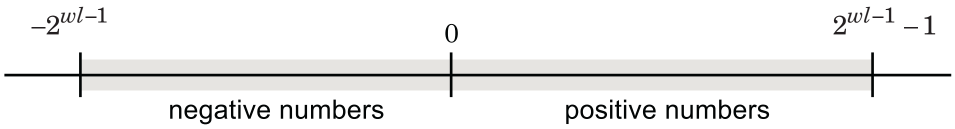
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl, scaling S and bias B is illustrated below:


$S\cdot\left(-2^{wl-1}\right)+B$
B
$S\cdot\left(2^{wl-1}-1\right)+B$
negative numbers
positive numbers

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. Refer to "Modulo Arithmetic" on page 1-10 for more information.

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

Precision

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

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

Rounding Methods

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, a rounding method is used to cast the value to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding method itself. To provide you with greater flexibility in the trade-off between cost and bias, Fixed-Point Designer software currently supports the following rounding methods:

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

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

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

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

- Bias — What is the expected value of the rounded values minus the original values: $\mathrm{E}\left(\hat{\theta}-\theta\right)$?
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)<0$ — The rounding method introduces a negative bias.
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)=0$ — The rounding method is unbiased.
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)>0$ — 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 Method in MATLAB	Fixed-Point Designer Rounding Mode in Simulink®	Cost	Bias	Possibility of Overflow
`ceil`	`Ceiling`	Low	Large positive	Yes
`convergent`	`Convergent`	High	Unbiased	Yes
`fix`	`Zero`	Low	• Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples	No
`floor`	`Floor`	Low	Large negative	No
`nearest`	`Nearest`	Moderate	Small positive	Yes
`round`	`Round`	High	• Small negative for negative samples • Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples	Yes
N/A	`Simplest` (Fixed-Point Designer only)	Low	Depends on the operation	No

Arithmetic Operations

In this section...

"Modulo Arithmetic" on page 1-10

"Two's Complement" on page 1-11

"Addition and Subtraction" on page 1-12

"Multiplication" on page 1-13

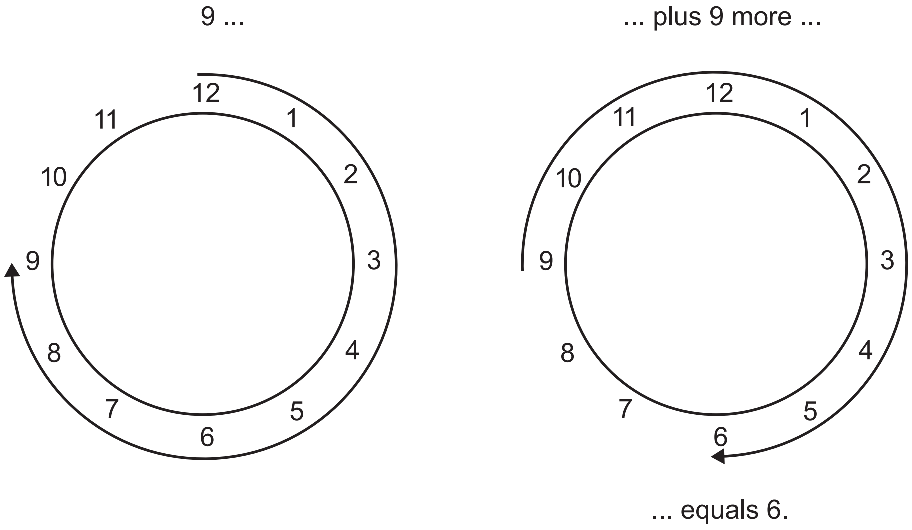
"Casts" on page 1-19

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

Modulo Arithmetic

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

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



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

Two’s Complement

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

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

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

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:

$$\begin{array}{rl} 00101 & \\ +1 & \\ \hline 00110 & (6) \end{array}$$

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):

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

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):

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

Carry bit
is discarded

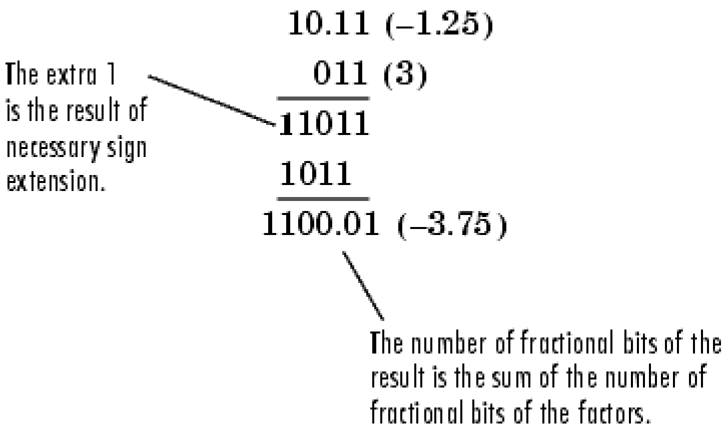
The default fimath has a value of `1` (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

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

Multiplication

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

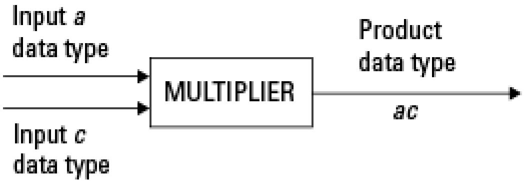
For example, consider the multiplication of 10.11 (-1.25) with 011 (3):


10.11 (–1.25)
011 (3)
11011
1011
1100.01 (–3.75)
The extra 1 is the result of necessary sign extension.
The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

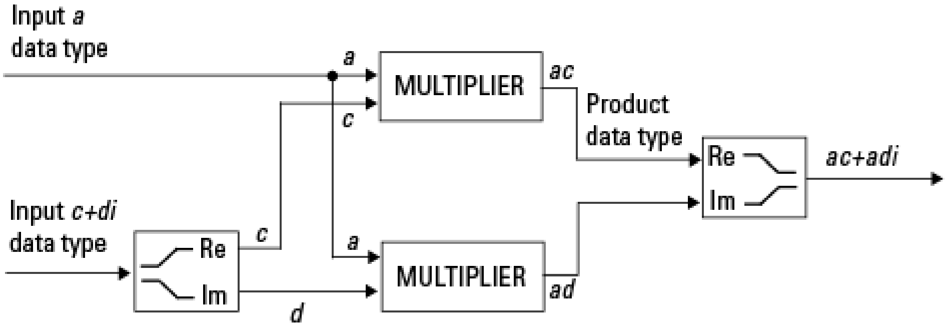
Multiplication Data Types

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.

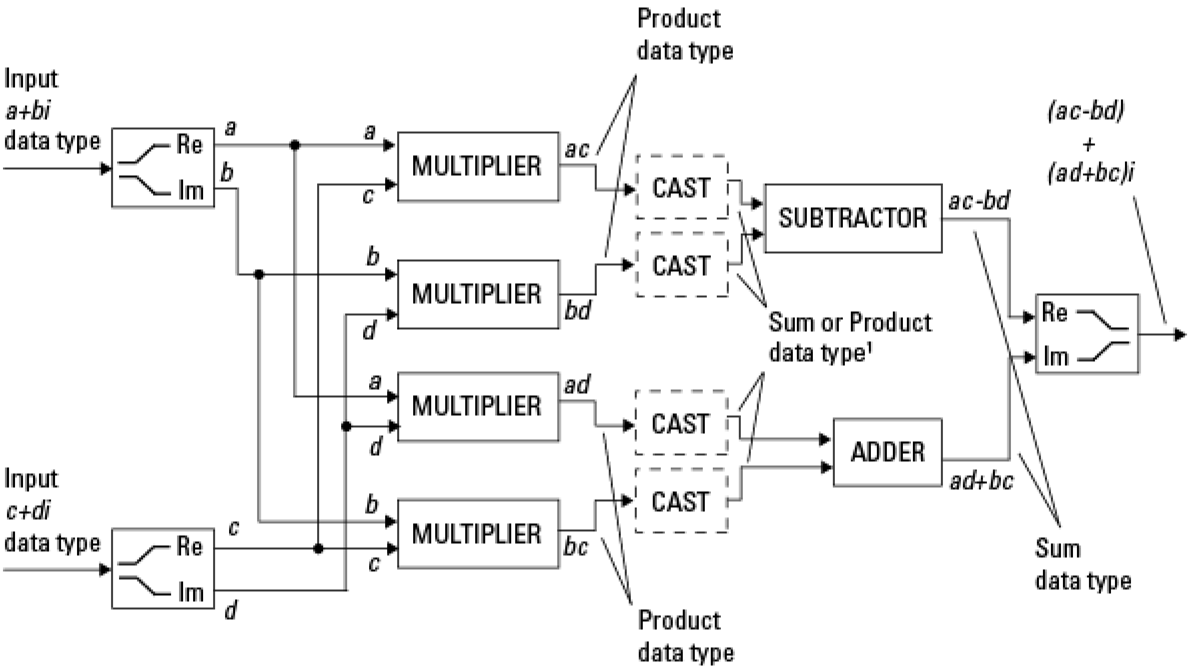
Real-Real Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of two real numbers. The software returns the output of this operation in the product data type, which is governed by the `fimath` object `ProductMode` property.



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. Note that the software returns the output of this operation in the sum data type, which is governed by the `fimath` object `SumMode` property. The intermediate product data type is determined by the `fimath` object `ProductMode` property.



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

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

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

for the subtractor, and

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

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

Multiplication with fimath

In the following examples, let

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

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

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

x =

     5


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

y = fi(10, T2, F)

y =

    10


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

z = x*y
```

```
z =

    50


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

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

```
x = fi(5,T1,F)

x =

     5


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

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

y =

  10.0000 + 2.0000i


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

```
z = x*y

z =

  50.0000 +10.0000i


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

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

```
x = 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
```

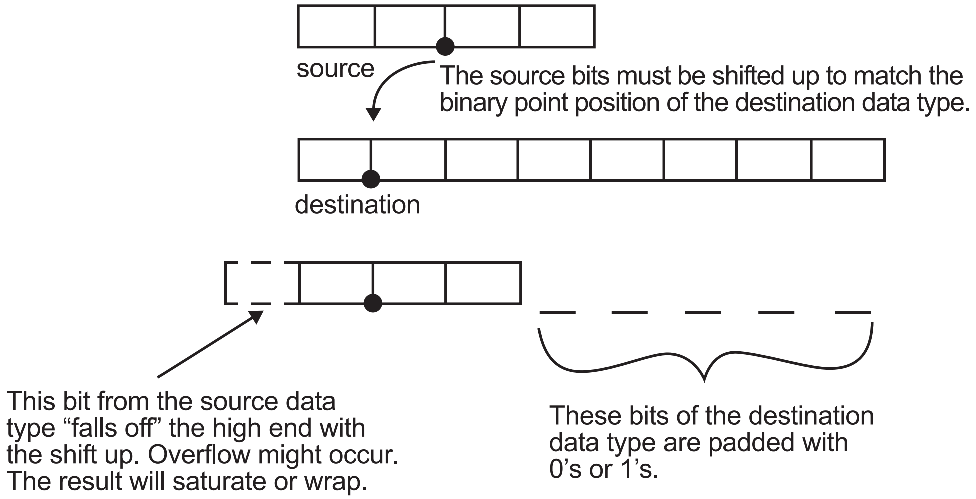
Casts

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

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

Casting from a Shorter Data Type to a Longer Data Type

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


source
The source bits must be shifted up to match the binary point position of the destination data type.
destination
This bit from the source data type “falls off” the high end with the shift up. Overflow might occur. The result will saturate or wrap.
These bits of the destination data type are padded with 0’s or 1’s.

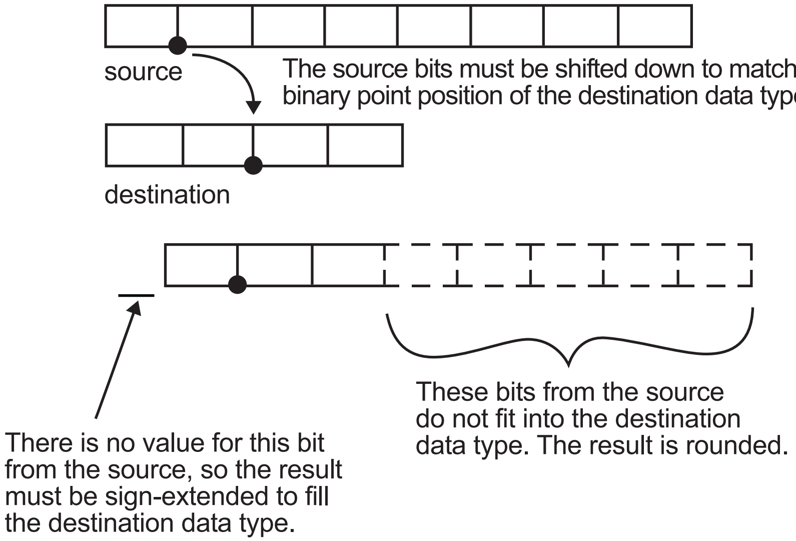
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either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


source
The source bits must be shifted down to match the binary point position of the destination data type.
destination
There is no value for this bit from the source, so the result must be sign-extended to fill the destination data type.
These bits from the source do not fit into the destination data type. The result is rounded.

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

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

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

fi Objects and C Integer Data Types

In this section...

"Integer Data Types" on page 1-22

"Unary Conversions" on page 1-24

"Binary Conversions" on page 1-25

"Overflow Handling" on page 1-28

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

Integer Data Types

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

C Integer Data Types

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

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

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

fi Integer Data Types

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

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

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

Unary Conversions

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

ANSI C Usual Unary Conversions

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

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

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

Binary Conversions

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

ANSI C Usual Binary Conversions

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

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

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

fi Usual Binary Conversions

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

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	double or single	• Signed = same as the original fi operand • WordLength = same as the original fi operand • FractionLength = set to best precision possible
fi	int8	• Signed = 1 • WordLength = 8 • FractionLength = 0

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	uint8	• Signed = 0 • WordLength = 8 • FractionLength = 0
fi	int16	• Signed = 1 • WordLength = 16 • FractionLength = 0
fi	uint16	• Signed = 0 • WordLength = 16 • FractionLength = 0
fi	int32	• Signed = 1 • WordLength = 32 • FractionLength = 0
fi	uint32	• Signed = 0 • WordLength = 32 • FractionLength = 0
fi	int64	• Signed = 1 • WordLength = 64 • FractionLength = 0
fi	uint64	• 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
`OverflowMode`	`'saturate'`	Overflows are saturated to the maximum or minimum value in the range.
	`'wrap'`	Overflows wrap using modulo arithmetic if unsigned, two’s complement wrap if signed.

fimath Object Properties Related to Overflow Handling	Property Value	Description
`ProductMode`	`'FullPrecision'`	Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than `MaxProductWordLength`. The rules for computing the resulting product word and fraction lengths are given in "ProductMode" in the Property Reference.
	`'KeepLSB'`	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. The resulting word length is determined by the `ProductWordLength` property. If `ProductWordLength` is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If `ProductWordLength` is less than is necessary for the full-precision product, then overflow occurs. The rule for computing the resulting product fraction length is given in "ProductMode" in the Property Reference.

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

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

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

2

Working with fi Objects

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

Ways to Construct fi Objects

In this section...

"Types of fi Constructors" on page 2-2

"Examples of Constructing fi Objects" on page 2-3

Types of fi Constructors

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

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

To get started, type

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

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

```
a =

     0


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

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

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

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

Examples of Constructing fi Objects

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

- fi
- sfi
- ufi

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

Constructing a fi Object with Property Name/Property Value Pairs

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

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

a =

    3.1415

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

```
        FractionLength: 13

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

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

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

Constructing a fi Object Using a numerictype Object

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

```
T = numerictype

T =


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

a = fi(pi, T)

a =

    1.0000


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

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

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

F =

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

a = fi(pi, T, F)

a =

    1.0000


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

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

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

Constructing a fi Object Using a fimath Object

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

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

F =


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

F.OverflowAction = 'Wrap'

F =


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

a = fi(pi, F)

a =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

You can also create `fi` objects using a `fimath` object while specifying various `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 string at a specific location in your file.

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

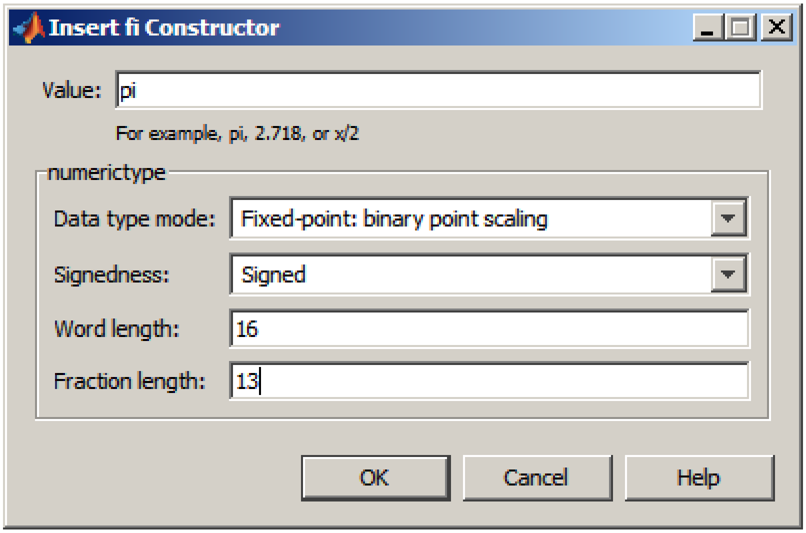
1 On the **Home** tab, in the **File** section, click **New > Script** to open the MATLAB Editor

2 On the **Editor** tab, in the **Edit** section, click fi ▾ in the **Insert** button group. Click the **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



Insert fi Constructor

Value: pi

For example, pi, 2.718, or x/2

numerictype

Data type mode: Fixed-point: binary point scaling

Signedness: Signed

Word length: 16

Fraction length: 13

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

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

Determining Property Precedence

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

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

T =


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

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

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

a =

    1.9999

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

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

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

b =

    3.1416

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

Copying a fi Object

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

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

```
a =

    3.1416

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

b =

    3.1416

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

Cast fi Objects

In this section...

"Overwriting by Assignment" on page 2-12

"Ways to Cast with MATLAB Software" on page 2-12

Overwriting by Assignment

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

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

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

A =

  32767

class(A)

ans =

int16
```

Ways to Cast with MATLAB Software

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

- Casting by Subscripted Assignment
- Casting by Conversion Function
- Casting with the Fixed-Point Designer `reinterpretcast` Function

Casting by Subscripted Assignment

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

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

A =

  127

class(A)

ans =

int8
```

The same is true for `fi` objects:

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

A =

   127
      s8,0
```

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

Casting by Conversion Function

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

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

A =

  127

class(A)

ans =

int8
```

The same is true for `fi` objects:

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

B =

       32767
      s16,0

A =

   127
      s8,0
```

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

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

T8 =
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0

T16 = numerictype(1,16,0)

T16 =


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

B = fi(32767,T16)

B =

       32767
      s16,0

A = fi(B, T8)

A =

   127
      s8,0
```

Casting with the reinterpretcast Function

You can convert fixed-point and built-in data types without changing the underlying data. The Fixed-Point 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], false, 8, 5)
T = numerictype(true, 8, 1);
A = reinterpretcast(B, T)

B =

    0.7813    1.0000    1.5625    4.0000

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

A =

   12.5000   16.0000   25.0000   -64.0000

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

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

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

binary_A =

00011001   00100000   00110010   10000000

binary_B =

00011001   00100000   00110010   10000000
```

fi Object Properties

In this section...

"Data Properties" on page 2-17

"fimath Properties" on page 2-17

"numerictype Properties" on page 2-19

"Setting fi Object Properties" on page 2-20

Data Properties

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

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

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

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

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

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

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

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

To learn more about these properties, see the "fimath Object Properties" in the Fixed-Point Designer Reference.

numerictype Properties

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

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

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

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

- Signedness — Whether a fi object is signed or unsigned

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

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

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

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

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

Setting fi Object Properties

You can set fi object properties in two ways:

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

Setting Fixed-Point Properties at Object Creation

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

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

a =
    3.1416


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

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

Using Direct Property Referencing with fi

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

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

```
a.WordLength

ans =

    16
```

To set the `OverflowMode` of `a`,

```
a.OverflowAction = 'Wrap'

a =

    3.1416


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

```
        FractionLength: 13

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

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

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

b =
    3.1415

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

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

b.fimath = []

b =
    3.1415

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

isfimathlocal(b)

ans =
     0
```

fi Object Functions

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

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

For example,

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

n =

  25736

h = hex(a)

h =

6488

a.hex

ans =

6488
```

Fixed-Point Topics

- “Set Up Fixed-Point Objects” on page 3-2
- “View Fixed-Point Number Circles” on page 3-18
- “Perform Binary-Point Scaling” on page 3-33
- “Develop Fixed-Point Algorithms” on page 3-39
- “Calculate Fixed-Point Sine and Cosine” on page 3-51
- “Calculate Fixed-Point Arctangent” on page 3-76
- “Compute Sine and Cosine Using CORDIC Rotation Kernel” on page 3-105
- “Perform QR Factorization Using CORDIC” on page 3-111
- “Compute Square Root Using CORDIC Hyperbolic Kernel” on page 3-151
- “Convert Cartesian to Polar Using CORDIC Vectoring Kernel” on page 3-161
- “Set Data Types Using Min/Max Instrumentation” on page 3-167
- “Convert Fast Fourier Transform (FFT) to Fixed Point” on page 3-181
- “Detect Limit Cycles in Fixed-Point State-Space Systems” on page 3-194
- “Compute Quantization Error” on page 3-208
- “Normalize Data for Lookup Tables” on page 3-218
- “Implement Fixed-Point Log2 Using Lookup Table” on page 3-224
- “Implement Fixed-Point Square Root Using Lookup Table” on page 3-229
- “Set Fixed-Point Math Attributes” on page 3-235

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 =

1100110011001100110011001100110011001100110011001101000000000000000000000000
```

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

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

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

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


ans =

1100110011001100110011001100110011001101


b = a*a


b =

        0.0100000000000045

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

bin(b)


ans =

101000111101011100001010001111010111000011110101110000101000111101011100001
```

Access to Data

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

DOUBLE(A)

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

```
ans =

              3.1416015625
```

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

A.DOUBLE = ...

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

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

```
a =

             2.71826171875

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

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

STOREDINTEGER(A)

```
storedInteger(a)
```

```
ans =

  22268
```

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

Relationship Between Stored Integer Value and Real-World Value

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

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

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

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

where

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

and

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

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

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

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

```
bin(a)


ans =

0101011011111100


oct(a)


ans =

053374
```

```
dec(a)
```

```
ans =

22268
```

```
hex(a)
```

```
ans =

56fc
```

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

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

fi(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 =

     10
```

```
storedInteger(b)
```

```
ans =

  20480
```

Specifying Properties with Parameter/Value Pairs

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

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

```
a =

          3.14159393310547

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

For more information on `fi` properties, type

```
help fi
```

or

```
doc fi
```

at the MATLAB command line.

Numeric Type Properties

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

```
T = numerictype


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)


ans =

011111111111111111111111111111111111111111111111111111111111111111111111111
```

The display of the `fi` object is also affected by MATLAB's `format` command. In particular, when displaying real-world values, it is handy to use

```
format long g
```

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

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

For more information on display preferences, type

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

or

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

at the MATLAB command line.

Cleanup

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

```
% Reset the fi display and logging preferences
fipref(fiprefAtStartOfThisExample);
set(0, 'format', originalFormat);
```

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) ... 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) + ... + b(2)*2^(1) + b(1)*2^(0)
```

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

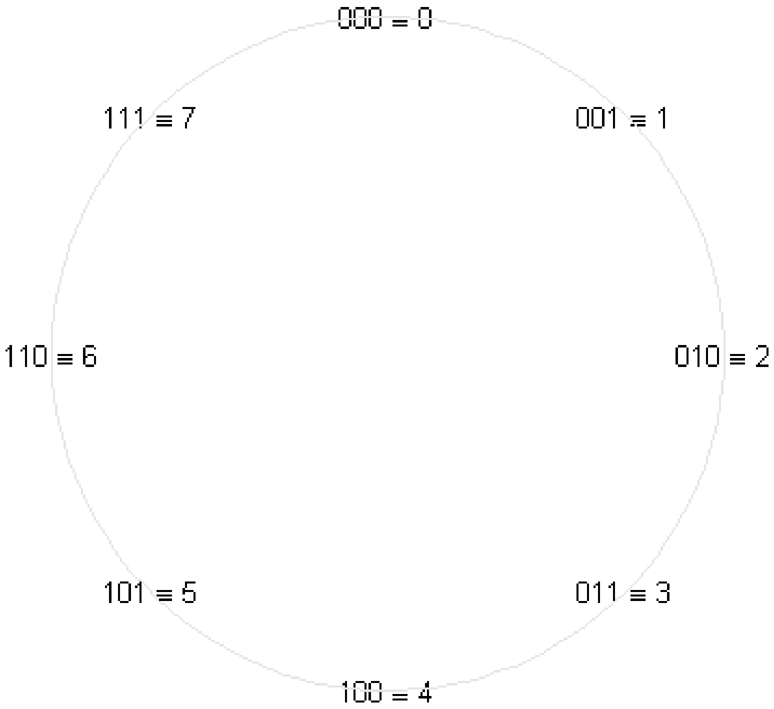
```
originalFormat = get(0, 'format'); format

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 =

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*2^-f
```

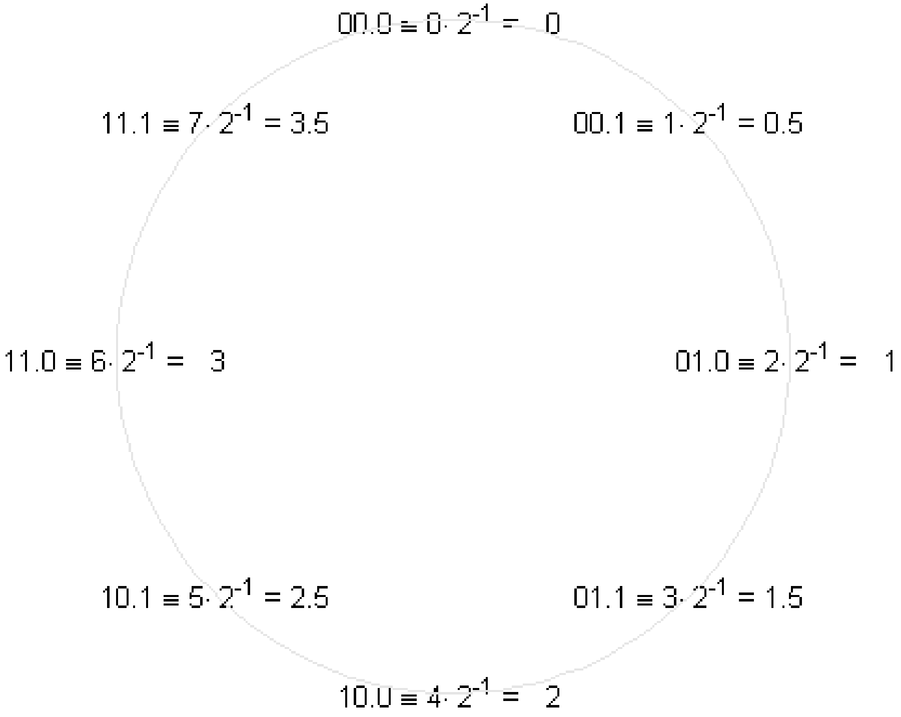
For example, let’s define a 3-bit unsigned fixed-point quantizer with a fractionlength of 1, and enumerate its range.

```
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 =

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);
```


$00.0 \equiv 0 \cdot 2^{-1} = 0$
$11.1 \equiv 7 \cdot 2^{-1} = 3.5$
$00.1 \equiv 1 \cdot 2^{-1} = 0.5$
$11.0 \equiv 6 \cdot 2^{-1} = 3$
$01.0 \equiv 2 \cdot 2^{-1} = 1$
$10.1 \equiv 5 \cdot 2^{-1} = 2.5$
$01.1 \equiv 3 \cdot 2^{-1} = 1.5$
$10.0 \equiv 4 \cdot 2^{-1} = 2$

Unsigned Fractional Fixed-Point.

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

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

```
  uf = u*2^-f
```

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

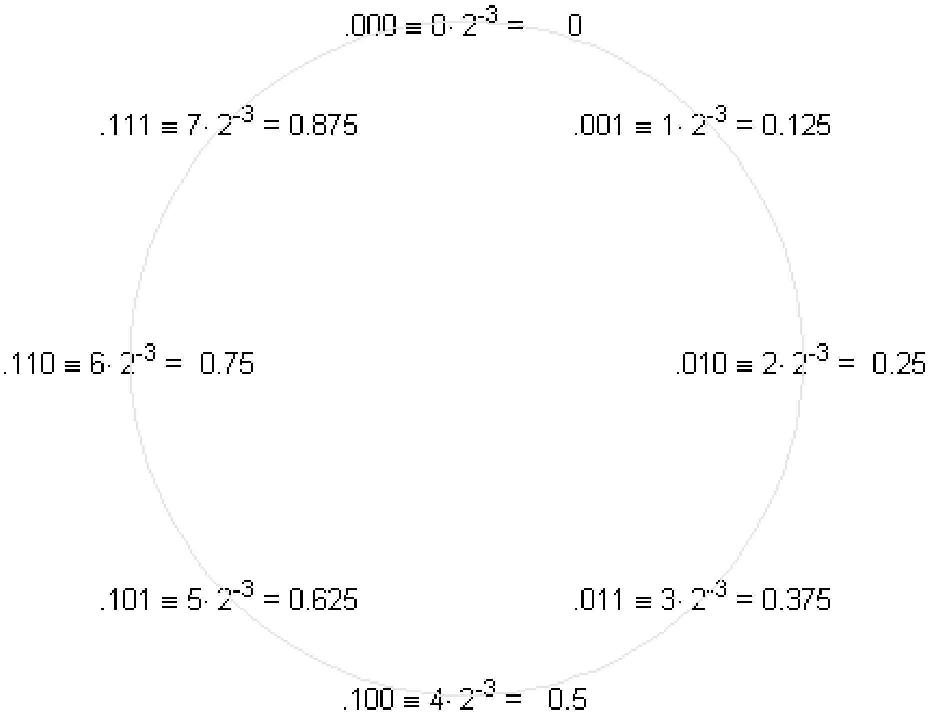
```
q = quantizer('ufixed',[3 3]);
[a,b] = range(q);
uf = (a:eps(q):b)'

% 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 =

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) ... 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)*2^(n-1) + b(n-1)*2^(n-2) + ... + b(2)*2^(1) + b(1)*2^(0)
```

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

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

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

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

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

```
s =

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


b =

100
101
```

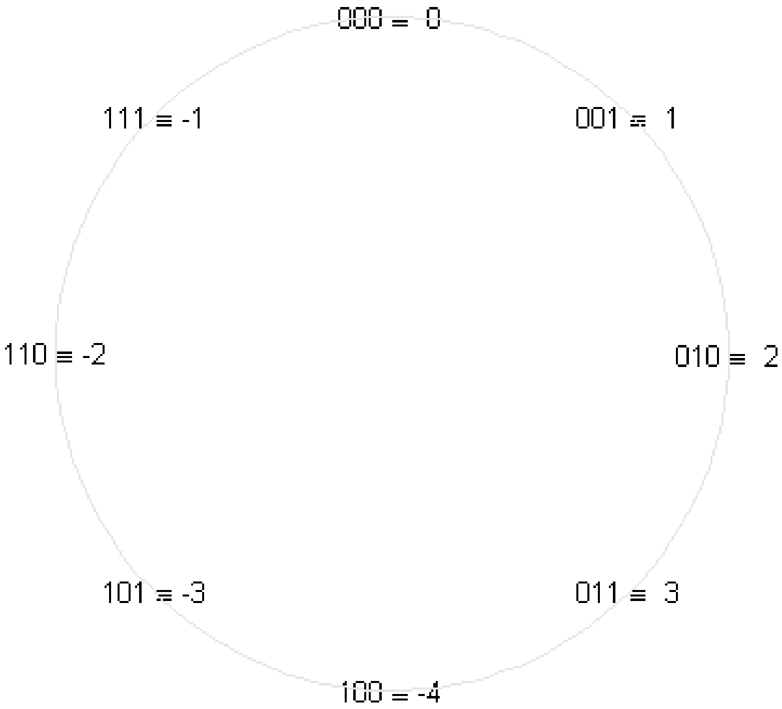
```
110
111
000
001
010
011
```

Signed Two's-Complement Integer Number Circle.

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

The reason for this ungainly looking definition of negative numbers is that addition of all numbers, both positive and negative, is carried out as if they were all positive, and then the n+1 carry bit is discarded. The result will be correct if there is no overflow.

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



Signed Fixed-Point.

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

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

```
sf = s*2^-f
```

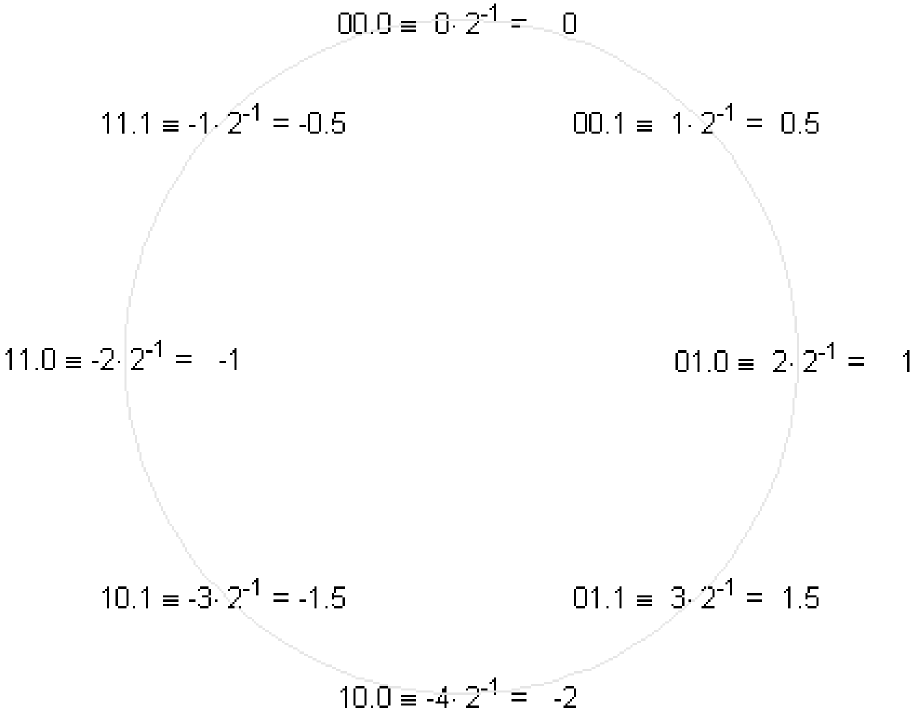
For example, let’s define a 3-bit signed fixed-point quantizer with a fractionlength of 1, and enumerate its range.

```
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 =

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 whos fractionlength 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.

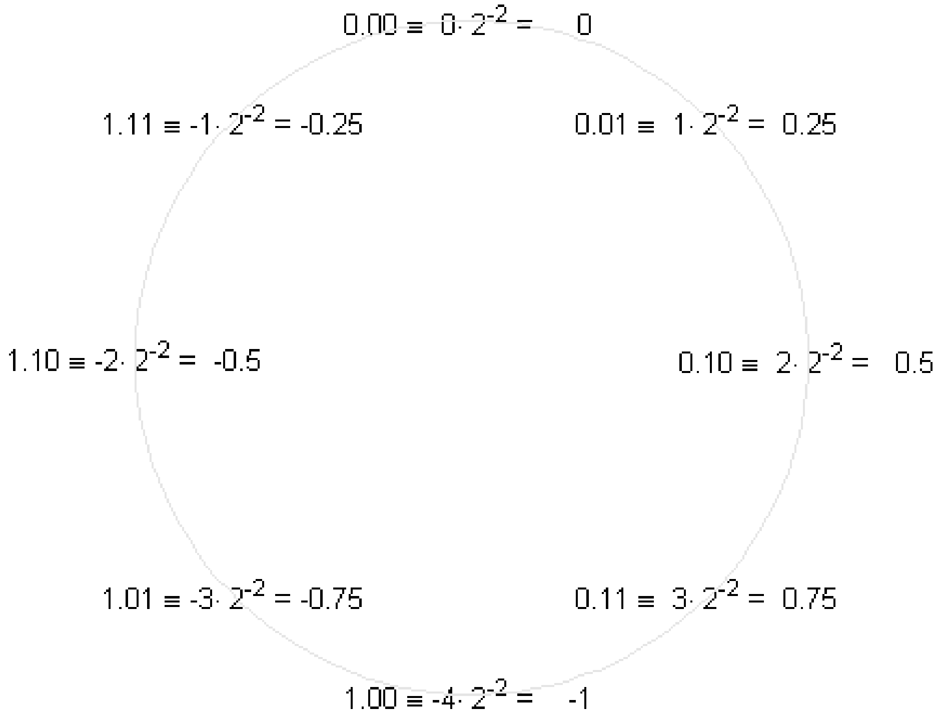
```
q = quantizer('fixed',[3 2]);
[a,b] = range(q);
sf = (a:eps(q):b)'

% 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 =

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);
```


$0.00 \equiv 0 \cdot 2^{-2} = 0$
$1.11 \equiv -1 \cdot 2^{-2} = -0.25$
$0.01 \equiv 1 \cdot 2^{-2} = 0.25$
$1.10 \equiv -2 \cdot 2^{-2} = -0.5$
$0.10 \equiv 2 \cdot 2^{-2} = 0.5$
$1.01 \equiv -3 \cdot 2^{-2} = -0.75$
$0.11 \equiv 3 \cdot 2^{-2} = 0.75$
$1.00 \equiv -4 \cdot 2^{-2} = -1$

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 =

   -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 =

   -1
```

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

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

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

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

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

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

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

```

c =

    -4

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

ans =

111


ans =

   -1
```

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

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

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

```
(1*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 =

    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)
```

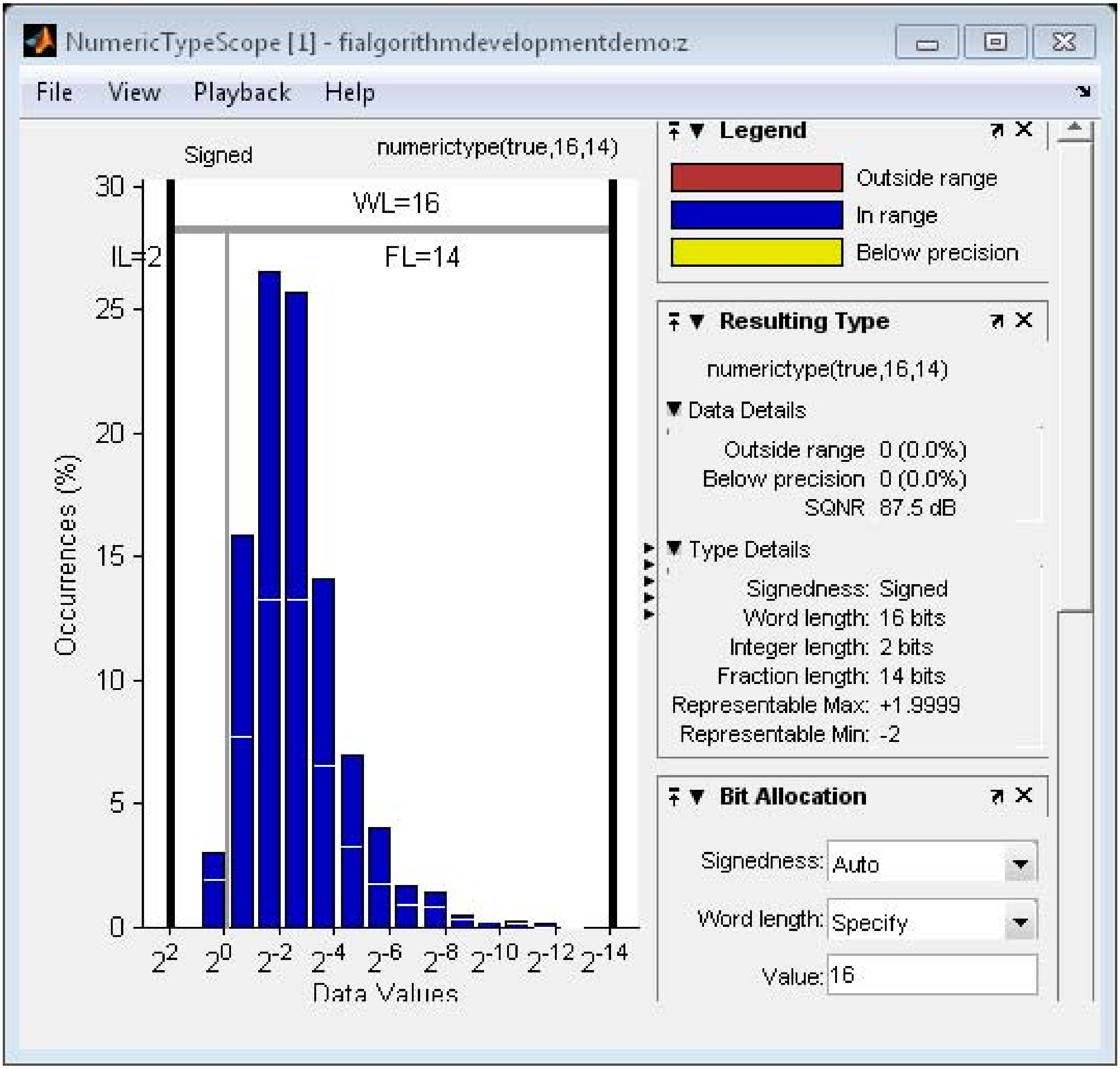
```
    y(k) =  b(1)*x(k) + z(1);
    z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
    z(2) =  b(3)*x(k)         - a(3)*y(k);
end

% 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 the state
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);
```



Legend

🟥	Outside range
🟦	In range
🟨	Below precision

Resulting Type

numerictype(true,16,14)

▼ Data Details

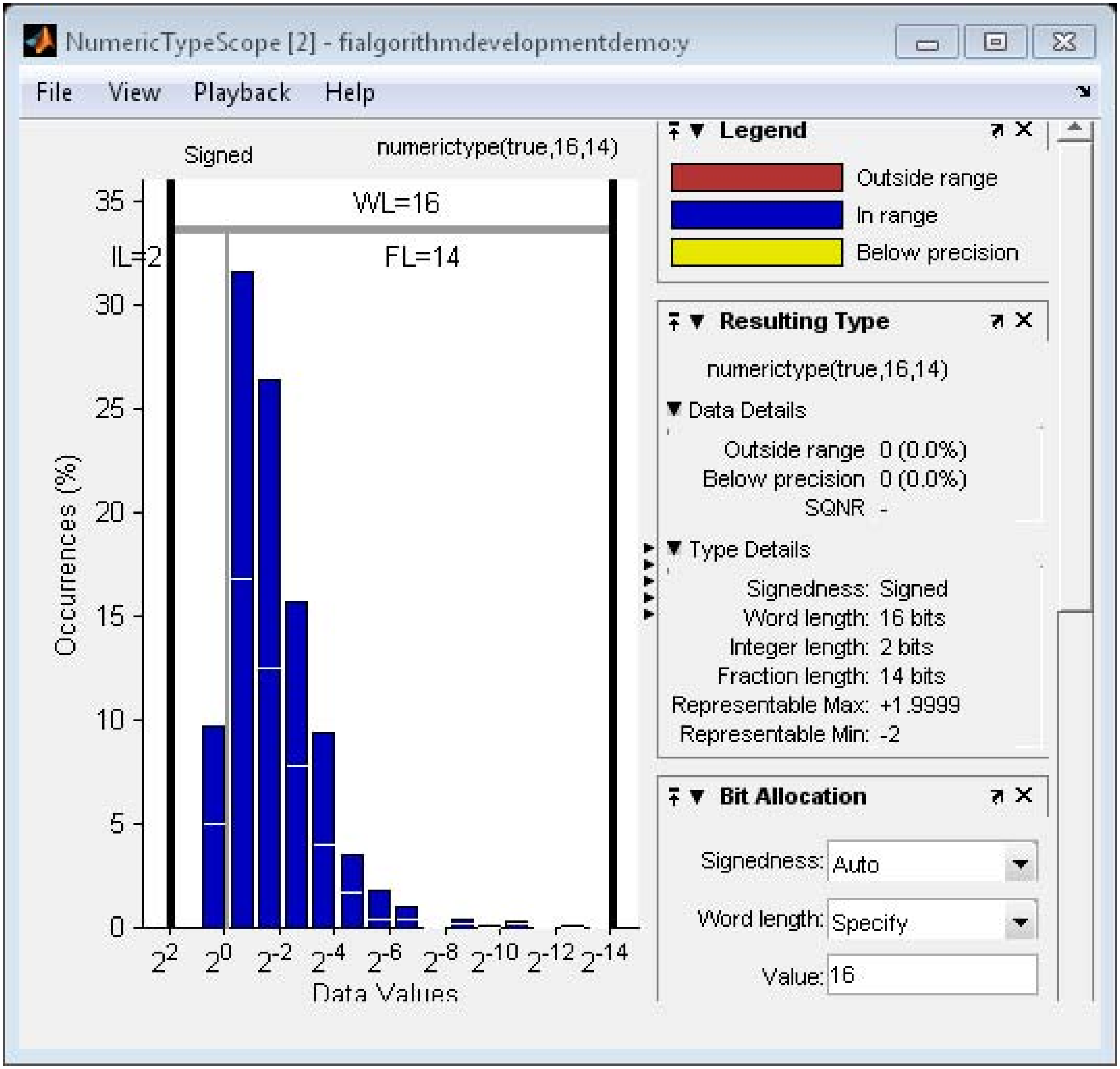
Outside range 0 (0.0%)
Below precision 0 (0.0%)
SQNR 87.5 dB

▼ Type Details

Signedness: Signed
Word length: 16 bits
Integer length: 2 bits
Fraction length: 14 bits
Representable Max: +1.9999
Representable Min: -2

Bit Allocation

Signedness: Auto
Word length: Specify
Value: 16



Legend

🟥	Outside range
🟦	In range
🟨	Below precision

Resulting Type

numerictype(true,16,14)

▼ Data Details

Outside range	0 (0.0%)
Below precision	0 (0.0%)
SQNR	-

▼ Type Details

Signedness:	Signed
Word length:	16 bits
Integer length:	2 bits
Fraction length:	14 bits
Representable Max:	+1.9999
Representable Min:	-2

Bit Allocation

Signedness:	Auto
Word length:	Specify
Value:	16

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

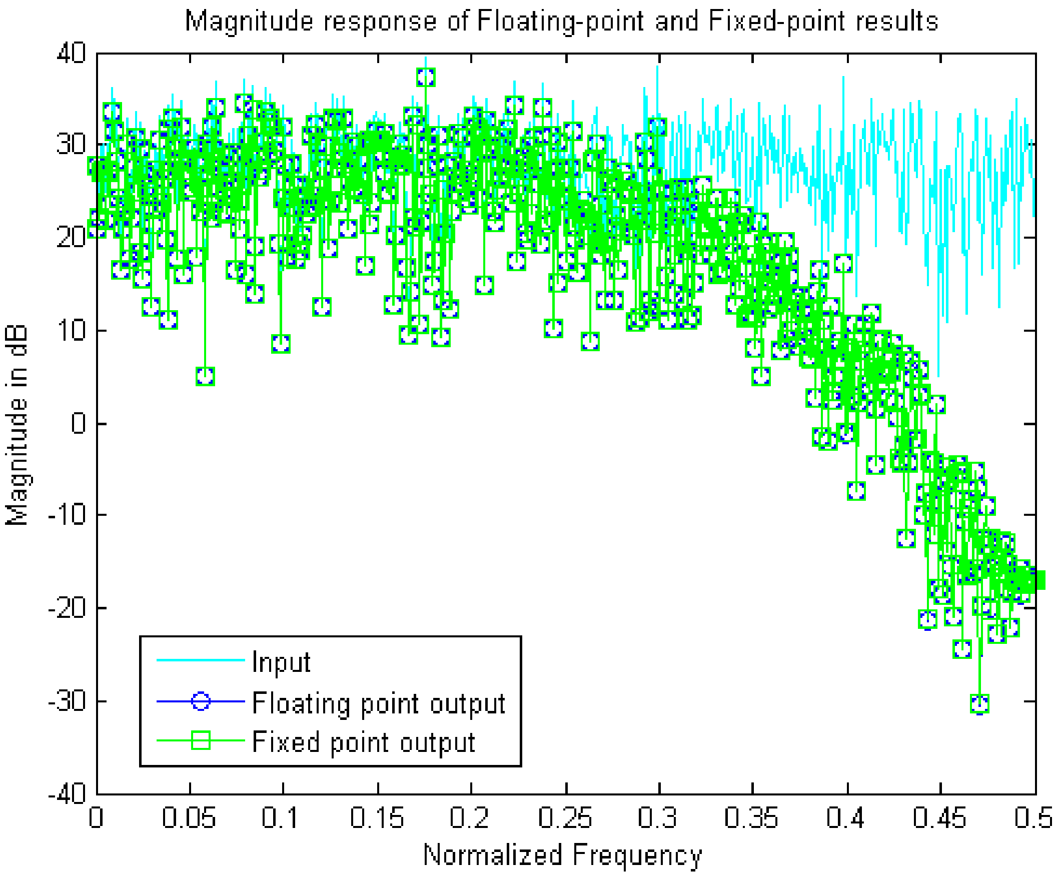
```
for k=1:length(x)
    y(k) =  b(1)*x(k) + z(1);
    z(1) = (b(2)*x(k) + z(2)) - a(2)*y(k);
    z(2) =  b(3)*x(k)         - a(3)*y(k);
end
% Reset the logging mode.
fp.LoggingMode = default_loggingmode;
```

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.

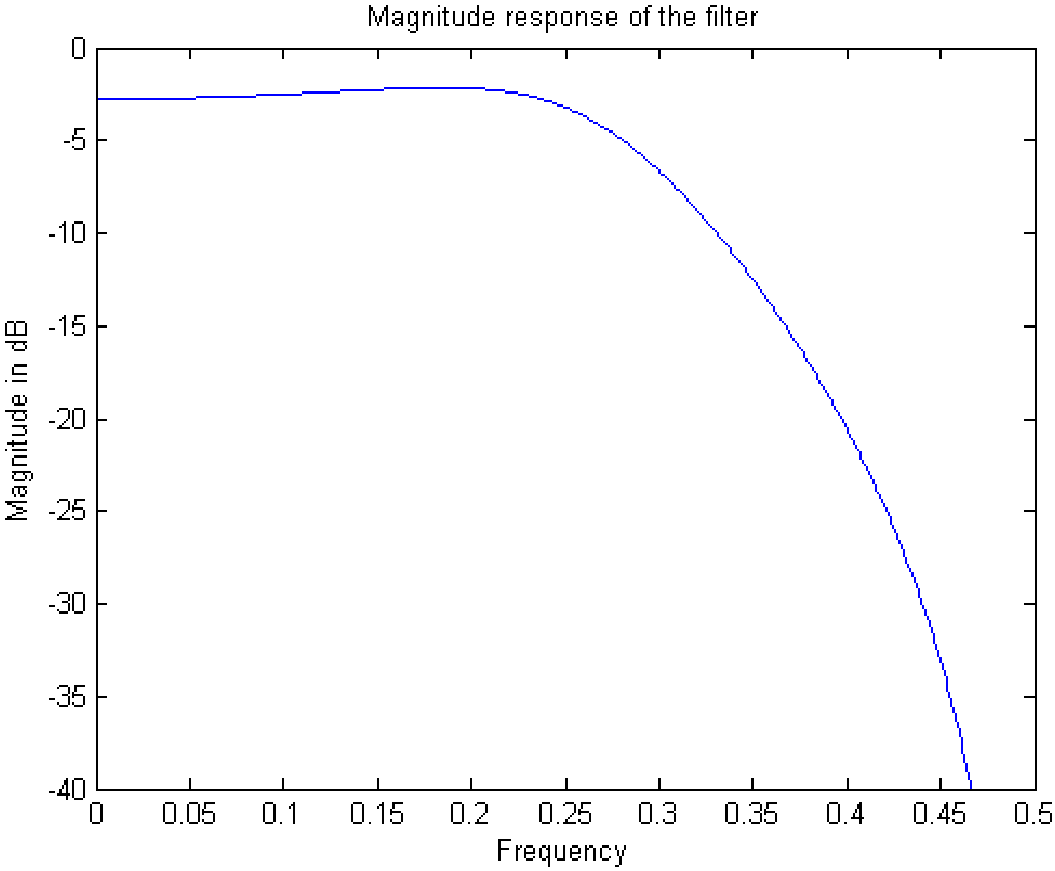
Compare and Plot the Floating-Point and Fixed-Point Results

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','Bes
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');
```

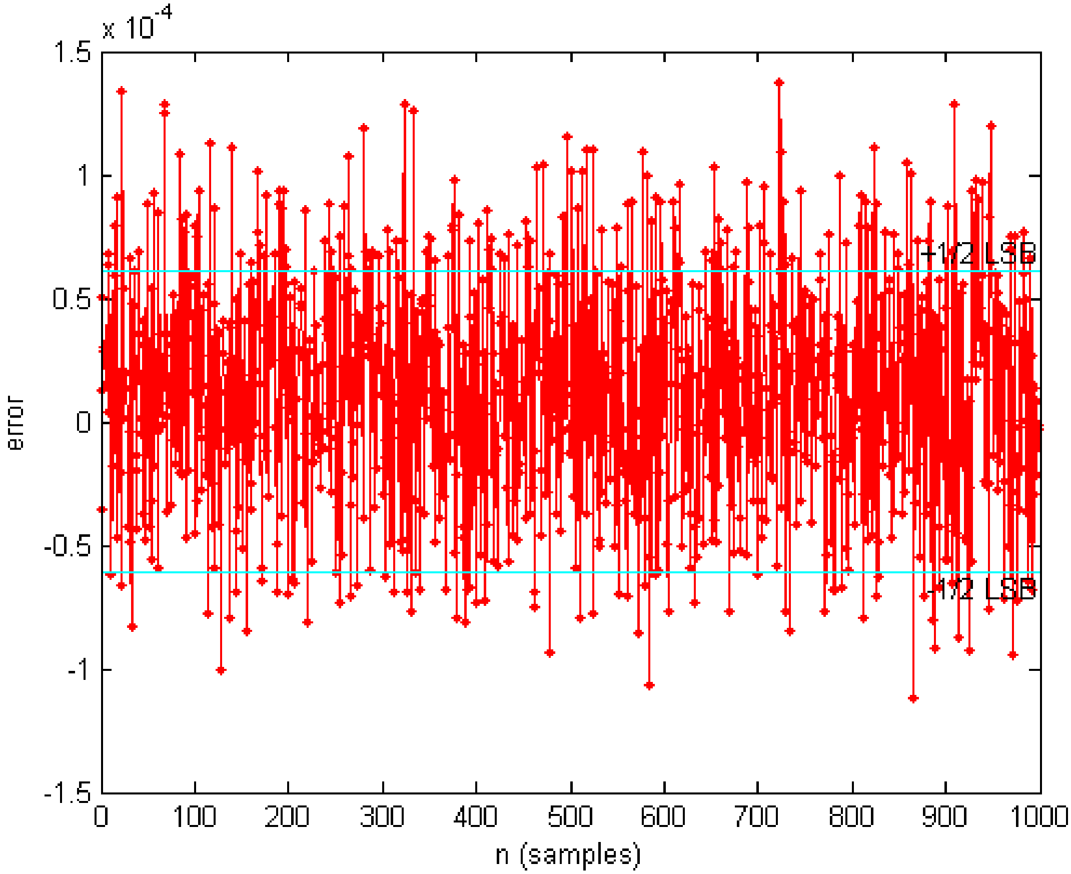


Notice that the high frequencies in the input signal are attenuated by the low-pass filter which is the expected behavior.

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
text(n(end),-e/2,'-1/2 LSB','HorizontalAlignment','right','VerticalAlignmen
xlabel('n (samples)'); ylabel('error')
```



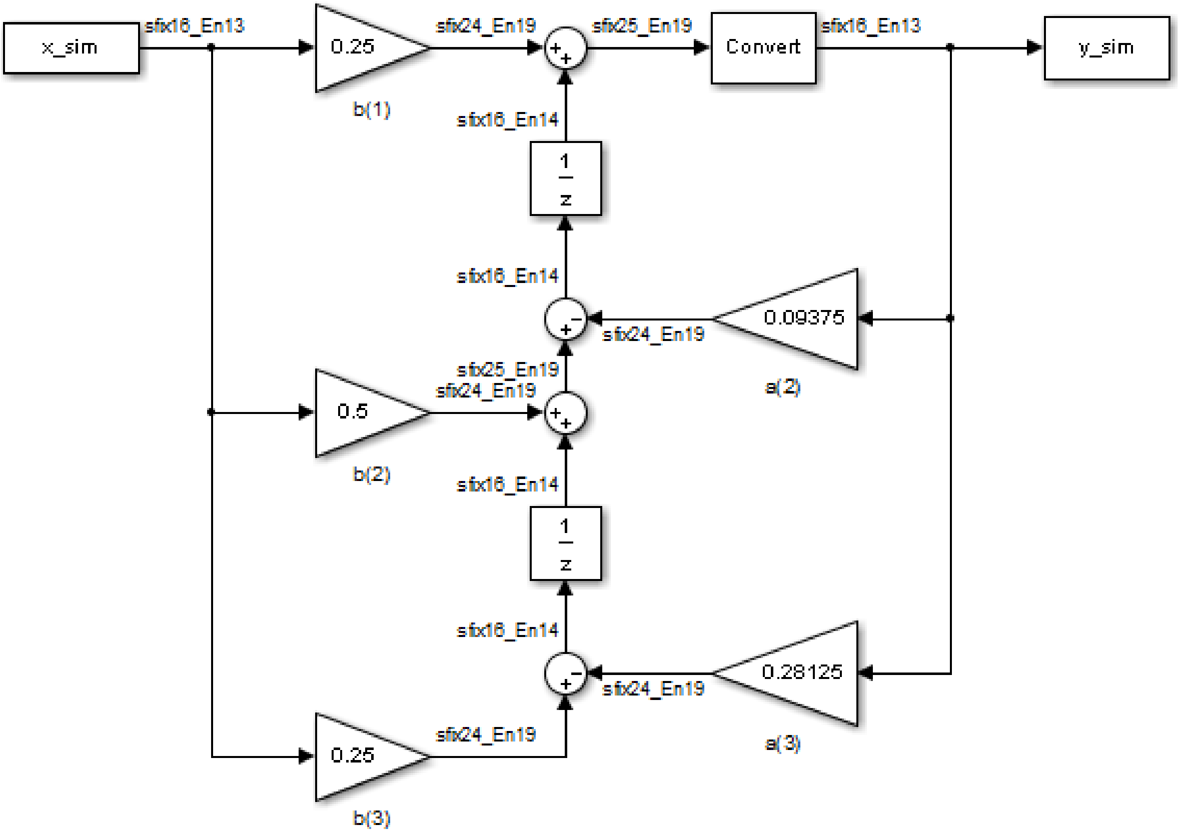
Simulink

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 =

     1
```



Copyright 2004-2010 The MathWorks, Inc.

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

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

CORDIC Rotation Computation Mode

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

In rotation mode, the CORDIC equations are:

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

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

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

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

$i = 0, 1, ..., N - 1$, 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 `cordiccexp`, `cordicsincos`, `cordicsin`, and `cordiccos` functions use quadrant correction (including possible extra negation) after the CORDIC iterations are completed.

Understanding the `CORDICSINCOS` Sine and Cosine Code

Introduction

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

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

Initialization

The `cordicsincos` function performs the following initialization steps:

- The angle input look-up table inpLUT is set to 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(:) = accumpos(z, inpLUT(idx));
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
    else
        z(:) = accumneg(z, inpLUT(idx));
        x(:) = accumneg(x, ytmp);
        y(:) = accumpos(y, xtmp);
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end
```

Visualizing the Sine-Cosine Rotation Mode CORDIC Iterations

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

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

Example

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

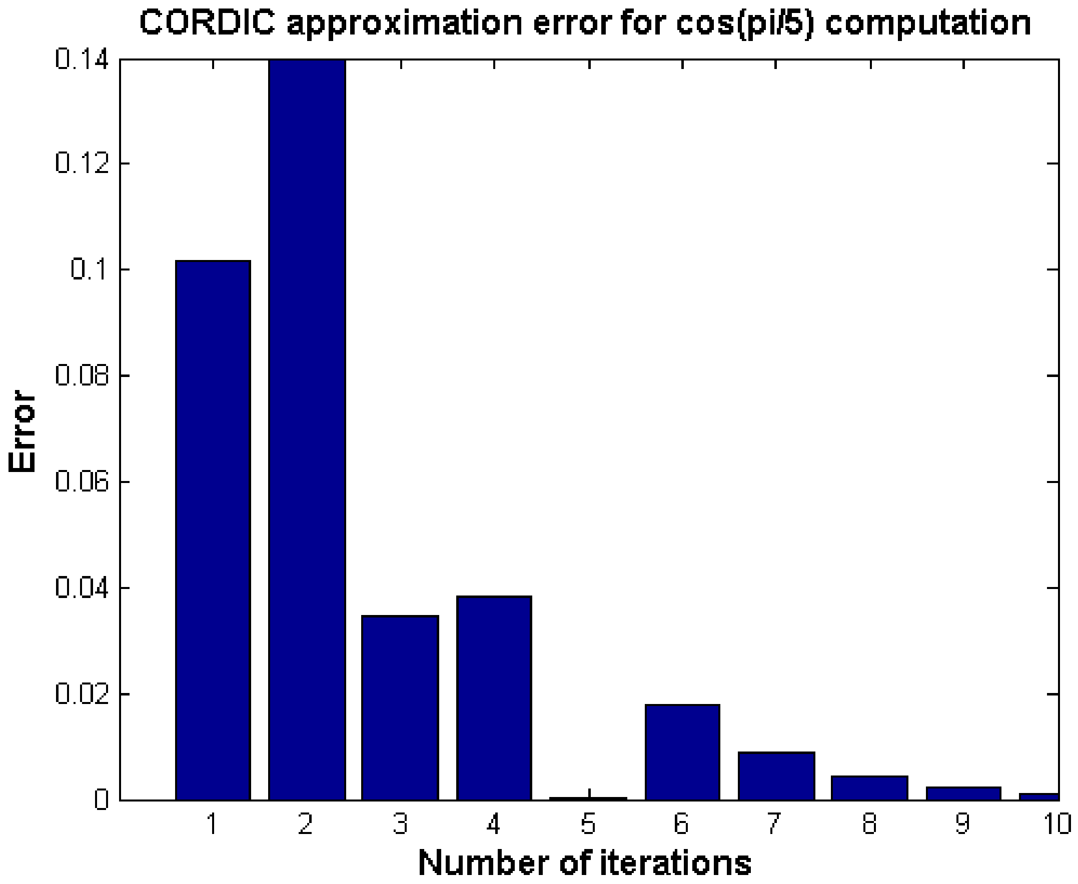
```
theta   = pi/5; % input angle in radians
niters  = 10;   % number of iterations
sinTh   = sin(theta); % reference result
cosTh   = cos(theta); % reference result
y_sin   = zeros(niters, 1);
sin_err = zeros(niters, 1);
x_cos   = zeros(niters, 1);
cos_err = zeros(niters, 1);
fprintf('\n\nNITERS \tERROR\n');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

```
NITERS  ERROR
```

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

Plot the CORDIC approximation error on a bar graph

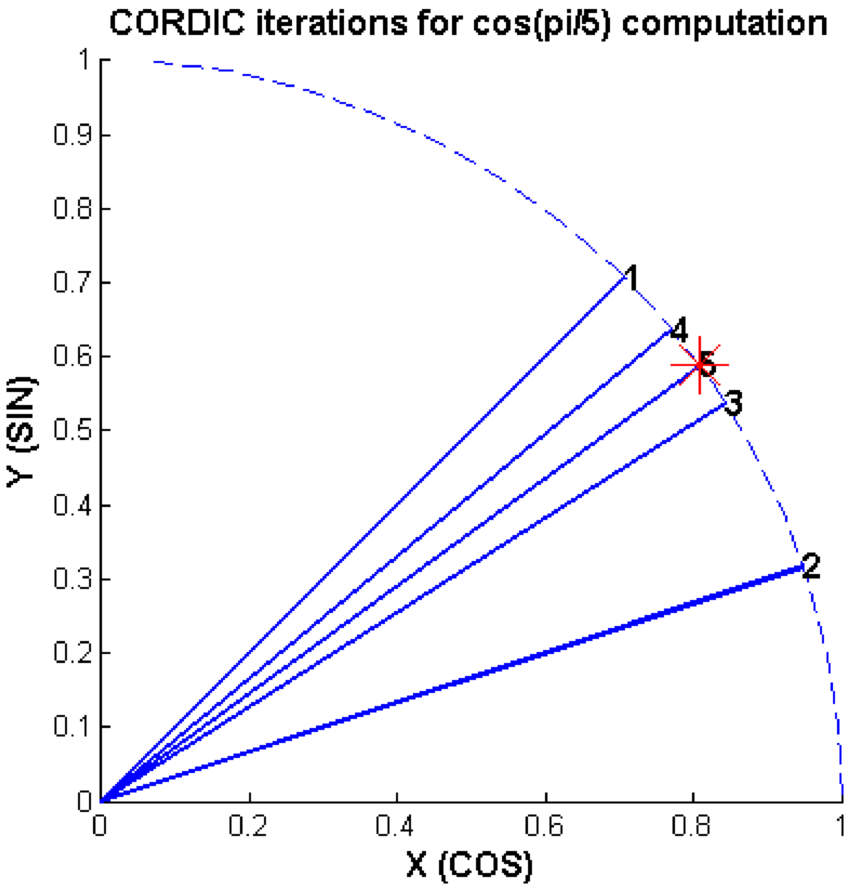
```
figure(1); clf;
bar(1:niters, cos_err(1:niters));
xlabel('Number of iterations','fontsize',12,'fontweight','b');
ylabel('Error','fontsize',12,'fontweight','b');
title('CORDIC approximation error for cos(pi/5) computation',...
    'fontsize',12,'fontweight','b');
axis([0 niters 0 0.14]);
```



Plot the X-Y results for 5 iterations

```
Niter2Draw = 5;
figure(2), clf, hold on
plot(cos(0:0.1:pi/2), sin(0:0.1:pi/2), 'b--'); % semi-circle
for i=1:Niter2Draw
    plot([0 x_cos(i)],[0 y_sin(i)], 'LineWidth', 2); % CORDIC iteration res
    text(x_cos(i),y_sin(i),int2str(i),'fontsize',12,'fontweight','b');
end
plot(cos(theta), sin(theta), 'r*', 'MarkerSize', 20); % IDEAL result
```

```
xlabel('X (COS)','fontsize',12,'fontweight','b')
ylabel('Y (SIN)','fontsize',12,'fontweight','b')
title('CORDIC iterations for cos(pi/5) computation',...
    'fontsize',12,'fontweight','b')
axis equal;
axis square;
```



Computing Fixed-point Sine with `cordicsin`

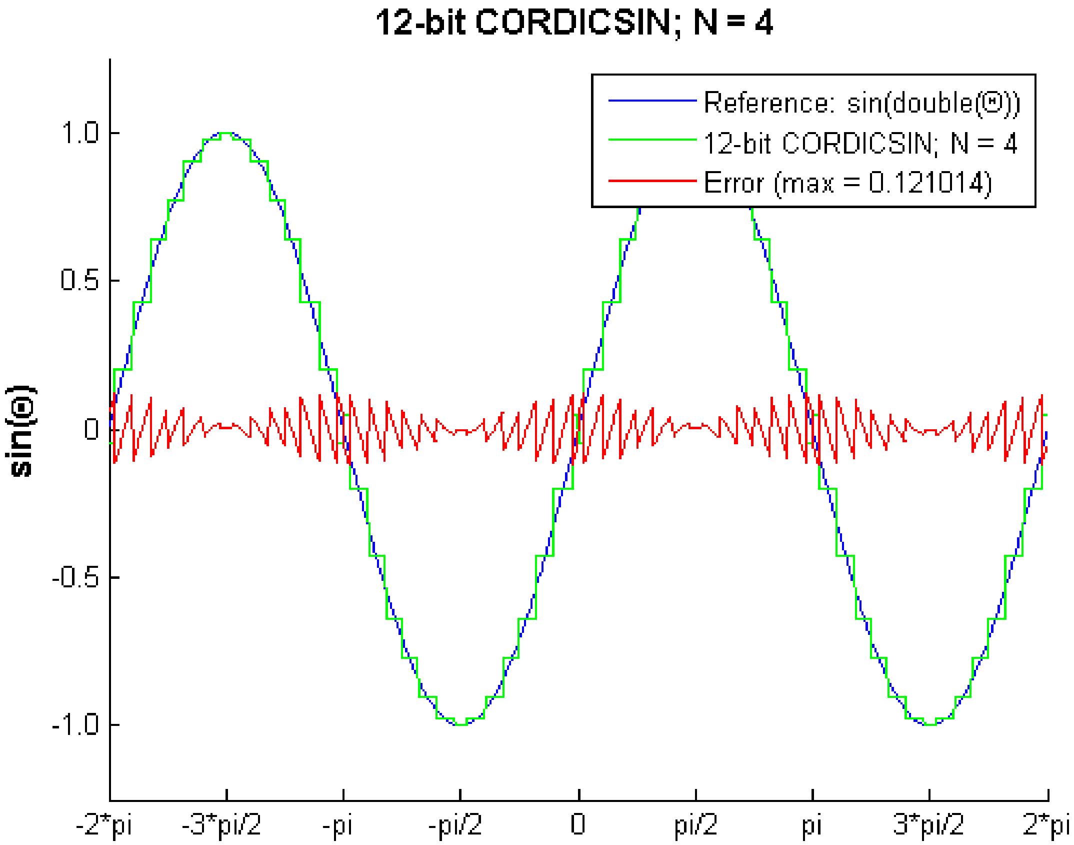
Create 1024 points between [-2*pi, 2*pi)

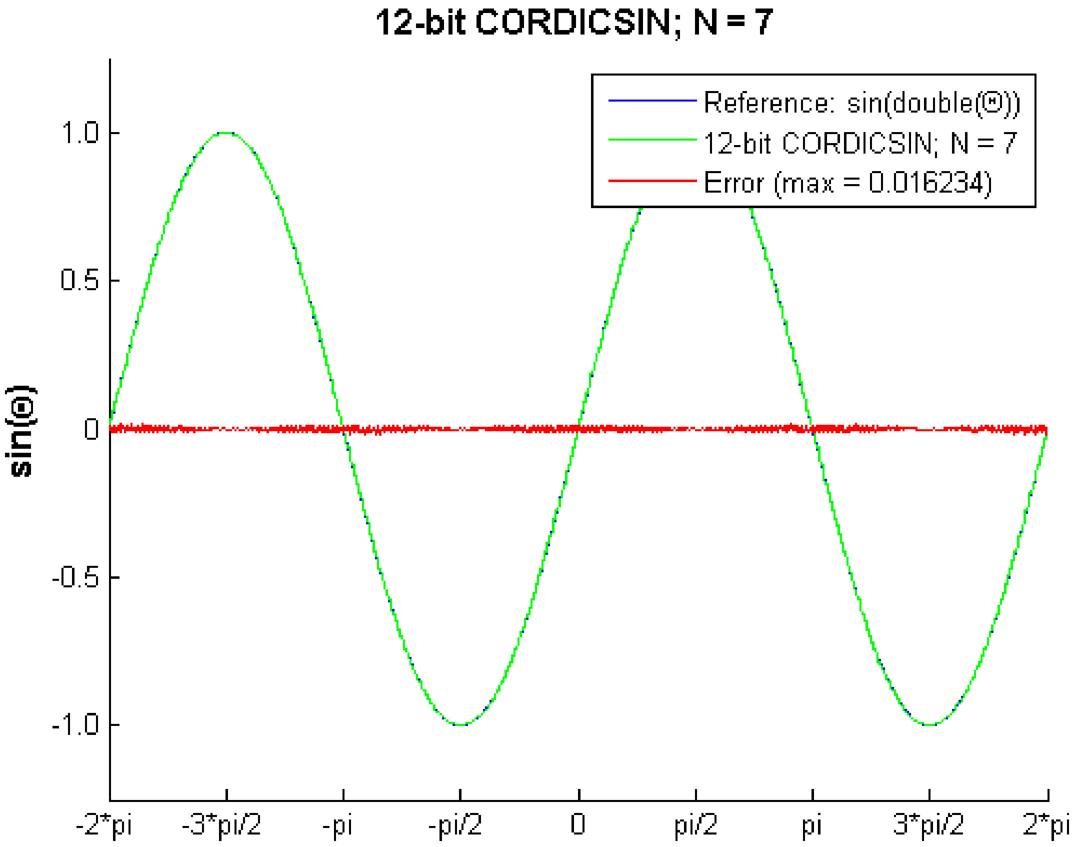
```
stepSize = pi/256;
thRadDbl = (-2*pi):stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDbl, 12);     % signed, 12-bit fixed-point values
sinThRef = sin(double(thRadFxp)); % reference results
```

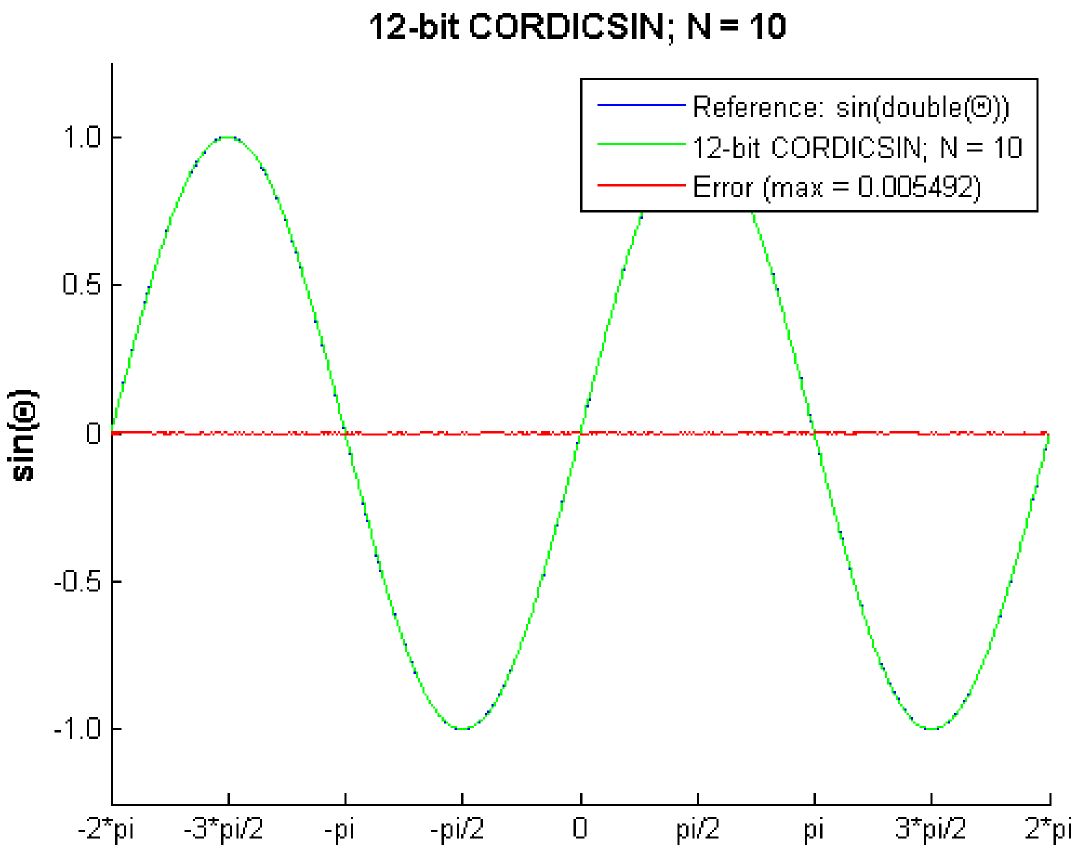
Compare fixed-point CORDIC vs. double-precision trig function results

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
```

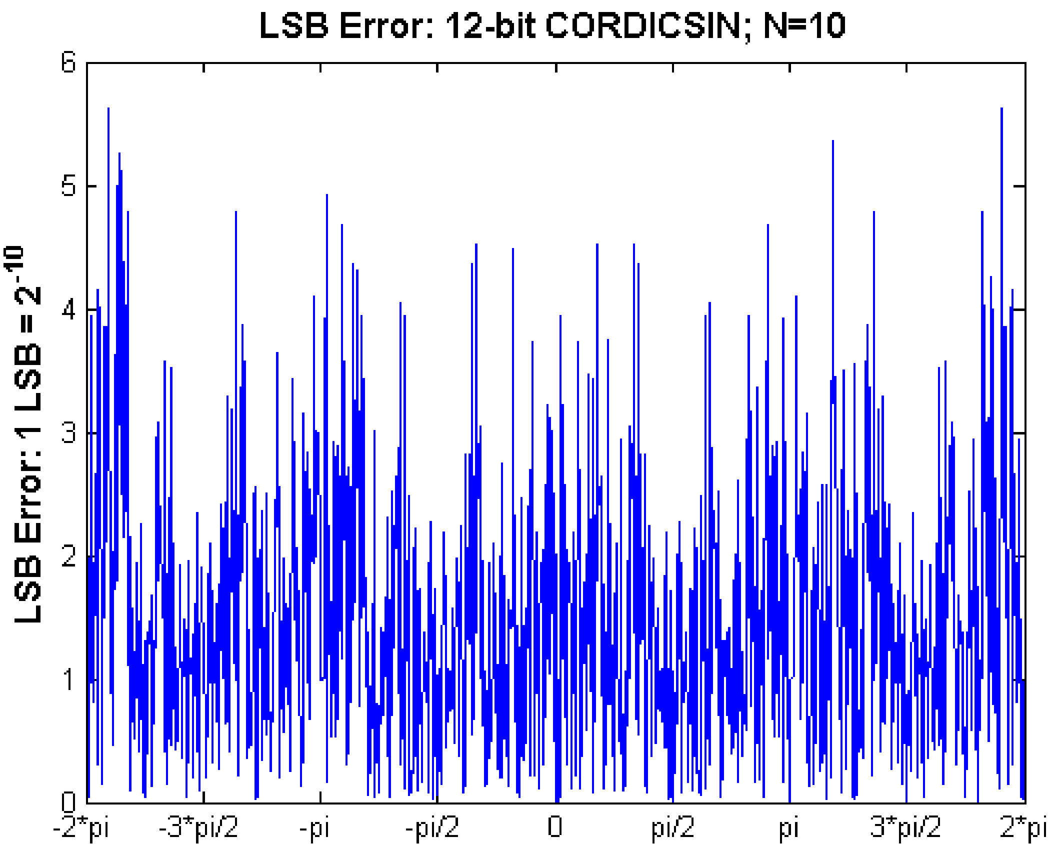

12-bit CORDICSIN; N = 4
Reference: sin(double(θ))
12-bit CORDICSIN; N = 4
Error (max = 0.121014)
sin(θ)
1.0
0.5
0
-0.5
-1.0
-2*pi
-3*pi/2
-pi
-pi/2
0
pi/2
pi
3*pi/2
2*pi


12-bit CORDICSIN; N = 7
Reference: sin(double(θ))
12-bit CORDICSIN; N = 7
Error (max = 0.016234)
sin(θ)
1.0
0.5
0
-0.5
-1.0
-2*pi
-3*pi/2
-pi
-pi/2
0
pi/2
pi
3*pi/2
2*pi



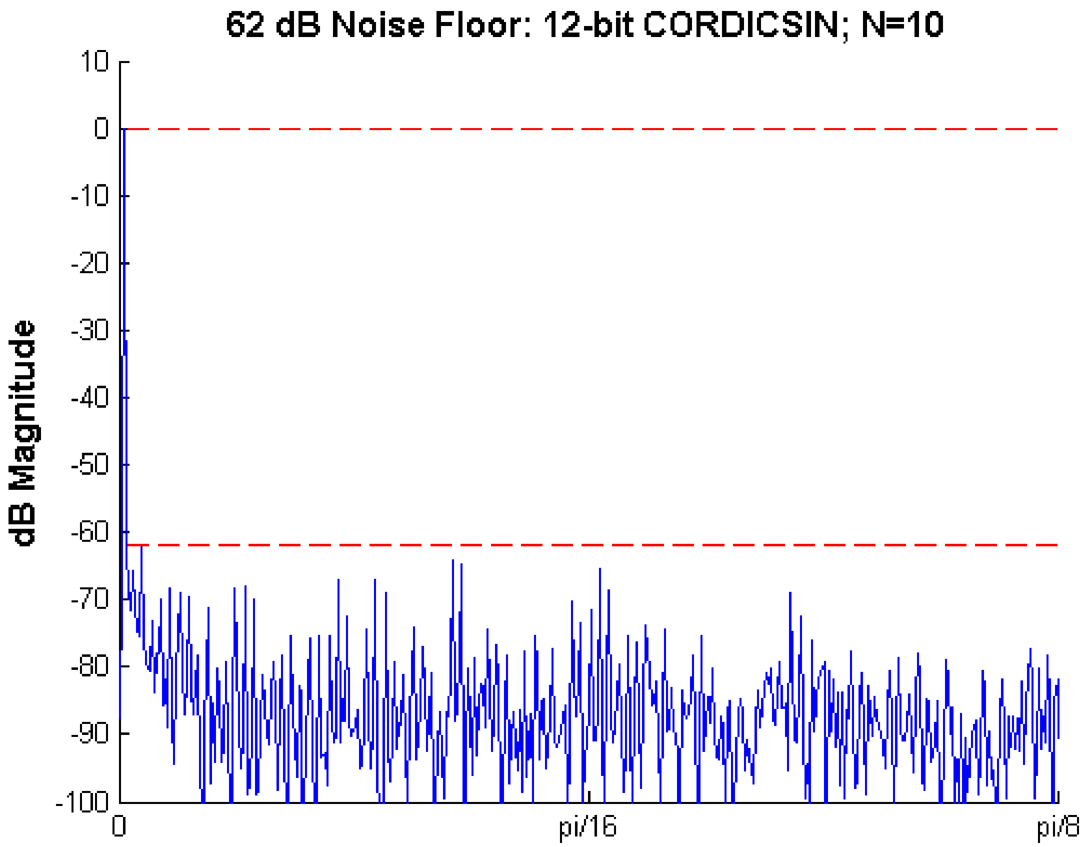
Compute the LSB Error for N = 10

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



Compute Noise Floor

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



Accelerating the Fixed-Point CORDICSINCOS Function with FIACCEL

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

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

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

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

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

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

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

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

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

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

```
cordicsincos_mex(thRadFxp);
telapsed_MEXcordicsincos = toc(tstart);
```

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

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

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

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

Calculating `SIN` and `COS` Using Lookup Tables

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

Single Lookup Table Based Approach

The `sin` and `cos` methods of the `fi` object in the Fixed-Point 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);
quarterSinDblFltPtVals  = (sin(2*pi*((0:63) ./ 256)))';
endpointQuantized_Plus1 = 1.0 - double(eps(fi(0,tblValsNT)));

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

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

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

Overview of Algorithm Implementation

The implementation of the Fixed-Point 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, 2pi]. For this purpose, a modulo-2pi operation is performed to obtain the fixed-point input value inpValInRange in the range [0, 2pi] and cast to in the best precision binary point scaled unsigned 16-bit fixed-point type numerictype(0,16,13):

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

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

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

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

```
normConst_NT = numerictype(0,32,31);
normConstant = fi(65536/51472, normConst_NT);
fullScaleIdx = normConstant * idxUFIX16;
idxUFIX16(:) = fullScaleIdx;
```

The top 8 most significant bits (MSBs) of this full-scale unsigned 16-bit index idxUFIX16 are used to directly index into the 8-bit sine lookup table. Two table lookups are performed, one at the computed table index location lutValBelow, and one at the next index location lutValAbove:

```
idxUint8MSBs = uint8(storedInteger(bitsliceget(idxUFIX16, 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 idxUFIX16 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(idxUFIX16,8,1), rFracNT);
rFraction    = idxFrac8LSBs;
```

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

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

Example

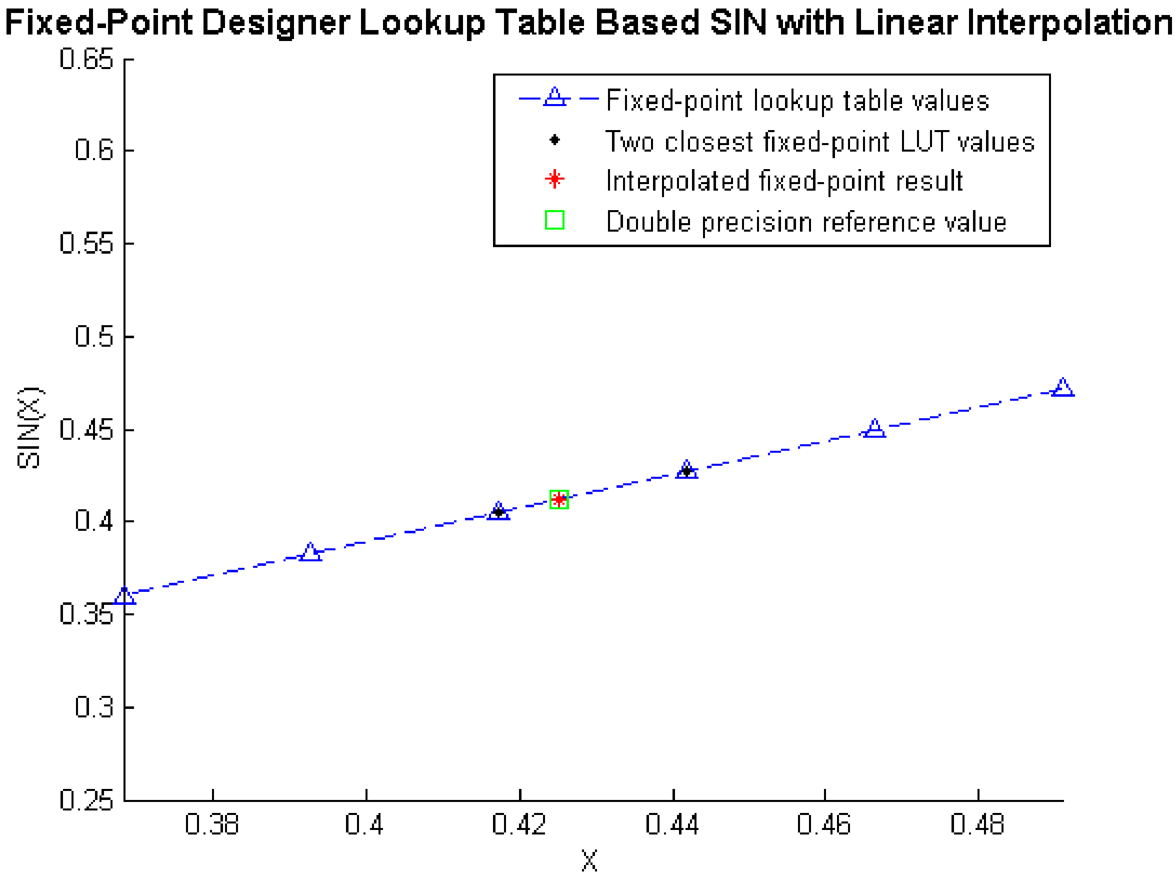
Using the above algorithm, here is an example of the lookup table and linear interpolation process used to compute the value of `SIN` for a fixed-point input `inpValInRange = 0.425` radians:

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

% Normalize its stored integer to get full-scale unsigned 16-bit integer in
idxUFIX16     = fi(storedInteger(inpValInRange), numerictype(0,16,0));
normConst_NT  = numerictype(0,32,31);
normConstant  = fi(65536/51472, normConst_NT);
fullScaleIdx  = normConstant * idxUFIX16;
idxUFIX16(:)  = fullScaleIdx;

% Do two table lookups using unsigned 8-bit integer index (i.e., 8 MSBs)
idxUint8MSBs  = uint8(storedInteger(bitsliceget(idxUFIX16, 16, 9)));
zeroBasedIdx  = int16(idxUint8MSBs);          % zero-based table index valu
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 remai
rFracNT       = numerictype(0,8,8); % fractional remainder data type
idxFrac8LSBs  = reinterpretcast(bitsliceget(idxUFIX16,8,1), 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 Interpolatio
    'fontsize',12,'fontweight','b');
```



Computing Fixed-point Sine Using `SIN`

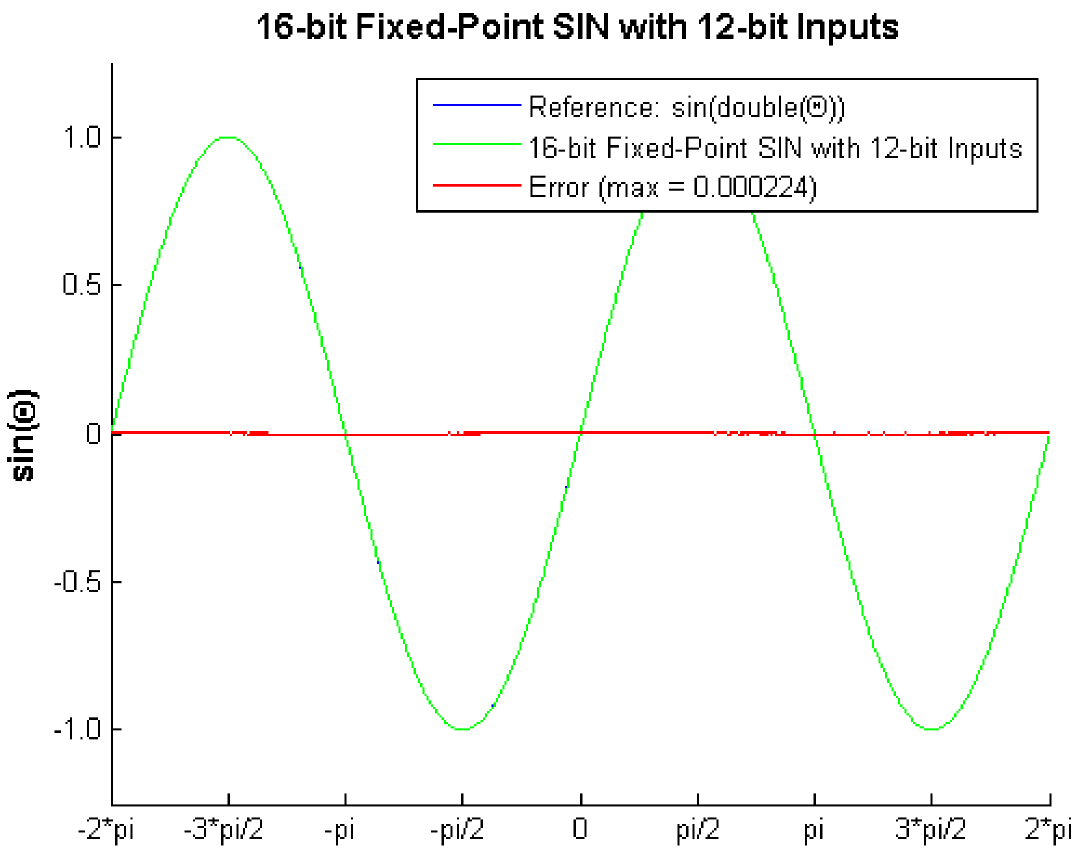
Create 1024 points between [-2*pi, 2*pi)

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

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

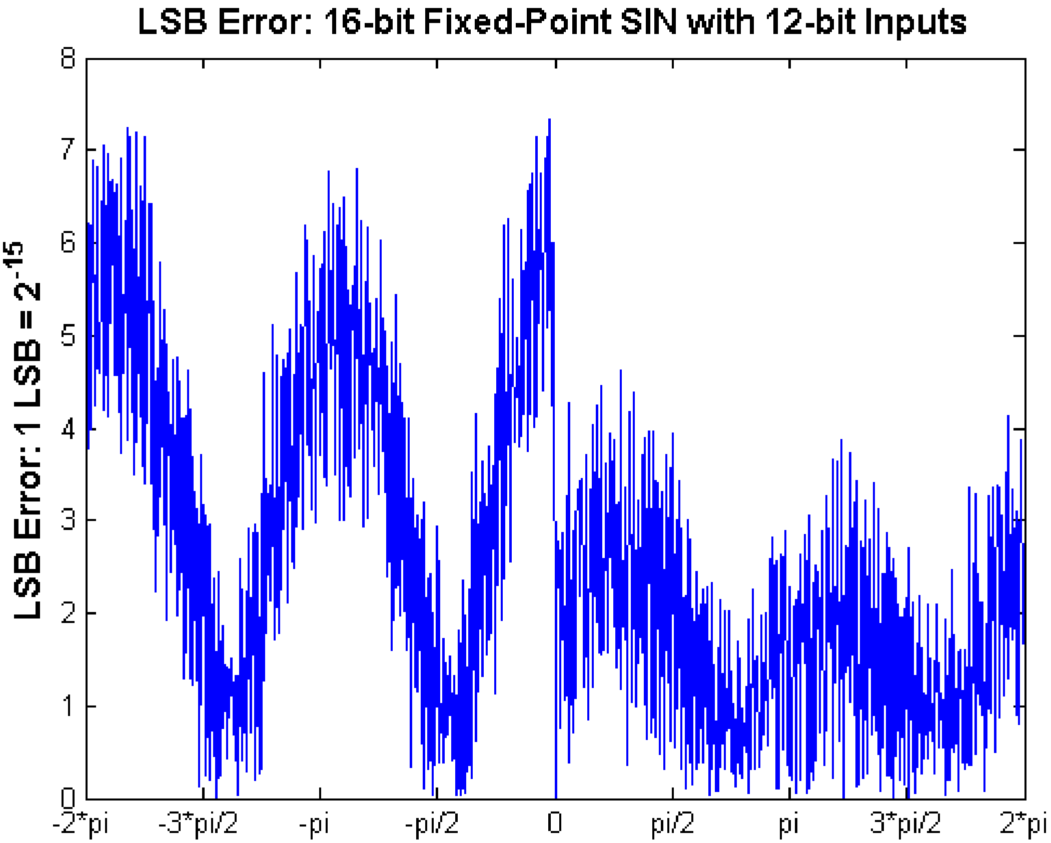
```
fxpSinTh  = sin(thRadFxp); % fixed-point results
```

```
sinThRef  = sin(double(thRadFxp)); % reference results
errSinRef = sinThRef - double(fxpSinTh);
figure; hold on; axis([-2*pi 2*pi -1.25 1.25]);
plot(thRadFxp, sinThRef,  'b');
plot(thRadFxp, fxpSinTh,  'g');
plot(thRadFxp, errSinRef, 'r');
ylabel('sin(\Theta)','fontsize',12,'fontweight','b');
set(gca,'XTick',-2*pi:pi/2:2*pi);
set(gca,'XTickLabel',...
    {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
    '0', 'pi/2', 'pi', '3*pi/2','2*pi'});
set(gca,'YTick',-1:0.5:1);
set(gca,'YTickLabel',{'-1.0','-0.5','0','0.5','1.0'});
ref_str = 'Reference: sin(double(\Theta))';
fxp_str = sprintf('16-bit Fixed-Point SIN with 12-bit Inputs');
err_str = sprintf('Error (max = %f)', max(abs(errSinRef)));
legend(ref_str, fxp_str, err_str);
title(fxp_str,'fontsize',12,'fontweight','b');
```



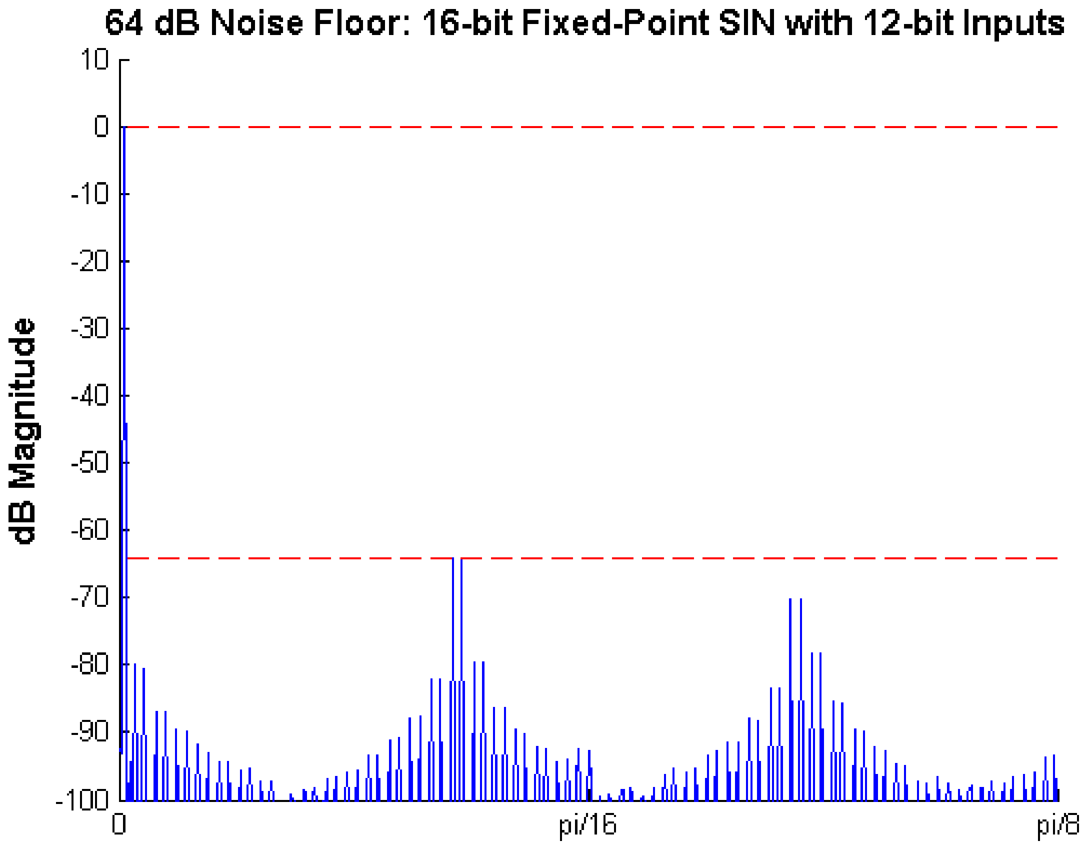
Compute the LSB Error

```
figure;
fracLen = fxpSinTh.FractionLength;
plot(thRadFxp, abs(errSinRef) * pow2(fracLen));
set(gca,'XTick',-2*pi:pi/2:2*pi);
set(gca,'XTickLabel',...
    {'-2*pi', '-3*pi/2', '-pi', '-pi/2', ...
    '0', 'pi/2', 'pi', '3*pi/2','2*pi'});
ylabel(sprintf('LSB Error: 1 LSB = 2^{-%d}',fracLen),'fontsize',12,'fontwei
title('LSB Error: 16-bit Fixed-Point SIN with 12-bit Inputs','fontsize',12,
axis([-2*pi 2*pi 0 8]);
```



Compute Noise Floor

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



Comparing the Costs of the Fixed-Point Approximation Algorithms

The fixed-point CORDIC algorithm requires the following operations:

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

The simplified single lookup table algorithm with nearest-neighbor linear interpolatiom 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:

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

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

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

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

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

As N approaches $+\infty$:

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

$$y_N = 0$$

$$z_N = z_0 + atan(y_0/x_0)$$

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

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 $z_N \approx atan(y_0/x_0)$.

Understanding the `CORDICATAN2` Code

Introduction

The `cordicatan2` function computes the four quadrant arctangent of the elements of x and y, where $-\pi \leq ATAN2(y, x) \leq +\pi$. `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 $z_N \approx atan(y_0/x_0)$

Shared Fixed-Point and Floating-Point CORDIC Kernel Code

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

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

Visualizing the Vectoring Mode CORDIC Iterations

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

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

Example

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

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

```
origFormat = get(0, 'format'); % store original format setting;
                               % restore this setting at the end.
format short
%
theta = 43*pi/180;  % input angle in radians
Niter = 10;         % number of iterations
inX   = cos(theta); % x coordinate of the input vector
inY   = sin(theta); % y coordinate of the input vector
%
% pre-allocate memories
zf = zeros(1, Niter);
xf = [inX, zeros(1, Niter)];
yf = [inY, zeros(1, Niter)];
angleLUT = atan(2.^-(0:Niter-1)); % pre-calculate the angle lookup table
%
% Call CORDIC vectoring kernel algorithm
for k = 1:Niter
   [xf(k+1), yf(k+1), zf(k)] = fixed.internal.cordic_vectoring_kernel_priva
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, E
        [(1:Niter); angleAccumulator(:)'; angleError(:)';log2(abs(zf(:)'-th
```

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

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

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

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

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

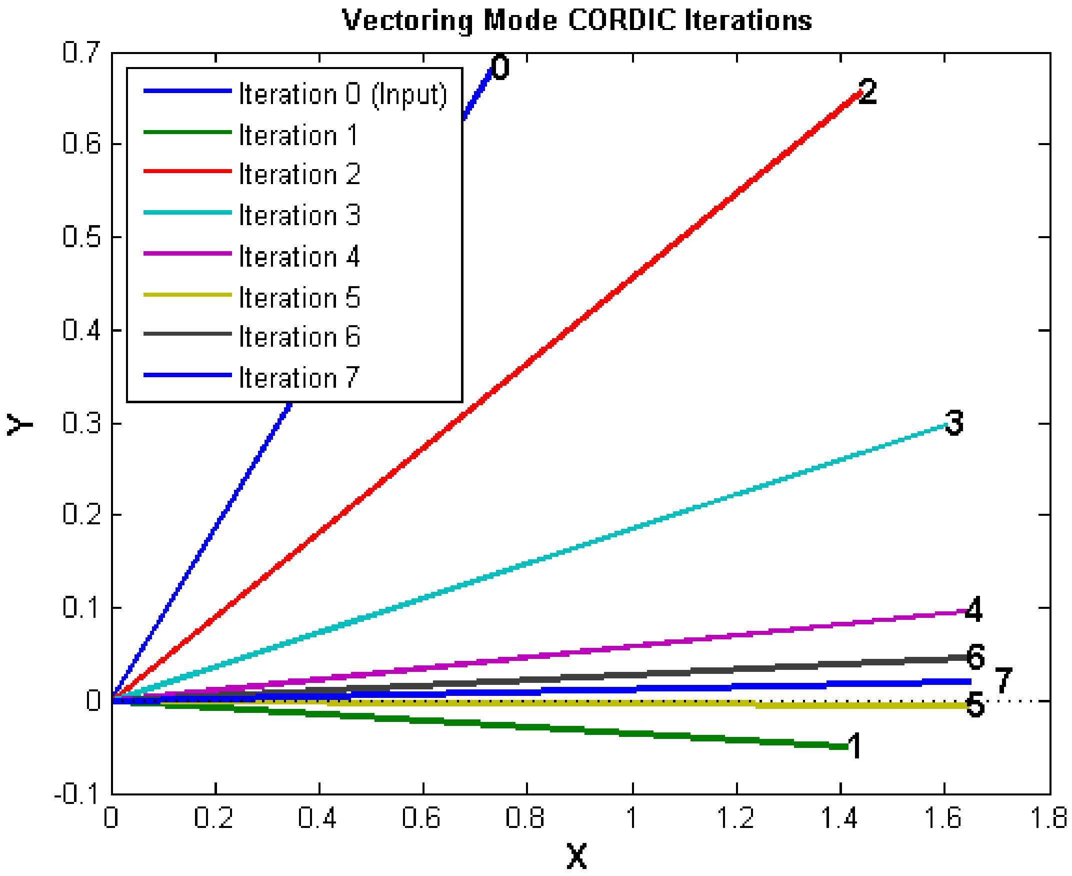
Note that y_n approaches 0, and x_n approaches $A_n\sqrt{x_0^2 + y_0^2} = A_n$, because $\sqrt{x_0^2 + y_0^2} = 1$.

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

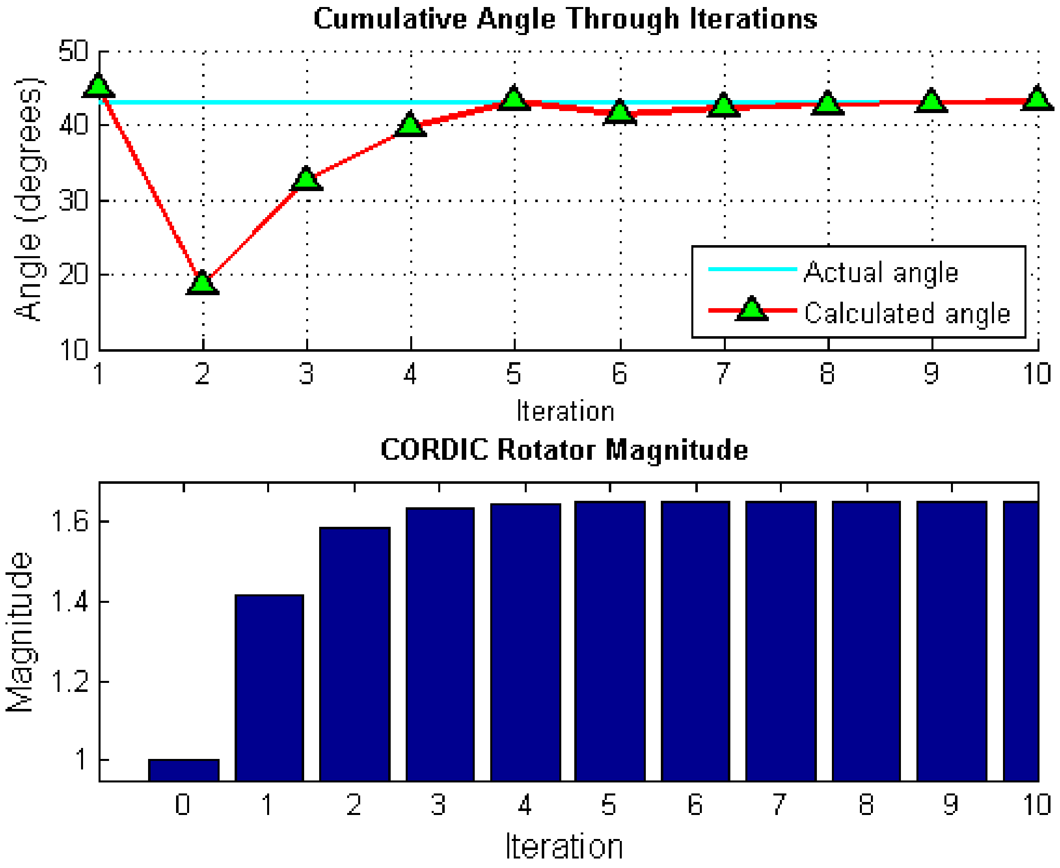
```
y_n =

   -0.0018
```

```
x_n = xf(end)
x_n =

    1.6468
```

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



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



Performing Overall Error Analysis of the CORDIC Algorithm

The overall error consists of two parts:

1. The algorithmic error that results from the CORDIC rotation angle being represented by a finite number of basic angles.

2. The quantization or rounding error that results from the finite precision representation of the angle lookup table, and from the finite precision arithmetic used in fixed-point operations.

Calculate the CORDIC Algorithmic Error

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

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

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

```
cordic_algErr_real_world_value =

     4.753112306290497e-04
```

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

```
cordic_algErr_bits = log2(cordic_algErr_real_world_value)
```

```
cordic_algErr_bits =

 -11.038839889583048
```

Relationship Between Number of Iterations and Precision

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

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

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

if i is equal to the word length of y and x0, then `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]);
```

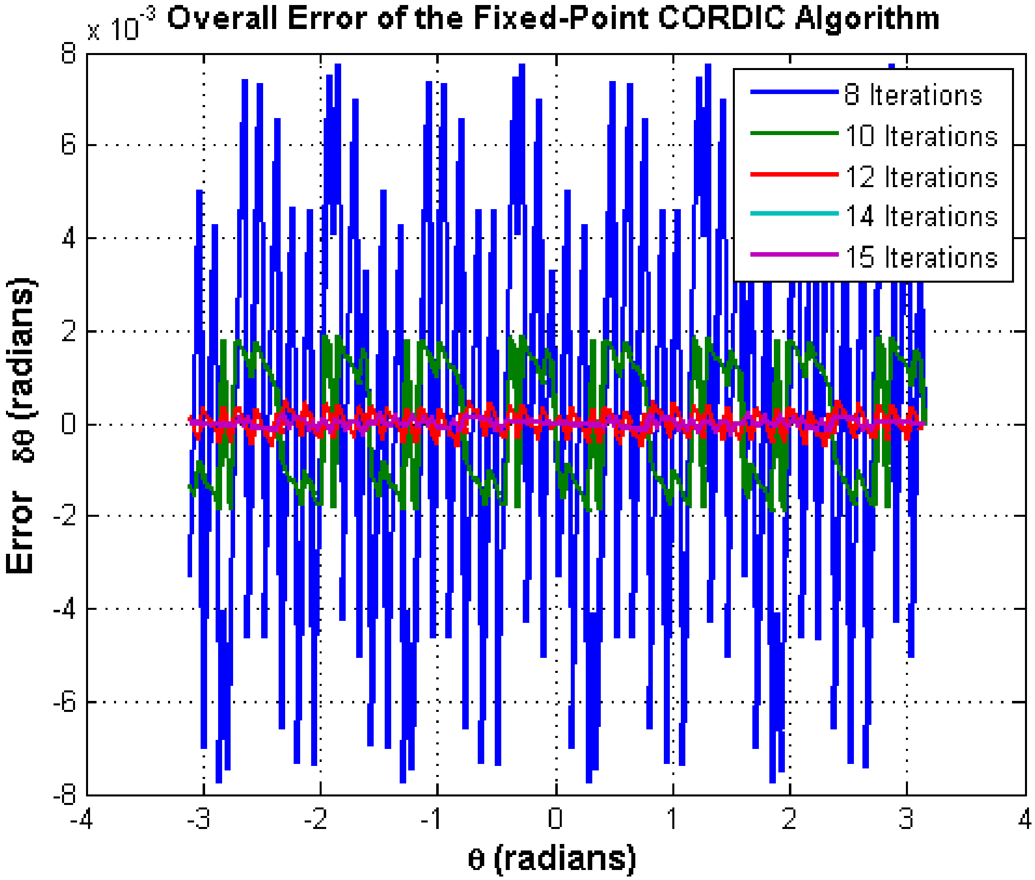
```
Iterations:  8, Max error in real-world-value: 0.00773633
```

```
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_cordicE
```

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

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



Accelerating the Fixed-Point `CORDICATAN2` Algorithm Using `FIACCEL`

You can generate a MEX function from MATLAB code using the MATLAB 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
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) = 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:

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

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

$$atan(x) = 0.970562748477141 * x - 0.189514164974601 * x^3.$$

The 5th order Chebyshev polynomial approximation is

$$atan(x) = 0.994949366116654 * x - 0.287060635532652 * x^3 + 0.078037176446441 * x^5.$$

The 7th order Chebyshev polynomial approximation is

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

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

Comparing the Algorithmic Error of the CORDIC and Polynomial Approximation Algorithms

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

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

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

```
constA3 = [0.970562748477141, -0.189514164974601]; % 3rd order
constA5 = [0.994949366116654,-0.287060635532652,0.078037176446441]; % 5th o
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 polynomia
zfltp5  = fidemo.poly_atan2(inYflt,inXflt,5,constA5); % 5th order polynomia
zfltp7  = fidemo.poly_atan2(inYflt,inXflt,7,constA7); % 7th order polynomia
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\
    [[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-worl
    [3:2:7; max_polyAlgErr(:)']);
```

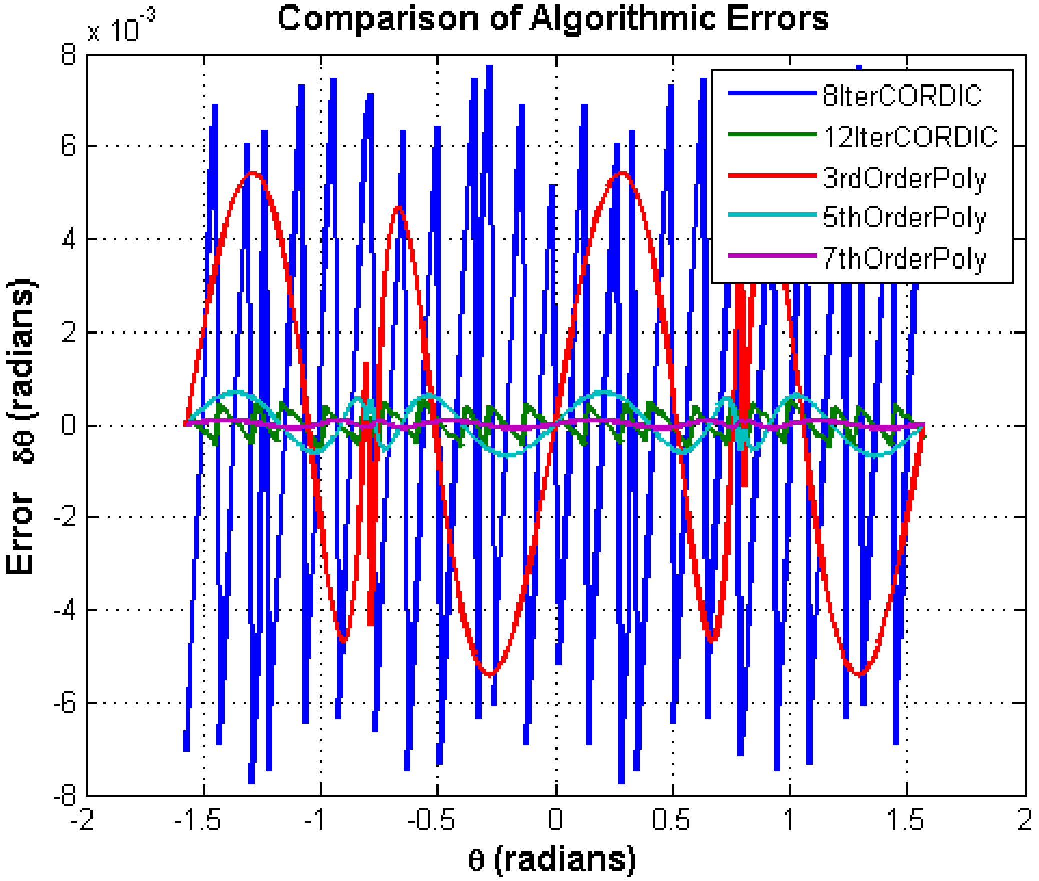
```
Order: 3, Polynomial approximation algorithmic error in real-world-value: O
Order: 5, Polynomial approximation algorithmic error in real-world-value: O
Order: 7, Polynomial approximation algorithmic error in real-world-value: 9
```

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

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

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

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



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

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

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

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

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

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

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

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

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

tmp2 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
tmp3 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodStr);
```

```
tmp2(:) = tmp*tmp;  % (y/x)^2
tmp3(:) = tmp2*tmp; % (y/x)^3

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

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

    if N == 7
        tmp7 = fi(0, 'numerictype',Tx, 'RoundingMethod', RoundingMethodS
        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', Rounding
end

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

Performing the Overall Error Analysis of the Polynomial Approximation Algorithm

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

Calculate the Quantization Error

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

Quantize the inputs and coefficients with convergent rounding:

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

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

```
ord    = 3:2:7; % using 3rd, 5th, 7th order polynomials
Tz     = numerictype(1, 16, 13); % output data type
zfix3p = fidemo.poly_atan2(inYfix,inXfix,ord(1),constAfix3,Tz,'Floor'); % 3
zfix5p = fidemo.poly_atan2(inYfix,inXfix,ord(2),constAfix5,Tz,'Floor'); % 5
zfix7p = fidemo.poly_atan2(inYfix,inXfix,ord(3),constAfix7,Tz,'Floor'); % 7
poly_quantErr = bsxfun(@minus, [zfltp3;zfltp5;zfltp7], double([zfix3p;zfix5
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
```

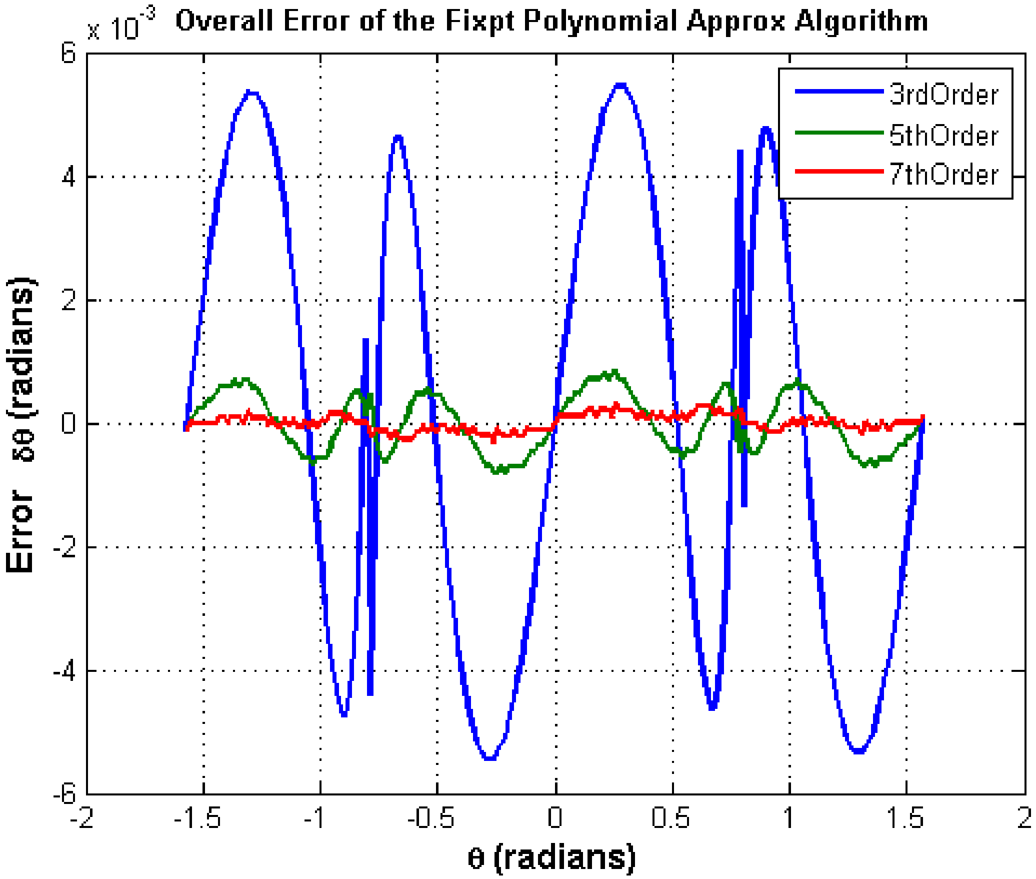
Calculate the Overall Error

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 = uint8(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 idxUFIX16 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(idxUFIX16,8,1), rFracNT);
rFraction    = idxFrac8LSBs;
```

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

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

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

Computing Fixed-point Argtangent Using ATAN2

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

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

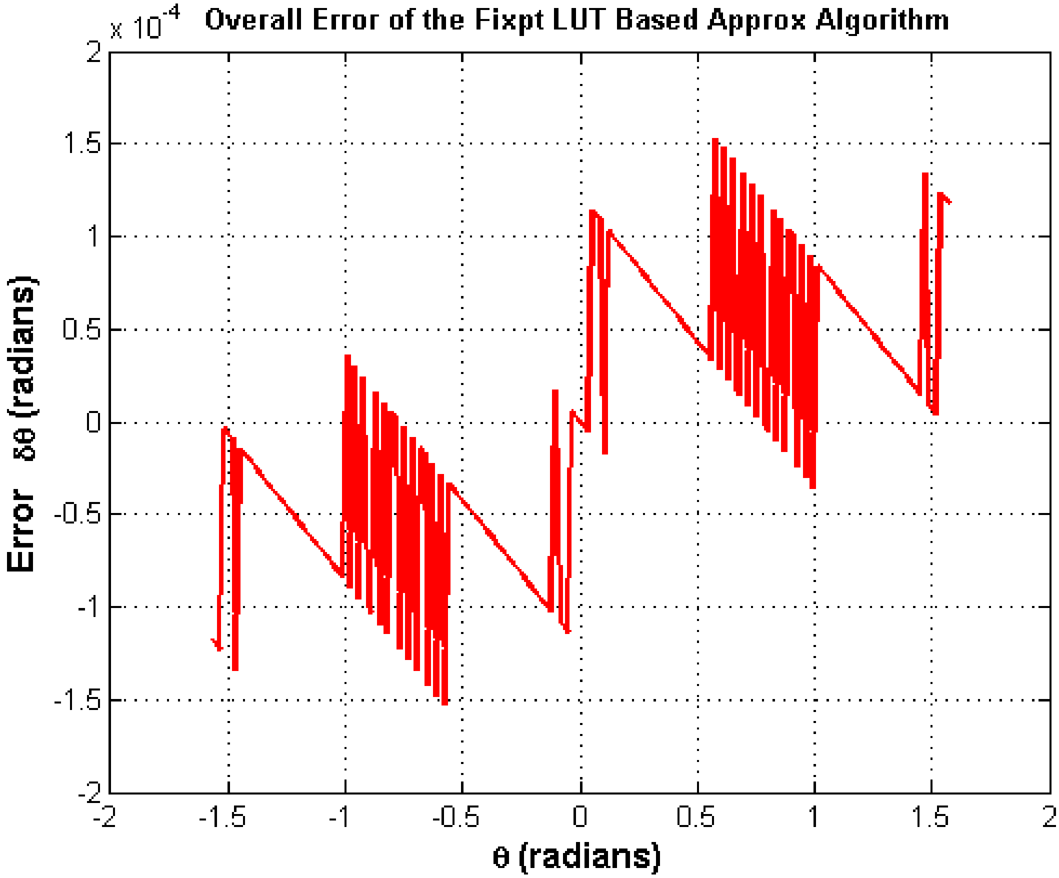
Calculate the Overall Error

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

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

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

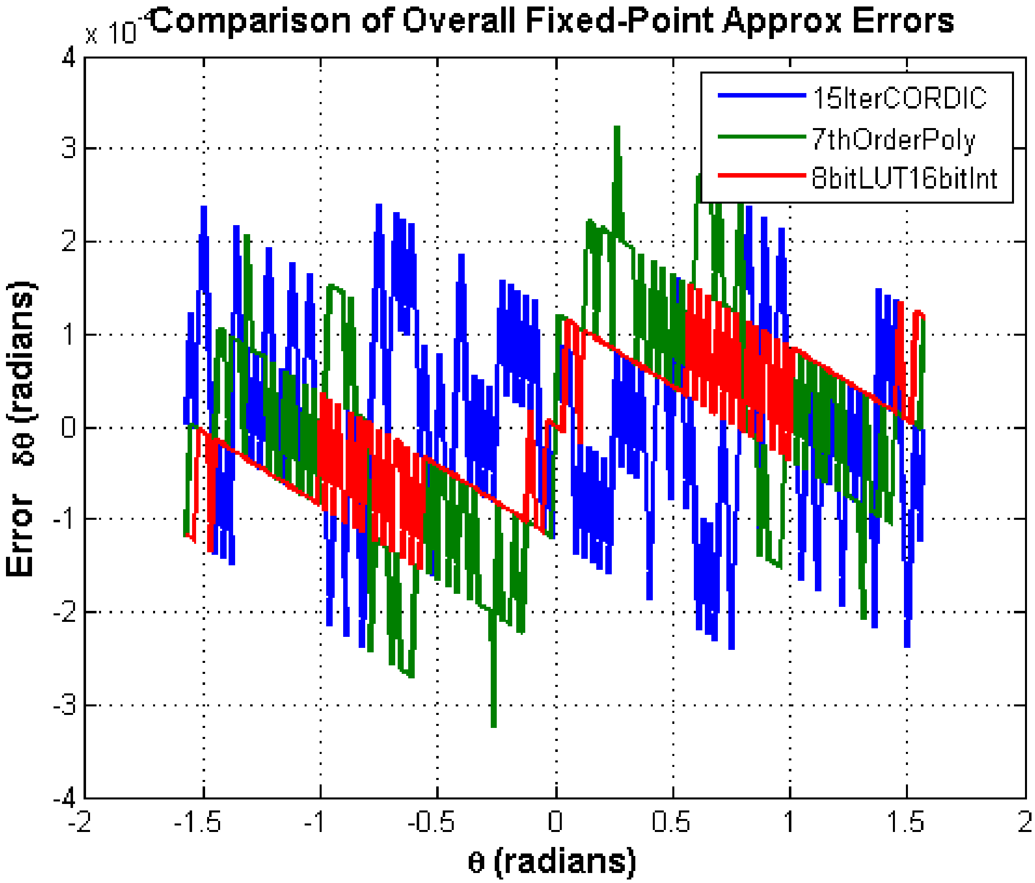
```
figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, lut_err)
```



Comparison of Overall Error Between the Fixed-Point Implementations

As was done previously, you can compute the overall error by comparing the fixed-point approximation(s) to the builtin `atan2` function. The ideal reference output is `zfltRef`.

```
zfixCDC15      = cordicatan2(inYfix, inXfix, 15);
cordic_15I_err = bsxfun(@minus, zfltRef, double(zfixCDC15));
poly_7p_err    = bsxfun(@minus, zfltRef, double(zfix7p));
figno = figno + 1;
fidemo.fixpt_atan2_demo_plot(figno, theta, cordic_15I_err, poly_7p_err, lut
```



Comparing the Costs of the Fixed-Point Approximation Algorithms

The fixed-point CORDIC algorithm requires the following operations:

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

The N-th order fixed-point Chebyshev polynomial approximation algorithm requires the following operations:

- 1 division
- (N+1) multiplications
- (N-1)/2 additions

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

- 1 division
- 2 table lookups
- 1 multiplication
- 2 additions

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

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

References

1. Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.

2. Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Compute Sine and Cosine Using CORDIC Rotation Kernel

This example shows how to compute sine and cosine using a CORDIC rotation kernel in MATLAB. CORDIC-based algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Introduction

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

The fixed-point CORDIC algorithm requires the following operations:

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

CORDIC Kernel Algorithm Using the Rotation Computation Mode

You can use a CORDIC rotation computing mode algorithm to calculate sine and cosine simultaneously, compute polar-to-cartesian conversions, and for other operations. In the rotation mode, the vector magnitude and an angle of rotation are known and the coordinate (X-Y) components are computed after rotation.

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

In rotation mode, the CORDIC equations are:

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

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

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

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

$i = 0, 1, ..., N-1$, 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 `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]).

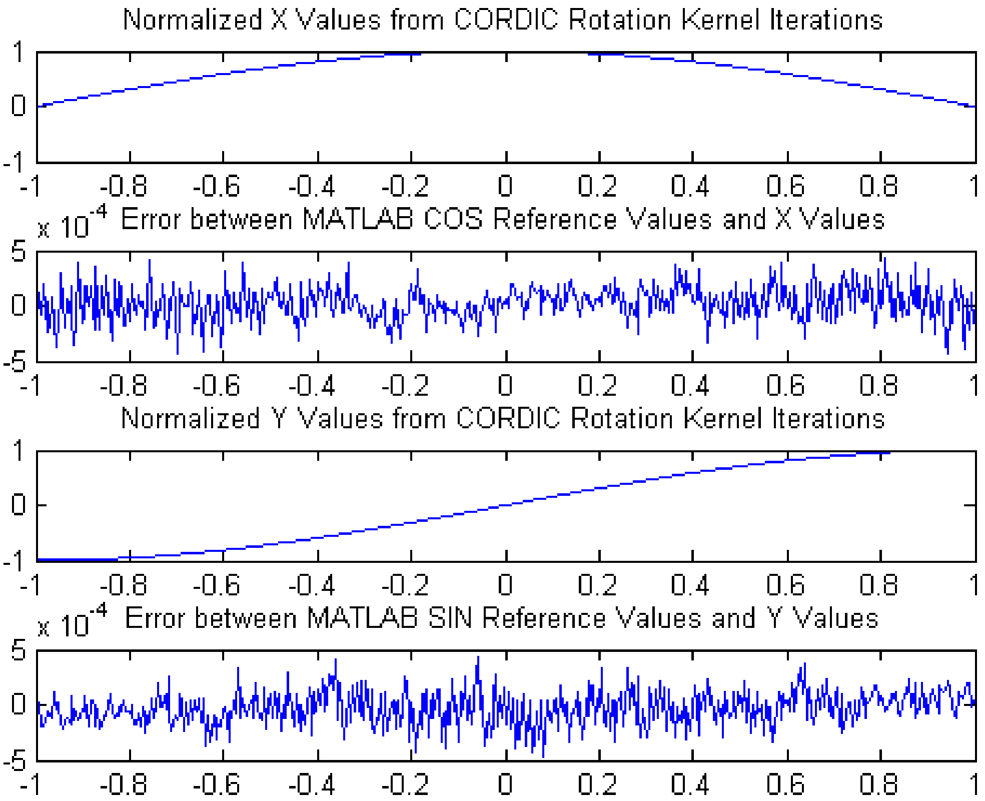
The CORDIC algorithm is usually run through a specified (constant) number of iterations since ending the CORDIC iterations early would break pipelined code, and the CORDIC gain A_n would not be constant because n would vary.

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

niters = 13; % Number of CORDIC iterations
inpLUT = fi(atan(2 .^ (-((0:(niters-1))'))) .* (2/pi), z_NT); % Normalized
AnGain = prod(sqrt(1+2.^(-2*(0:(niters-1))))); % CORDIC gain
inv_An = 1 / AnGain; % 1/A_n inverse of CORDIC gain

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

% Plot the CORDIC-approximated sine and cosine values
figure;
subplot(411);
plot(thNorm, x_out);
axis([-1 1 -1 1]);
title('Normalized X Values from CORDIC Rotation Kernel Iterations');
subplot(412);
thetaRadians = pi/2 .* thNorm; % real-world range [-pi/2 pi/2] angle values
plot(thNorm, cos(thetaRadians) - double(x_out));
title('Error between MATLAB COS Reference Values and X Values');
subplot(413);
plot(thNorm, y_out);
axis([-1 1 -1 1]);
title('Normalized Y Values from CORDIC Rotation Kernel Iterations');
subplot(414);
plot(thNorm, sin(thetaRadians) - double(y_out));
title('Error between MATLAB SIN Reference Values and Y Values');
```



References

1 Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.

2 Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Perform QR Factorization Using CORDIC

This example shows how to write MATLAB code that works for both floating-point and fixed-point data types. The algorithm used in this example is the QR factorization implemented via CORDIC (Coordinate Rotation Digital Computer).

A good way to write an algorithm intended for a fixed-point target is to write it in MATLAB using builtin floating-point types so you can verify that the algorithm works. When you refine the algorithm to work with fixed-point types, then the best thing to do is to write it so that the same code continues working with floating-point. That way, when you are debugging, then you can switch the inputs back and forth between floating-point and fixed-point types to determine if a difference in behavior is because of fixed-point effects such as overflow and quantization versus an algorithmic difference. Even if the algorithm is not well suited for a floating-point target (as is the case of using CORDIC in the following example), it is still advantageous to have your MATLAB code work with floating-point for debugging purposes.

In contrast, you may have a completely different strategy if your target is floating point. For example, the QR algorithm is often done in floating-point with Householder transformations and row or column pivoting. But in fixed-point it is often more efficient to use CORDIC to apply Givens rotations with no pivoting.

This example addresses the first case, where your target is fixed-point, and you want an algorithm that is independent of data type because it is easier to develop and debug.

In this example you will learn various coding methods that can be applied across systems. The significant design patterns used in this example are the following:

- Data Type Independence: the algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.
- Overflow Prevention: method to guarantee not to overflow. This demonstrates how to prevent overflows in fixed-point.

- Solving Systems of Equations: method to use computational efficiency. Narrow your code scope by isolating what you need to define.

The main part in this example is an implementation of the QR factorization in fixed-point arithmetic using CORDIC for the Givens rotations. The algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.

The QR factorization of M-by-N matrix A produces an M-by-N upper triangular matrix R and an M-by-M orthogonal matrix Q such that `A = Q*R`. A matrix is upper triangular if it has all zeros below the diagonal. An M-by-M matrix Q is orthogonal if `Q'*Q = 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*R = A`.

```
function [Q,R] = cordicqr(A,niter)
  Kn = inverse_cordic_growth_constant(niter);
  [m,n] = size(A);
  R = A;
  Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
  Q(:) = eye(m);                          % Initialize Q
  for j=1:n
    for i=j+1:m
      [R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
          cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
    end
  end
end
```

This function was written to be independent of data type. It works equally well with builtin floating-point types (double and single) and with the fixed-point `fi` object.

One of the trickiest aspects of writing data-type independent code is to specify data type and size for a new variable. In order to preserve data types without having to explicitly specify them, the output R was set to be the same as input A, like this:

```
R = A;
```

In addition to being data-type independent, this function was written in such a way that MATLAB Coder™ will be able to generate efficient C code from it. In MATLAB, you most often declare and initialize a variable in one step, like this:

```
Q = eye(m)
```

However, `Q=eye(m)` would always produce Q as a double-precision floating point variable. If A is fixed-point, then we want Q to be fixed-point; if A is single, then we want Q to be single; etc.

Hence, you need to declare the type and size of Q in one step, and then initialize it in a second step. This gives MATLAB Coder the information it needs to create an efficient C program with the correct types and sizes. In the finished code you initialize output Q to be an M-by-M identity matrix and the same data type as A, like this:

```
Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
Q(:) = eye(m);                          % Initialize Q
```

The coder.nullcopy function declares the size and type of Q without initializing it. The expansion of the first column of A with repmat won't appear in code generated by MATLAB; it is only used to specify the size. The repmat function was used instead of A(:,1:m) because A may have more rows than columns, which will be the case in a least-squares problem. You have to be sure to always assign values to every element of an array when you declare it with coder.nullcopy, because if you don't then you will have uninitialized memory.

You will notice this pattern of assignment again and again. This is another key enabler of data-type independent code.

The heart of this function is applying orthogonal Givens rotations in-place to the rows of R to zero out sub-diagonal elements, thus forming an upper-triangular matrix. The same rotations are applied in-place to the columns of the identity matrix, thus forming orthogonal Q. The Givens rotations are applied using the cordicgivens function, as defined in the next section. The rows of R and columns of Q are used as both input and output to the cordicgivens function so that the computation is done in-place, overwriting R and Q.

```
[R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
    cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
```

Defining the CORDIC Givens Rotation

The cordicgivens function applies a Givens rotation by performing CORDIC iterations to rows x=R(j,j:end), y=R(i,j:end) around the angle defined by x(1)=R(j,j) and y(1)=R(i,j) where i>j, thus zeroing out R(i,j). The same rotation is applied to columns u = Q(:,j) and v = Q(:,i), thus forming the orthogonal matrix Q.

```
function [x,y,u,v] = cordicgivens(x,y,u,v,niter,Kn)
  if x(1)<0
    % Compensation for 3rd and 4th quadrants
    x(:) = -x;  u(:) = -u;
    y(:) = -y;  v(:) = -v;
  end
  for i=0:niter-1
    x0 = x;
    u0 = u;
    if y(1)<0
      % Counter-clockwise rotation
      % x and y form R,         u and v form Q
      x(:) = x - bitsra(y, i);  u(:) = u - bitsra(v, i);
      y(:) = y + bitsra(x0,i);  v(:) = v + bitsra(u0,i);
    else
      % Clockwise rotation
      % x and y form R,         u and v form Q
      x(:) = x + bitsra(y, i);  u(:) = u + bitsra(v, i);
      y(:) = y - bitsra(x0,i);  v(:) = v - bitsra(u0,i);
    end
  end
  % Set y(1) to exactly zero so R will be upper triangular without round of
  % 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 Kn = 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.
  0     1.000000000000000                   0   1.000000000000000
  1     1.414213562373095   0.414213562373095   0.707106781186547  -0.292893
  2     1.581138830084190   0.166925267711095   0.632455532033676  -0.074651
  3     1.629800601300662   0.048661771216473   0.613571991077896  -0.018883
  4     1.642484065752237   0.012683464451575   0.608833912517752  -0.004738
  5     1.645688915757255   0.003204850005018   0.607648256256168  -0.001185
  6     1.646492278712479   0.000803362955224   0.607351770141296  -0.000296
  7     1.646693254273644   0.000200975561165   0.607277644093526  -0.000074
  8     1.646743506596901   0.000050252323257   0.607259112298893  -0.000018
  9     1.646756070204878   0.000012563607978   0.607254479332562  -0.000004
 10     1.646759211139822   0.000003140934944   0.607253321089875  -0.000001
 11     1.646759996375617   0.000000785235795   0.607253031529134  -0.000000
 12     1.646760192684695   0.000000196309077   0.607252959138945  -0.000000
 13     1.646760241761972   0.000000049077277   0.607252941041397  -0.000000
 14     1.646760254031292   0.000000012269320   0.607252936517010  -0.000000
 15     1.646760257098622   0.000000003067330   0.607252935385914  -0.000000
 16     1.646760257865455   0.000000000766833   0.607252935103139  -0.000000
 17     1.646760258057163   0.000000000191708   0.607252935032446  -0.000000
 18     1.646760258105090   0.000000000047927   0.607252935014772  -0.000000
 19     1.646760258117072   0.000000000011982   0.607252935010354  -0.000000
 20     1.646760258120067   0.000000000002995   0.607252935009249  -0.000000
 21     1.646760258120816   0.000000000000749   0.607252935008973  -0.000000
 22     1.646760258121003   0.000000000000187   0.607252935008904  -0.000000
 23     1.646760258121050   0.000000000000047   0.607252935008887  -0.000000
 24     1.646760258121062   0.000000000000012   0.607252935008883  -0.000000
 25     1.646760258121065   0.000000000000003   0.607252935008882  -0.000000
 26     1.646760258121065   0.000000000000001   0.607252935008881  -0.000000
 27     1.646760258121065                   0   0.607252935008881
 28     1.646760258121065                   0   0.607252935008881
 29     1.646760258121065                   0   0.607252935008881
 30     1.646760258121065                   0   0.607252935008881
 31     1.646760258121065                   0   0.607252935008881
 32    1.646760258121065                     0   0.607252935008881
```

Comparing CORDIC to the Standard Givens Rotation

The cordicgivens function is numerically equivalent to the following standard Givens rotation algorithm from Golub & Van Loan, *Matrix Computations*. In the cordicqr function, if you replace the call to cordicgivens with a call to givensrotation, then you will have the standard Givens QR algorithm.

```
function [x,y,u,v] = givensrotation(x,y,u,v)
  a = x(1); b = y(1);
  if b==0
    % No rotation necessary.  c = 1; s = 0;
    return;
  else
    if abs(b) > abs(a)
      t = -a/b; s = 1/sqrt(1+t^2); c = s*t;
    else
      t = -b/a; c = 1/sqrt(1+t^2); s = c*t;
    end
  end
  x0 = x;             u0 = u;
  % x and y form R,   u and v form Q
  x(:) = c*x0 - s*y;  u(:) = c*u0 - s*v;
  y(:) = s*x0 + c*y;  v(:) = s*u0 + c*v;
end
```

The givensrotation function uses division and square root, which are expensive in fixed-point, but good for floating-point algorithms.

Example of CORDIC Rotations

Here is a 3-by-3 example that follows the CORDIC rotations through each step of the algorithm. The algorithm uses orthogonal rotations to zero out the subdiagonal elements of R using the diagonal elements as pivots. The same rotations are applied to the identity matrix, thus producing orthogonal Q such that Q*R = A.

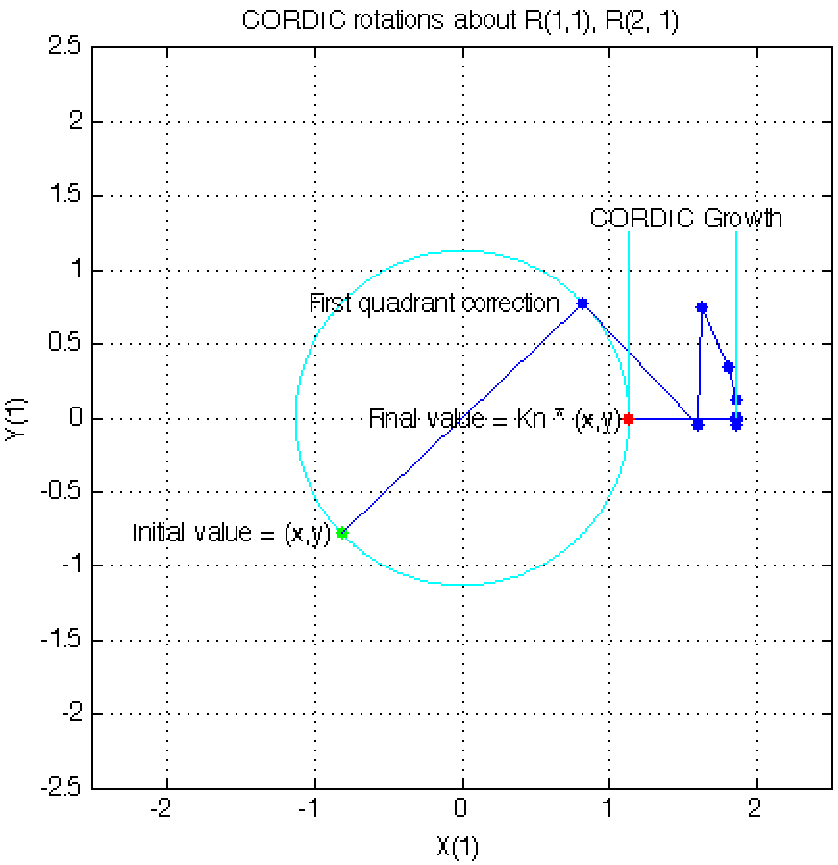
Let A be a random 3-by-3 matrix, and initialize R = A, and Q = eye(3).

```
  R = A = [-0.8201    0.3573   -0.0100
           -0.7766   -0.0096   -0.7048
           -0.7274   -0.6206   -0.8901]

      Q = [ 1         0         0
            0         1         0
            0         0         1]
```

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
```

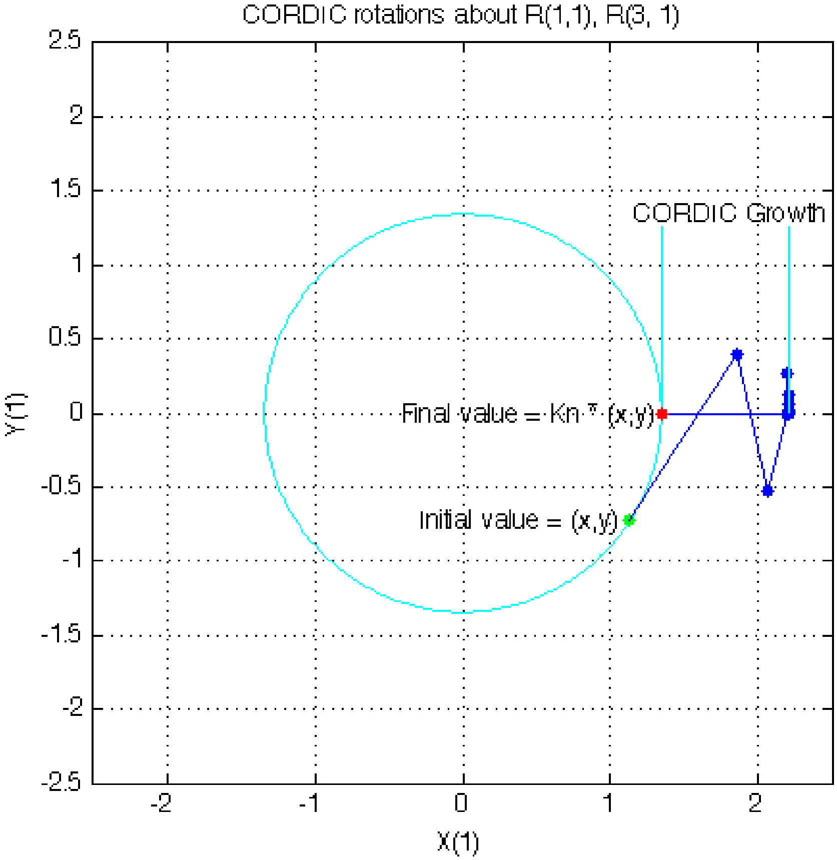
In the following plot, you can see the growth in x in each of the CORDIC iterations. The growth is factored out at the last step by multiplying it by `Kn = 0.60725`. You can see that `y(1)` iterates to 0. Initially, the point `[x(1), y(1)]` is in the third quadrant, and is reflected into the first quadrant before the start of the CORDIC iterations.



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

```
    R before the second rotation           R after the second rotation
x  [1.1294    -0.2528    0.4918]   ->    x [1.3434    0.1235    0.8954]
         0     0.2527    0.5049                   0    0.2527    0.5049
y [-0.7274]   -0.6206   -0.8901    ->    y [      0   -0.6586   -0.4820]

    Q before the second rotation           Q after the second rotation
    u                       v              u                   v
```

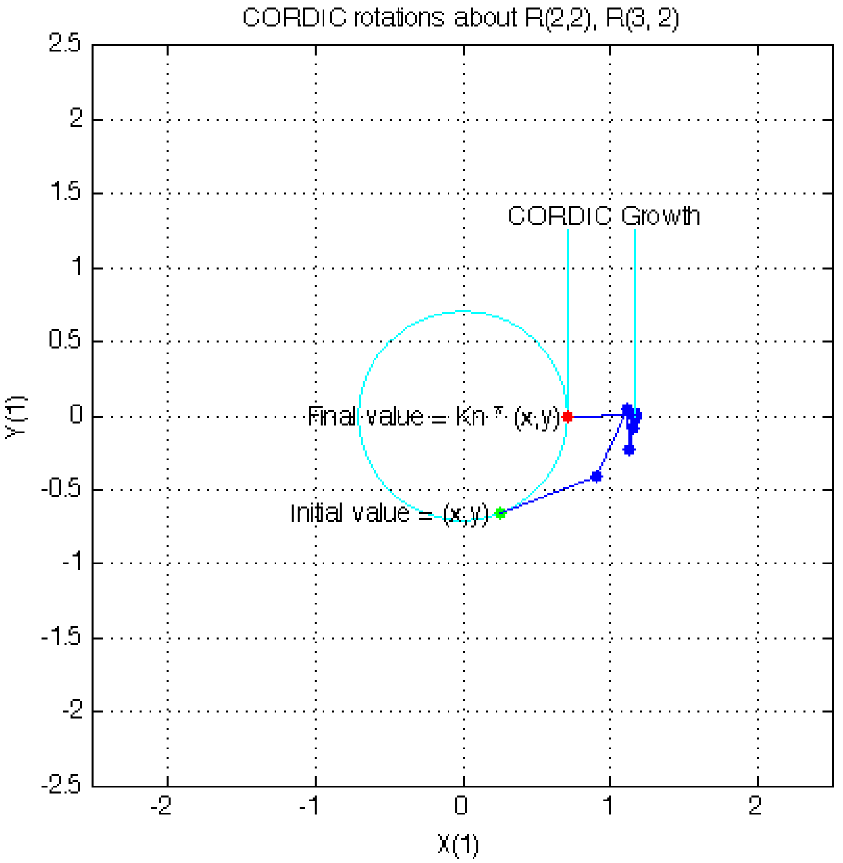
```
[-0.7261]   0.6876        [0]               [-0.6105]   0.6876  [-0.3932]
[-0.6876]  -0.7261        [0]     ->        [-0.5781]  -0.7261  [-0.3723]
[      0]        0        [1]               [-0.5415]        0  [ 0.8407]
```



The third rotation is about the second and third row of R and the second and third column of Q. Element `R(2,2)` is the pivot and `R(3,2)` rotates to 0.

```
     R before the third rotation            R after the third rotation
     1.3434     0.1235     0.8954           1.3434     0.1235     0.8954
 x        0  [ 0.2527     0.5049]   ->   x        0   [0.7054     0.6308]
```

```
y        0   [-0.6586    -0.4820]    ->    y        0    [      0    0.2987]

   Q before the third rotation              Q after the third rotation
                u          v                             u          v
  -0.6105   [ 0.6876] [-0.3932]            -0.6105   [ 0.6134] [ 0.5011]
  -0.5781   [-0.7261] [-0.3723]    ->      -0.5781   [ 0.0875] [-0.8113]
  -0.5415   [      0] [ 0.8407]            -0.5415   [-0.7849] [ 0.3011]
```



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

```
R =
   1.3434    0.1235    0.8954
        0    0.7054    0.6308
        0         0    0.2987

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]=size(A)`, and `[Q,R]=givensqr(A)`. Then

```
max(abs(R(:))) <= sqrt(m) * max(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

```
norm(A(:,j)) <= sqrt(m) * max(abs(A(:))).
```

That is, for all j

```
norm(A(:,j))  = sqrt(A(1,j)^2 + A(2,j)^2 + ... + A(m,j)^2)
             <= sqrt( m * max(abs(A(:)))^2)
              = sqrt(m) * max(abs(A(:))).
```

and so for all i,j

```
abs(R(i,j)) <= norm(A(:,j)) <= sqrt(m) * max(abs(A(:))).
```

Hence, it is also true for the largest element of R

```
max(abs(R(:))) <= sqrt(m) * max(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

```
abs(R(i,j)) <= sqrt(m) * upperbound(A)
```

Note that sqrt(m)*upperbound(A) is also an upper bound for the elements of A:

```
abs(A(i,j)) <= upperbound(A) <= sqrt(m)*upperbound(A)
```

Therefore, when picking fixed-point data types, sqrt(m)*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 = [7    -7     7     7
     7     7    -7     7
     7    -7    -7    -7
     7     7     7    -7];
```

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(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 =

    0.5000   -0.5000    0.5000    0.5000
    0.5000    0.5000   -0.5000    0.5000
    0.5000   -0.5000   -0.5000   -0.5000
    0.5000    0.5000    0.5000   -0.5000


R =

   14.0000    0.0000   -0.0000   -0.0000
         0   14.0000   -0.0000    0.0000
         0         0   14.0000    0.0000
         0         0         0   14.0000
```

Another simple example of attaining maximum growth is a matrix that has all identical elements, like a matrix of all ones. A matrix of ones will get rotated into `1*sqrt(m)` in the first row and zeros elsewhere. For example, this 9-by-5 matrix will have all `1*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 `log2(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*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*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*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
    0.0000    1.0000         0
         0         0    1.0000
```

```
Q*Q'
```

```
ans =

    1.0000    0.0000    0.0000
    0.0000    1.0000   -0.0000
    0.0000   -0.0000    1.0000
```

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

```
Q0*R0 - A
```

```
ans =

   1.0e-15 *

   -0.1110   -0.1110   -0.1110
   -0.1110   -0.1110   -0.1110
   -0.1110   -0.1110   -0.1110
```

```
Q*R - A
```

```
ans =

   1.0e-15 *

   -0.2220    0.2220    0.2220
    0.4441         0         0
    0.2220    0.2220    0.2220
```

Solving Systems of Equations Without Forming Q

Given matrices A and B, you can use the QR factorization to solve for X in the following equation:

```
A*X = B.
```

If A has more rows than columns, then X will be the least-squares solution. If X and B have more than one column, then several solutions can be computed at the same time. If `A = Q*R` is the QR factorization of A, then the solution can be computed by back-solving

```
R*X = C
```

where `C = Q'*B`. Instead of forming Q and multiplying to get `C = Q'*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 Hyperbolic Kernel

This example shows how to compute square root using a CORDIC hyperbolic kernel algorithm in MATLAB. CORDIC-based algorithms are critical to many embedded applications, including motor controls, navigation, signal processing, and wireless communications.

Introduction

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

The fixed-point CORDIC algorithm requires the following operations:

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

Note that for hyperbolic CORDIC-based algorithms, such as square root, certain iterations (i = 4, 7, 10, ..., 3k+1, ...) are repeated to achieve result convergence. There is an additional cost of 3 additions for each of those repeated iterations.

CORDIC Kernel Algorithms Using Hyperbolic Computation Modes

You can use a CORDIC computing mode algorithm to calculate hyperbolic functions, such as hyperbolic trigonometric, square root, log, exp, etc.

In hyperbolic rotation mode, the CORDIC equations are:

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

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

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

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

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

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

- $x_N \approx A_N(x_0 \cosh z_0 + y_0 \sinh z_0)$
- $y_N \approx A_N(y_0 \cosh z_0 + x_0 \sinh z_0)$
- $z_N \approx 0$

Where:

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

Typically N is chosen to be a large-enough constant value. Thus, A_N may be pre-computed.

In hyperbolic vectoring mode, the CORDIC equations are as above, but with $d_i = +1$ if $y_i < 0$, and -1 otherwise;

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

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

Note that the rotations in the hyperbolic coordinate system do not converge. It has been shown, however, that convergence is achieved if certain iterations (i = 4, 7, 10, ..., 3k+1, ...) are repeated.

Efficient MATLAB Implementation of a CORDIC Hyperbolic Vectoring Algorithm

A MATLAB code implementation example of the CORDIC Hyperbolic Vectoring algorithm follows (for the case of scalar x, y, and z). This same code can be used for both fixed-point and floating-point operation.

CORDIC Hyperbolic Vectoring Kernel

```
function [x, y, z] = cordic_hyperbolic_vectoring_kernel(x, y, z, inpLUT, n)
% Perform CORDIC hyperbolic vectoring kernel algorithm for N iterations.
k = 3; % Used for REPEAT rotations at (idx == 4, 7, 10, ..., 3k+1, ...)
for idx = 1:n
    xtmp = bitsra(x, idx); % multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % multiply by 2^(-idx)

     if y < 0
         x(:) = accumpos(x, ytmp);
         y(:) = accumpos(y, xtmp);
         z(:) = accumneg(z, inpLUT(idx));
     else
         x(:) = accumneg(x, ytmp);
         y(:) = accumneg(y, xtmp);
         z(:) = accumpos(z, inpLUT(idx));
     end

     if k > 0
         k = k-1; % Decrease '3k+1' counter
     else
         k = 3;   % Re-start '3k+1' counter and REPEAT rotation
         if y < 0
             x(:) = accumpos(x, ytmp);
             y(:) = accumpos(y, xtmp);
             z(:) = accumneg(z, inpLUT(idx));
         else
             x(:) = accumneg(x, ytmp);
             y(:) = accumneg(y, xtmp);
             z(:) = accumpos(z, inpLUT(idx));
        end
     end
 end % idx loop
```

CORDIC-Based Square Root Computation

Square Root Computation Using the CORDIC Hyperbolic Kernel

The judicious choice of initial values allows the CORDIC kernel hyperbolic vectoring mode algorithm to compute square root.

First, the following initialization steps are performed:

- The input look-up table `inpLUT` is set to `atanh(2 .^ -(1:N))`.
- x_0 is set to $v + 0.25$.
- y_0 is set to $v - 0.25$.
- z_0 is set to zero.

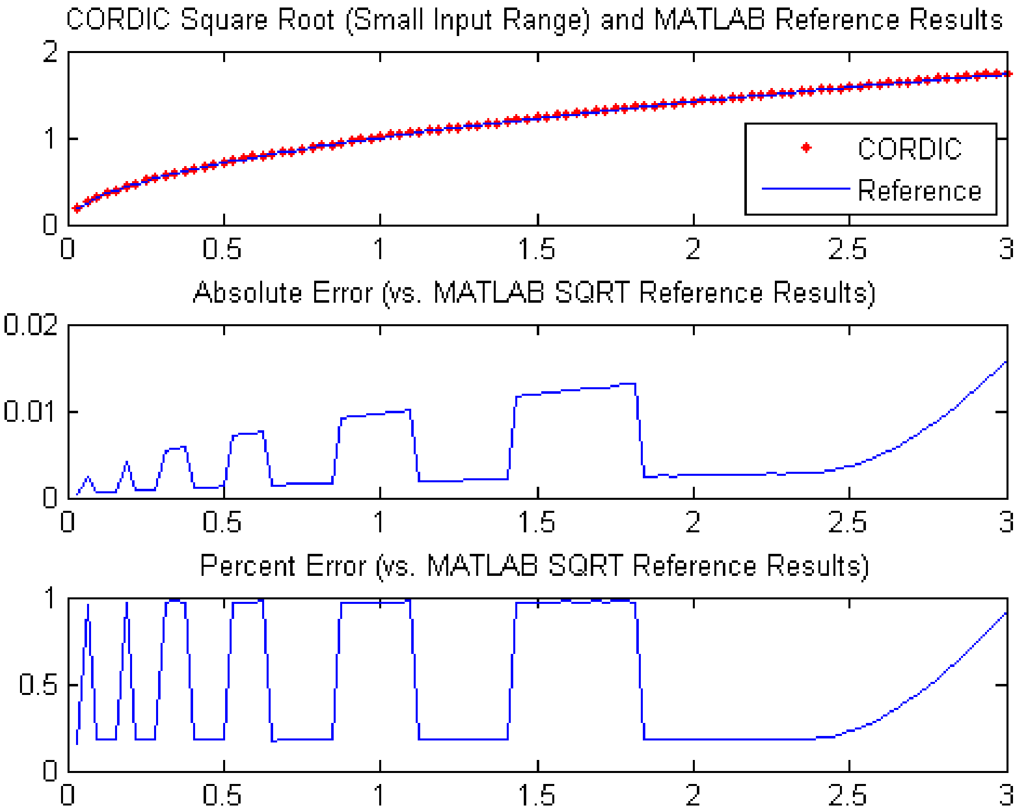
After N iterations, these initial values lead to the following output as N approaches $+\infty$:

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

Example

Use CORDIC to compute the square root of `v_fix`:

```
xyNT  = numerictype(1,20,16);
v_fix = fi(((2^-5):(2^-5):3.0), xyNT); % fixed-point input values
niter = 10; % note that iterations 4, 7, and 10 will be repeated
hpLUT = atanh(2 .^ -(1:niter));
z_NT  = numerictype(1,24,23);
lutFP = fi(hpLUT, z_NT);
x_sqr = fi(zeros(size(v_fix)), xyNT); % X array pre-allocation
y_sqr = fi(zeros(size(v_fix)), xyNT); % Y array pre-allocation
z_sqr = fi(zeros(size(v_fix)), z_NT); % Z array pre-allocation

for idx = 1:length(v_fix)
    x_in = fi(accumpos(v_fix(idx), 0.25)); % v + 0.25 in same data type
    y_in = fi(accumneg(v_fix(idx), 0.25)); % v - 0.25 in same data type
    z_in = fi(0, z_NT);

    [x_sqr(idx), y_sqr(idx), z_sqr(idx)] = ...
        fidemo.cordic_hyperbolic_vectoring_kernel(...
            x_in, y_in, z_in, lutFP, niter);
end

% Get the Real World Value (RWV) of the CORDIC outputs for comparison
% and plot the error between the MATLAB reference and CORDIC sqrt values
An_hp = 0.5 .* prod(sqrt(1+2.^(-2*(0:(niter-1)))));
x_cdc = double(x_sqr) ./ An_hp; % CORDIC sqrt results (scaled by An_hp)
v_ref = double(v_fix);
x_ref = sqrt(v_ref); % MATLAB sqrt reference results
figure;
subplot(311);
plot(v_ref, x_cdc, 'r.');
hold on;
plot(v_ref, x_ref, 'b-');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root (Small Input Range) and MATLAB Reference Results'
hold off;
subplot(312);
absErr = abs(x_ref - x_cdc);
plot(v_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');
subplot(313);
plot(v_ref, 100 .* (absErr ./ x_ref));
title('Percent Error (vs. MATLAB SQRT Reference Results)');
```



Overcoming Algorithm Input Range Limitations

For many square root algorithms, the input value v is typically normalized to a [0.5, 2) range, using a fixed word length normalization. This additional pre-processing step may be used to support large input value ranges. The CORDIC-based square root algorithm implementation in particular goes unstable for larger input values.

As described in the example Implement Fixed-Point Square Root Using Lookup Table, arbitrary positive inputs u may be expressed as $u = v * 2^n$, where n is an even positive integer value. Simple post-processing may then be used to adjust corresponding output values.

Here is a step-by-step procedure for arbitrary input ranges, assuming a fixed input word length:

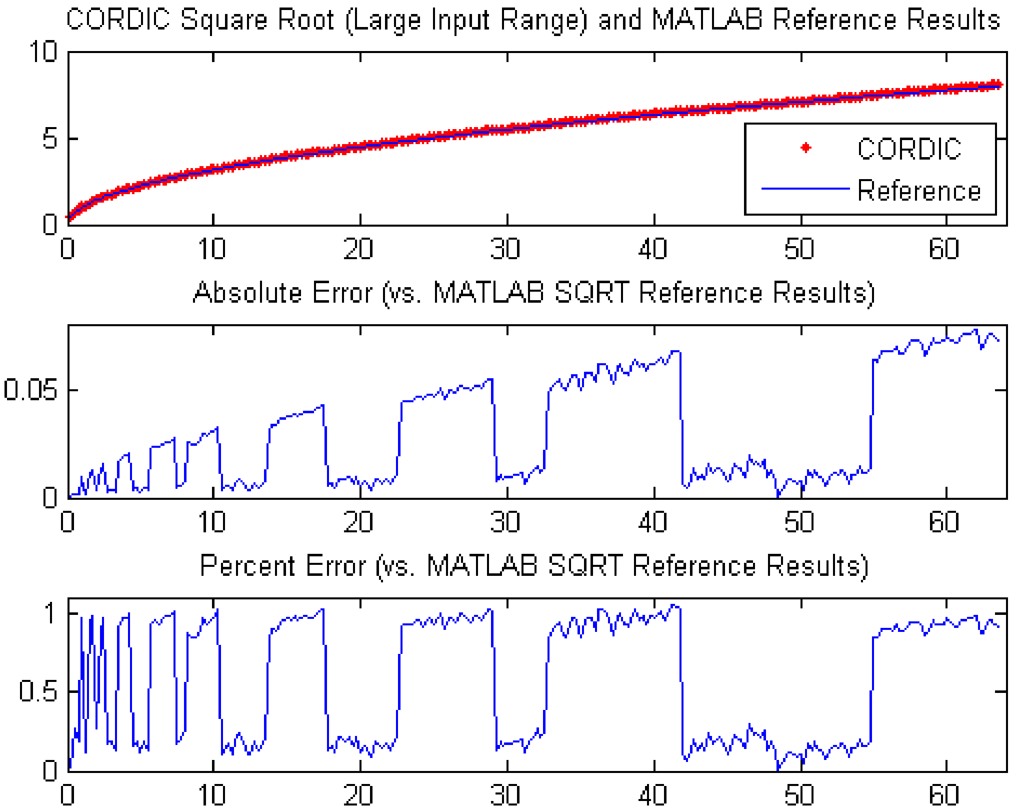
1. Declare the number of bits in a byte, `B`, as a constant. In this example, $B = 8$.
2. If the input value is >= 2.0 then use the function `fi_normalize_unsigned_8_bit_byte()` described in example Normalize Data for Lookup Tables to normalize the input $u > 0$ such that $u = v * 2^n$, $0.5 <= v < 2$, and n is an even positive integer value.
3. Compute the CORDIC-based square root of v, using the algorithm described above.
4. Scale the intermediate result by $2^{n/2}$ using `bitsll`.
5. For plotting and comparison purposes, scale the output by A_N.

Example

Compute the square root of 8-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_len    = 254;
u_ref    = linspace(0.25, 63.5, u_len);
u_in_arb = fi(u_ref,0,8); % 8-bit unsigned fixed-point input data
twoFixPt = fi(2, numerictype(u_in_arb));
sqrt_ref = sqrt(double(u_in_arb)); % MATLAB sqrt reference results
niter    = 8; % note that iterations 4, 7, and 10 will be repeated
An_hp    = 0.5 .* prod(sqrt(1+2.^(-2*(0:(niter-1)))));
hpLUT    = atanh(2 .^ -(1:niter));
z_NT     = numerictype(1,16,15);
lutFP    = fi(hpLUT, z_NT);
xyNT     = numerictype(1,16,12);
results  = zeros(u_len, 2);
results(:,2) = sqrt_ref(:);
```

```
for idx = 1:u_len
    v_fix = fi(0, xyNT); % Pre-allocate next in-range input value
    input = u_in_arb(idx);

    isInputOutOfRange = (input >= twoFixPt);
    if (isInputOutOfRange)
        % Outside of CORDIC SQRT algorithm range limit
        [v_fix(1), n] = fi_normalize_unsigned_8_bit_byte(input);
        isodd         = int8(storedInteger(bitand(fi(1,1,8,0),fi(n))));
        v_fix(1)      = bitsra(v_fix,isodd);
        n             = n + isodd;
        leftShiftVal  = bitsra(n,1); % Scale output by 2^(n/2)
    else
        % Inside of CORDIC SQRT algorithm range limit
        v_fix(1)     = input; % Cast to v_fix type
        leftShiftVal = 0;     % No output rescaling
    end

    x_in = fi(accumpos(v_fix, 0.25)); % v + 0.25 in same data type
    y_in = fi(accumneg(v_fix, 0.25)); % v - 0.25 in same data type
    z_in = fi(0, z_NT);

    [x_out, y_out, z_out] = ...
        fidemo.cordic_hyperbolic_vectoring_kernel(...
            x_in, y_in, z_in, lutFP, niter);

    x_tmp = bitsll(x_out, leftShiftVal); % Scaled fixed-point output

    % Compute the equivalent Real World Value result for plotting
    results(idx,1) = double(x_tmp) ./ An_hp;
end

% Plot the Real World Value (RWV) of CORDIC and MATLAB reference results
figure;
subplot(311);
plot(u_ref, results(:,1), 'r.');
hold on;
plot(u_ref, results(:,2), 'b-');
legend('CORDIC', 'Reference', 'Location', 'SouthEast');
title('CORDIC Square Root (Large Input Range) and MATLAB Reference Results'
axis([0 64 0 10]);
hold off;
subplot(312);
absErr = abs(results(:,2) - results(:,1));
plot(u_ref, absErr);
title('Absolute Error (vs. MATLAB SQRT Reference Results)');
axis([0 64 0 0.08]);
subplot(313);
plot(u_ref, 100 .* (absErr ./ results(:,2)));
title('Percent Error (vs. MATLAB SQRT Reference Results)');
axis([0 64 0 1.1]);
```



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 `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;

$i = 0, 1, ..., N-1$, 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 $\theta = \mathrm{atan}(y_0/x_0)$.

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 $+\infty$:

- $x_N \approx A_N\sqrt{x_0^2 + y_0^2}$
- $z_N \approx \mathrm{atan}(y_0/x_0)$

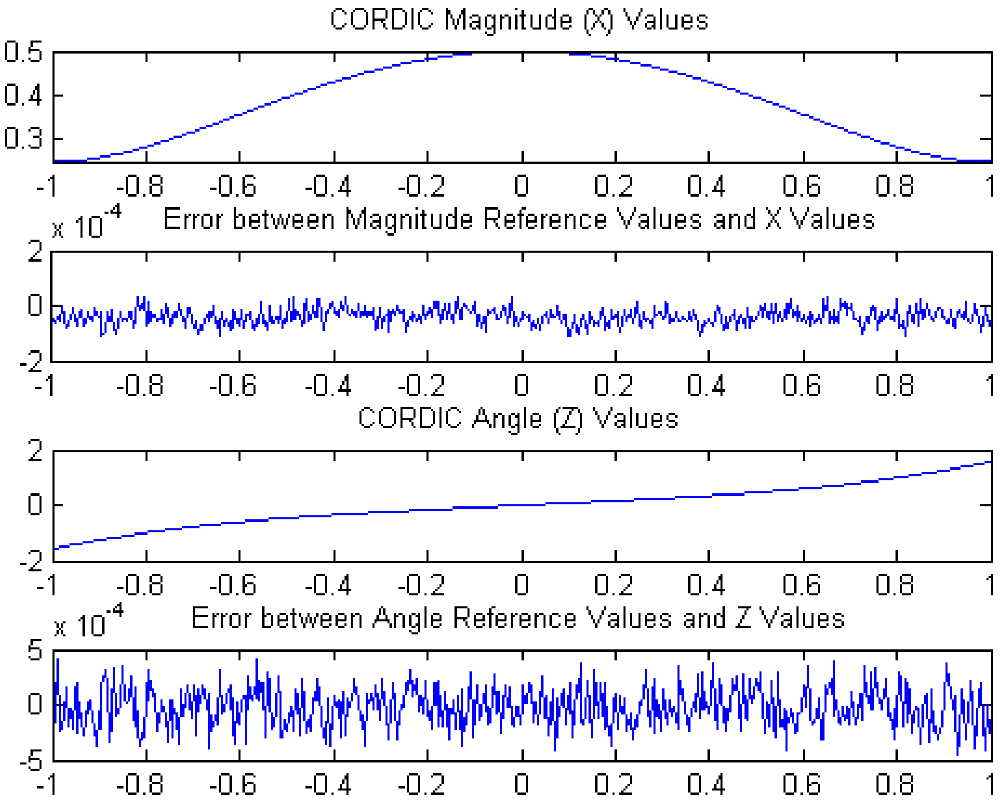
Other vectoring-kernel-based function approximations are possible via pre- and post-processing and using other initial conditions (see [1,2]).

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

niters = 13; % Number of CORDIC iterations
inpLUT = fi(atan(2 .^ (-((0:(niters-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, niters);
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:(niters-1))))); % CORDIC gain
x_c2p_RWV    = (1/AnGain) .* double(x_c2p); % Magnitude (scaled by CORDIC g
z_c2p_RWV    =   (pi/2)   .* double(z_c2p); % Angles (in radian units)
[thRWV,rRWV] = cart2pol(double(inXfix), double(inYfix)); % MATLAB reference
magnitudeErr = rRWV - x_c2p_RWV;
angleErr     = thRWV - z_c2p_RWV;
```

```
figure;
subplot(411);
plot(thNorm, x_c2p_RWV);
axis([-1 1 0.25 0.5]);
title('CORDIC Magnitude (X) Values');
subplot(412);
plot(thNorm, magnitudeErr);
title('Error between Magnitude Reference Values and X Values');
subplot(413);
plot(thNorm, z_c2p_RWV);
title('CORDIC Angle (Z) Values');
subplot(414);
plot(thNorm, angleErr);
title('Error between Angle Reference Values and Z Values');
```



References

1 Jack E. Volder, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, Volume EC-8, September 1959, pp330-334.

2 Ray Andraka, A survey of CORDIC algorithm for FPGA based computers, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, pp191-200

Set Data Types Using Min/Max Instrumentation

This example shows how to set fixed-point data types by instrumenting MATLAB code for min/max logging and using the tools to propose data types.

The functions you will use are:

- `buildInstrumentedMex` - Build MEX function with instrumentation enabled
- `showInstrumentationResults` - Show instrumentation results
- `clearInstrumentationResults` - Clear instrumentation results

The Unit Under Test

The function that you convert to fixed-point in this example is a second-order direct-form 2 transposed filter. You can substitute your own function in place of this one to reproduce these steps in your own work.

```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

For a MATLAB function to be instrumented, it must be suitable for code generation. For information on code generation, see the reference page for `buildInstrumentedMex`. A MATLAB Coder™ license is not required to use `buildInstrumentedMex`.

In this function the variables `y` and `z` are used as both inputs and outputs. This is an important pattern because:

- You can set the data type of `y` and `z` outside the function, thus allowing you to re-use the function for both fixed-point and floating-point types.
- The generated C code will create `y` and `z` as references in the function argument list. For more information about this pattern, see the

documentation under Code Generation from MATLAB > User's Guide > Generating Efficient and Reusable Code > Generating Efficient Code > Eliminating Redundant Copies of Function Inputs.

Run the following code to copy the test function into a temporary directory so this example doesn't interfere with your own work.

```
tempdirObj = fidemo.fiTempdir('fi_instrumentation_fixed_point_filter_demo')

copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
                  'fi_2nd_order_df2t_filter.m'),'.','f');
```

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

```
FIPREF_STATE = 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 `fimath` with floor rounding and wrap overflow settings to control arithmetic, which leads to more efficient C code.

```
b = fi(num,1,16); b.fimath = F;
a = fi(den,1,16); a.fimath = F;
```

Hard-code the filter coefficients into the implementation of this filter by passing them as constants to the `buildInstrumentedMex` command.

```
B = coder.Constant(b);
A = coder.Constant(a);
```

Data Types Determined by the Values of the Coefficients and Inputs

The values of the coefficients and values of the inputs determine the data types of output `y` and state vector `z`. Create them with a scaled double datatype so their values will attain full range and you can identify potential overflows and propose data types.

```
yisd = fi(zeros(N,1),1,16,15,'DataType','ScaledDouble','fimath',F);
zisd = fi(zeros(2,1),1,16,15,'DataType','ScaledDouble','fimath',F);
```

Instrument the MATLAB Function as a Scaled-Double MEX Function

To instrument the MATLAB code, you create a MEX function from the MATLAB function using the `buildInstrumentedMex` command. The inputs to `buildInstrumentedMex` are the same as the inputs to `fiaccel`, but `buildInstrumentedMex` has no `fi`-object restrictions. The output of buildInstrumentedMex is a MEX function with instrumentation inserted, so when the MEX function is run, the simulated minimum and maximum values are recorded for all named variables and intermediate values.

Use the '-o' option to name the MEX function that is generated. If you do not use the '-o' option, then the MEX function is the name of the MATLAB function with '_mex' appended. You can also name the MEX function the same as the MATLAB function, but you need to remember that MEX functions take precedence over MATLAB functions and so changes to the MATLAB function will not run until either the MEX function is re-generated, or the MEX function is deleted and cleared.

```
buildInstrumentedMex fi_2nd_order_df2t_filter ...
    -o filter_scaled_double ...
    -args {B,A,x,yisd,zisd}
```

Test Bench with Chirp Input

The test bench for this system is set up to run chirp and step signals. In general, test benches for systems should cover a wide range of input signals.

The first test bench uses a chirp input. A chirp signal is a good representative input because it covers a wide range of frequencies.

```
t = linspace(0,1,N);       % Time vector from 0 to 1 second
f1 = N/2;                  % Target frequency of chirp set to Nyquist
xchirp = sin(pi*f1*t.^2);  % Linear chirp from 0 to Fs/2 Hz in 1 second
x(:) = xchirp;             % Cast the chirp to fixed-point
```

Run the Instrumented MEX Function to Record Min/Max Values

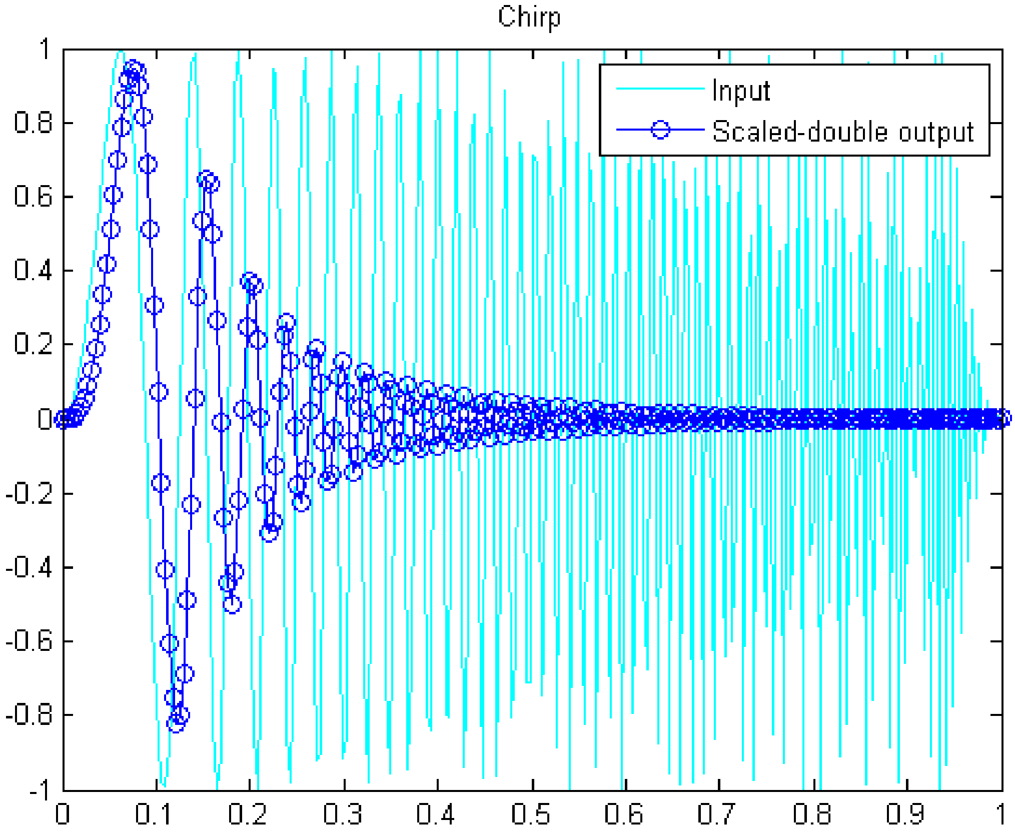
The instrumented MEX function must be run to record minimum and maximum values for that simulation run. Subsequent runs accumulate the instrumentation results until they are cleared with clearInstrumentationResults.

Note that the numerator and denominator coefficients were compiled as constants so they are not provided as input to the generated MEX function.

```
ychirp = filter_scaled_double(x,yisd,zisd);
```

The plot of the filtered chirp signal shows the lowpass behavior of the filter with these particular coefficients. Low frequencies are passed through and higher frequencies are attenuated.

```
clf
plot(t,x,'c',t,ychirp,'bo-')
title('Chirp')
legend('Input','Scaled-double output')
figure(gcf); drawnow;
```



Show Instrumentation Results with Proposed Fraction Lengths for Chirp

The showInstrumentationResults command displays the code generation report with instrumented values. The input to showInstrumentationResults is the name of the instrumented MEX function for which you wish to show results.

This is the list of options to the showInstrumentationResults command:

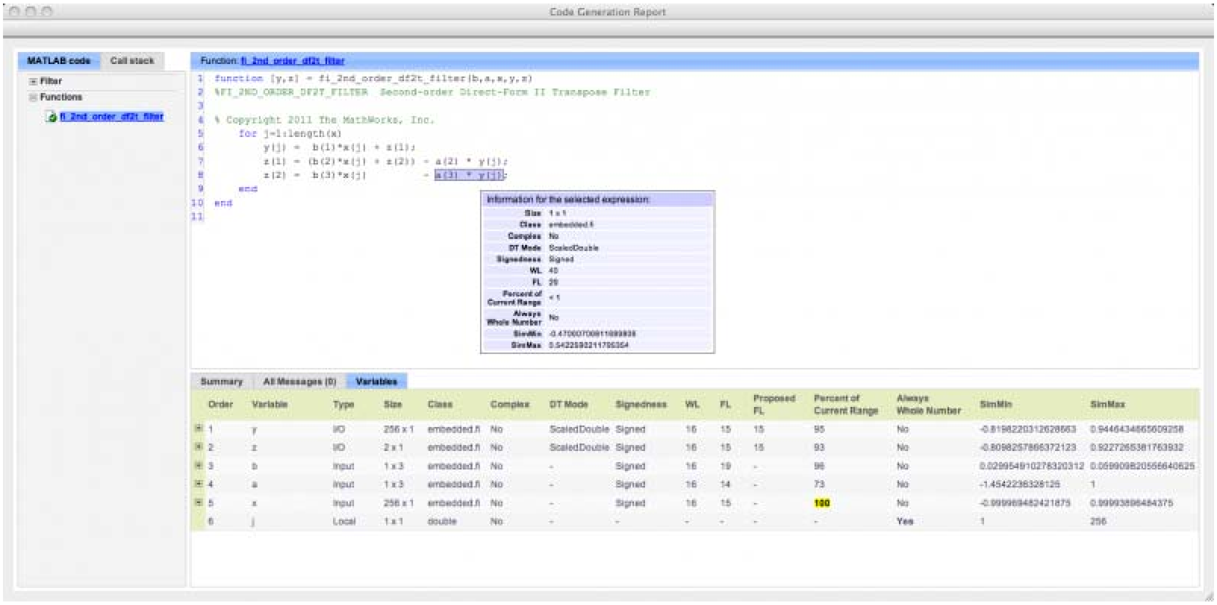
- -defaultDT T Default data type to propose for doubles, where T is a numerictype object, or one of the strings {remainFloat, double, single, int8, int16, int32, int64, uint8, uint16, uint32, uint64}. The default is remainFloat.
- -nocode Do not show MATLAB code in the printable report. Display only 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.



```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
%FI_2ND_ORDER_DF2T_FILTER  Second-order Direct-Form II Transpose Filter

% Copyright 2011 The MathWorks, Inc.
    for j=1:length(x)
        y(j) =  b(1)*x(j) + z(1);
        z(1) =  (b(2)*x(j) + z(2)) - a(2) * y(j);
        z(2) =  b(3)*x(j)          - a(3) * y(j);
    end
end
```

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	-0.47000700911889938
SimMax	0.5422593211795354

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Proposed FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
⊞ 1	y	I/O	256 x 1	embedded.fi	No	ScaledDouble	Signed	16	15	15	95	No	-0.8198220312628663	0.9446434665609258
⊞ 2	z	I/O	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	-0.999969482421875	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.

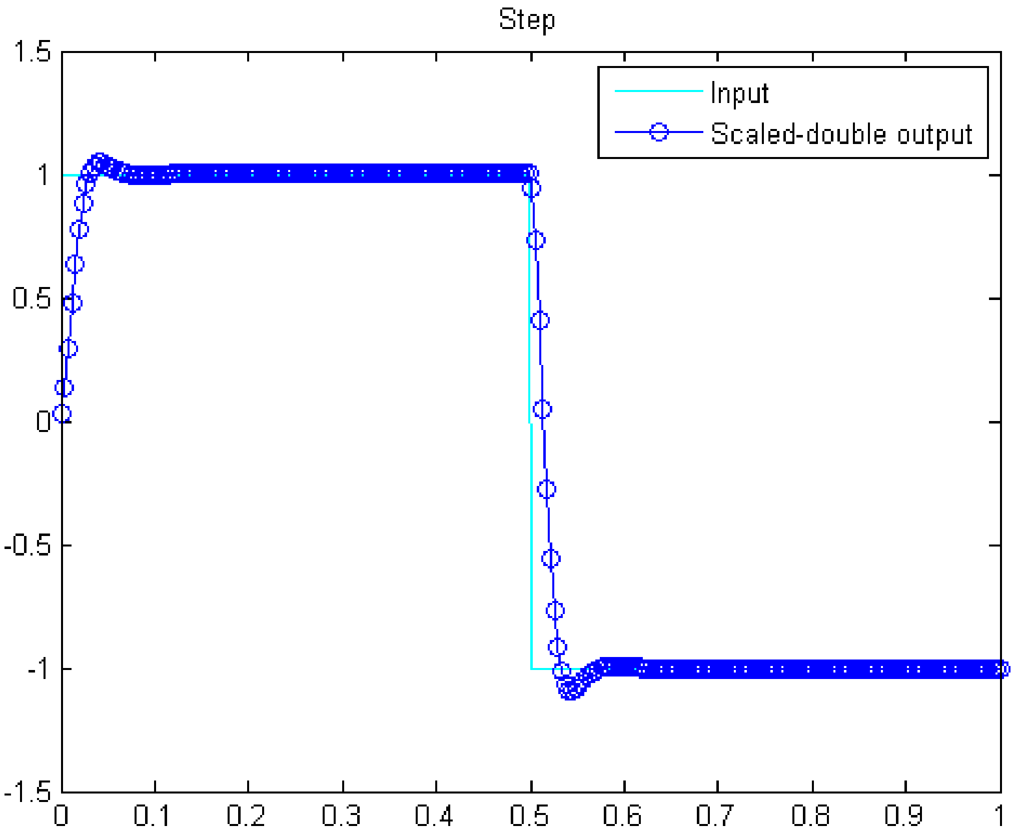
```
xstep = [ones(N/2,1);-ones(N/2,1)];
x(:) = xstep;
```

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
```

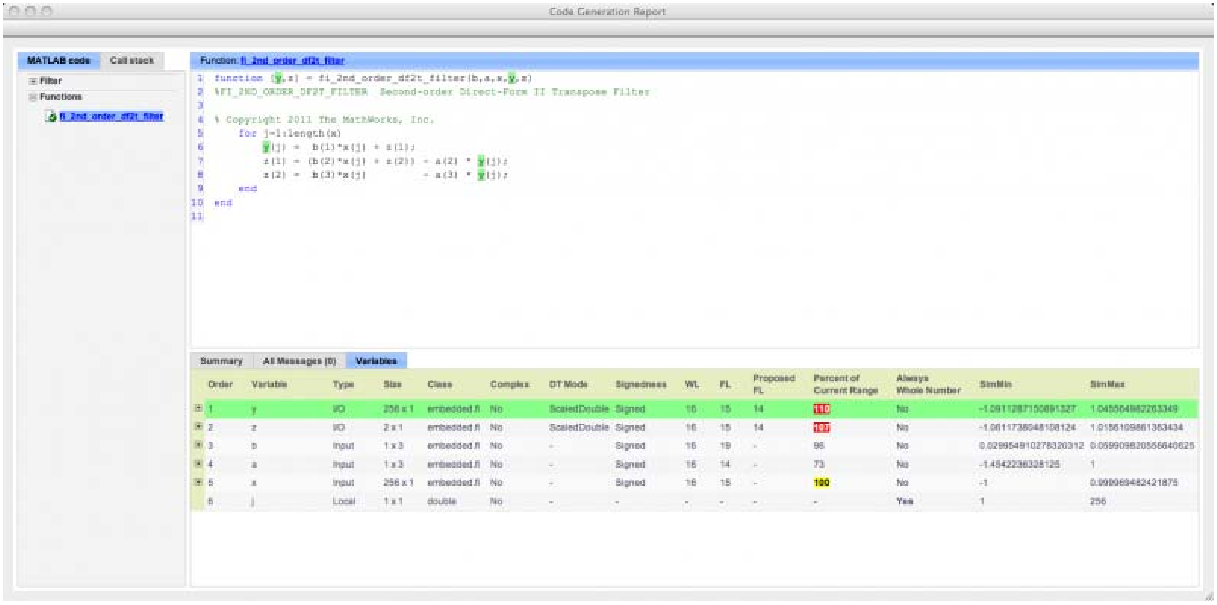
```
plot(t,x,'c',t,ystep,'bo-')
title('Step')
legend('Input','Scaled-double output')
figure(gcf); drawnow;
```



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

Function: fi_2nd_order_df2t_filter

```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
%FI_2ND_ORDER_DF2T_FILTER  Second-order Direct-Form II Transpose Filter

% Copyright 2011 The MathWorks, Inc.
    for j=1:length(x)
        y(j) =  b(1)*x(j) + z(1);
        z(1) = (b(2)*x(j) + z(2)) - a(2) * y(j);
        z(2) =  b(3)*x(j)         - a(3) * y(j);
    end
end
```

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Proposed FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	I/O	256 x 1	embedded.fi	No	ScaledDouble	Signed	16	15	14	110	No	-1.0911287150891327	1.0455604982263349
2	z	I/O	2 x 1	embedded.fi	No	ScaledDouble	Signed	16	15	14	107	No	-1.0611738048108124	1.0156109861363434
3	b	Input	1 x 3	embedded.fi	No	-	Signed	16	19	-	98	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.

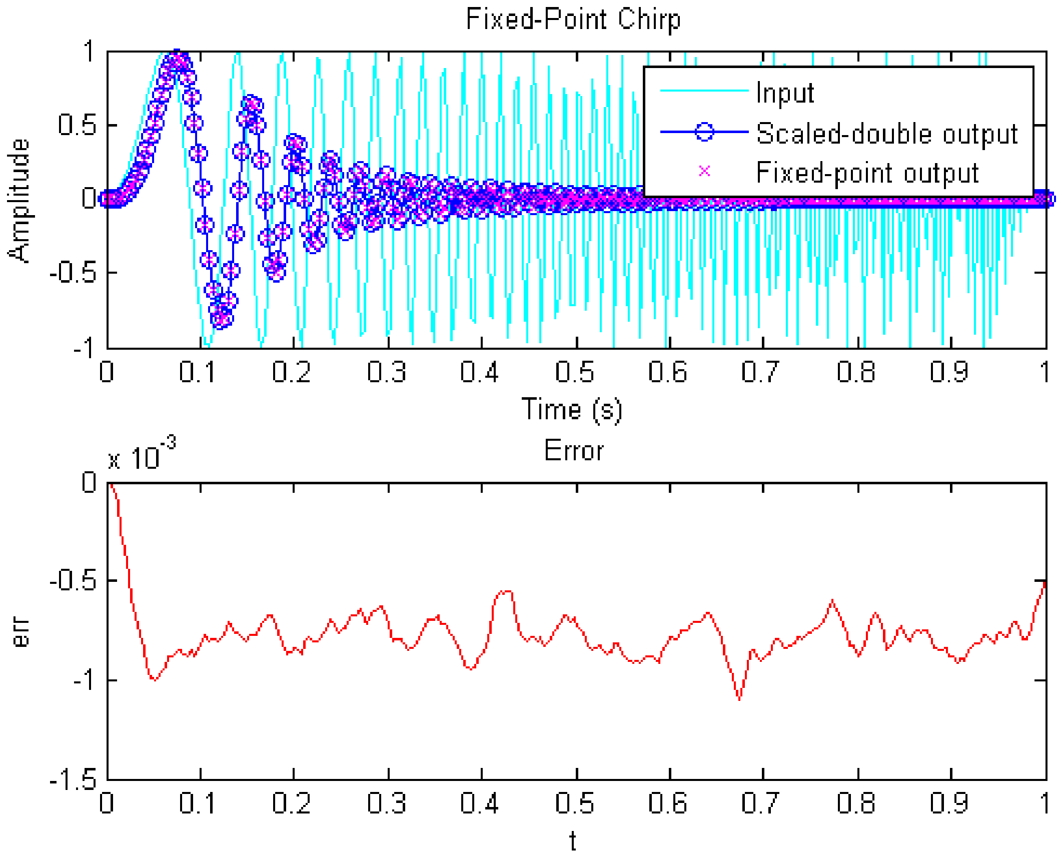
Validate with Chirp Input

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);
```

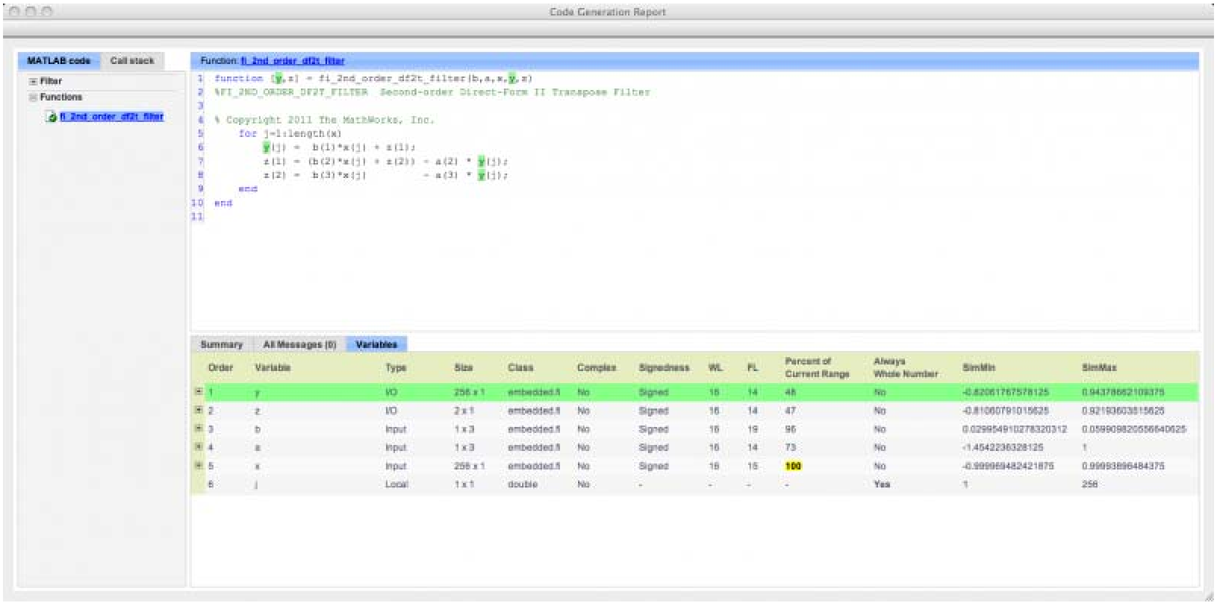
Compare the fixed-point outputs to the scaled-double outputs to verify that they meet your design criteria.

```
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
```



```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
%FI_2ND_ORDER_DF2T_FILTER  Second-order Direct-Form II Transpose Filter

% Copyright 2011 The MathWorks, Inc.
    for j=1:length(x)
        y(j) =  b(1)*x(j) + z(1);
        z(1) = (b(2)*x(j) + z(2)) - a(2) * y(j);
        z(2) =  b(3)*x(j)         - a(3) * y(j);
    end
end
```

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	I/O	256 x 1	embedded.fi	No	Signed	16	14	48	No	-0.82061767578125	0.94378662109375
2	z	I/O	2 x 1	embedded.fi	No	Signed	16	14	47	No	-0.81060791015625	0.92193603515625
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	-0.999969482421875	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.

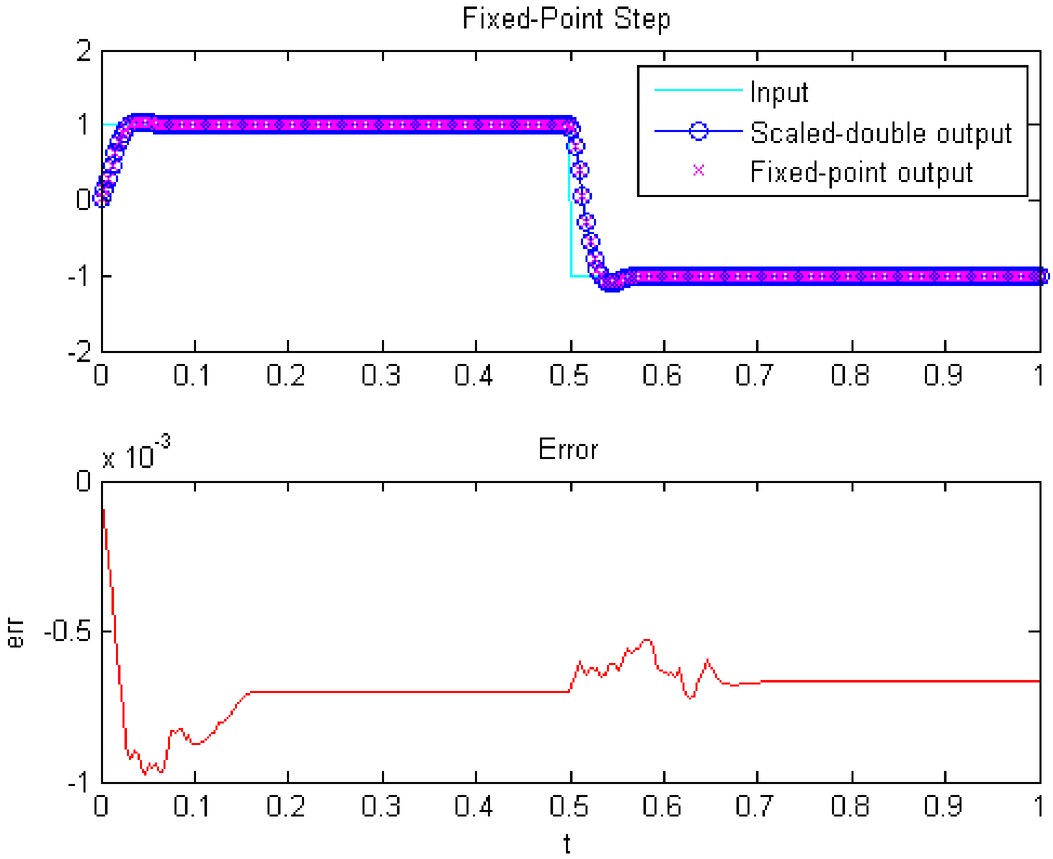
```
clearInstrumentationResults filter_fixed_point
```

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);
```

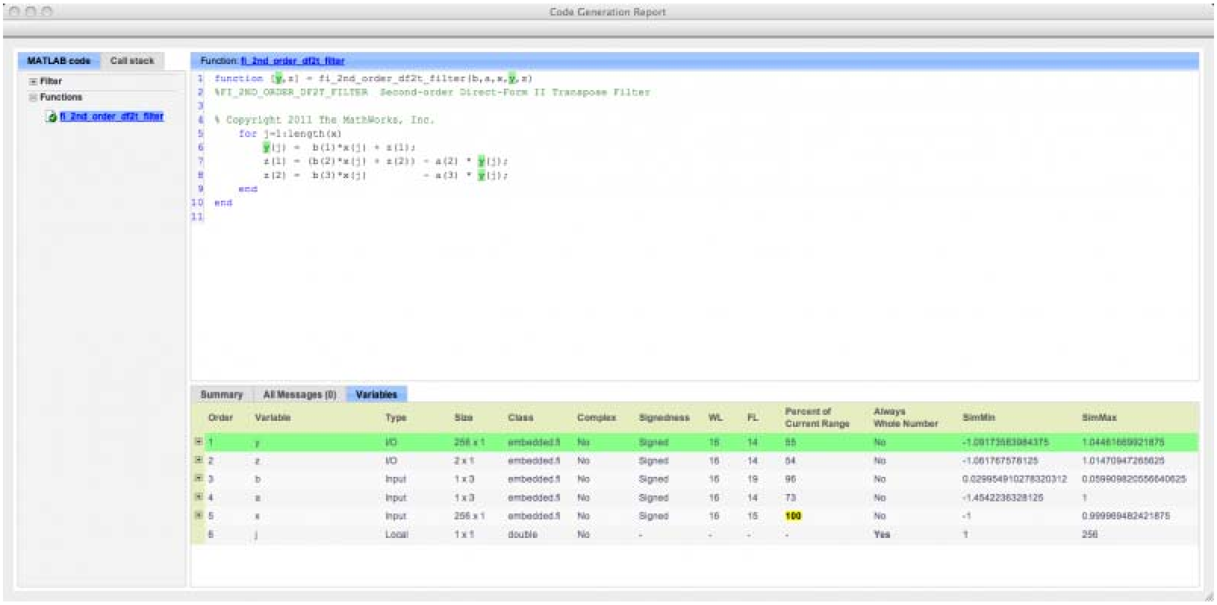
Plot the fixed-point outputs against the scaled-double outputs to verify that they meet your design criteria.

```
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
```



```
function [y,z] = fi_2nd_order_df2t_filter(b,a,x,y,z)
%FI_2ND_ORDER_DF2T_FILTER  Second-order Direct-Form II Transpose Filter

% Copyright 2011 The MathWorks, Inc.
    for j=1:length(x)
        y(j) =  b(1)*x(j) + z(1);
        z(1) = (b(2)*x(j) + z(2)) - a(2) * y(j);
        z(2) =  b(3)*x(j)         - a(3) * y(j);
    end
end

```

Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
1	y	I/O	256 x 1	embedded.fi	No	Signed	16	14	55	No	-1.09173583984375	1.04461669921875
2	z	I/O	2 x 1	embedded.fi	No	Signed	16	14	54	No	-1.061767578125	1.01470947265625
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	-1	0.999969482421875
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');

copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo',...
                  'fi_m_radix2fft_algorithm1_6_2.m'),'.','f');
copyfile(fullfile(matlabroot,'toolbox','fixedpoint','fidemos',...
                  'fi_m_radix2fft_withscaling.m'),'.','f');
```

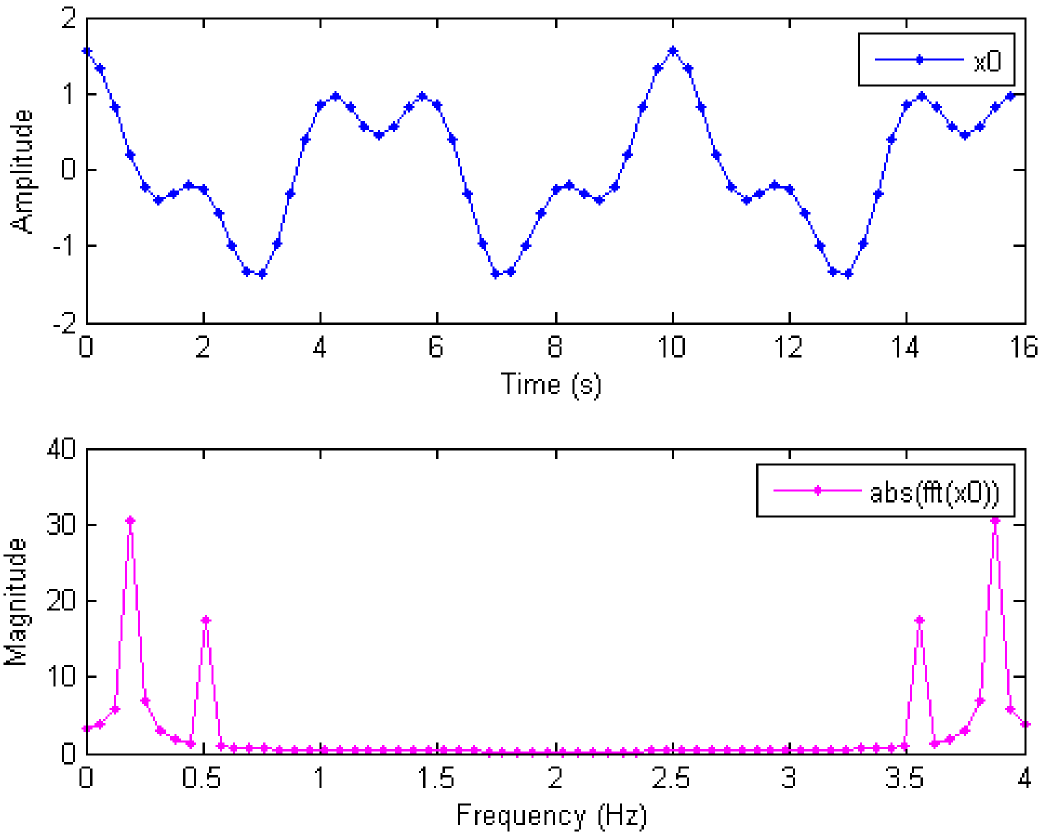
Run the following code to capture current states, and reset the global states.

```
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 requir
                                            % the input to be complex
y  = fft(x0);                               % Frequency-domain transformati

figure(gcf); clf
subplot(211); plot(t,real(x0),'b.-'); xlabel('Time (s)'); ylabel('Amplitude
subplot(212); plot(f,abs(y),'m.-'); xlabel('Frequency (Hz)'); ylabel('Magni
```



The peaks at 0.2 and 0.5 Hz in the frequency plot correspond to the two sinusoids of the time-domain signal at those frequencies.

Note the reflected peaks at 3.5 and 3.8 Hz. When the input to an FFT is real-valued, as it is in this case, then the output `y` is conjugate-symmetric:

$$y(k) = \mathrm{conj}(y(N - k)).$$

There are many different implementations of the FFT, each having its own costs and benefits. You may find that a different algorithm is better for your application than the one given here. This algorithm is used to provide you with an example of how you might begin your own exploration.

This example uses the decimation-in-time unit-stride FFT shown in Algorithm 1.6.2 on page 45 of the book *Computational Frameworks for the Fast Fourier Transform* by Charles Van Loan.

In pseudo-code, the algorithm in the textbook is as follows.

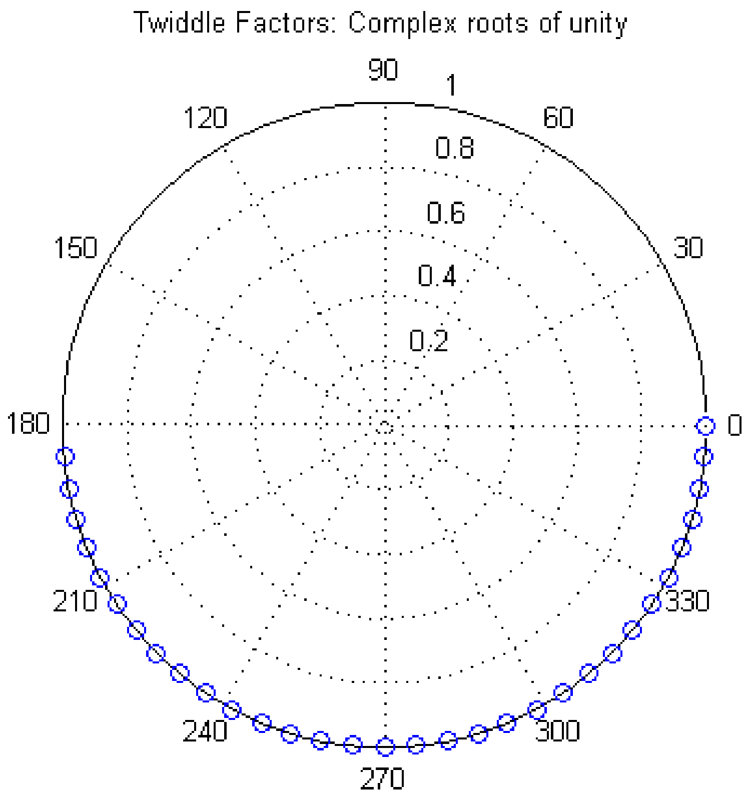
Algorithm 1.6.2. If x is a complex vector of length n and $n = 2^t$, then the following algorithm overwrites x with $F_n x$.

```
$x = P_n x$
$w = w_n^{(long)}$          (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
```

The textbook algorithm uses zero-based indexing. F_n is an n-by-n Fourier-transform matrix, P_n is an n-by-n bit-reversal permutation matrix, and w is a complex vector of twiddle factors. The twiddle factors, w , are complex roots of unity computed by the following algorithm:

```
function w = fi_radix2twiddles(n)
t = log2(n);
if floor(t) ~= t
  error('N must be an exact power of two.');
end
w = zeros(n-1,1);
k=1;
```

```
L=2;
% Equation 1.4.11, p. 34
while L<=n
  theta = 2*pi/L;
  % Algorithm 1.4.1, p. 23
  for j=0:(L/2 - 1)
    w(k) = complex( cos(j*theta), -sin(j*theta) );
    k = k + 1;
  end
  L = L*2;
end

figure(gcf);clf
w0 = fidemo.fi_radix2twiddles(n);
polar(angle(w0),abs(w0),'o')
title('Twiddle Factors: Complex roots of unity')
```



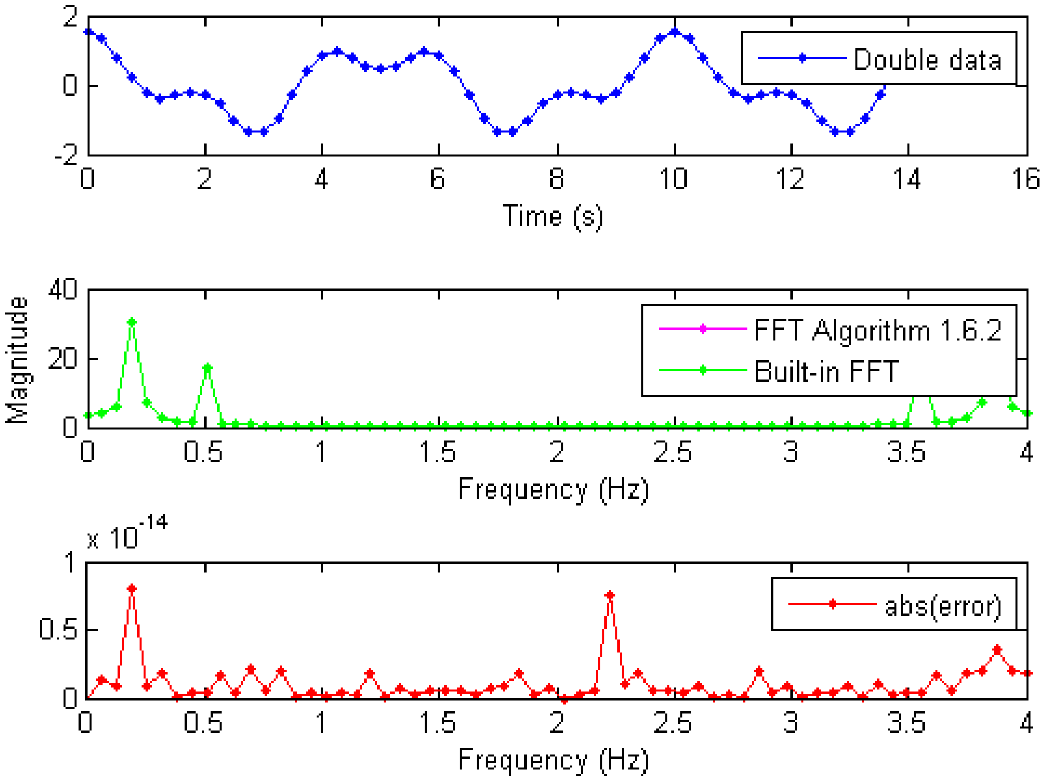
Verify Floating-Point Code

To implement the algorithm in MATLAB, you can use the `fidemo.fi_bitreverse` function to bit-reverse the input sequence, and you must add one to the indices to convert them from zero-based to one-based.

```
function x = fi_m_radix2fft_algorithm1_6_2(x, w)
n = length(x);  t = log2(n);
x = fidemo.fi_bitreverse(x,n);
for q=1:t
  L = 2^q; r = n/L; L2 = L/2;
  for k=0:(r-1)
    for j=0:(L2-1)
      temp          = w(L2-1+j+1) * x(k*L+j+L2+1);
      x(k*L+j+L2+1) = x(k*L+j+1)  - temp;
      x(k*L+j+1)    = x(k*L+j+1)  + temp;
    end
  end
end
```

To verify that you correctly implemented the algorithm in MATLAB, run a known signal through it and compare the results to the results produced by the MATLAB FFT function.

```
y = fi_m_radix2fft_algorithm1_6_2(x0, w0);

y0 = fft(x0); % MATLAB's built-in FFT for comparison

fidemo.fi_fft_demo_plot(real(x0),y,y0,Fs,'Double data', {'FFT Algorithm 1.6
```



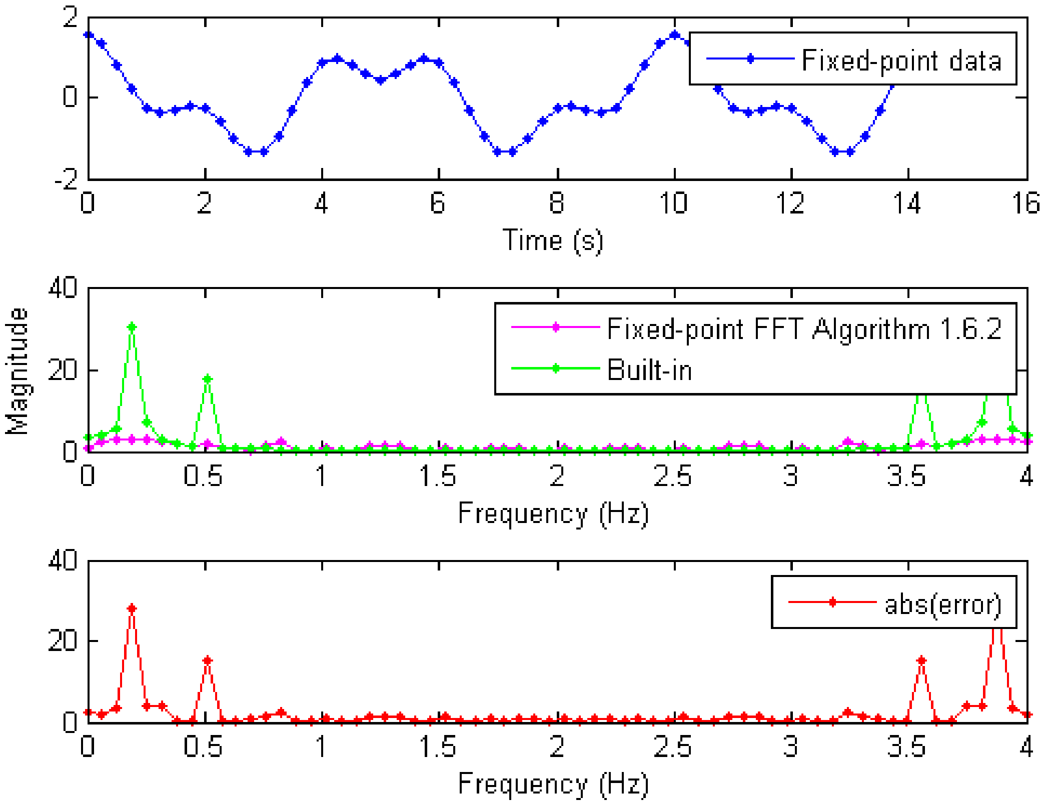
Because the error is within tolerance of the MATLAB built-in FFT function, you know you have correctly implemented the algorithm.

Identify Fixed-Point Issues

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

```
x = sfi(x0);  % Convert to signed fixed-point
w = sfi(w0);  % Convert to signed fixed-point
```

```
% Re-run the same algorithm with the fixed-point inputs
y  = fi_m_radix2fft_algorithm1_6_2(x,w);
fidemo.fi_fft_demo_plot(real(x),y,y0,Fs,'Fixed-point data', ...
                         {'Fixed-point FFT Algorithm 1.6.2','Built-in'});
```



Note that the magnitude plot (center) of the fixed-point FFT does not resemble the plot of the built-in FFT. The error (bottom plot) is much larger than what you would expect to see for round off error, so it is likely that overflow has occurred.

Use Min/Max Instrumentation to Identify Overflows

To instrument the MATLAB code, you create a MEX function from the MATLAB function using the buildInstrumentedMex command. The inputs to buildInstrumentedMex are the same as the inputs to fiaccel, but buildInstrumentedMex has no fi-object restrictions. The output of buildInstrumentedMex is a MEX function with instrumentation inserted, so when the MEX function is run, the simulated minimum and maximum values are recorded for all named variables and intermediate values.

The '-o' option is used to name the MEX function that is generated. If the '-o' option is not used, then the MEX function is the name of the MATLAB function with '_mex' appended. You can also name the MEX function the same as the MATLAB function, but you need to remember that MEX functions take precedence over MATLAB functions and so changes to the MATLAB function will not run until either the MEX function is re-generated, or the MEX function is deleted and cleared.

Create the input with a scaled double datatype so its values will attain full range and you can identify potential overflows.

```
x_scaled_double = fi(x0,'DataType','ScaledDouble');
buildInstrumentedMex fi_m_radix2fft_algorithm1_6_2 ...
    -o fft_instrumented -args {x_scaled_double w}
```

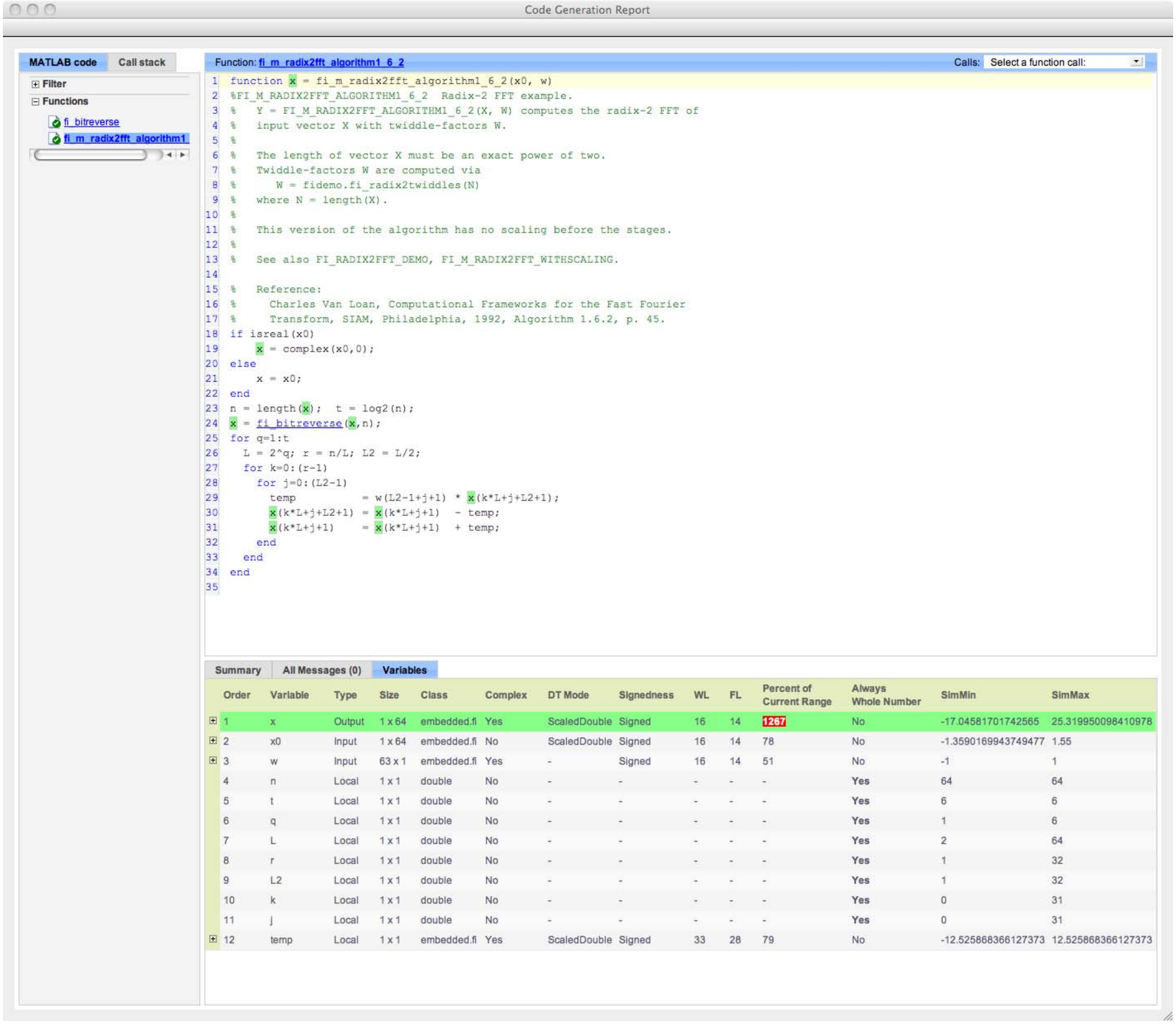
Run the instrumented MEX function to record min/max values.

```
y_scaled_double = fft_instrumented(x_scaled_double,w);
```

Show the instrumentation results.

```
showInstrumentationResults fft_instrumented
```

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



```
function x = fi_m_radix2fft_algorithm1_6_2(x0, 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.
%
%   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.
if isreal(x0)
    x = complex(x0,0);
else
    x = x0;
end
n = length(x);  t = log2(n);
x = 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
```

	Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
⊞	1	x	Output	1 x 64	embedded.fi	Yes	ScaledDouble	Signed	16	14	1267	No	-17.04581701742565	25.319950098410978
⊞	2	x0	Input	1 x 64	embedded.fi	No	ScaledDouble	Signed	16	14	78	No	-1.3590169943749477	1.55
⊞	3	w	Input	63 x 1	embedded.fi	Yes	-	Signed	16	14	51	No	-1	1
	4	n	Local	1 x 1	double	No	-	-	-	-	-	Yes	64	64
	5	t	Local	1 x 1	double	No	-	-	-	-	-	Yes	6	6
	6	q	Local	1 x 1	double	No	-	-	-	-	-	Yes	1	6
	7	L	Local	1 x 1	double	No	-	-	-	-	-	Yes	2	64
	8	r	Local	1 x 1	double	No	-	-	-	-	-	Yes	1	32
	9	L2	Local	1 x 1	double	No	-	-	-	-	-	Yes	1	32
	10	k	Local	1 x 1	double	No	-	-	-	-	-	Yes	0	31
	11	j	Local	1 x 1	double	No	-	-	-	-	-	Yes	0	31
⊞	12	temp	Local	1 x 1	embedded.fi	Yes	ScaledDouble	Signed	33	28	79	No	-12.525868366127373	12.525868366127373

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 & Schafer 1989, equation 9.101], [Welch 1969].

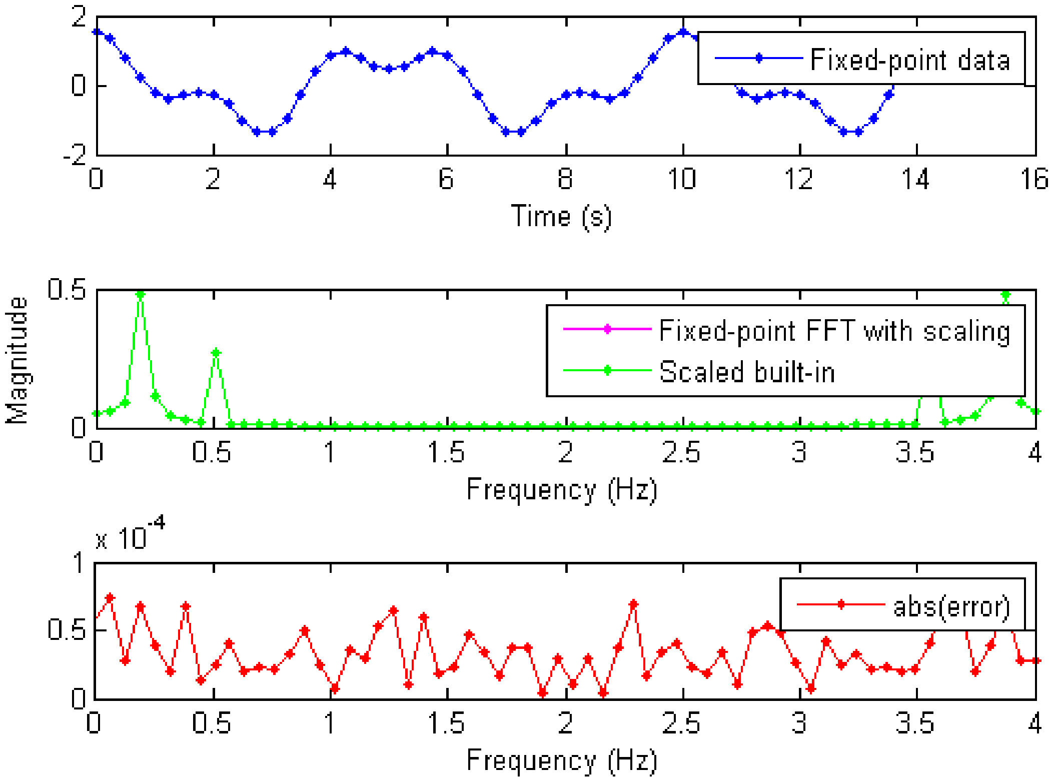
However, if you scale the input to each of the stages of the FFT by 1/2, you can obtain an overall scaling of 1/n and produce a noise-to-signal ratio proportional to n [Oppenheim & Schafer 1989, equation 9.105], [Welch 1969].

An efficient way to scale by 1/2 in fixed-point is to right-shift the data. To do this, you use the bit shift right arithmetic function `bitsra`. After scaling each stage of the FFT, and optimizing the index variable computation, your algorithm becomes:

```
function x = fi_m_radix2fft_withscaling(x, w)
n = length(x);  t = log2(n);
x = fidemo.fi_bitreverse(x,n);
% Generate index variables as integer constants so they are not computed in
% the loop.
LL = int32(2.^(1:t)); rr = int32(n./LL); LL2 = int32(LL./2);
for q=1:t
    L = LL(q); r = rr(q); L2 = LL2(q);
    for k=0:(r-1)
        for j=0:(L2-1)
            temp          = w(L2-1+j+1) * x(k*L+j+L2+1);
            x(k*L+j+L2+1) = bitsra(x(k*L+j+1) - temp, 1);
            x(k*L+j+1)    = bitsra(x(k*L+j+1) + temp, 1);
        end
    end
end
```

Run the scaled algorithm with fixed-point data.

```
x = sfi(x0);
w = sfi(w0);

y = fi_m_radix2fft_withscaling(x,w);

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



You can see that the scaled fixed-point FFT algorithm now matches the built-in FFT to a tolerance that is expected for 16-bit fixed-point data.

References

Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform,* SIAM, 1992.

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

Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing,* Prentice Hall, 1989.

Peter D. Welch, "A Fixed-Point Fast Fourier Transform Error Analysis," IEEE Transactions on Audio and Electroacoustics, Vol. AU-17, No. 2, June 1969, pp. 151-157.

Run the following code to restore the global states.

```
fipref(FIPREF_STATE);
clearInstrumentationResults fft_instrumented
clear fft_instrumented
```

Run the following code to delete the temporary directory.

```
tempdirObj.cleanUp;
```

Detect Limit Cycles in Fixed-Point State-Space Systems

This example shows how to analyze a fixed-point state-space system to detect limit cycles.

The example focuses on detecting large scale limit cycles due to overflow with zero inputs and highlights the conditions that are sufficient to prevent such oscillations.

References:

[1] Richard A. Roberts and Clifford T. Mullis, "Digital Signal Processing", Addison-Wesley, Reading, Massachusetts, 1987, ISBN 0-201-16350-0, Section 9.3.

[2] S. K. Mitra, "Digital Signal Processing: A Computer Based Approach", McGraw-Hill, New York, 1998, ISBN 0-07-042953-7.

Select a State-Space Representation of the System.

We observe that the system is stable by observing that the eigenvalues of the state-transition matrix A have magnitudes less than 1.

```
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

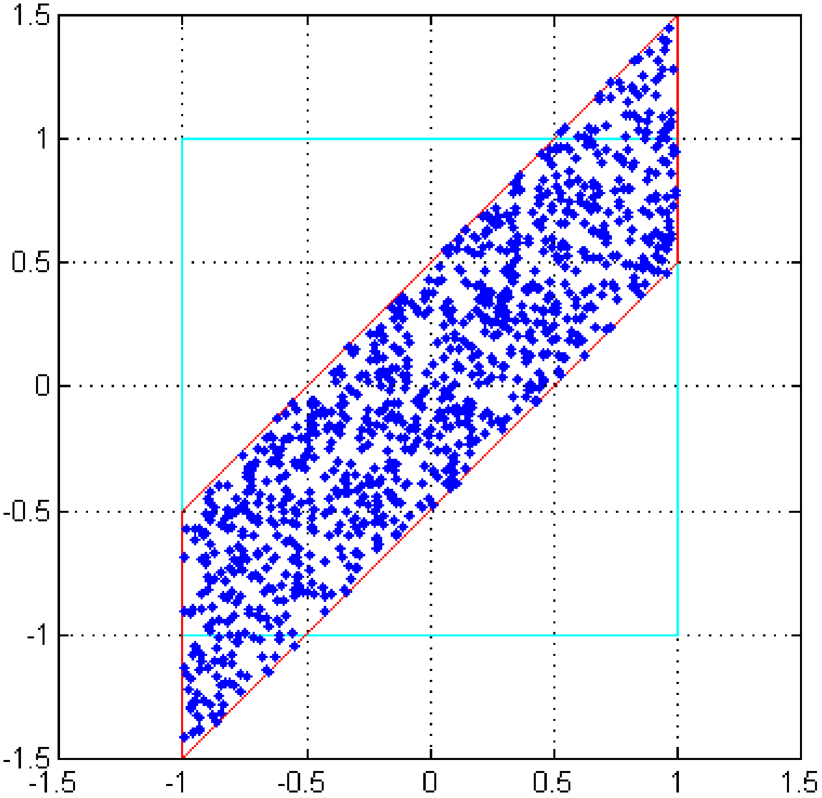
```
type(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','fisiso
function [y,z] = fisisostatespacefilter(A,B,C,D,x,z)
%FISISOSTATESPACEFILTER Single-input, single-output statespace filter
% [Y,Zf] = FISISOSTATESPACEFILTER(A,B,C,D,X,Zi) filters data X with
% initial conditions Zi with the state-space filter defined by matrices
% A, B, C, D.  Output Y and final conditions Zf are returned.

%   Copyright 2004-2011 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $

y = x;
z(:,2:length(x)+1) = 0;
for k=1:length(x)
  y(k)      = C*z(:,k) + D*x(k);
  z(:,k+1) = A*z(:,k) + B*x(k);
end
```

Floating-Point Filter

Create a floating-point filter and observe the trajectory of the states.

First, we choose random states within the unit square and observe where they are projected after one step of being multiplied by the state-transition matrix A.

```
rng('default');
clf
x1 = [-1 1 1 -1 -1];
y1 = [-1 -1 1 1 -1];
plot(x1,y1,'c')
axis([-1.5 1.5 -1.5 1.5]); axis square; grid;
hold on

% Plot the projection of the square
p = A*[x1;y1];
plot(p(1,:),p(2,:),'r')

r = 2*rand(2,1000)-1;
pr = A*r;
plot(pr(1,:),pr(2,:),'.')
```



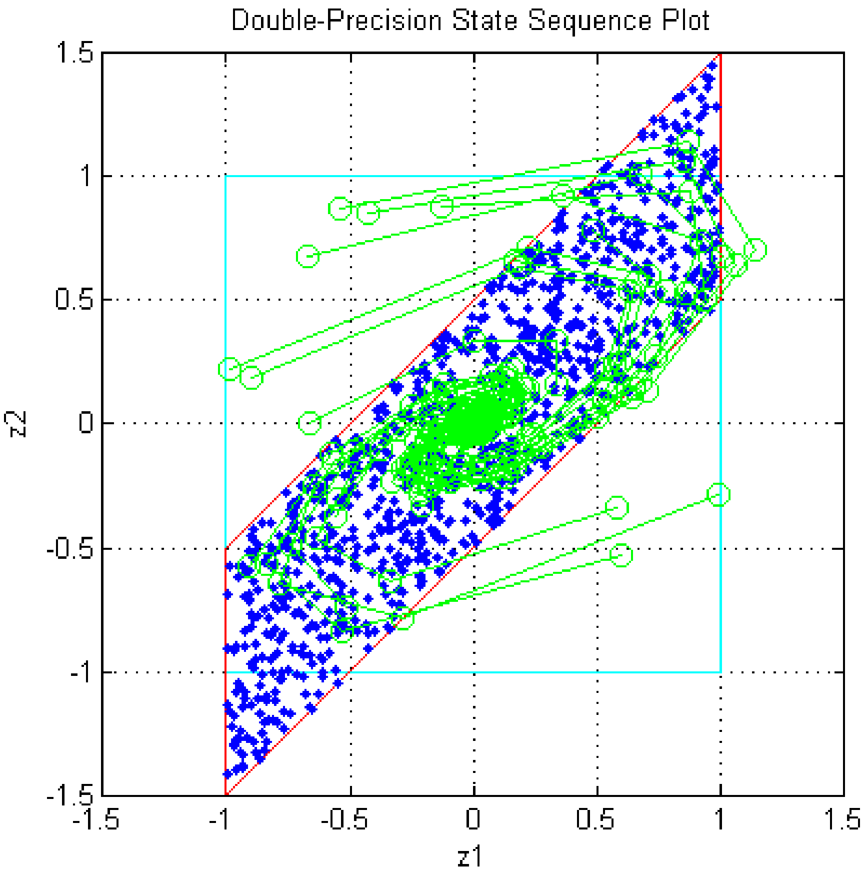
Random Initial States Followed Through Time

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)
```

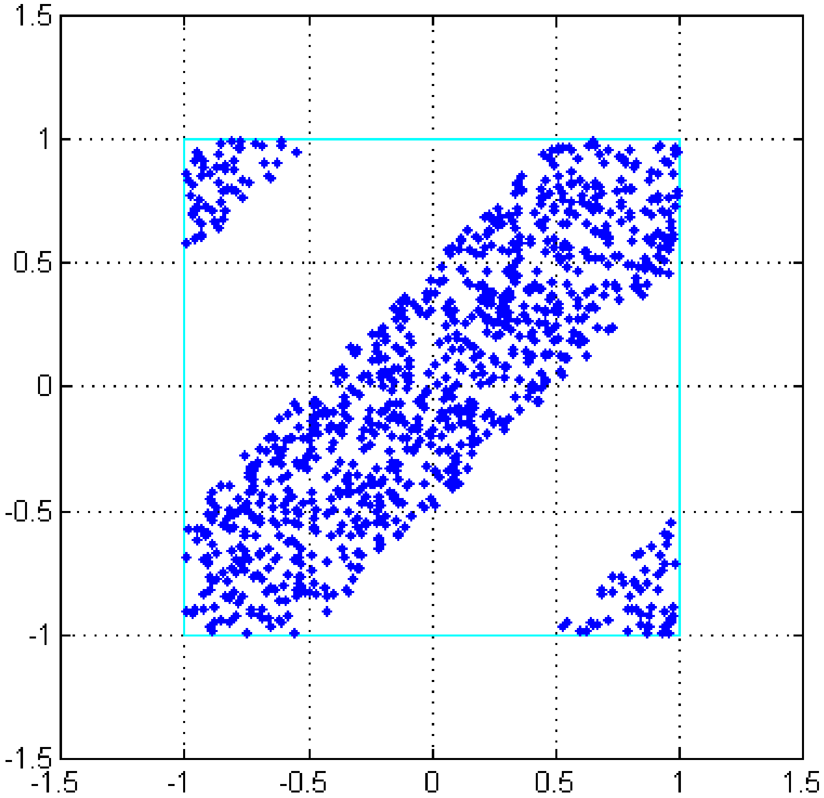
```
A =

         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,:),'.')
```

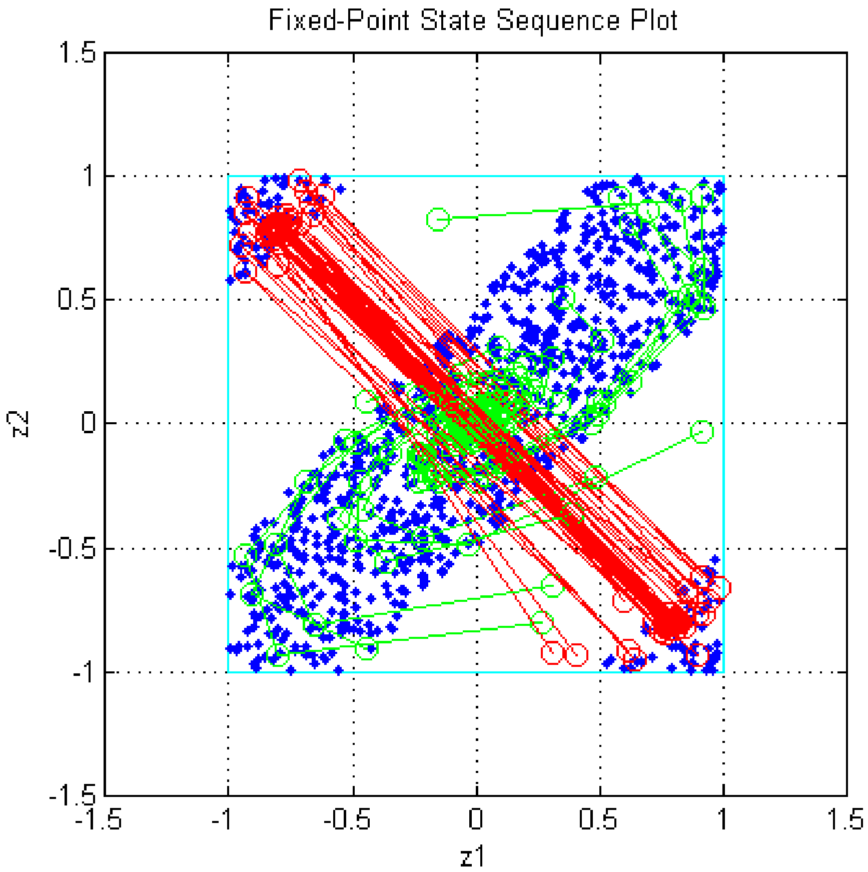


Execute the Fixed-Point Filter.

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., abs(eig(A))<1,
- the matrix A is normal i.e., A'*A = A*A'.

Note that for the current representation, the second condition does not hold.

Apply Similarity Transform to Create a Normal A

We now apply a similarity transformation to the original system that will create a normal state-transition matrix A2.

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

Similarity transformations preserve eigenvalues, as a result of which the system transfer function of the transformed system remains same as before. However, the transformed state transformation matrix A2 is normal.

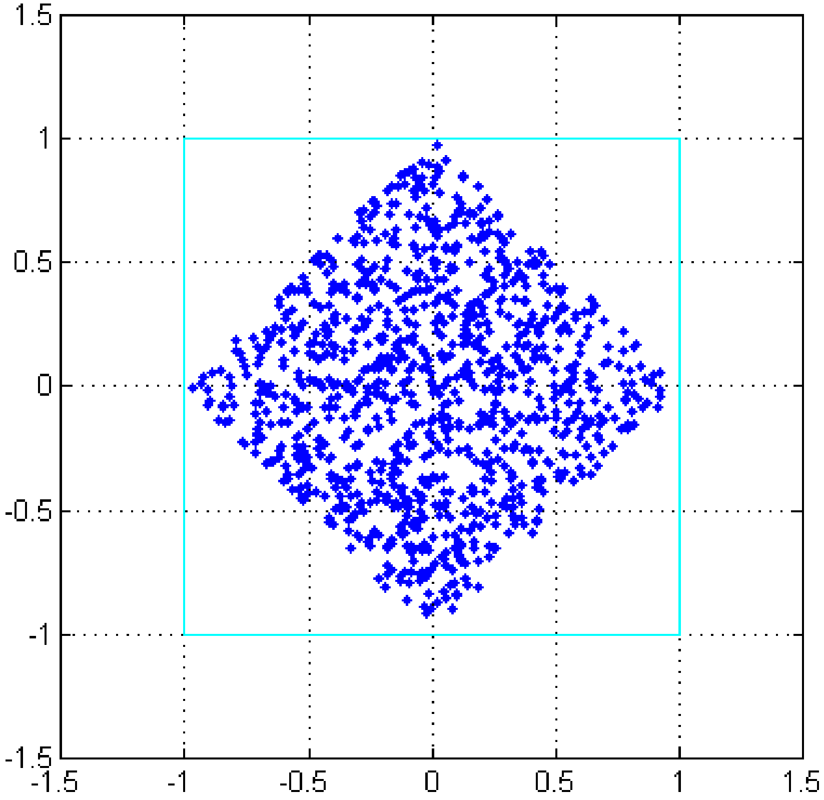
Check for Limit Cycles on the Transformed System.

Plot the Projection of the Square of the Normal-Form System

Now the projection of random initial states inside the unit square all contract uniformly. This is the result of the state transition matrix A2 being normal. The states are also rotated by 90 degrees counterclockwise.

```
clf
r = 2*rand(2,1000)-1;
pr = A2*r;
plot([-1 1 1 -1 -1],[-1 -1 1 1 -1],'c')
axis([-1.5 1.5 -1.5 1.5]); axis square; grid;
hold on
```

```
plot(pr(1,:),pr(2,:),'.')
```

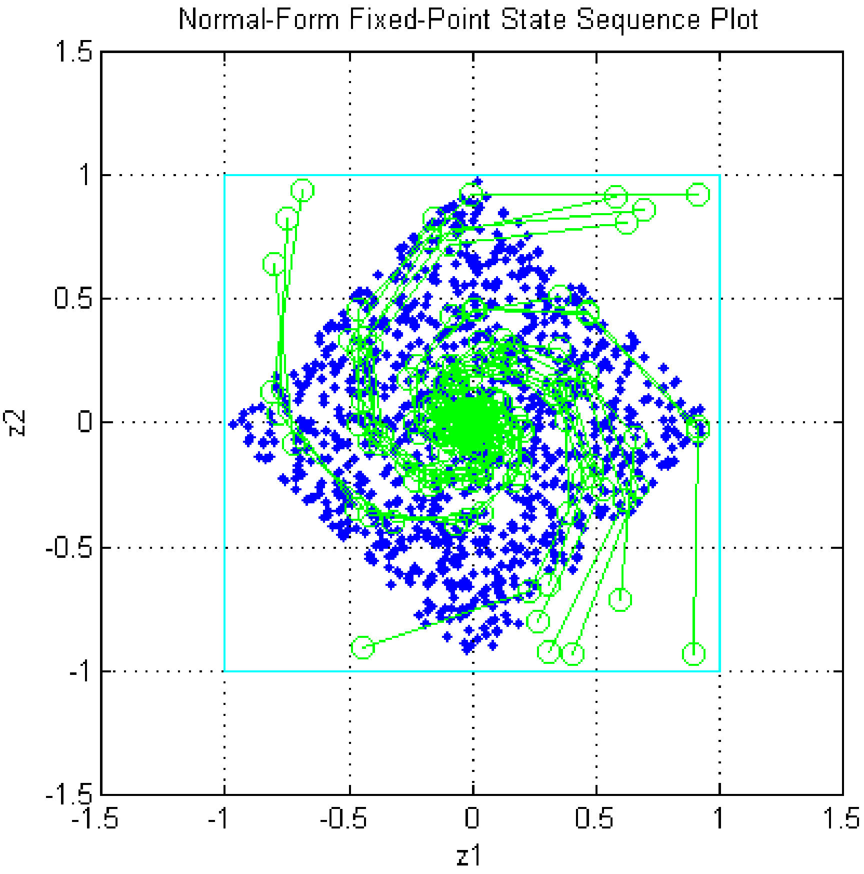


Plot the State Sequence

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

```
x = fi(zeros(10,1),1,16,15,'fimath',F);
zi = fi([0;0],1,16,15,'fimath',F);
q = assignmentquantizer(zi);
e = double(eps(zi));
```

```
rng('default');
for k=1:20
  y = x;
  zi(:) = randquant(q,size(A,1),1);
  [y,zf] = fidemo.fisisostatespacefilter(A2,B2,C2,D2,x,zi);
  if abs(double(zf(end)))>0.5, c='ro-'; else, c='go-'; end
  plot(zf(1,:), zf(2,:),c,'markersize',8);
end
title('Normal-Form Fixed-Point State Sequence Plot');
xlabel('z1'); ylabel('z2')
```



Trying this for other randomly chosen initial states illustrates that there is no region from which the filter is unable to recover.

```
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);
```

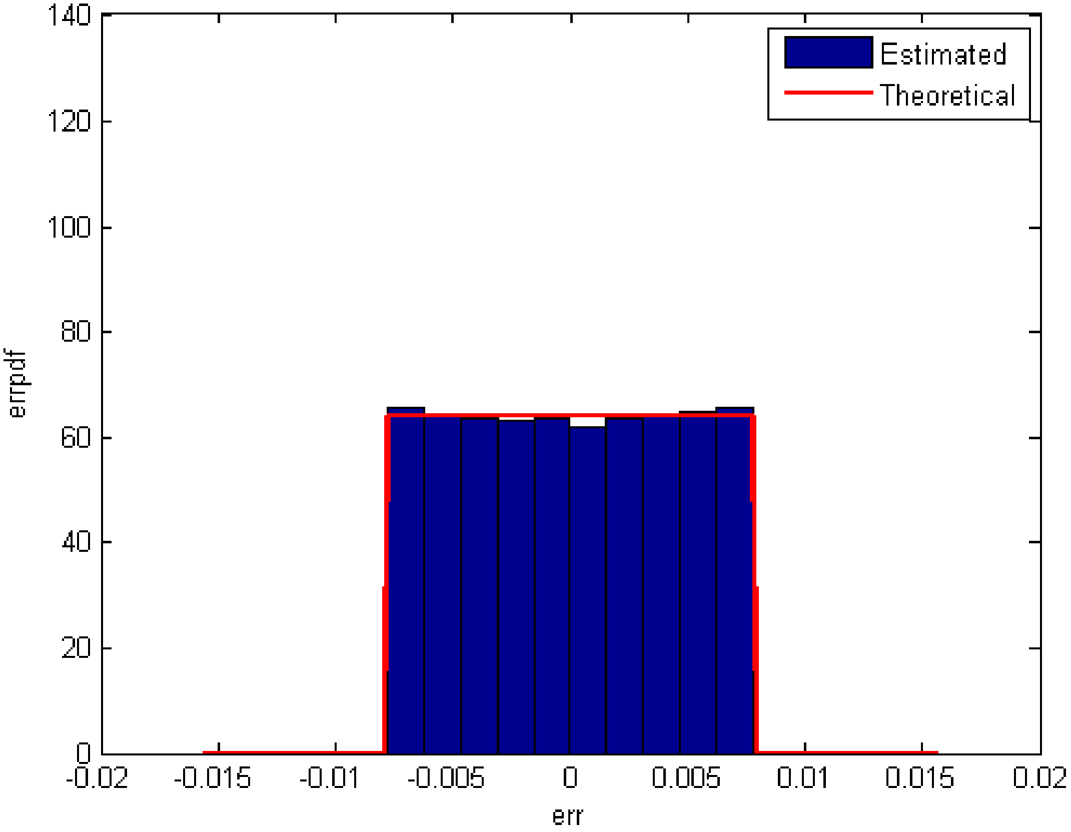
Fix: Round Towards Zero.

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
```

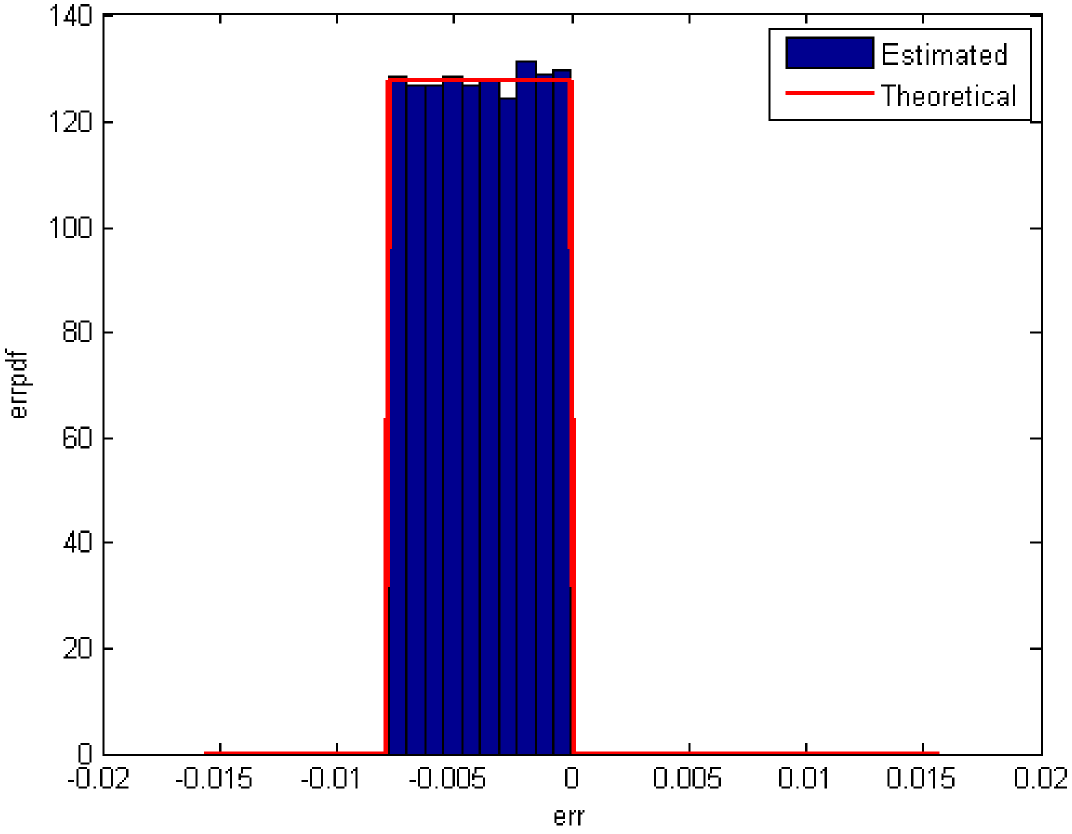


Floor: Round Towards Minus Infinity.

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)
```

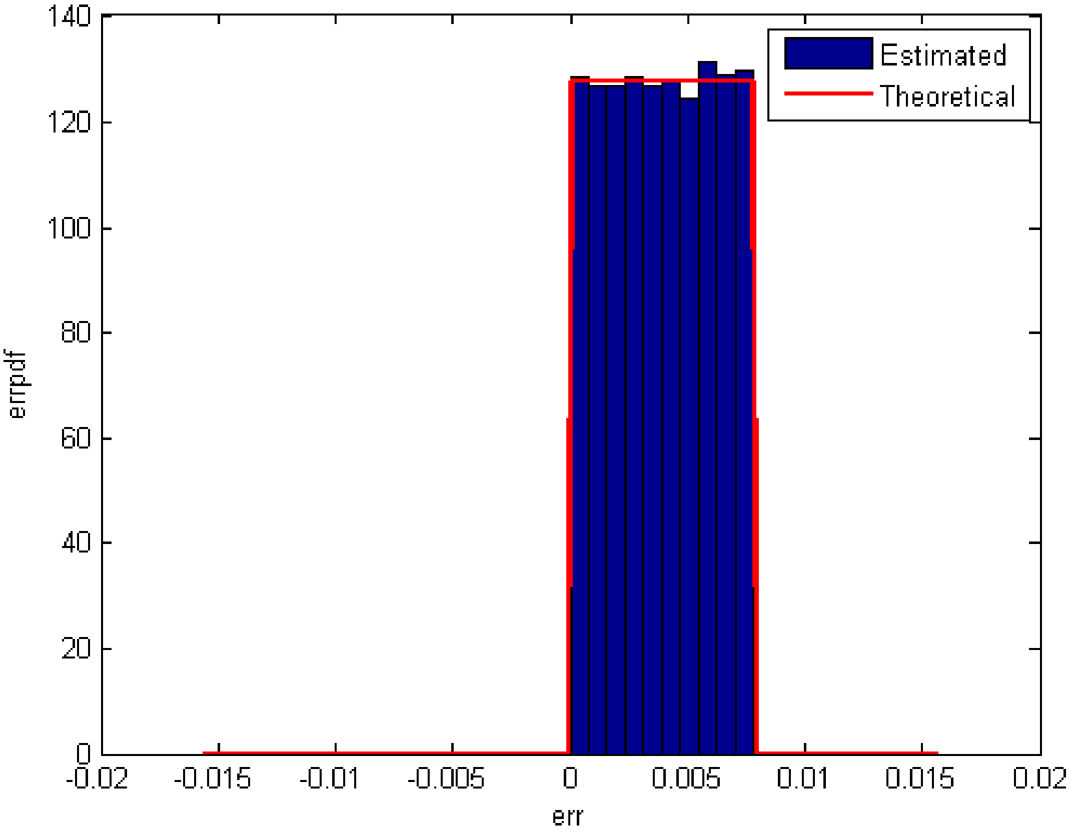
```
Estimated   error variance (dB) = -52.9148
Theoretical error variance (dB) = -52.936
Estimated   mean = -0.0038956
Theoretical mean = -0.0039063
```



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
```



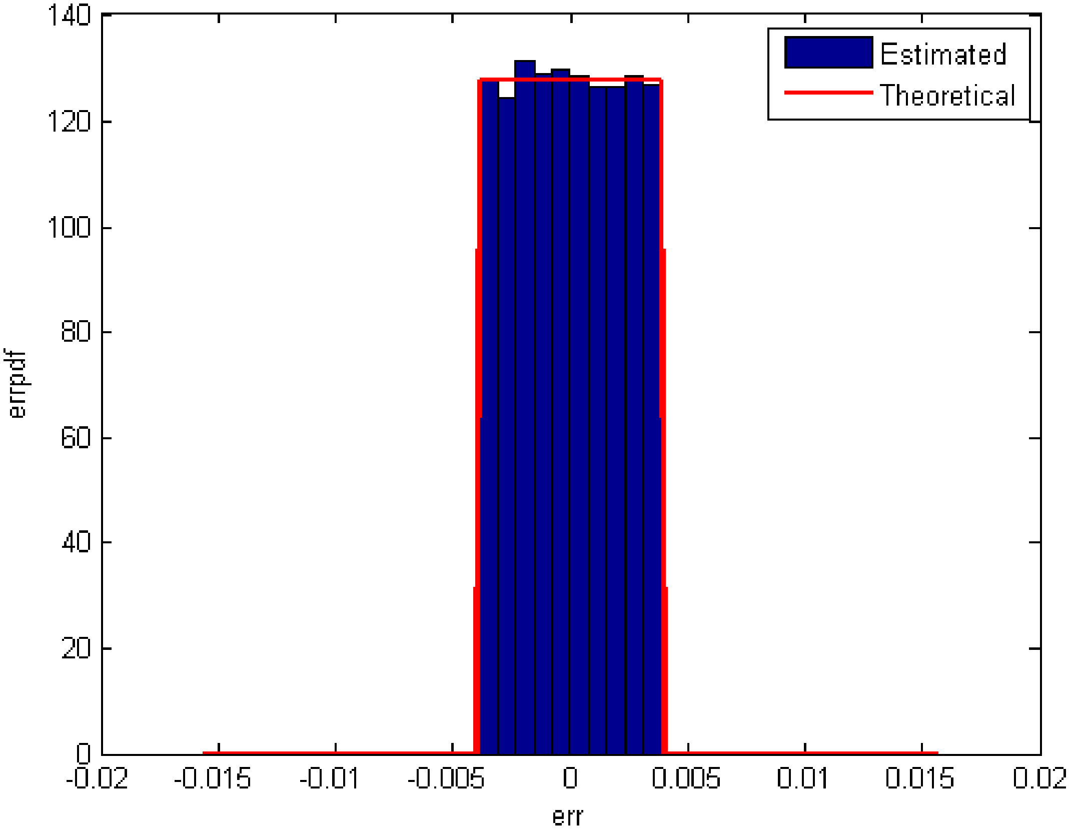
Round: Round to Nearest. In a Tie, Round to Largest Magnitude.

Round is more accurate than floor, but all values smaller than 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.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
```

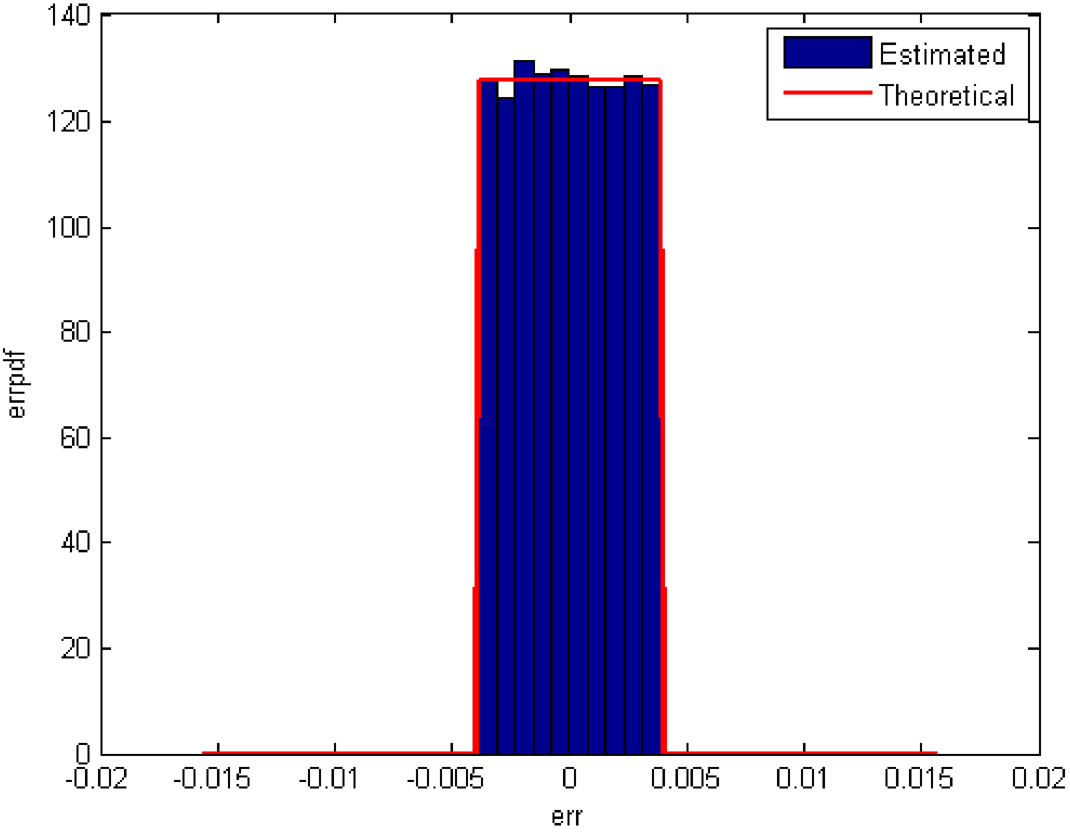
```
Estimated   error variance (dB) = -52.9579
Theoretical error variance (dB) = -52.936
Estimated   mean = -2.212e-06
Theoretical mean = 0
```



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

Convergent rounding eliminates the bias introduced by ordinary "round" caused by always rounding the tie in the same direction.

```
q = quantizer('convergent',[8 7]);
err = quantize(q,u) - u;
f_t = errpdf(q,xi);
mu_t = errmean(q);
v_t  = errvar(q);
% Theoretical variance = eps(q)^2 / 12
% Theoretical mean     = 0
fidemo.qerrordemoplot(q,f_t,xi,mu_t,v_t,err)

Estimated   error variance (dB) = -52.9579
Theoretical error variance (dB) = -52.936
Estimated   mean = -2.212e-06
Theoretical mean = 0
```



Comparison of Nearest vs. Convergent

The error probability density function for convergent rounding is difficult to distinguish from that of round-to-nearest by looking at the plot.

The error p.d.f. of convergent is

```
f(err) = 1/eps(q),  for -eps(q)/2 <= err <= eps(q)/2, and 0 otherwise
```

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

```
f(err) = 1/eps(q),  for -eps(q)/2 <  err <= eps(q)/2, and 0 otherwise
```

Note that the error p.d.f. of convergent is symmetric, while round is slightly biased towards the positive.

The only difference is the direction of rounding in a tie.

```
x=[-3.5:3.5]';
[x convergent(x) nearest(x)]
```

```
ans =

   -3.5000   -4.0000   -3.0000
   -2.5000   -2.0000   -2.0000
   -1.5000   -2.0000   -1.0000
   -0.5000         0         0
    0.5000         0    1.0000
    1.5000    2.0000    2.0000
    2.5000    2.0000    3.0000
    3.5000    4.0000    4.0000
```

Plot Helper Function

The helper function that was used to generate the plots in this example is listed below.

```
type(fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','qerror
```

```
function qerrordemoplot(q,f_t,xi,mu_t,v_t,err)
```

```
%QERRORDEMOPLOT  Plot function for QERRORDEMO.
%    QERRORDEMOPLOT(Q,F_T,XI,MU_T,V_T,ERR) produces the plot and display us
%    the example function QERRORDEMO, where Q is the quantizer whos attribu
%    being analyzed; F_T is the theoretical quantization error probability
%    density function for quantizer Q computed by ERRPDF; XI is the domain
%    values being evaluated by ERRPDF; MU_T is the theoretical quantization
%    error mean of quantizer Q computed by ERRMEAN; V_T is the theoretical
%    quantization error variance of quantizer Q computed by ERRVAR; and ERR
%    is the error generated by quantizing a random signal by quantizer Q.
%
%    See QERRORDEMO for examples of use.

%    Copyright 1999-2012 The MathWorks, Inc.

v=10*log10(var(err));
disp(['Estimated   error variance (dB) = ',num2str(v)]);
disp(['Theoretical error variance (dB) = ',num2str(10*log10(v_t))]);
disp(['Estimated   mean = ',num2str(mean(err))]);
disp(['Theoretical mean = ',num2str(mu_t)]);
[n,c]=hist(err);
figure(gcf)
bar(c,n/(length(err)*(c(2)-c(1))),'hist');
line(xi,f_t,'linewidth',2,'color','r');
% Set the ylim uniformly on all plots
set(gca,'ylim',[0 max(errpdf(quantizer(q.format,'nearest'),xi)*1.1)])
legend('Estimated','Theoretical')
xlabel('err'); ylabel('errpdf')
```

Normalize Data for Lookup Tables

This example shows how to normalize data for use in lookup tables.

Lookup tables are a very efficient way to write computationally-intense functions for fixed-point embedded devices. For example, you can efficiently implement logarithm, sine, cosine, tangent, and square-root using lookup tables. You normalize the inputs to these functions to produce a smaller lookup table, and then you scale the outputs by the normalization factor. This example shows how to implement the normalization function that is used in examples Implement Fixed-Point Square Root Using Lookup Table and Implement Fixed-Point Log2 Using Lookup Table.

Setup

To assure that this example does not change your preferences or settings, this code stores the original state, and you will restore it at the end.

```
originalFormat = get(0, 'format'); format long g
originalWarningState = warning('off','fixed:fi:underflow');
originalFiprefState = 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 <= 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-poin
    u = removefimath(u);
    NLZLUT = number_of_leading_zeros_look_up_table();
    word_length = u.WordLength;
    u_fraction_length = u.FractionLength;
    B = 8;
    leftshifts=int8(0);
    % Reinterpret the input as an unsigned integer.
    T_unsigned_integer = numerictype(0, word_length, 0);
    v = reinterpretcast(u,T_unsigned_integer);
    F = fimath('OverflowAction','Wrap',...
               'RoundingMethod','Floor',...
               'SumMode','KeepLSB',...
               'SumWordLength',v.WordLength);
    v = setfimath(v,F);
    % Unroll the loop in generated code so there will be no branching.
    for k = coder.unroll(1:ceil(word_length/B))
        % For each iteration, see how many leading zeros are in the high
        % byte of V, and shift them out to the left. Continue with the
        % shifted V for as many bytes as it has.
        %
        % The index is the high byte of the input plus 1 to make it a
        % one-based index.
        index = int32(bitsra(v, word_length - B) + uint8(1));
        % Index into the number-of-leading-zeros lookup table.  This lookup
        % table takes in a byte and returns the number of leading zeros in
        % binary representation.
        shiftamount = NLZLUT(index);
        % Left-shift out all the leading zeros in the high byte.
        v = bitsll(v,shiftamount);
        % Update the total number of left-shifts
        leftshifts = leftshifts+shiftamount;
    end
    % The input has been left-shifted so the most-significant-bit is a 1.
    % Reinterpret the output as unsigned with one integer bit, so
    % that 1 <= x < 2.
    T_x = numerictype(0,word_length,word_length-1);
    x = reinterpretcast(v, T_x);
    x = removefimath(x);
    % Let Q = int(u).  Then u = Q*2^(-u_fraction_length),
    % and x = Q*2^leftshifts * 2^(1-word_length).  Therefore,
    % u = x*2^n, where n is defined as:
    n = word_length -  u_fraction_length - leftshifts - 1;
end
```

Number-of-Leading-Zeros Lookup Table

Function `number_of_leading_zeros_look_up_table` is used by `fi_normalize_unsigned_8_bit_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 ...
                   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 = 1.001101000000000_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  0000000    Left-shifted by 1 additional bit, for a total of 9.
```

Reinterpret these bits as unsigned fixed-point with 15 fractional bits.

```
x = 1.001101000000000_2 = 1.203125
```

The value for n is the word-length of u, minus the fraction length of u, minus the number of left shifts, minus 1.

```
n = 16 - 8 - 9 - 1 = -2.
```

And so your result is:

```
[x,n] = fi_normalize_unsigned_8_bit_byte(u)
```

```
x =

                  1.203125

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

n =

   -2
```

Comparing binary values, you can see that x has the same bits as u, left-shifted by 9 bits.

```
binary_representation_of_u = bin(u)
binary_representation_of_x = bin(x)
```

```
binary_representation_of_u =

0000000001001101


binary_representation_of_x =

1001101000000000
```

Cleanup

Restore original state.

```
set(0, 'format', originalFormat);
warning(originalWarningState);
fipref(originalFiprefState);
```

Implement Fixed-Point Log2 Using Lookup Table

This example shows how to implement fixed-point log2 using a lookup table. Lookup tables generate efficient code for embedded devices.

Setup

To assure that this example does not change your preferences or settings, this code stores the original state, and you will restore it at the end.

```
originalFormat = get(0, 'format'); format long g
originalWarningState = warning('off','fixed:fi:underflow');
originalFiprefState = get(fipref); reset(fipref)
```

Log2 Implementation

The log2 algorithm is summarized here.

1 Declare the number of bits in a byte, B, as a constant. In this example, B=8.

2 Use function fi_normalize_unsigned_8_bit_byte() described in example Normalize Data for Lookup Tables to normalize the input u>0 such that u = x * 2^n and 1 <= 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 = uint8(x_B)-2^(B-1) + 1 is used as an index to LOG2LUT so that log2(x_B) can be evaluated by looking up the index log2(x_B) = 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 reinterpretcast().

6 Finally, compute the output using the lookup table and linear interpolation:

```
log2( u ) = log2( x * 2^n )
          = n + log2( x )
          = n + LOG2LUT( i ) + r * ( LOG2LUT( i+1 ) - LOG2LUT( i ) )
```

```
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 = eml.nullcopy(fi(zeros(size(u)), numerictype(LOG2LUT), fimath(LOG2LU
    B = 8; % Number of bits in a byte
    w = u.WordLength;
    for k = 1:prod(size(u))
        assert(u(k)>0,'Input must be positive.');
        % Normalize the input such that u = x * 2^n and 1 <= x < 2
        [x,n] = fi_normalize_unsigned_8_bit_byte(u(k));
        % Extract the high byte of x
        high_byte = uint8( storedInteger(bitsliceget(x, w, w - B + 1)) );
        % Convert the high byte into an index for LOG2LUT
        i =  high_byte - 2^(B-1) + 1;
        % Interpolate between points.
        % The upper byte was used for the index into LOG2LUT
        % The remaining bits make up the fraction between points.
        T_unsigned_fraction = numerictype(0, w-B, w-B);
        r = reinterpretcast(bitsliceget(x,w-B,1), T_unsigned_fraction);
        y(k) = n + LOG2LUT(i) + ...
               r*(LOG2LUT(i+1) - LOG2LUT(i)) ;
    end
    % Remove fimath from the output to insulate the caller from math settin
    % declared inside this function.
    y = removefimath(y);
end
```

Log2 Lookup Table

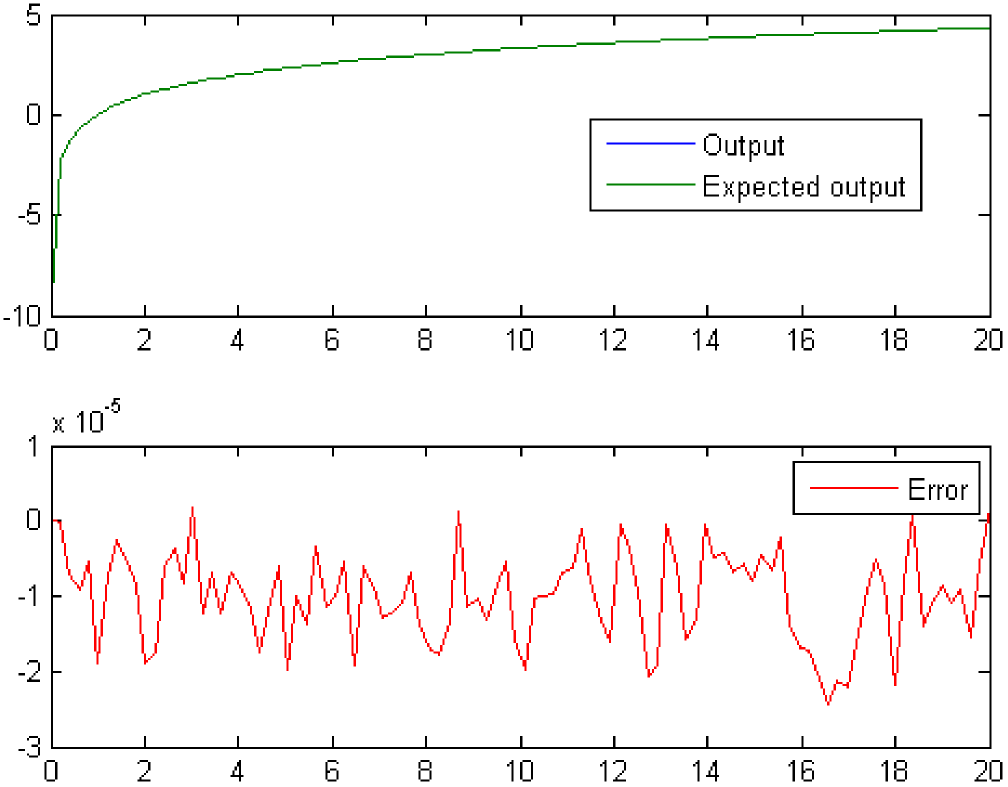
Function `log2_lookup_table` loads the lookup table of `log2` values. You can create the table by running:

```
B = 8;
log2_table = log2((2^(B-1) : 2^(B)) / 2^(B - 1))
```

```
function LOG2LUT = log2_lookup_table()
    B = 8;  % Number of bits in a byte
    % log2_table = log2((2^(B-1) : 2^(B)) / 2^(B - 1))
    log2_table = [0.000000000000000   0.011227255423254   0.022367813028454
                  0.044394119358453   0.055282435501190   0.066089190457773
                  0.087462841250339   0.098032082960527   0.108524456778169
                  0.129283016944966   0.139551352398794   0.149747119504682
                  0.169925001442312   0.179909090014934   0.189824558880017
                  0.209453365628950   0.219168520462162   0.228818690495881
                  0.247927513443586   0.257387842692652   0.266786540694901
                  0.285402218862248   0.294620748891627   0.303780748177103
                  0.321928094887362   0.330916878114617   0.339850002884625
                  0.357552004618084   0.366322214245816   0.375039431346925
                  0.392317422778760   0.400879436282184   0.409390936137702
                  0.426264754702098   0.434628227636725   0.442943495848728
                  0.459431618637297   0.467605550082997   0.475733430966398
                  0.491853096329675   0.499845887083205   0.507794640198696
                  0.523561956057013   0.531381460516312   0.539158811108031
                  0.554588851677637   0.562242424221073   0.569855608330948
                  0.584962500721156   0.592457037268080   0.599912842187128
                  0.614709844115208   0.622051819456376   0.629356620079610
                  0.643856189774725   0.651051691178929   0.658211482751795
                  0.672425341971496   0.679480099505446   0.686500527183218
                  0.700439718141092   0.707359132080883   0.714245517666123
                  0.727920454563199   0.734709620225838   0.741466986401147
                  0.754887502163469   0.761551232444479   0.768184324776926
                  0.781359713524660   0.787902559391432   0.794415866350106
                  0.807354922057604   0.813781191217037   0.820178962415188
                  0.832890014164742   0.839203788096944   0.845490050944375
                  0.857980995127572   0.864186144654280   0.870364719583405
                  0.882643049361841   0.888743248898259   0.894817763307943
                  0.906890595608518   0.912889336229962   0.918863237274595
                  0.930737337562886   0.936637939002571   0.942514505339240
                  0.954196310386875   0.960001932068081   0.965784284662087
                  0.977279923499916   0.982993574694310   0.988684686772166
                  1.000000000000000];

     % Cast to fixed point with the most accurate rounding method
     WL = 4*B;  % Word length
     FL = 2*B;  % Fraction length
     LOG2LUT = fi(log2_table,1,WL,FL,'RoundingMethod','Nearest');
     % Set fimath for the most efficient math operations
     F = fimath('OverflowAction','Wrap',...
                'RoundingMethod','Floor',...
                'SumMode','SpecifyPrecision',...
                'SumWordLength',WL,...
                'SumFractionLength',FL,...
                'ProductMode','SpecifyPrecision',...
                'ProductWordLength',WL,...
                'ProductFractionLength',2*FL);
     LOG2LUT = setfimath(LOG2LUT,F);
 end
```

Example

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

y = fi_log2lookup_8_bit_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 <= 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 = uint8(x_B)-2^(B-2) + 1 is used as an index to SQRTLUT so that sqrt(x_B) can be evaluated by looking up the index sqrt(x_B) = SQRTLUT(i).

5. Use the remainder, r = x - x_B, interpreted as a fraction, to linearly interpolate between SQRTLUT(i) and the next value in the table SQRTLUT(i+1). The remainder, r, is created by extracting the lower w - B bits of x, where w denotes the word-length of x. It is interpreted as a fraction by using function reinterpretcast().

6. Finally, compute the output using the lookup table and linear interpolation:

```
sqrt( u ) = sqrt( x * 2^n )
          = sqrt(x) * 2^(n/2)
          = ( SQRTLUT( i ) + r * ( SQRTLUT( i+1 ) - SQRTLUT( i ) ) ) * 2^(n

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(SQRT
    B = 8; % Number of bits in a byte
    w = u.WordLength;
    for k = 1:prod(size(u))
        assert(u(k)>=0,'Input must be non-negative.');
        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 = int8(storedInteger(bitand(fi(1,1,8,0),fi(n))));
            x = bitsra(x,isodd);
            n = n + isodd;
            % Extract the high byte of x
            high_byte = uint8( storedInteger(bitsliceget(x, w, w - B + 1))
            % Convert the high byte into an index for SQRTLUT
            i =  high_byte - 2^(B-2) + 1;
            % The upper byte was used for the index into SQRTLUT.
            % The remainder, r, interpreted as a fraction, is used to
            % linearly interpolate between points.
            T_unsigned_fraction = numerictype(0, w-B, w-B);
            r = reinterpretcast(bitsliceget(x,w-B,1), T_unsigned_fraction);
            y(k) = bitshift((SQRTLUT(i) + r*(SQRTLUT(i+1) - SQRTLUT(i))),..
                            bitsra(n,1));
        end
    end
    % Remove fimath from the output to insulate the caller from math settin
    % declared inside this function.
    y = removefimath(y);
end
```

Square Root Lookup Table

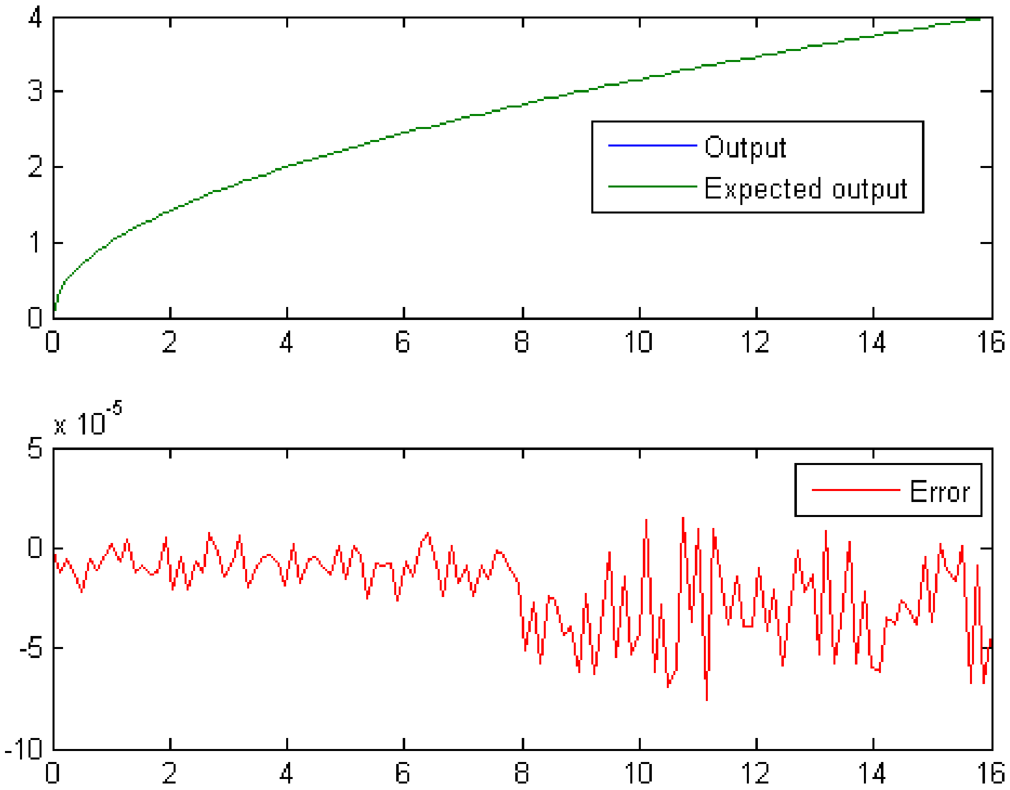
Function `sqrt_lookup_table` loads the lookup table of square-root values. You can create the table by running:

```
sqrt_table = sqrt( (2^(B-2):2^(B))/2^(B-1) );

function SQRTLUT = sqrt_lookup_table()
    B = 8;  % Number of bits in a byte
    % sqrt_table = sqrt( (2^(B-2):2^(B))/2^(B-1) )
    sqrt_table = [0.707106781186548   0.712609640686961   0.718070330817254
                  0.728868986855663   0.734208757779421   0.739509972887452
                  0.750000000000000   0.755190373349661   0.760345316287277
                  0.770551750371122   0.775604602874429   0.780624749799800
                  0.790569415042095   0.795495128834866   0.800390529679106
                  0.810092587300983   0.814900300650331   0.819679815537750
                  0.829156197588850   0.833854004007896   0.838525491562421
                  0.847791247890659   0.852386356061616   0.856956825050130
                  0.866025403784439   0.870524267324007   0.875000000000000
                  0.883883476483184   0.888291900221993   0.892678553567856
                  0.901387818865997   0.905711046636840   0.910013736160065
                  0.918558653543692   0.922801441264588   0.927024810886958
                  0.935414346693485   0.939581023648307   0.943729304408844
                  0.951971638232989   0.956066158798647   0.960143218483576
                  0.968245836551854   0.972271824131503   0.976281209488332
                  0.984250984251476   0.988211768802619   0.992156741649222
                  1.000000000000000   1.003898650263063   1.007782218537319
                  1.015504800579495   1.019344151893756   1.023169096484056
                  1.030776406404415   1.034559084827928   1.038327982864759
                  1.045825033167594   1.049553476484167   1.053268721647045
                  1.060660171779821   1.064336647870400   1.068000468164691
                  1.075290658380328   1.078917281352004   1.082531754730548
                  1.089724735885168   1.093303480283494   1.096870548424015
                  1.103970108290981   1.107502821666834   1.111024302164449
                  1.118033988749895   1.121522402807898   1.125000000000000
                  1.131923142267177   1.135368882786559   1.138804197393037
                  1.145643923738960   1.149048519428140   1.152443057161611
                  1.159202311936963   1.162567202358642   1.165922381636102
                  1.172603939955857   1.175930482639174   1.179247641507075
                  1.185854122563142   1.189143599402528   1.192424001771182
                  1.198957880828180   1.202211503854459   1.205456345124119
                  1.211919964354082   1.215138880951474   1.218349293101120
                  1.224744871391589   1.227930169024281   1.231107225224513
                  1.237436867076458   1.240589577579950   1.243734296383275
                  1.250000000000000   1.253121103485214   1.256234452640111
                  1.262438117295260   1.265528545707287   1.268611445636527
                  1.274754878398196   1.277815518766305   1.280868845744950
                  1.286953767623375   1.289985465034393   1.293010054098575
                  1.299038105676658   1.302041665999979   1.305038313613819
                  1.311011060212689   1.313987252601790   1.316956719106592
                  1.322875655532295   1.325825214724777   1.328768226591831
                  1.334634781503914   1.337558409939543   1.340475661845451
                  1.346291201783626   1.349189571557681   1.352081728298996
                  1.357847561400027   1.360721316067327   1.363589014329464
                  1.369306393762915   1.372156150006259   1.375000000000000
                  1.380670127148408   1.383496476323666   1.386317063301177
                  1.391941090707505   1.394744600276337   1.397542485937369
                  1.403121520040228   1.405902734900249   1.408678458698081
                  1.414213562373095];
    % Cast to fixed point with the most accurate rounding method
    WL = 4*B;  % Word length
    FL = 2*B;  % Fraction length
    SQRTLUT = fi(sqrt_table, 1, WL, FL, 'RoundingMethod','Nearest');
    % Set fimath for the most efficient math operations
    F = fimath('OverflowAction','Wrap',...
               'RoundingMethod','Floor',...
               'SumMode','KeepLSB',...
               'SumWordLength',WL,...
               'ProductMode','KeepLSB',...
               'ProductWordLength',WL);
    SQRTLUT = setfimath(SQRTLUT, F);
end
```

Example

```
u = fi(linspace(0,128,1000),0,16,12);

y = fi_sqrtlookup_8_bit_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','RoundingMeth
>> A = fi(pi);
>> B = fi(2);
>> C = A * setfimath(B,F)

C =

     6.2832

           DataTypeMode: Fixed-point: binary point scaling
             Signedness: Signed
             WordLength: 32
         FractionLength: 26

         RoundingMethod: Floor
         OverflowAction: Wrap
            ProductMode: KeepLSB
      ProductWordLength: 32
                SumMode: FullPrecision
       MaxSumWordLength: 128
```

Note that variable `B` is not changed.

```
>> B

B =

      2

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

Removing FIMATH Conflict in a Loop

You can compute products and sums to match the accumulator of a DSP with floor rounding and wrap overflow, and use nearest rounding and saturate overflow on the output. To avoid mismatched `fimath` errors, you can remove the `fimath` on the output variable when it is used in a computation with the other variables.

MATLAB Code

In this example, the products are 32-bits, and the accumulator is 40-bits, keeping the least-significant-bits with floor rounding and wrap overflow like C's native integer rules. The output uses nearest rounding and saturate overflow.

```
function [y,z] = setfimath_removefimath_in_a_loop(b,a,x,z)
    % Setup
    F_floor = fimath('RoundingMethod','Floor',...
           'OverflowAction','Wrap',...
           'ProductMode','KeepLSB',...
           'ProductWordLength',32,...
           'SumMode','KeepLSB',...
           'SumWordLength',40);
    F_nearest = fimath('RoundingMethod','Nearest',...
        'OverflowAction','Wrap');
    % Set fimaths that are local to this function
    b = setfimath(b,F_floor);
    a = setfimath(a,F_floor);
    x = setfimath(x,F_floor);
    z = setfimath(z,F_floor);
    % Create y with nearest rounding
    y = coder.nullcopy(fi(zeros(size(x)),true,16,14,F_nearest));
    % Algorithm
    for j=1:length(x)
        % Nearest assignment into y
        y(j) =  b(1)*x(j) + z(1);
        % Remove y's fimath conflict with other fimaths
        z(1) = (b(2)*x(j) + z(2)) - a(2) * removefimath(y(j));
        z(2) =  b(3)*x(j)         - a(3) * removefimath(y(j));
    end
    % Cleanup: Remove fimath from outputs
    y = removefimath(y);
    z = removefimath(z);
end
```

Code Generation Instructions

If you have MATLAB Coder software, you can generate C code with the specificed hardware characteristics using the following commands.

```
N = 256;
t = 1:N;
xstep = [ones(N/2,1);-ones(N/2,1)];
num = [0.0299545822080925  0.0599091644161849  0.0299545822080925];
den = [1                  -1.4542435862515900  0.5740619150839550];

b = fi(num,true,16);
a = fi(den,true,16);
x = fi(xstep,true,16,15);
zi = fi(zeros(2,1),true,16,14);

B = coder.Constant(b);
A = coder.Constant(a);

config_obj = coder.config('lib');
config_obj.GenerateReport = true;
config_obj.LaunchReport = true;
config_obj.TargetLang = 'C';
config_obj.GenerateComments = true;
config_obj.GenCodeOnly = true;
config_obj.HardwareImplementation.ProdBitPerChar=8;
config_obj.HardwareImplementation.ProdBitPerShort=16;
config_obj.HardwareImplementation.ProdBitPerInt=32;
config_obj.HardwareImplementation.ProdBitPerLong=40;

codegen -config config_obj setfimath_removefimath_in_a_loop -args {B,A,x,zi
```

Generated C Code

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
void setfimath_removefimath_in_a_loop(const int16_T x[256], int16_T z[2],
  int16_T y[256])
{
  int32_T j;
  int40_T i0;
  int16_T b_y;

  /* Setup */
  /* Set fimaths that are local to this function */
  /* Create y with nearest rounding */
  /* Algorithm */
  for (j = 0; j < 256; j++) {
    /* Nearest assignment into y */
    i0 = 15705 * x[j] + ((int40_T)z[0] << 20);
    b_y = (int16_T)((int32_T)(i0 >> 20) + ((i0 & 524288L) != 0L));

    /* Remove y's fimath conflict with other fimaths */
    z[0] = (int16_T)(((31410 * x[j] + ((int40_T)z[1] << 20)) - ((int40_T)(-
      * b_y) << 6)) >> 20);
    z[1] = (int16_T)((15705 * x[j] - ((int40_T)(9405 * b_y) << 6)) >> 20);
    y[j] = b_y;
  }

  /* Cleanup: Remove fimath from outputs */
}
```

Polymorphic Code

You can write MATLAB code that can be used for both floating-point and fixed-point types using setfimath and removefimath.

```
function y = user_written_function(u)
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB');
    u = setfimath(u,F);
    % Algorithm
    y = u + u;
    % Cleanup
    y = removefimath(y);
end
```

Fixed Point Inputs

When the function is called with fixed-point inputs, then `fimath F` is used for the arithmetic, and the output has no attached `fimath`.

```
>> u = fi(pi/8,true,16,15,'RoundingMethod','Convergent');
>> y = user_written_function(u)

y =

           0.785400390625

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 15
```

Generated C Code for Fixed Point

If you have MATLAB Coder software, you can generate C code using the following commands.

```
>> u = fi(pi/8,true,16,15,'RoundingMethod','Convergent');
>> codegen user_written_function -args {u} -config:lib -launchreport
```

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
int32_T user_written_function(int16_T u)
{
  /* Setup */
  /* Algorithm */
  /* Cleanup */
  return u + u;
}
```

Double Inputs

Since `setfimath` and `removefimath` are pass-through for floating-point types, the `user_written_function` example works with floating-point types, too.

```
function y = user_written_function(u)
```

```
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB');
    u = setfimath(u,F);
    % Algorithm
    y = u + u;
    % Cleanup
    y = removefimath(y);
end
```

Generated C Code for Double

When compiled with floating-point input, you get the following generated C code.

```
>> codegen user_written_function -args {0} -config:lib -launchreport

real_T user_written_function(real_T u)
{
  return u + u;
}
```

Where the `real_T` type is defined as a `double`:

```
typedef double real_T;
```

More Polymorphic Code

This function is written so that the output is created to be the same type as the input, so both floating-point and fixed-point can be used with it.

```
function y = user_written_sum_polymorphic(u)
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB',...
        'SumWordLength',32);

     u = setfimath(u,F);
     if isfi(u)
         y = fi(0,true,32,get(u,'FractionLength'),F);
     else
         y = zeros(1,1,class(u));
     end

     % Algorithm
     for i=1:length(u)
         y(:) = y + u(i);
     end

     % Cleanup
     y = removefimath(y);

end
```

Fixed Point Generated C Code

If you have MATLAB Coder software, you can generate fixed-point C code using the following commands.

```
>> u = fi(1:10,true,16,11);
>> codegen user_written_sum_polymorphic -args {u} -config:lib -launchreport
```

Functions `fimath`, `setfimath` and `removefimath` control the fixed-point math, but the underlying data contained in the variables does not change and so the generated C code does not produce any data copies.

```
int32_T user_written_sum_polymorphic(const int16_T u[10])
{
  int32_T y;
  int32_T i;

  /* Setup */
  y = 0;

  /* Algorithm */
  for (i = 0; i < 10; i++) {
    y += u[i];
  }
  /* Cleanup */
  return y;
}
```

Floating Point Generated C Code

If you have MATLAB Coder software, you can generate floating-point C code using the following commands.

```
>> u = 1:10;
>> codegen user_written_sum_polymorphic -args {u} -config:lib -launchreport
```

```
real_T user_written_sum_polymorphic(const real_T u[10])
{
  real_T y;
  int32_T i;

  /* Setup */
  y = 0.0;

  /* Algorithm */
  for (i = 0; i < 10; i++) {
    y += u[i];
  }

  /* Cleanup */
  return y;
}
```

Where the `real_T` type is defined as a `double`:

```
typedef double real_T;
```

SETFIMATH on Integer Types

Following the established pattern of treating built-in integers like `fi` objects, `setfimath` converts integer input to the equivalent `fi` with attached `fimath`.

```
>> u = int8(5);
>> codegen user_written_u_plus_u -args {u} -config:lib -launchreport
```

```
function y = user_written_u_plus_u(u)
```

```
    % Setup
    F = fimath('RoundingMethod','Floor',...
        'OverflowAction','Wrap',...
        'SumMode','KeepLSB',...
        'SumWordLength',32);
    u = setfimath(u,F);
    % Algorithm
    y = u + u;
    % Cleanup
    y = removefimath(y);
end
```

The output type was specified by the `fimath` to be 32-bit.

```
int32_T user_written_u_plus_u(int8_T u)
{
  /* Setup */
  /* Algorithm */
  /* Cleanup */
  return u + u;
}
```

Working with fimath Objects

- "fimath Object Construction" on page 4-2
- "fimath Object Properties" on page 4-6
- "fimath Properties Usage for Fixed-Point Arithmetic" on page 4-11
- "fimath for Rounding and Overflow Modes" on page 4-20
- "fimath for Sharing Arithmetic Rules" on page 4-22
- "fimath ProductMode and SumMode" on page 4-25

fimath Object Construction

In this section...

"fimath Object Syntaxes" on page 4-2

"Building fimath Object Constructors in a GUI" on page 4-4

fimath Object Syntaxes

The arithmetic attributes of a `fi` object are defined by a local `fimath` object, which is attached to that `fi` object. If a `fi` object has no local `fimath`, the following default fimath values are used:

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

You can create `fimath` objects in Fixed-Point 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 4-7.

Building fimath Object Constructors in a GUI

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

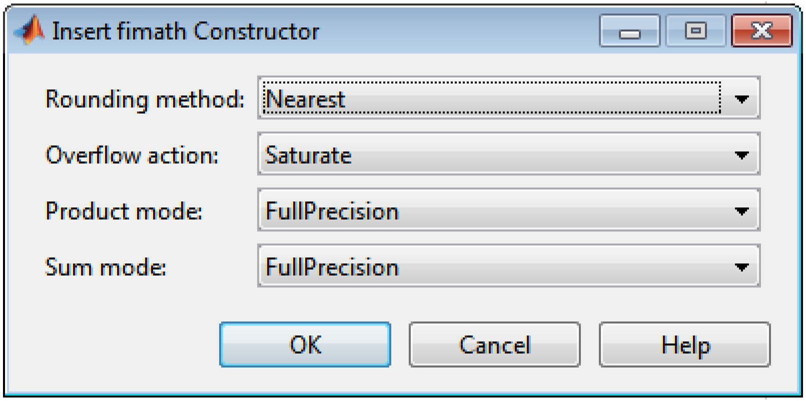
For example, to create a `fimath` object that uses convergent rounding and wraps on overflow, perform the following steps:

1. 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 fi ▾ 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`



Insert fimath Constructor

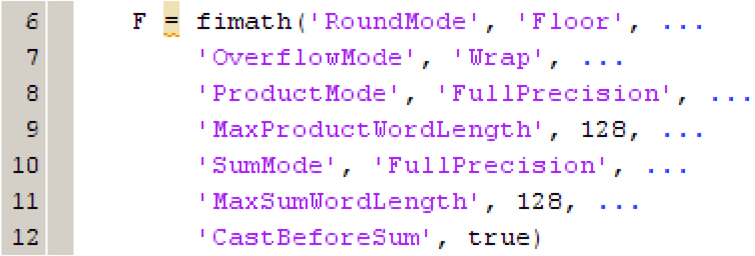
Rounding method: Nearest

Overflow action: Saturate

Product mode: FullPrecision

Sum mode: FullPrecision

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



```
    F = fimath('RoundMode', 'Floor', ...
        'OverflowMode', 'Wrap', ...
        'ProductMode', 'FullPrecision', ...
        'MaxProductWordLength', 128, ...
        'SumMode', 'FullPrecision', ...
        'MaxSumWordLength', 128, ...
        'CastBeforeSum', true)
```

fimath Object Properties

In this section...

"Math, Rounding, and Overflow Properties" on page 4-6

"Setting fimath Object Properties" on page 4-7

Math, Rounding, and Overflow Properties

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

Property	Description
`CastBeforeSum`	Whether both operands are cast to the sum data type before addition
`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

Property	Description
`SumBias`	Bias of the sum data type
`SumFixedExponent`	Fixed exponent of the sum data type
`SumFractionLength`	Fraction length, in bits, of the sum data type
`SumMode`	Defines how the sum data type is determined
`SumSlope`	Slope of the sum data type
`SumSlopeAdjustmentFactor`	Slope adjustment factor of the sum data type
`SumWordLength`	Word length, in bits, of the sum data type

For details about these properties, refer to the “fi Object Properties” on page 2-17. To learn how to specify properties for `fimath` objects in Fixed-Point Designer software, refer to “Setting fimath Object Properties” on page 4-7.

Setting fimath Object Properties

- “Setting fimath Properties at Object Creation” on page 4-7
- “Using Direct Property Referencing with fimath” on page 4-8
- “Setting fimath Properties in the Model Explorer” on page 4-8

Setting fimath Properties at Object Creation

You can set properties of `fimath` objects at the time of object creation by including properties after the arguments of the `fimath` constructor function.

For example, to set the overflow action to `Saturate` and the rounding method to `Convergent`,

```
F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent')

F =
        RoundingMethod: Convergent
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Using Direct Property Referencing with fimath

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

For example, to get the `RoundingMethod` of `F`,

```
F.RoundingMethod

ans =

Convergent
```

To set the `OverflowAction` of `F`,

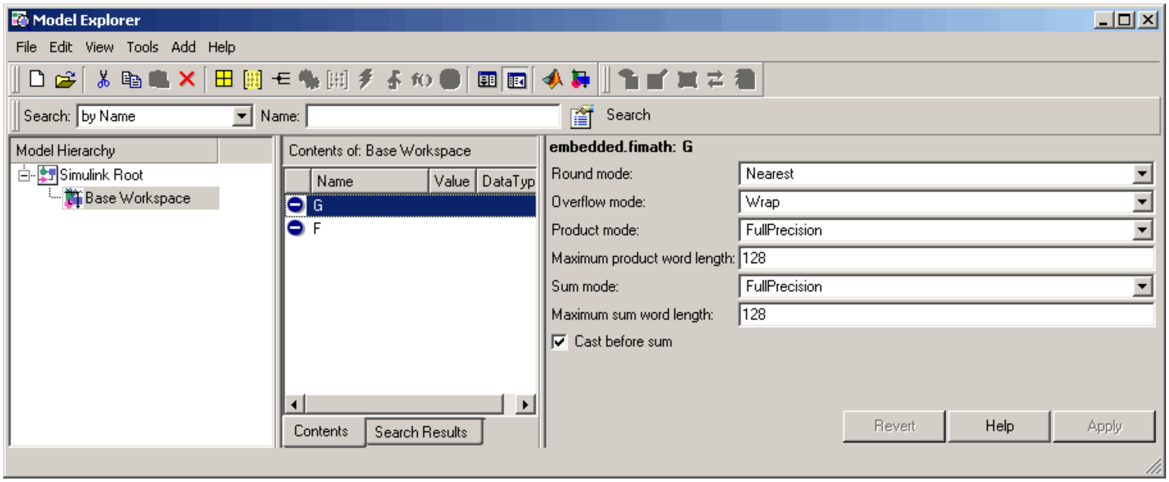
```
F.OverflowAction = 'Wrap'

F =


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

Setting fimath Properties in the Model Explorer

You can view and change the properties for any `fimath` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View** > **Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The following figure shows the Model Explorer when you define the following `fimath` objects in the MATLAB workspace:

```
F = fimath

F =


         RoundingMethod: Nearest
         OverflowAction: Saturate
            ProductMode: FullPrecision
                SumMode: FullPrecision
G = fimath('OverflowAction','Wrap')

G =

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



embedded.fimath: G

Round mode: Nearest
Overflow mode: Wrap
Product mode: FullPrecision
Maximum product word length: 128
Sum mode: FullPrecision
Maximum sum word length: 128
☑ Cast before sum

Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a `fimath` object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

For more information on working with the Model Explorer, see the following sections of the Fixed-Point Designer documentation:

- "Specifying Fixed-Point Parameters in the Model Explorer" on page 9-79
- "Sharing Models with Fixed-Point MATLAB Function Blocks" on page 9-82

fimath Properties Usage for Fixed-Point Arithmetic

In this section...

"fimath Rules for Fixed-Point Arithmetic" on page 4-11

"Binary-Point Arithmetic" on page 4-13

"[Slope Bias] Arithmetic" on page 4-17

fimath Rules for Fixed-Point Arithmetic

`fimath` properties define the rules for performing arithmetic operations on `fi` objects. The `fimath` properties that govern fixed-point arithmetic operations can come from a local `fimath` object or the `fimath` default values.

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

The following sections discuss how `fi` objects with local `fimath` objects interact with `fi` objects without local fimath.

Binary Operations

In binary fixed-point operations such as `c = a + b`, the following rules apply:

- If both `a` and `b` have no local fimath, the operation uses default fimath values to perform the fixed-point arithmetic. The output `fi` object `c` also has no local fimath.
- If either `a` or `b` has a local `fimath` object, the operation uses that `fimath` object to perform the fixed-point arithmetic. The output `fi` object `c` has the same local `fimath` object as the input.

Unary Operations

In unary fixed-point operations such as `b = abs(a)`, the following rules apply:

- If `a` has no local fimath, the operation uses default fimath values to perform the fixed-point arithmetic. The output `fi` object `b` has no local fimath.

- If a has a local fimath object, the operation uses that fimath object to perform the fixed-point arithmetic. The output fi object b has the same local fimath object as the input a.

When you specify a fimath object in the function call of a unary fixed-point operation, the operation uses the fimath object you specify to perform the fixed-point arithmetic. For example, when you use a syntax such as b = abs(a,F) or b = sqrt(a,F), the abs and sqrt operations use the fimath object F to compute intermediate quantities. The output fi object b always has no local fimath.

Concatenation Operations

In fixed-point concatenation operations such as c = [a b], c = [a;b] and c = bitconcat(a,b), the following rule applies:

- The fimath properties of the left-most fi object in the operation determine the fimath properties of the output fi object c.

For example, consider the following scenarios for the operation d = [a b c]:

- If a is a fi object with no local fimath, the output fi object d also has no local fimath.
- If a has a local fimath object, the output fi object d has the same local fimath object.
- If a is not a fi object, the output fi object d inherits the fimath properties of the next left-most fi object. For example, if b is a fi object with a local fimath object, the output fi object d has the same local fimath object as the input fi object b.

fimath Object Operations: add, mpy, sub

The output of the fimath object operations add, mpy, and sub always have no local fimath. The operations use the fimath object you specify in the function call, but the output fi object never has a local fimath object.

MATLAB Function Block Operations

Fixed-point operations performed with the MATLAB Function block use the same rules as fixed-point operations performed in MATLAB.

All input signals to the MATLAB Function block that you treat as `fi` objects associate with whatever you specify for the **MATLAB Function block fimath** parameter. When you set this parameter to `Same as MATLAB`, your `fi` objects do not have local fimath. When you set the **MATLAB Function block fimath** parameter to `Specify other`, you can define your own set of `fimath` properties for all `fi` objects in the MATLAB Function block to associate with. You can choose to treat only fixed-point input signals as `fi` objects or both fixed-point and integer input signals as `fi` objects. See "Using fimath Objects in MATLAB Function Blocks" on page 9-80.

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 4-11 for more information.

```
a = fi(pi);
b = fi(8);
isequal(a.fimath, b.fimath)

ans =

     1

a + b

ans =

   11.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 19
        FractionLength: 13
```

To perform arithmetic with +, -, .*, or *, two fi operands must also have the same data type. For example, you can perform addition on two fi objects with data type double, but not on an object with data type double and one with data type single:

```
a = fi(3, 'DataType', 'double')

a =

     3
```

```
          DataTypeMode: Double

b = fi(27, 'DataType', 'double')

b =

    27

          DataTypeMode: Double

a + b

ans =

    30

          DataTypeMode: Double

c = fi(12, 'DataType', 'single')

c =

    12

          DataTypeMode: Single

a + c
??? Math operations are not allowed on FI objects with
  different data types.
```

Fixed-point `fi` object operands do not have to have the same scaling. You can perform binary math operations on a `fi` object with a fixed-point data type and a `fi` object with a scaled doubles data type. In this sense, the scaled double data type acts as a fixed-point data type:

```
a = fi(pi)

a =

    3.1416
```

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

b = fi(magic(2), ...
'DataTypeMode', 'Scaled double: binary point scaling')

b =

     1     3
     4     2


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12

a + b

ans =

    4.1416    6.1416
    7.1416    5.1416


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 13
```

Use the `divide` function to perform division with doubles, singles, or binary point-only scaling `fi` objects.

[Slope Bias] Arithmetic

Fixed-Point 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.

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 4-22
- “Local fimath Usage to Share Arithmetic Rules” on page 4-22

Sharing fimath properties across fi objects ensures that the fi objects are using the same arithmetic rules and helps you avoid “mismatched fimath” errors.

Default fimath Usage to Share Arithmetic Rules

You can ensure that your fi objects are all using the same fimath properties by not specifying any local fimath. To assure no local fimath is associated with a fi object, you can:

- Create a fi object using the fi constructor without specifying any fimath properties in the constructor call. For example:

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

- Create a fi object using the sfi or ufi constructor. All fi objects created with these constructors have no local fimath.

  ```
  b = sfi(pi)
  ```

- Use removefimath to remove a local fimath object from an existing fi object.

Local fimath Usage to Share Arithmetic Rules

You can also use a fimath object to define common arithmetic rules that you would like to use for multiple fi objects. You can then create your fi objects, using the same fimath object for each. To do so, you must also create a numerictype object to define a common data type and scaling. Refer to “numerictype Object Construction” on page 6-2 for more information on numerictype objects. The following example shows the creation of a 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 4-25

"FullPrecision" on page 4-26

"KeepLSB" on page 4-27

"KeepMSB" on page 4-28

"SpecifyPrecision" on page 4-30

Example Setup

The examples in the sections of this topic show the differences among the four settings of the `ProductMode` and `SumMode` properties:

- `FullPrecision`
- `KeepLSB`
- `KeepMSB`
- `SpecifyPrecision`

To follow along, first set the following preferences:

```
p = fipref;
p.NumericTypeDisplay = 'short';
p.FimathDisplay = 'none';
p.LoggingMode = 'on';
F = fimath('OverflowAction','Wrap',...
      'RoundingMethod','Floor',...
      'CastBeforeSum',false);
warning off
format compact
```

Next, define `fi` objects `a` and `b`. Both have signed 8-bit data types. The fraction length gets chosen automatically for each `fi` object to yield the best possible precision:

```
a = fi(pi, true, 8)
```

```
a =
    3.1563
      s8,5

b = fi(exp(1), true, 8)
b =
    2.7188
      s8,5
```

FullPrecision

Now, set `ProductMode` and `SumMode` for `a` and `b` to `FullPrecision` and look at some results:

```
F.ProductMode = 'FullPrecision';
F.SumMode = 'FullPrecision';
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5

b
b =
    2.7188   %010.10111
      s8,5

a*b
ans =
    8.5811   %001000.1001010011
      s16,10

a+b
ans =
    5.8750   %0101.11100
      s9,5
```

In `FullPrecision` mode, the product word length grows to the sum of the word lengths of the operands. In this case, each operand has 8 bits, so the product word length is 16 bits. The product fraction length is the sum of the fraction lengths of the operands, in this case 5 + 5 = 10 bits.

The sum word length grows by one most significant bit to accommodate the possibility of a carry bit. The sum fraction length aligns with the fraction lengths of the operands, and all fractional bits are kept for full precision. In this case, both operands have 5 fractional bits, so the sum has 5 fractional bits.

KeepLSB

Now, set `ProductMode` and `SumMode` for `a` and `b` to `KeepLSB` and look at some results:

```
F.ProductMode = 'KeepLSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepLSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5

b
b =
    2.7188   %010.10111
      s8,5

a*b
ans =
    0.5811   %00.1001010011
      s12,10

a+b
ans =
    5.8750   %0000101.11100
      s12,5
```

In KeepLSB mode, you specify the word lengths and the least significant bits of results are automatically kept. This mode models the behavior of integer operations in the C language.

The product fraction length is the sum of the fraction lengths of the operands. In this case, each operand has 5 fractional bits, so the product fraction length is 10 bits. In this mode, all 10 fractional bits are kept. Overflow occurs because the full-precision result requires 6 integer bits, and only 2 integer bits remain in the product.

The sum fraction length aligns with the fraction lengths of the operands, and in this model all least significant bits are kept. In this case, both operands had 5 fractional bits, so the sum has 5 fractional bits. The full-precision result requires 4 integer bits, and 7 integer bits remain in the sum, so no overflow occurs in the sum.

KeepMSB

Now, set ProductMode and SumMode for a and b to KeepMSB and look at some results:

```
F.ProductMode = 'KeepMSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepMSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5

b
b =
    2.7188   %010.10111
      s8,5

a*b
ans =
    8.5781   %001000.100101
```

```
      s12,6

a+b
ans =
    5.8750   %0101.11100000
      s12,8
```

In KeepMSB mode, you specify the word lengths and the most significant bits of sum and product results are automatically kept. This mode models the behavior of many DSP devices where the product and sum are kept in double-wide registers, and the programmer chooses to transfer the most significant bits from the registers to memory after each operation.

The full-precision product requires 6 integer bits, and the fraction length of the product is adjusted to accommodate all 6 integer bits in this mode. No overflow occurs. However, the full-precision product requires 10 fractional bits, and only 6 are available. Therefore, precision is lost.

The full-precision sum requires 4 integer bits, and the fraction length of the sum is adjusted to accommodate all 4 integer bits in this mode. The full-precision sum requires only 5 fractional bits; in this case there are 8, so there is no loss of precision.

This example shows that, in KeepMSB mode the fraction length changes regardless of whether or not an overflow occurs. The fraction length is set to the amount needed to represent the product in case both terms use the maximum possible value (18+18-16=20 in this example).

```
F = fimath('SumMode','KeepMSB','ProductMode','KeepMSB',...
    'ProductWordLength',16,'SumWordLength',16);
a=fi(100,1,16,-2,'fimath',F);
a*a

ans =

     0

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

```
        FractionLength: -20

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: KeepMSB
     ProductWordLength: 16
               SumMode: KeepMSB
         SumWordLength: 16
         CastBeforeSum: true
```

SpecifyPrecision

Now set ProductMode and SumMode for a and b to SpecifyPrecision and look at some results:

```
F.ProductMode = 'SpecifyPrecision';
F.ProductWordLength = 8;
F.ProductFractionLength = 7;
F.SumMode = 'SpecifyPrecision';
F.SumWordLength = 8;
F.SumFractionLength = 7;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5

b
b =
    2.7188   %010.10111
      s8,5

a*b
ans =
    0.5781   %0.1001010
      s8,7

a+b
ans =
   -0.1250   %1.1110000
      s8,7
```

In SpecifyPrecision mode, you must specify both word length and fraction length for sums and products. This example unwisely uses fractional formats for the products and sums, with 8-bit word lengths and 7-bit fraction lengths.

The full-precision product requires 6 integer bits, and the example specifies only 1, so the product overflows. The full-precision product requires 10 fractional bits, and the example only specifies 7, so there is precision loss in the product.

The full-precision sum requires 4 integer bits, and the example specifies only 1, so the sum overflows. The full-precision sum requires 5 fractional bits, and the example specifies 7, so there is no loss of precision in the sum.

5

Working with fipref Objects

- "fipref Object Construction" on page 5-2
- "fipref Object Properties" on page 5-3
- "fi Object Display Preferences Using fipref" on page 5-5
- "Underflow and Overflow Logging Using fipref" on page 5-7
- "Data Type Override Preferences Using fipref" on page 5-12

fipref Object Construction

The `fipref` object defines the display and logging attributes for all `fi` objects. You can use the `fipref` constructor function to create a new object.

To get started, type

```
P = fipref
```

to create a default `fipref` object.

```
P =
         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'
```

The syntax

```
P = fipref(...'PropertyName','PropertyValue'...)
```

allows you to set properties for a `fipref` object at object creation with property name/property value pairs.

Your `fipref` settings persist throughout your MATLAB session. Use `reset(fipref)` to return to the default settings during your session. Use `savefipref` to save your display preferences for subsequent MATLAB sessions.

fipref Object Properties

In this section...

"Display, Data Type Override, and Logging Properties" on page 5-3

"fipref Object Properties Setting" on page 5-3

Display, Data Type Override, and Logging Properties

The following properties of `fipref` objects are always writable:

- `FimathDisplay` — Display options for the local `fimath` attributes of a `fi` object
- `DataTypeOverride` — Data type override options
- `LoggingMode` — Logging options for operations performed on `fi` objects
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object
- `NumberDisplay` — Display options for the value of a `fi` object

These properties are described in detail in the "fi Object Properties" on page 2-17. To learn how to specify properties for `fipref` objects in Fixed-Point 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 6-7
- “numerictype Structure of Fixed-Point Objects” on page 6-11
- “numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects” on page 6-14

numerictype Object Construction

In this section...

"numerictype Object Syntaxes" on page 6-2

"Example: Construct a numerictype Object with Property Name and Property Value Pairs" on page 6-3

"Example: Copy a numerictype Object" on page 6-4

"Example: Build numerictype Object Constructors in a GUI" on page 6-5

numerictype Object Syntaxes

`numerictype` objects define the data type and scaling attributes of `fi` objects, as well as Simulink signals and model parameters. You can create `numerictype` objects in Fixed-Point 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 string at a specific location in your file.

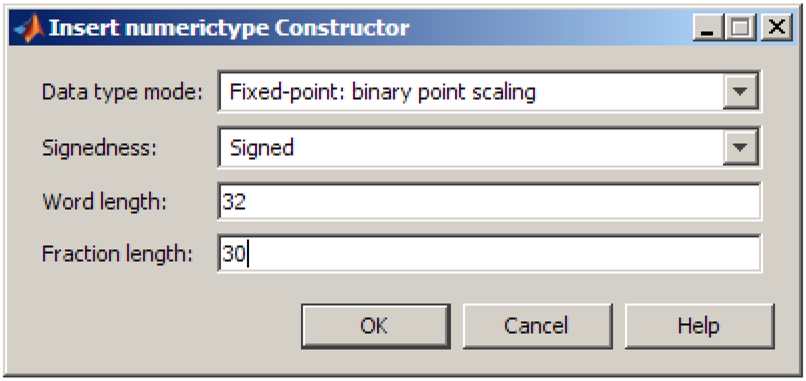
For example, to create a signed `numerictype` object with binary-point scaling, a word length of 32 bits and a fraction length of 30 bits, perform the following steps:

1. 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 fi ▼ 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`



Insert numerictype Constructor

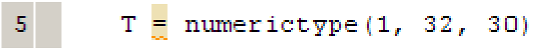
Data type mode: Fixed-point: binary point scaling

Signedness: Signed

Word length: 32

Fraction length: 30

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



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

numerictype Object Properties

In this section...
"Data Type and Scaling Properties" on page 6-7
"Set numerictype Object Properties" on page 6-8

Data Type and Scaling Properties

All properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object become read only after the `fi` object has been created. Any `numerictype` properties of a `fi` object that are unspecified at the time of `fi` object creation are automatically set to their default values. The properties of a `numerictype` object are:

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Signedness` — Signed, unsigned, or auto
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

These properties are described in detail in the "fi Object Properties" on page 2-17. To learn how to specify properties for `numerictype` objects in Fixed-Point Designer software, refer to "Set numerictype Object Properties" on page 6-8.

Set numerictype Object Properties

Setting numerictype Properties at Object Creation

You can set properties of `numerictype` objects at the time of object creation by including properties after the arguments of the `numerictype` constructor function.

For example, to set the word length to 32 bits and the fraction length to 30 bits,

```
T = numerictype('WordLength', 32, 'FractionLength', 30)

T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Use Direct Property Referencing with numerictype Objects

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

For example, to get the word length of `T`,

```
T.WordLength

ans =

32
```

To set the fraction length of `T`,

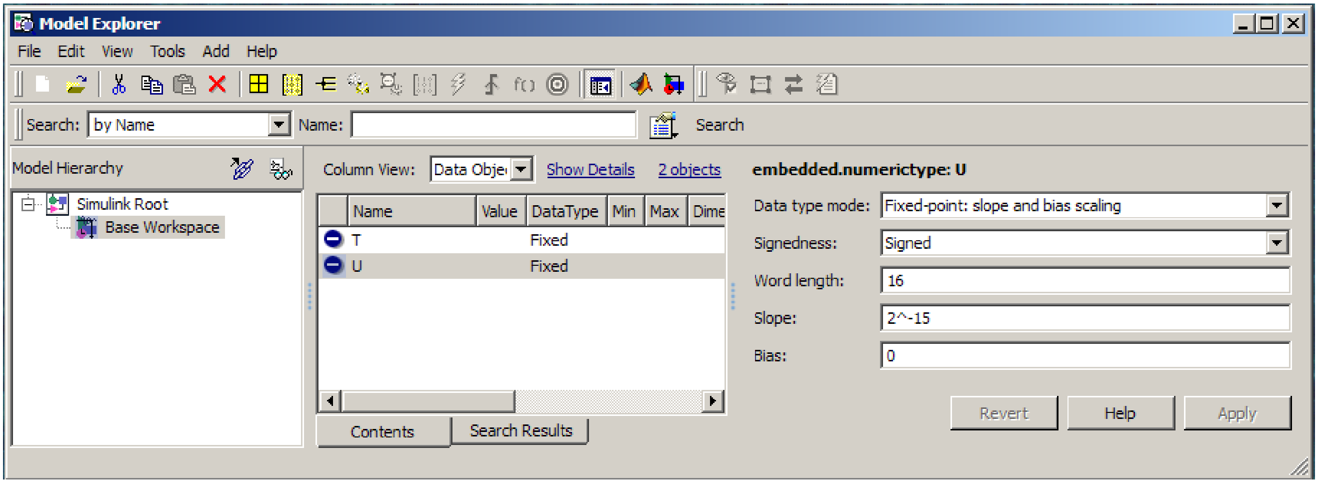
```
T.FractionLength = 31

T =
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 31
```

Set numerictype Properties in the Model Explorer

You can view and change the properties for any `numerictype` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View** > **Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The figure below shows the Model Explorer when you define the following `numerictype` objects in the MATLAB workspace:

```
T = numerictype

T =


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

U = numerictype('DataTypeMode', 'Fixed-point: slope and bias')

U =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^-15
                  Bias: 0
```



Search: by Name

Name:

Column View: Data Obje

Name	Value	DataType	Min	Max	Dime
T		Fixed			
U		Fixed			

embedded.numerictype: U

Data type mode: Fixed-point: slope and bias scaling

Signedness: Signed

Word length: 16

Slope: 2^-15

Bias: 0

Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a `numerictype` object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

numerictype Structure of Fixed-Point Objects

In this section...

"Valid Values for numerictype Structure Properties" on page 6-11

"Properties That Affect the Slope" on page 6-13

"Stored Integer Value and Real World Value" on page 6-13

Valid Values for numerictype Structure Properties

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

Note When you create a `fi` object, any unspecified field of the `numerictype` object reverts to its default value. Thus, if the `DataTypeMode` is set to `unspecified scaling`, it defaults to `binary point scaling` when the `fi` object is created. If the `Signedness` property of the `numerictype` object is set to `Auto`, it defaults to `Signed` when the `fi` object is created.

DataTypeMode	DataType	Scaling	Signedness	Word-Length	Fraction-Length	Slope	Bias
Fixed-point data types							

`Fixed-point: binary point scaling`	`Fixed`	`BinaryPoint`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	Positive or negative integer	2^(-fraction length)	0

`Fixed-point: slope and bias scaling`	`Fixed`	`SlopeBias`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	`N/A`	Any floating-point number	Any floating-point number

DataTypeMode	DataType	Scaling	Signedness	Word-Length	Fraction-Length	Slope	Bias

`Fixed-point: unspecified scaling`	`Fixed`	`Unspecified`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`

Scaled double data types							

`Scaled double: binary point scaling`	`ScaledDouble`	`BinaryPoint`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	Positive or negative integer	2^(-fraction length)	0

`Scaled double: slope and bias scaling`	`ScaledDouble`	`SlopeBias`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	`N/A`	Any floating-point number	Any floating-point number

`Scaled double: unspecified scaling`	`ScaledDouble`	`Unspecified`	`Signed`
			`Unsigned`
			`Auto`
				Positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`

Built-in data types							

`Double`	`double`	`N/A`	`1`
			`true`
				`64`	`0`	`1`	`0`

`Single`	`single`	`N/A`	`1`
			`true`
				`32`	`0`	`1`	`0`

`Boolean`	`boolean`	`N/A`	`0`
			`false`
				`1`	`0`	`1`	`0`

You cannot change the `numerictype` properties of a `fi` object after `fi` object creation.

Properties That Affect the Slope

The **Slope** field of the `numerictype` structure is related to the `SlopeAdjustmentFactor` and `FixedExponent` properties by

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

The `FixedExponent` and `FractionLength` properties are related by

$$fixed\ exponent = -fraction\ length$$

If you set the `SlopeAdjustmentFactor`, `FixedExponent`, or `FractionLength` property, the **Slope** field is modified.

Stored Integer Value and Real World Value

The `numerictype` `StoredIntegerValue` and `RealWorldValue` properties are related according to

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

which is equivalent to

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

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

numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects

You can use a numerictype object to define common data type and scaling rules that you would like to use for many fi objects. You can then create multiple fi objects, using the same numerictype object for each.

Example 1

In the following example, you create a numerictype object T with word length 32 and fraction length 28. Next, to ensure that your fi objects have the same numerictype attributes, create fi objects a and b using your numerictype object T.

```
format long g
T = numerictype('WordLength',32,'FractionLength',28)

T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28

a = fi(pi,T)

a =

           3.1415926553309


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28

b = fi(pi/2, T)
```

```
b =

           1.5707963258028


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28
```

Example 2

In this example, start by creating a numerictype object T with [Slope Bias] scaling. Next, use that object to create two fi objects, c and d with the same numerictype attributes:

```
T = numerictype('Scaling','slopebias','Slope', 2^2, 'Bias', 0)

T =

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

c = fi(pi, T)

c =

     4

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

d = fi(pi/2, T)
```

```
d =
 
 
     0
 
          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^2
                  Bias: 0
```

Working with quantizer Objects

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

Constructing quantizer Objects

You can use quantizer objects to quantize data sets. You can create quantizer objects in Fixed-Point 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 2-17 for more details about these properties, including their possible values.

For example, to create a fixed-point `quantizer` object with

- The `Format` property value set to `[16,14]`
- The `OverflowAction` property value set to `'Saturate'`
- The `RoundingMethod` property value set to `'Ceiling'`

type

```
q = quantizer('datamode','fixed','format',[16,14],...
    'OverflowMode','saturate','RoundMode','ceil')
```

You do not have to include `quantizer` object property names when you set `quantizer` object property values.

For example, you can create `quantizer` object `q` from the previous example by typing

```
q = quantizer('fixed',[16,14],'saturate','ceil')
```

Note You do not have to include default property values when you construct a `quantizer` object. In this example, you could leave out `'fixed'` and `'saturate'`.

Quantizing Data with quantizer Objects

You construct a `quantizer` object to specify the quantization parameters to use when you quantize data sets. You can use the `quantize` function to quantize data according to a `quantizer` object's specifications.

Once you quantize data with a `quantizer` object, its state values might change.

The following example shows

- How you use `quantize` to quantize data
- How quantization affects `quantizer` object states
- How you reset `quantizer` object states to their default values using `reset`

1 Construct an example data set and a `quantizer` object.

```
format long g
rng('default');
x = randn(100,4);
q = quantizer([16,14]);
```

2 Retrieve the values of the `maxlog` and `noverflows` states.

```
q.maxlog

ans =

    -1.79769313486232e+308

q.noverflows

ans =

     0
```

   Note that `maxlog` is equal to `-realmax`, which indicates that the quantizer `q` is in a reset state.

3 Quantize the data set according to the `quantizer` object's specifications.

```
y = quantize(q,x);
Warning: 626 overflow(s) occurred in the fi quantize operation.
```

4 Check the values of `maxlog` and `noverflows`.

```
q.maxlog

ans =

          1.99993896484375

q.noverflows

ans =

    626
```

Note that the maximum logged value was taken after quantization, that is, `q.maxlog == max(y)`.

5 Reset the `quantizer` states and check them.

```
reset(q)
q.maxlog

ans =

    -1.79769313486232e+308

q.noverflows

ans =

     0
```

Transformations for Quantized Data

You can convert data values from numeric to hexadecimal or binary according to a `quantizer` object's specifications.

Use

- `num2bin` to convert data to binary
- `num2hex` to convert data to hexadecimal
- `hex2num` to convert hexadecimal data to numeric
- `bin2num` to convert binary data to numeric

For example,

```
q = quantizer([3 2]);
x = [0.75   -0.25
     0.50   -0.50
     0.25   -0.75
       0      -1 ];
b = num2bin(q,x)

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

produces all two's complement fractional representations of 3-bit fixed-point numbers.

Fixed-Point Conversion

- “Fixed-Point Conversion Workflows” on page 8-2
- “Code Coverage” on page 8-3
- “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros” on page 8-7
- “Propose Data Types Based on Simulation Ranges” on page 8-37
- “Propose Data Types Based on Derived Ranges” on page 8-55
- “Command-Line Fixed-Point Conversion Workflow” on page 8-70
- “Command-Line Fixed-Point Conversion Best Practices” on page 8-72

Fixed-Point Conversion Workflows

In this section...

"Command-Line Workflow" on page 8-2

"Automated Project Workflow" on page 8-2

You can convert floating-point MATLAB code to fixed point manually or, if you have a MATLAB Coder™ license, you can use the project fixed-point conversion tool to convert from floating-point MATLAB code to fixed-point C code.

Command-Line 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 integrated into other projects. For more information, see "Command-Line Fixed-Point Conversion Workflow" on page 8-70 and "Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros" on page 8-7.

Automated Project Workflow

If you are new to fixed-point modeling and you are looking for a direct path from floating-point MATLAB to integer-based C or HDL code, use the automated project workflow (requires MATLAB Coder license). Using this automated workflow, the tool provides data type proposals based on simulation ranges, static ranges, or both. For more information, see "Propose Data Types Based on Simulation Ranges" on page 8-37 and "Propose Data Types Based on Derived Ranges" on page 8-55.

Code Coverage

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files should 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 verify that your test file is exercising the algorithm adequately. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. If you simulate multiple test files in one run, the tool displays cumulative coverage. However, if you specify multiple test files but run them one at a time, the tool displays the coverage of the file that ran last.

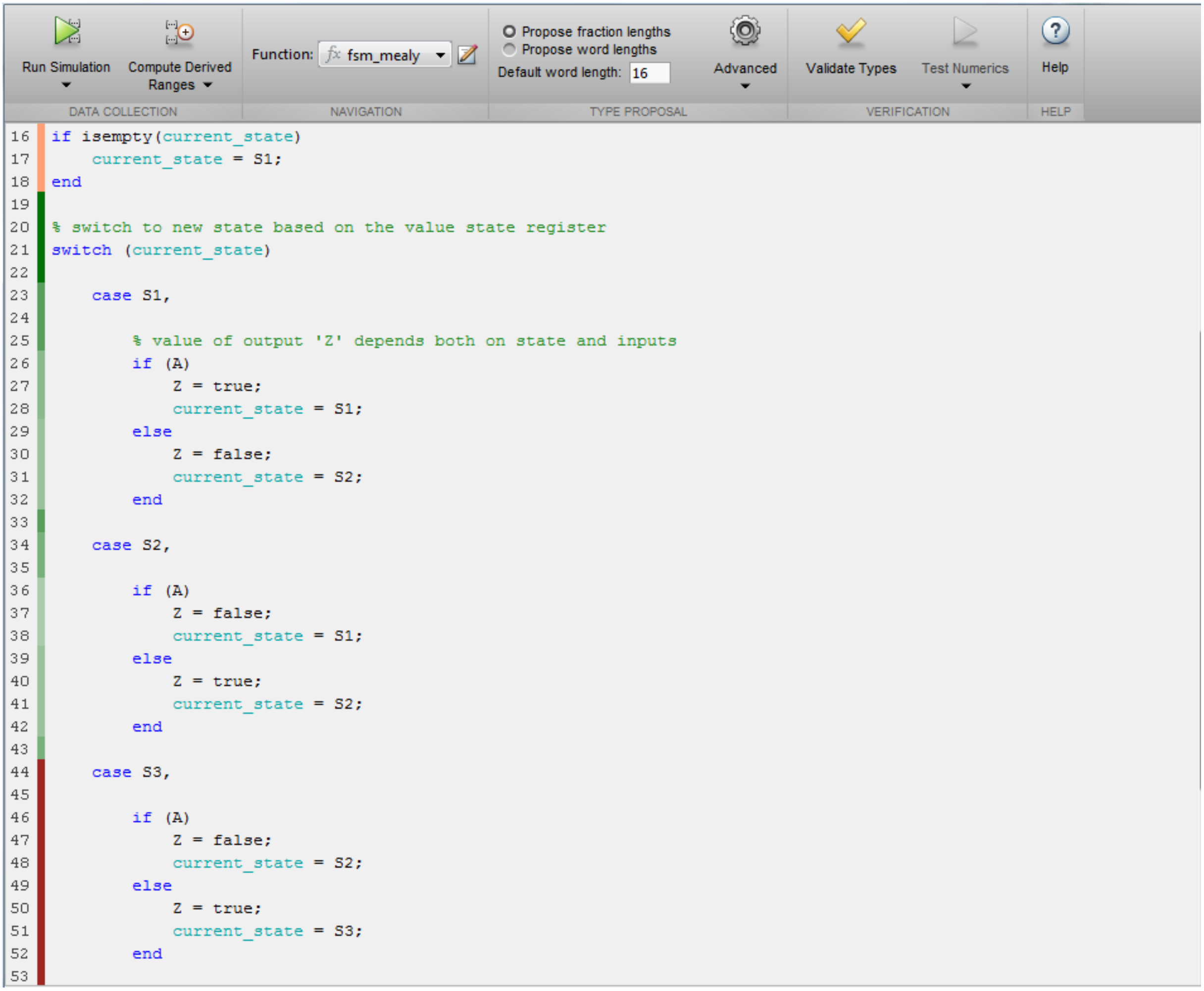
Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage might speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

1 Click **Run Simulation**.

2 Clear `Show code coverage`.

The tool covers basic MATLAB control constructs and shows statement coverage for basic blocks of code. The tool displays a color-coded coverage bar to the left of the code.

Coverage Bar Color	How Often Code is Executed During Test File Simulation
Dark green	Always
Light green	Sometimes
Orange	Once
Red	Never

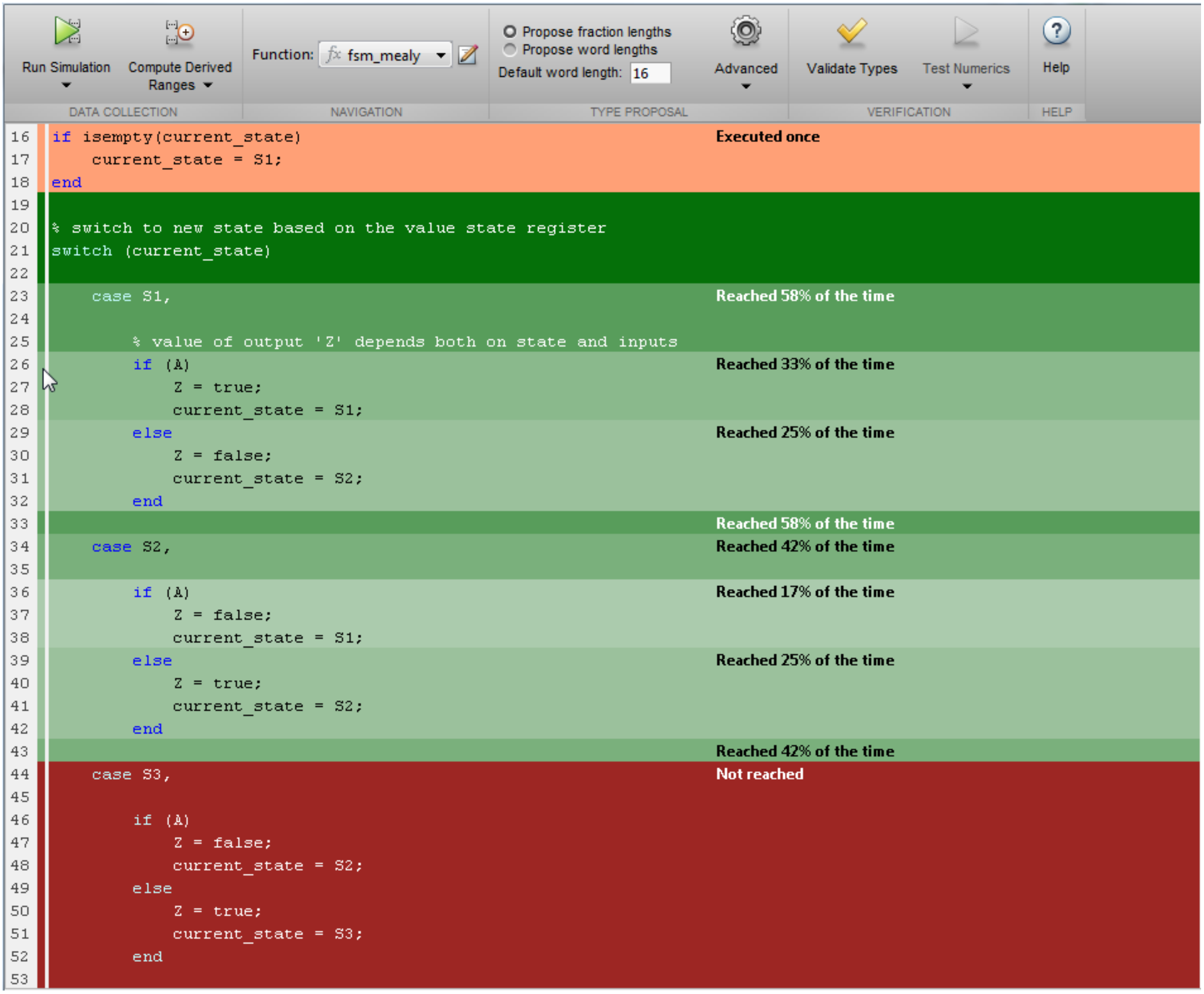


Function: fsm_mealy

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

```
if isempty(current_state)
    current_state = S1;
end

% switch to new state based on the value state register
switch (current_state)

    case S1,

        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S2;
        end

    case S2,

        if (A)
            Z = false;
            current_state = S1;
        else
            Z = true;
            current_state = S2;
        end

    case S3,

        if (A)
            Z = false;
            current_state = S2;
        else
            Z = true;
            current_state = S3;
        end

```

When you position your cursor over the coverage bar, the color highlighting extends over the code and the tool displays more information about how often the code is executed. For MATLAB constructs that affect control flow (if-elseif-else, switch-case, for-continue-break, return), it displays statement coverage as a percentage coverage for basic blocks inside these constructs.



```
if isempty(current_state)                                         Executed once
    current_state = S1;
end

% switch to new state based on the value state register
switch (current_state)

    case S1,                                                      Reached 58% of the time

        % value of output 'Z' depends both on state and inputs
        if (A)                                                    Reached 33% of the time
            Z = true;
            current_state = S1;
        else                                                      Reached 25% of the time
            Z = false;
            current_state = S2;
        end
                                                                  Reached 58% of the time
    case S2,                                                      Reached 42% of the time

        if (A)                                                    Reached 17% of the time
            Z = false;
            current_state = S1;
        else                                                      Reached 25% of the time
            Z = true;
            current_state = S2;
        end
                                                                  Reached 42% of the time
    case S3,                                                      Not reached

        if (A)
            Z = false;
            current_state = S2;
        else
            Z = true;
            current_state = S3;
        end

```

To verify that your test file is testing your algorithm over the intended operating range, review the code coverage results and take action as described in the following table.

Coverage Bar Color	Action Required
Dark green	None
Light green	Review percentage coverage and verify that it is reasonable based on your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.
Orange	This is expected behavior for initialization code, for example, the initialization of persistent variables. For other cases, verify that this behavior is reasonable for your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.
Red	If the code that is not executed is an error condition, this is acceptable behavior. If the code should be executed, modify the test file or add another test file to extend coverage. If the code is written conservatively and has upper and lower boundary limits and you cannot modify the test file to reach this code, add static minimum and maximum values (see "Computing Derived Ranges").

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) + ... + b(end)*x(n-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=mod(n-1,length(b))+1, for n=1, 2, 3, ...

For example, let length(b) = 3, and initialize p and z to:

```
p = 0, z = [ 0    0    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)  0    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) x(2)  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) x(2) 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) x(2) 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) x(5) 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) x(5) 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_ex

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 norm(b,1) = sum(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];
```

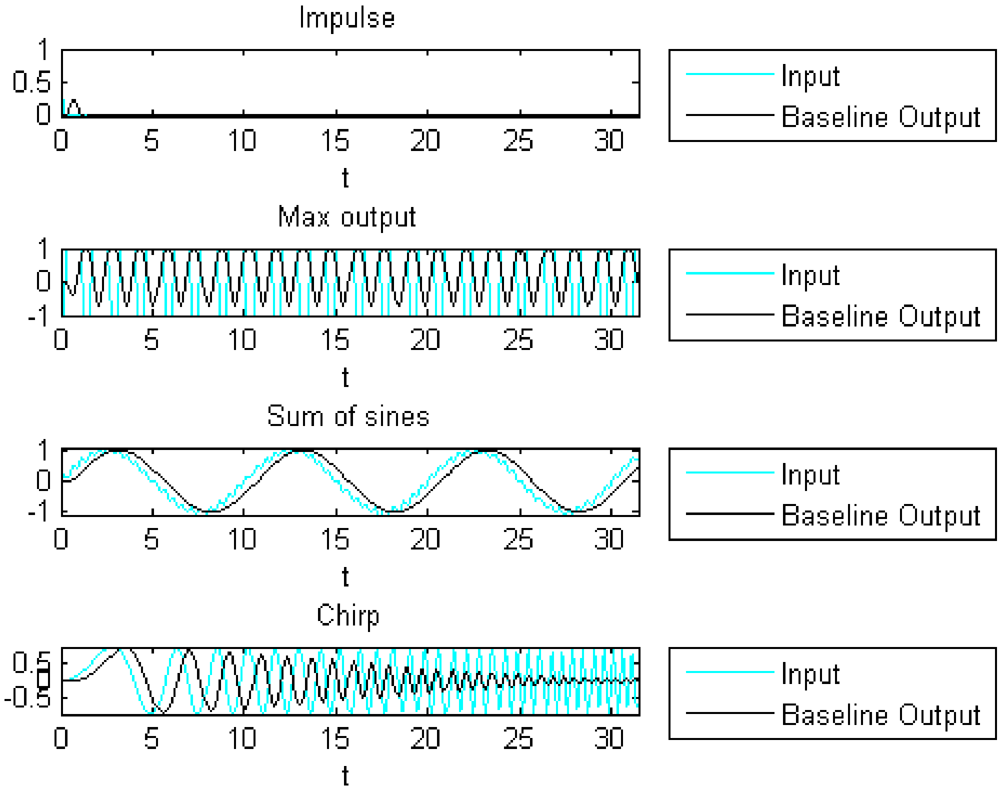
Call the original function

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 streami
    % 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),rese
     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(:,
```

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);
```

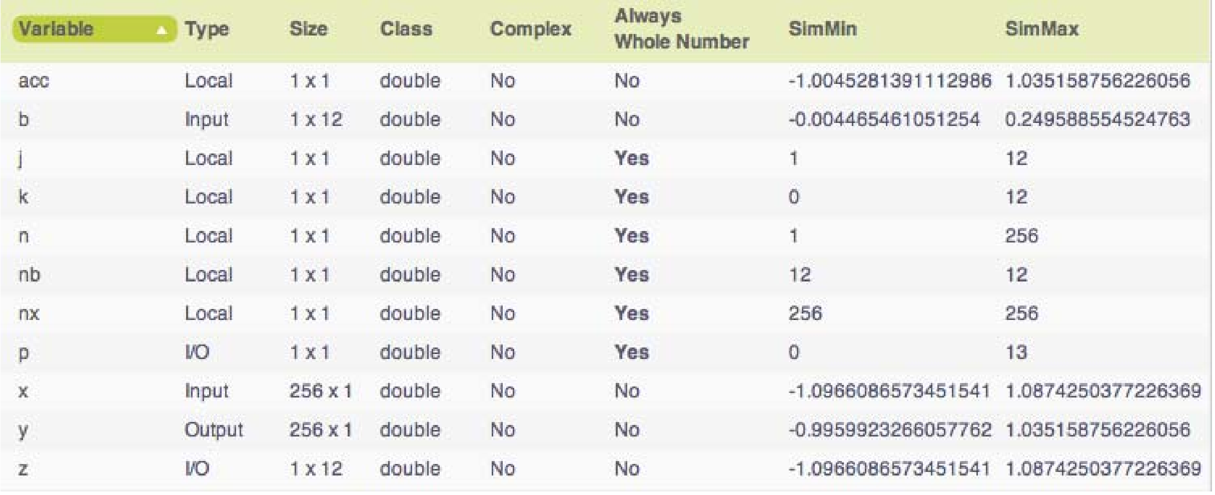
Show types

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

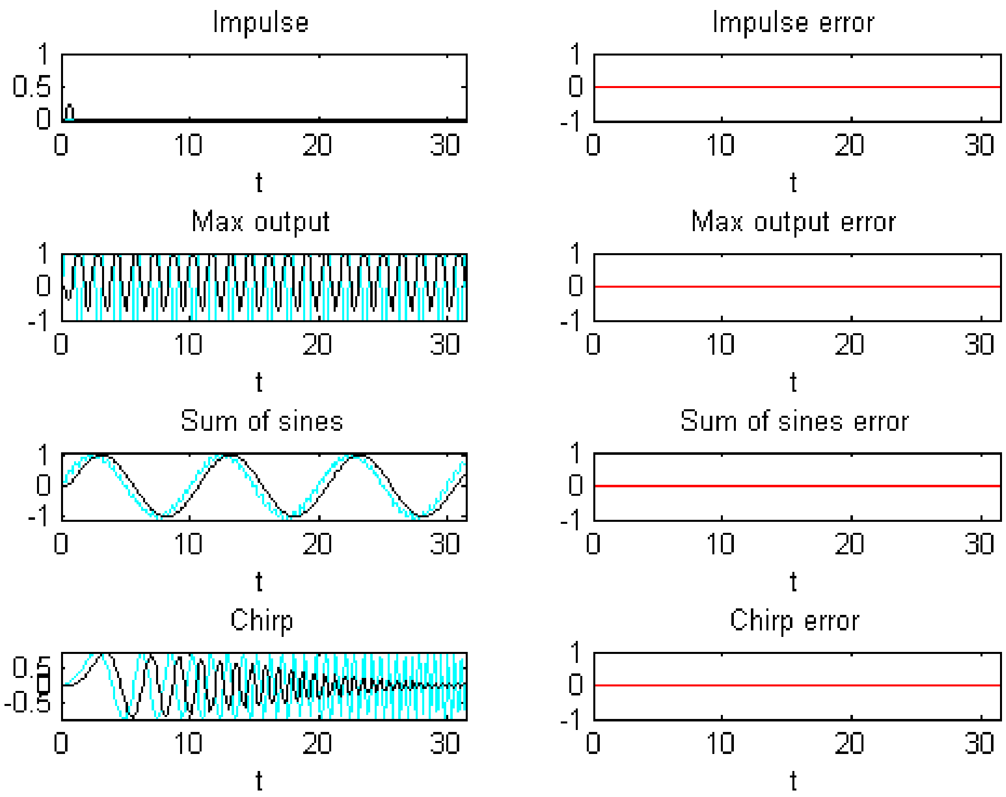


Variable ▲	Type	Size	Class	Complex	Always Whole Number	SimMin	SimMax
acc	Local	1 x 1	double	No	No	-1.0045281391112986	1.035158756226056
b	Input	1 x 12	double	No	No	-0.004465461051254	0.249588554524763
j	Local	1 x 1	double	No	**Yes**	1	12
k	Local	1 x 1	double	No	**Yes**	0	12
n	Local	1 x 1	double	No	**Yes**	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);
        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.

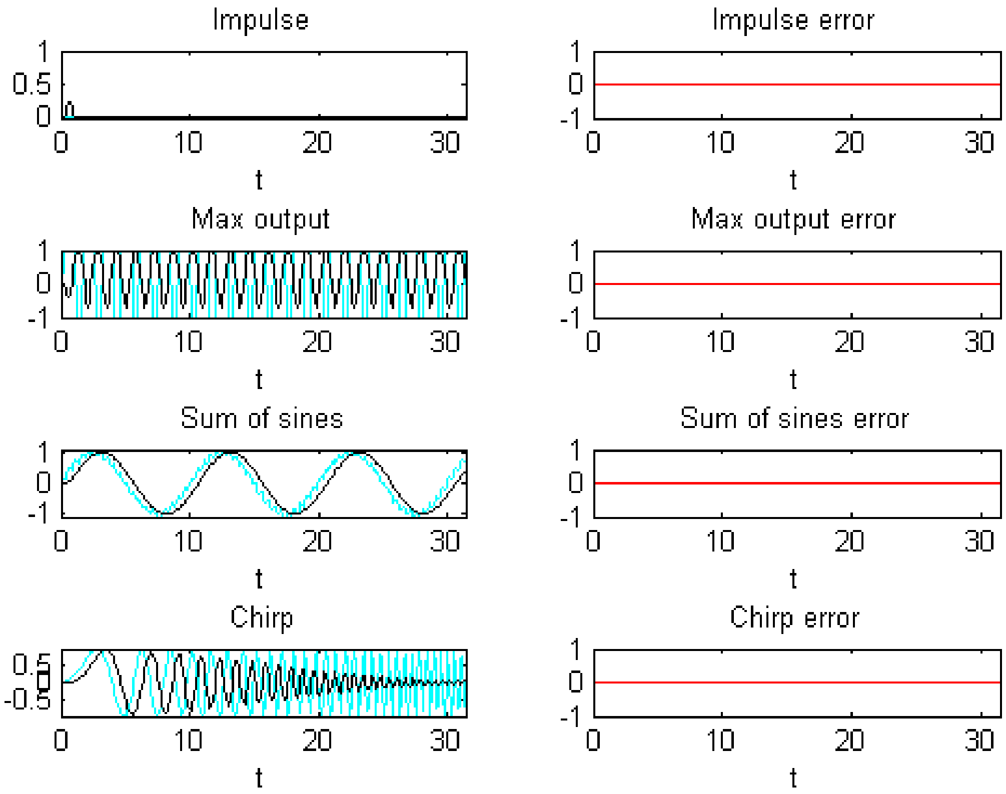
The function call z1 = zeros(size(b),’like’,T1.z) creates an array of zeros the same size as b with the properties of variable T1.z. Initially, T1.z is a double defined in function fir_filt_circ_buff_original_types, but it is re-defined as a fixed-point type later in this example.

```
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),

y1 = fir_filt_circ_buff_typed_test(b,x);
fir_filt_circ_buff_plot2(3,titles,t,x,y0,y1)
```

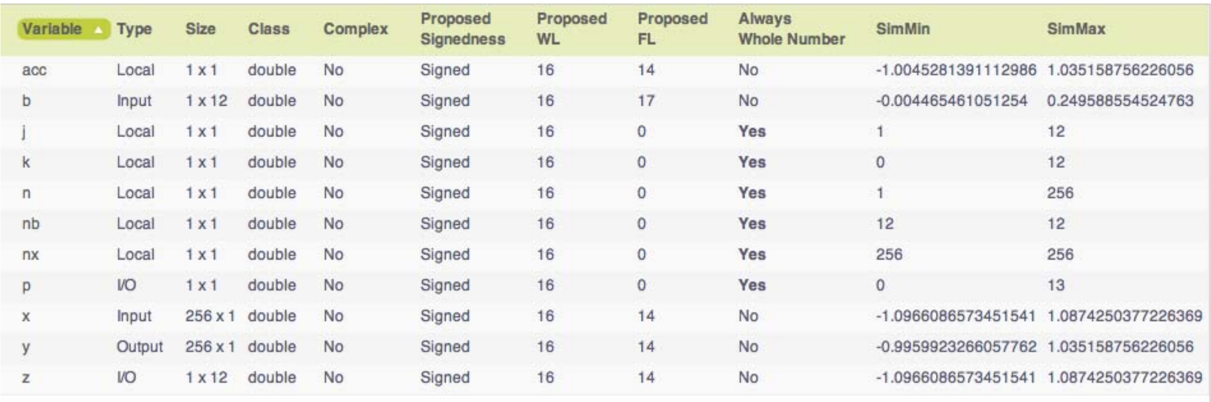


Propose data types from simulation min/max logs

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

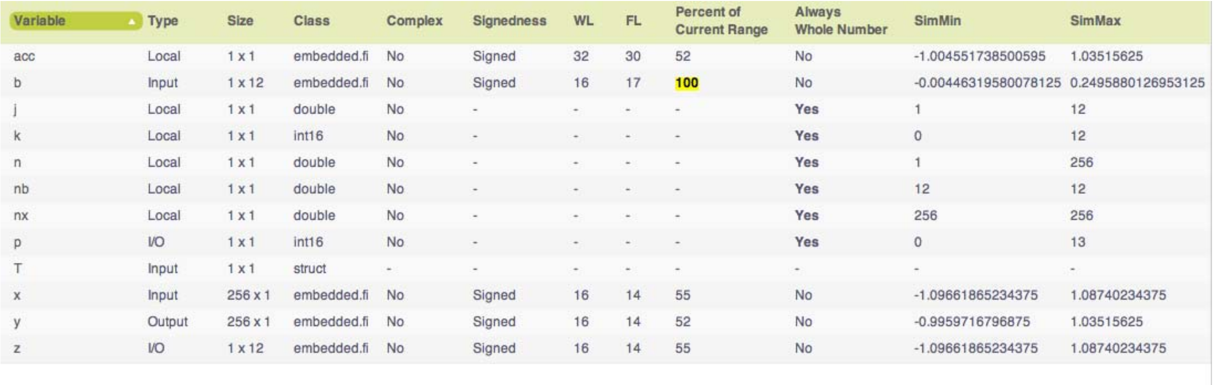
```
buildInstrumentedMex fir_filt_circ_buff_typed_entry_point -args {b, x(:,1),

[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

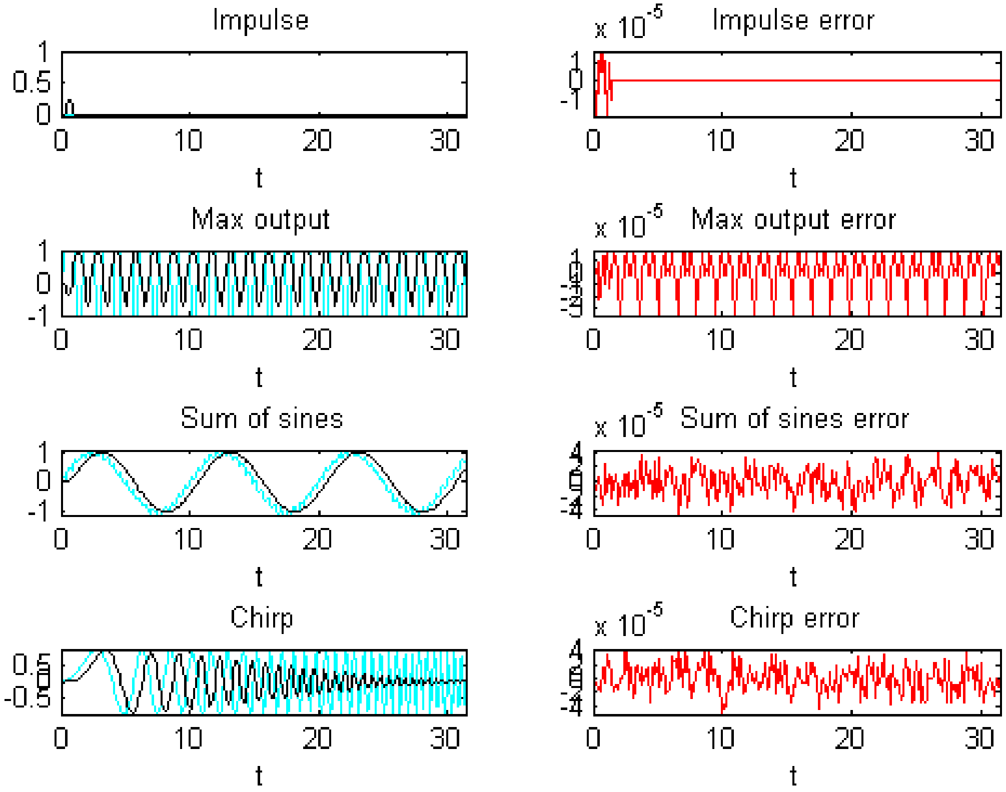


Variable ▲	Type	Size	Class	Complex	Signedness	WL	FL	Percent of Current Range	Always Whole Number	SimMin	SimMax
acc	Local	1 x 1	embedded.fi	No	Signed	32	30	52	No	-1.004551738500595	1.03515625
b	Input	1 x 12	embedded.fi	No	Signed	16	17	**100**	No	-0.00446319580078125	0.2495880126953125
j	Local	1 x 1	double	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
```

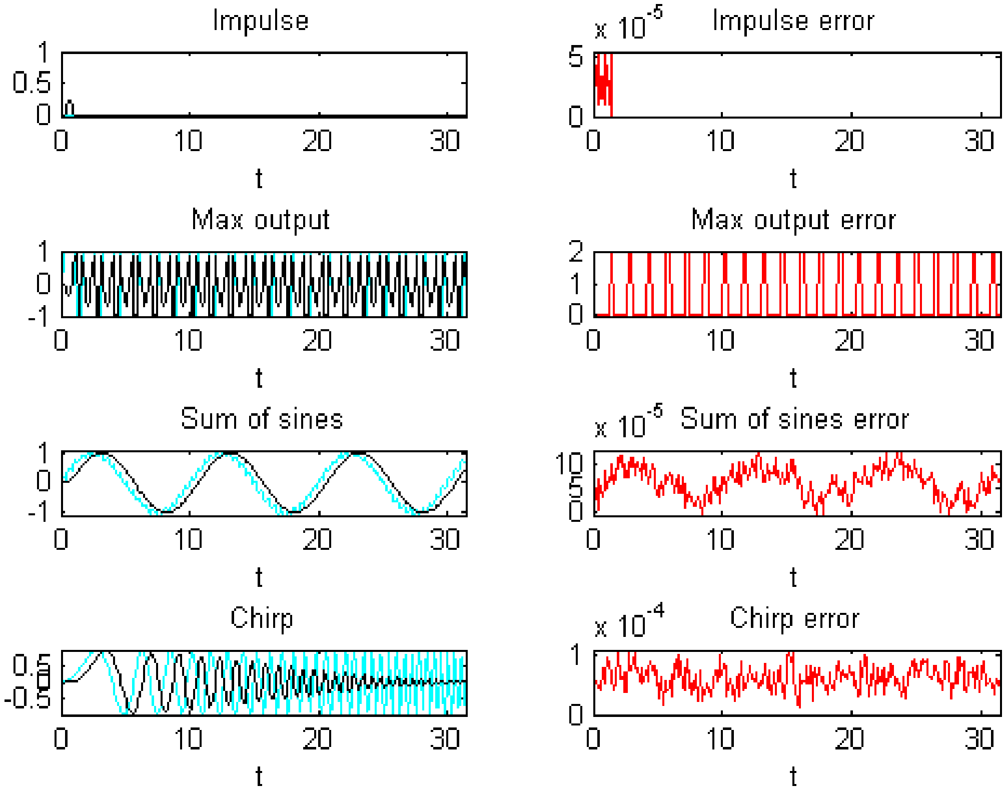
Test the native C-code types

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:

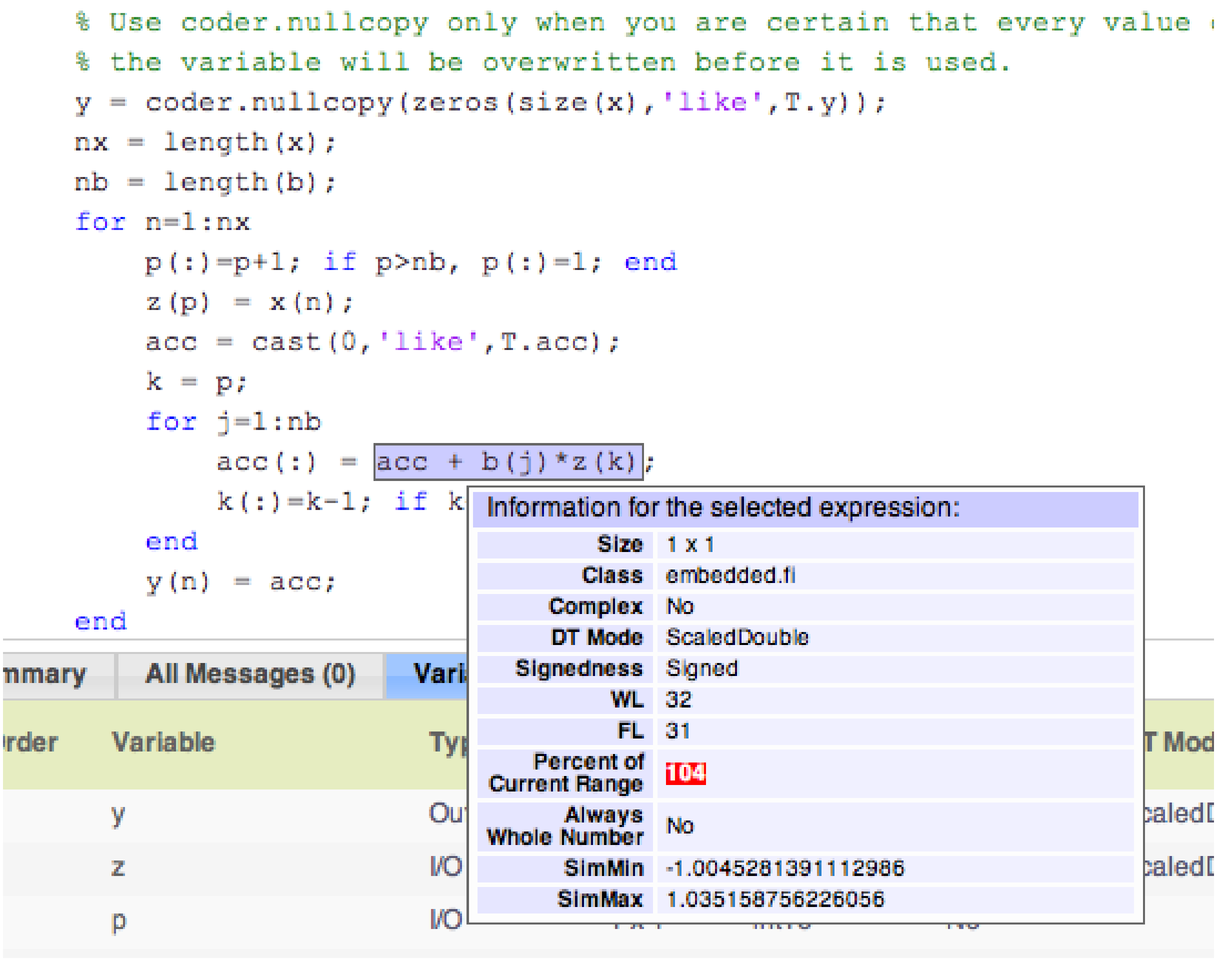
- Select the entry-point function, fir_filt_circ_buff_typed_entry_point
- Select fir_filt_circ_buff_typed_codegen in the following line of code:

```
[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 every value
% 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
```

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=fi([],true,16,16,F);
```

Then the sum in this line of MATLAB code no longer overflows:

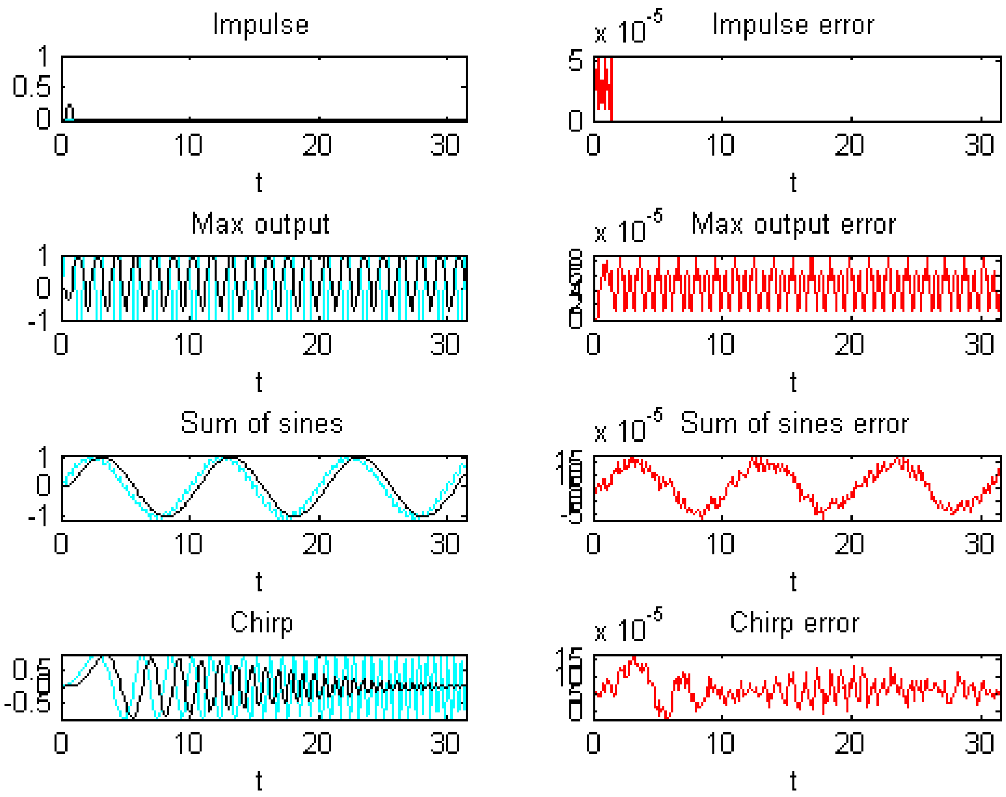
```
acc(:) = acc + b(j)*z(k);
```

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

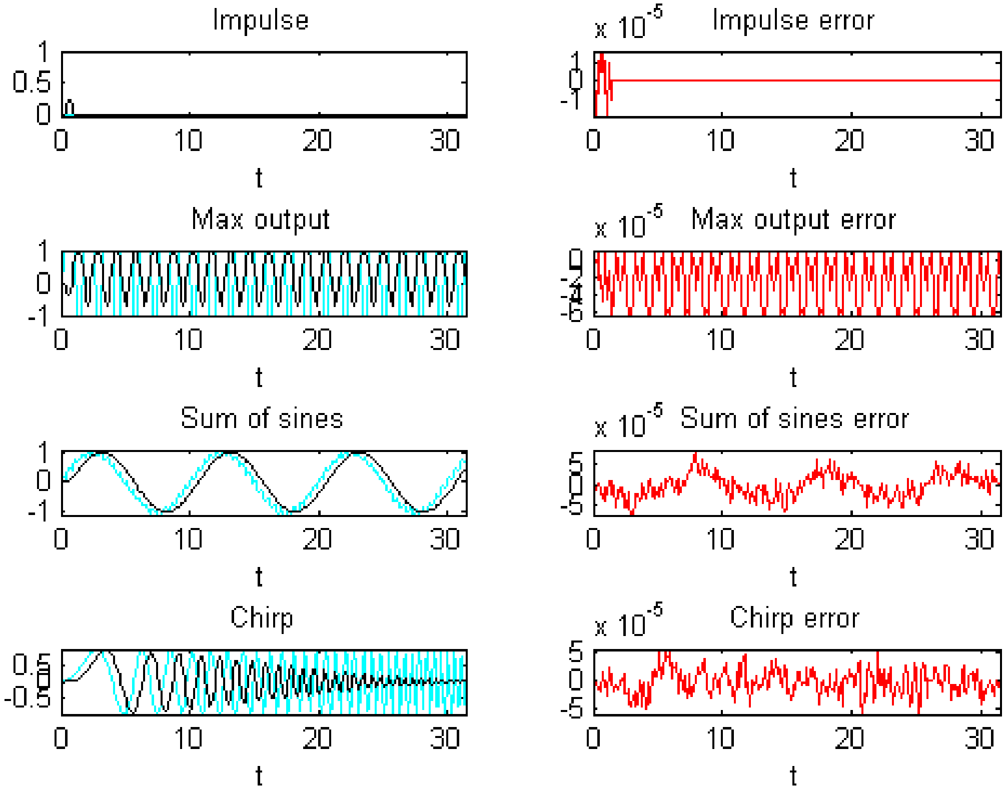
If the bias is not acceptable in your application, then change the rounding method to 'Nearest' to eliminate the bias. Rounding to nearest generates slightly more complicated C-code, but it may be necessary for you if you want to eliminate the bias and have a smaller error.

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 -l
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[
  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;
```

Propose Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

Before generating C code, you must set up the C compiler. See "Set Up C Compiler" on page 9-16.

For instructions on installing MathWorks® products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver` .

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\coder\fun_with_matlab`.
2. Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
   ```

3. Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	fun_with_matlab.m	Entry-point MATLAB function
Test file	fun_with_matlab_test.m	MATLAB script that tests fun_with_matlab.m

The fun_with_matlab Function

```
function y = fun_with_matlab(x) %#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 fun_with_matlab_test Script

The test script runs the fun_with_matlab 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.

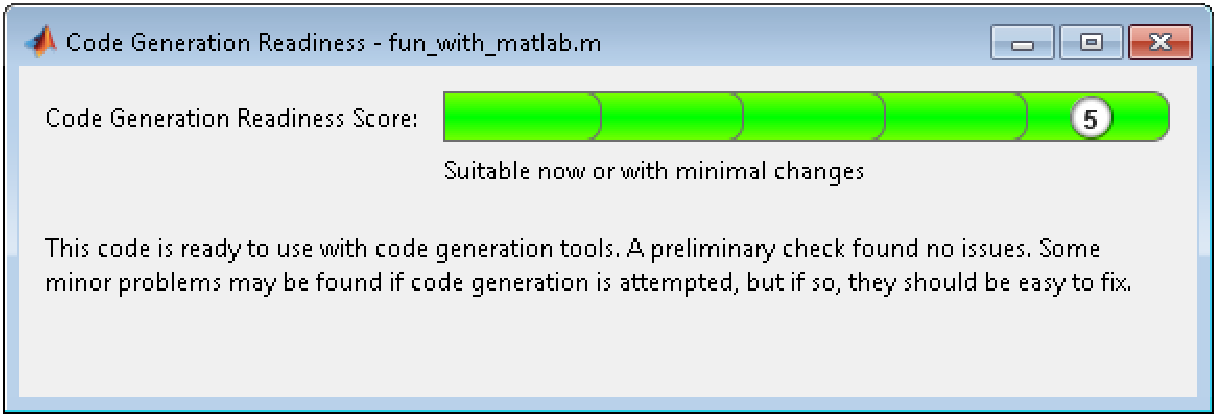
```
% fun_with_matlab_test
%
% Define representative inputs
N = 256;                     % Number of points
t = linspace(0,1,N);         % Time vector from 0 to 1 second
f1 = N/2;                    % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
```

```
x_step = ones(1,N);         % Step
x_impulse = zeros(1,N);     % Impulse
x_impulse(1)=1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i=1:size(x,1)
  y(i,:) = fun_with_matlab(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i=1:size(x,1)
  subplot(size(x,1),1,i);
  plot(t,x(i,:),t,y(i,:));
  title(titles{i})
  legend('Input','Output');
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.');
```

Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select `Check Code Generation Readiness`.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.



Code Generation Readiness Score:

Suitable now or with minimal changes

This code is ready to use with code generation tools. A preliminary check found no issues. Some minor problems may be found if code generation is attempted, but if so, they should be easy to fix.

If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “Detect and Debug Code Generation Errors” on page 9-25.

Create and set up a MATLAB Coder Project

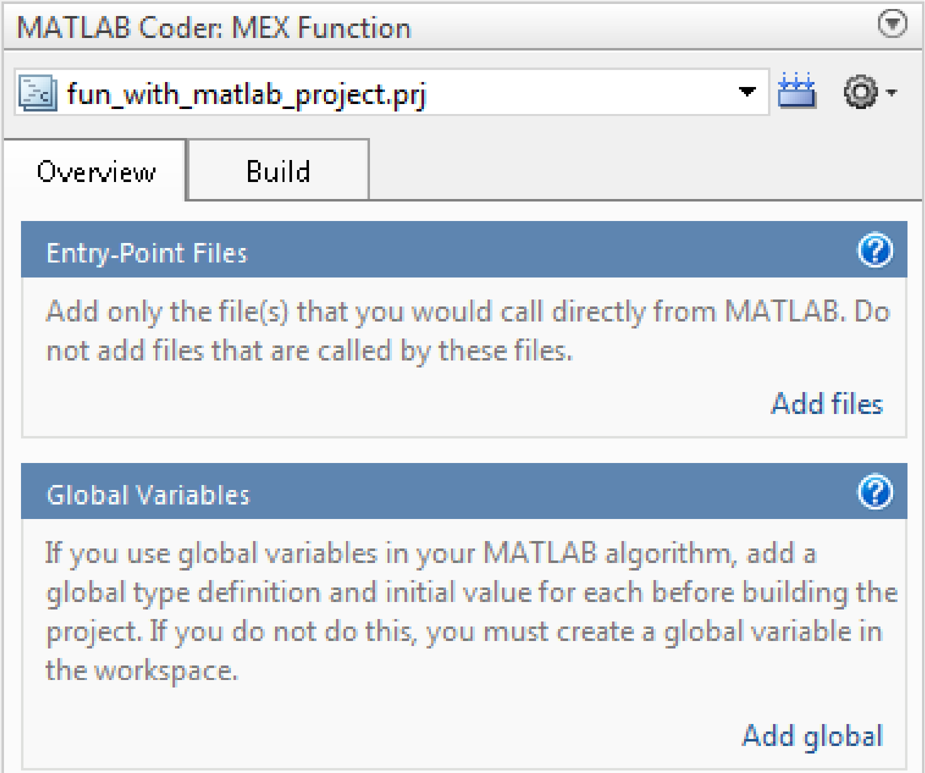
1 Navigate to the work folder that contains the file for this example.

2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `fun_with_matlab_project.prj`.

   Alternatively, at the MATLAB command line, enter

   ```
   coder -new fun_with_matlab_project.prj
   ```

   By default, the project opens in the MATLAB workspace.



MATLAB Coder: MEX Function

fun_with_matlab_project.prj

Entry-Point Files

Add only the file(s) that you would call directly from MATLAB. Do not add files that are called by these files.

Global Variables

If you use global variables in your MATLAB algorithm, add a global type definition and initial value for each before building the project. If you do not do this, you must create a global variable in the workspace.

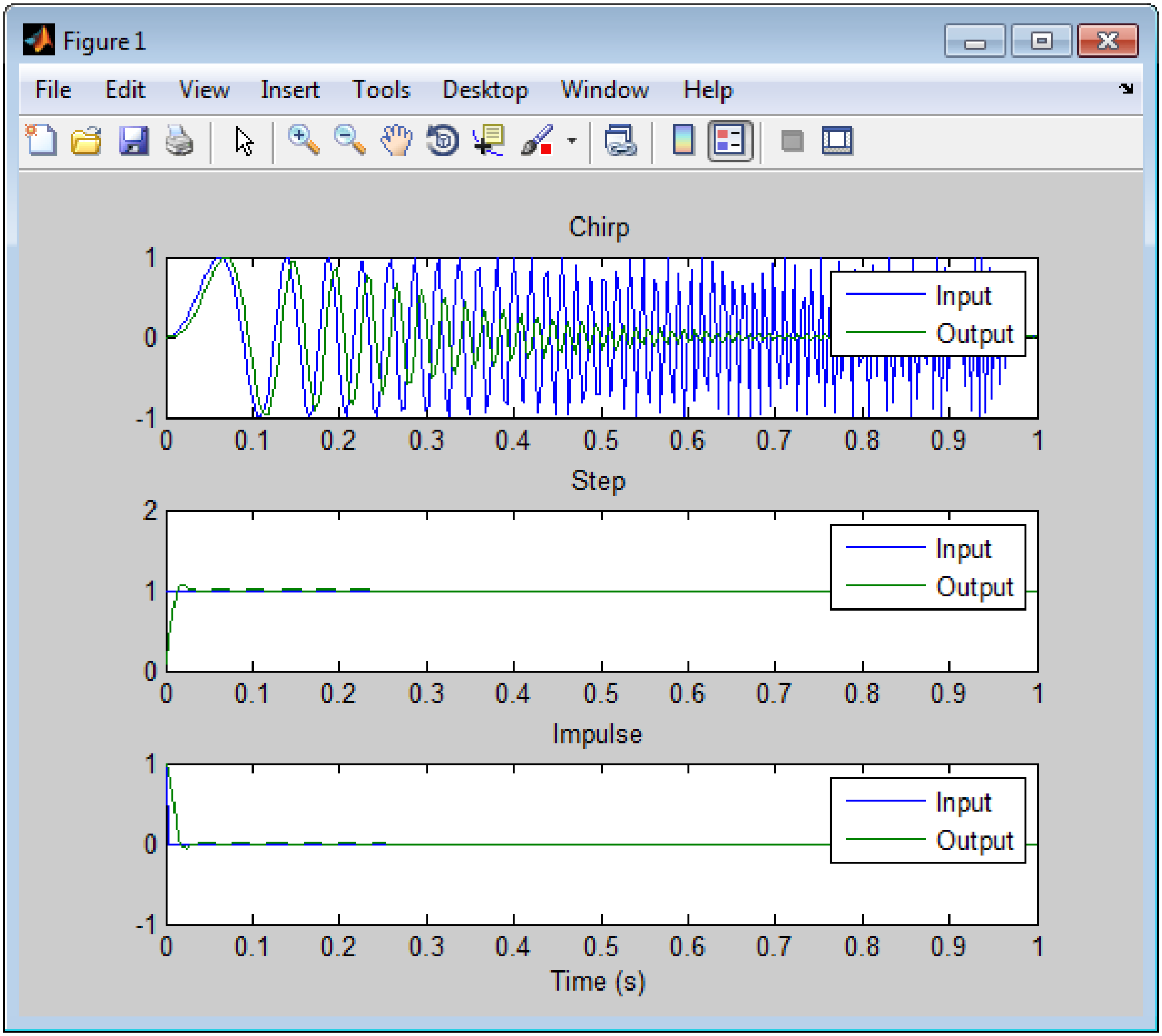
3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

Define Input Types

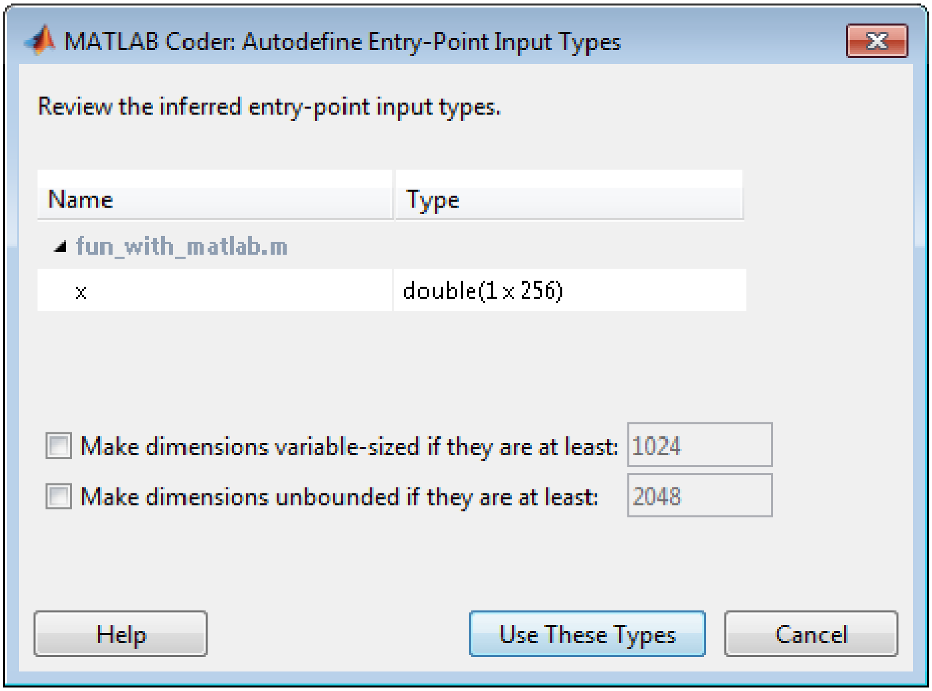
1 On the project **Overview** tab, click the **Autodefine types** link.

2 In the Autodefine Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them.



MATLAB Coder: Autodefine Entry-Point Input Types

Review the inferred entry-point input types.

Name	Type
◢ fun_with_matlab.m	
x	double(1 x 256)

□ Make dimensions variable-sized if they are at least: 1024

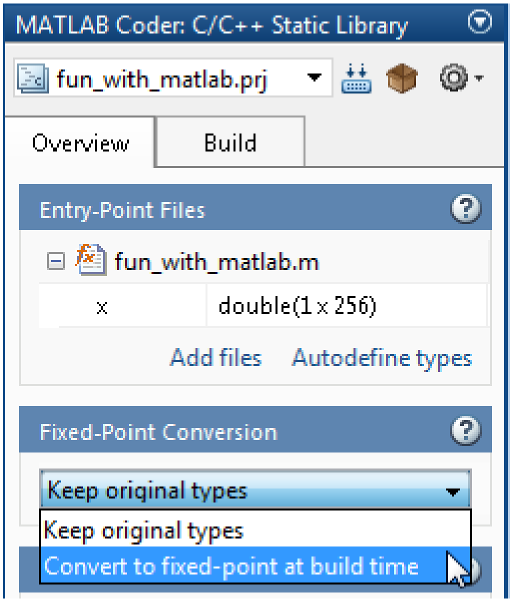
□ Make dimensions unbounded if they are at least: 2048

3 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

Fixed-Point Conversion

1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.



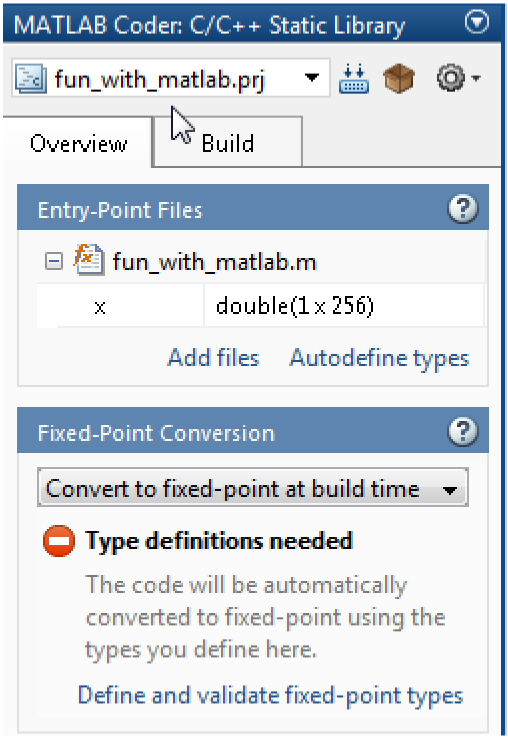
fun_with_matlab.m

x	double(1 x 256)

Fixed-Point Conversion: Keep original types

- Keep original types
- Convert to fixed-point at build time

The project indicates that you must first define the fixed-point data types.



x double(1 x 256)

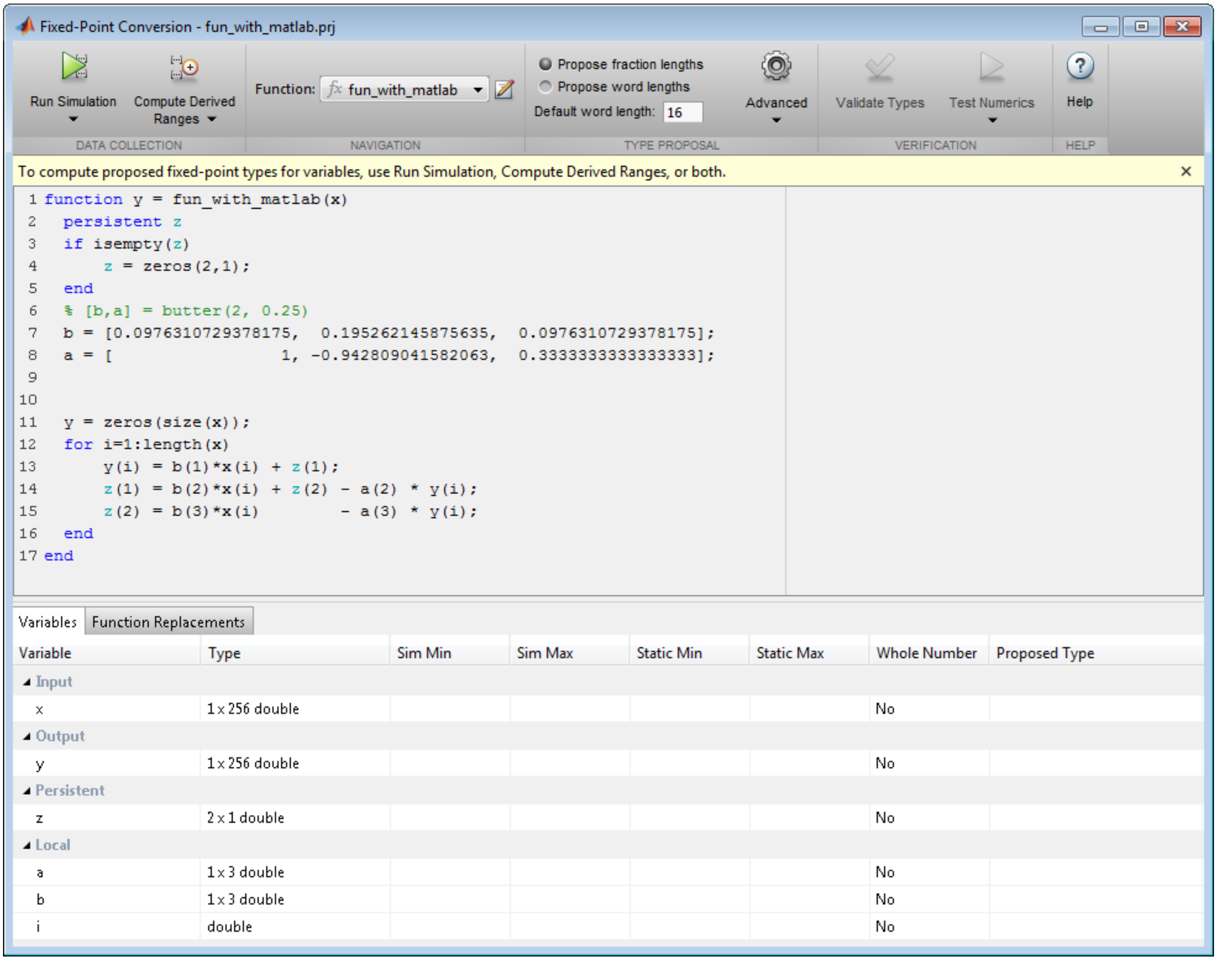
Convert to fixed-point at build time

Type definitions needed

The code will be automatically converted to fixed-point using the types you define here.

2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information”.



Function: fun_with_matlab

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.

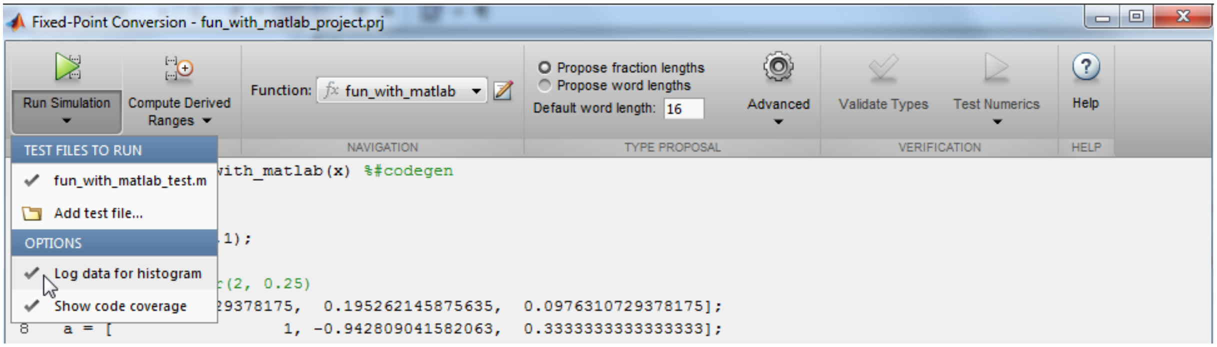
```
function y = fun_with_matlab(x)
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                  1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double					No	
Output							
y	1 x 256 double					No	
Persistent							
z	2 x 1 double					No	
Local							
a	1 x 3 double					No	
b	1 x 3 double					No	
i	double					No	

If the MEX function generation fails, the tool provides error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation”.

3 Click **Run Simulation** and verify that the `fun_with_matlab_test` file is selected as a test file to run. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results. To clear results, right-click the **Variables** tab and select `Reset entire table`.

4 Click **Run Simulation** and select `Log data for histogram`.

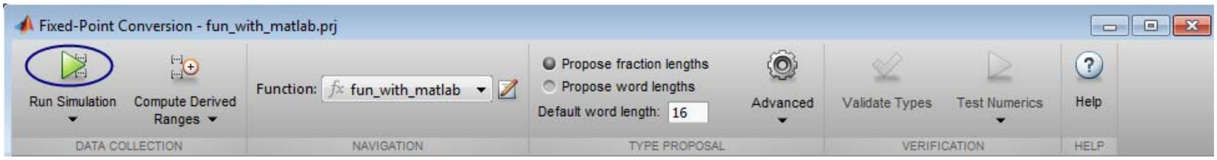


● Propose fraction lengths
○ Propose word lengths
Default word length: 16

```
vith_matlab(x) %#codegen

1);

:(2, 0.25)
29378175,   0.195262145875635,   0.0976310729378175];
    a = [              1, -0.942809041582063,   0.333333333333333];
```

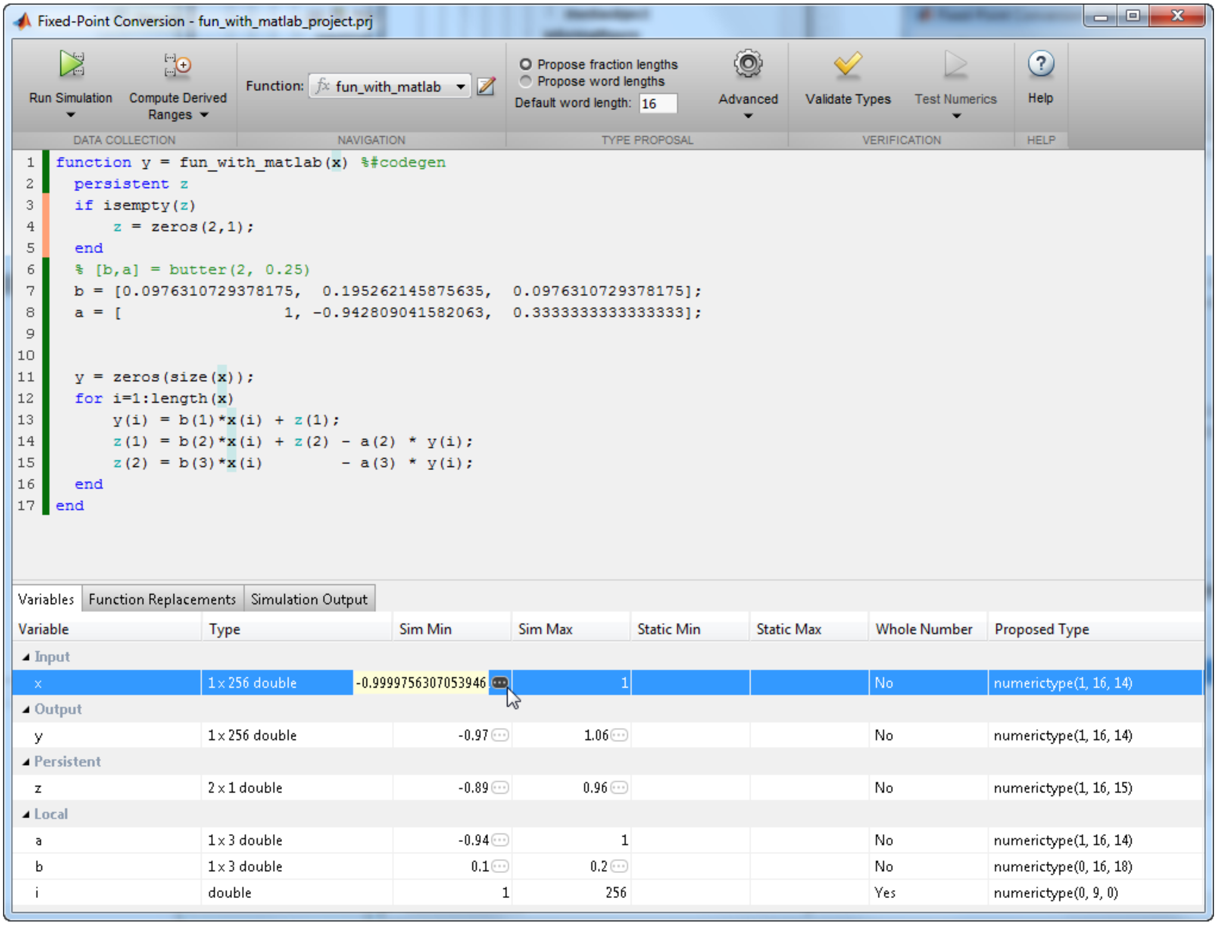
By default, the `Show code coverage` option is selected. This option provides code coverage information that helps you verify that your test file is testing your algorithm over the intended operating range.

5 Click the Run Simulation button.



● Propose fraction lengths
○ Propose word lengths
Default word length: 16

The simulation runs and the conversion tool 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. Here, the dark green line to the left of the code indicates that the code is run every time the algorithm is executed. The orange bar indicates that the code next to it is executed only once. In this example, this is the expected behavior because the code is initializing a persistent variable. If your test file is not covering all your code, update the test or add more test files. For more information, see "Code Coverage" on page 8-3.



● Propose fraction lengths
○ Propose word lengths
Default word length: 16

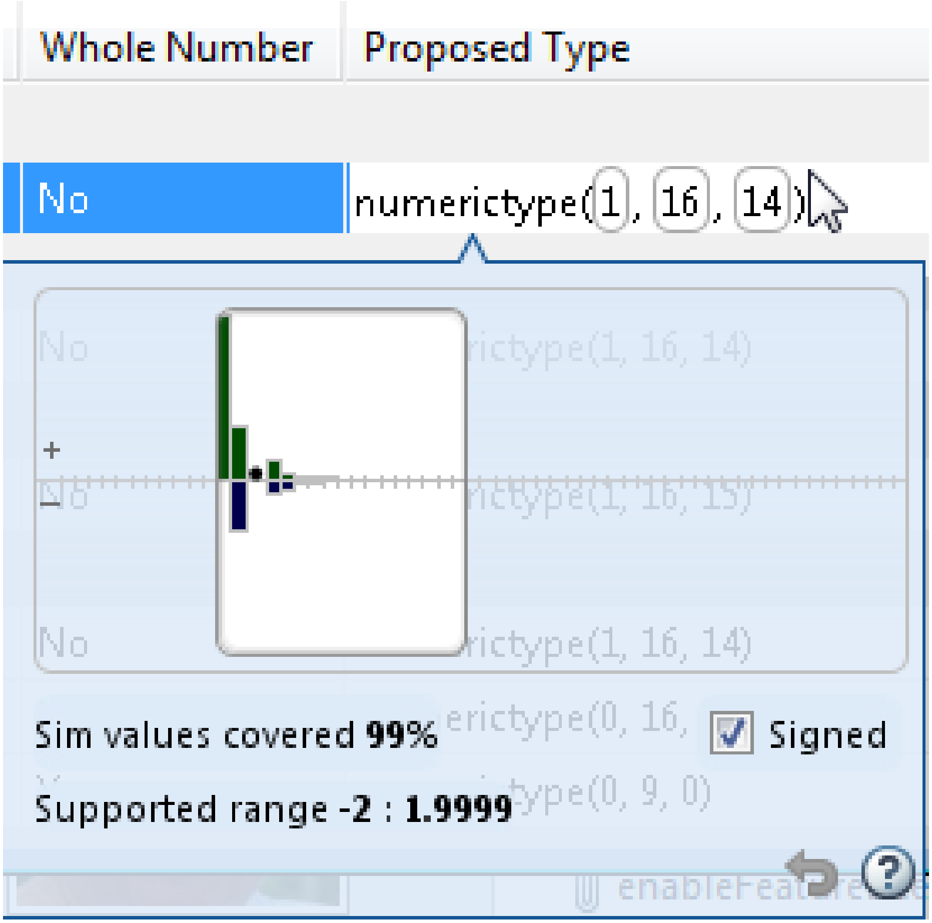
```
function y = fun_with_matlab(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                 1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double	-0.9999756307053946	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							
a	1 x 3 double	-0.94 …	1			No	numerictype(1, 16, 14)
b	1 x 3 double	0.1 …	0.2 …			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)

If a value has ... next to it, the value is rounded. Place your cursor over the ... to view the actual value.

The tool 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 **Validate Types** option is now enabled.

6 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.



Whole Number	Proposed Type
No	numerictype(1, 16, 14)

Sim values covered **99%**

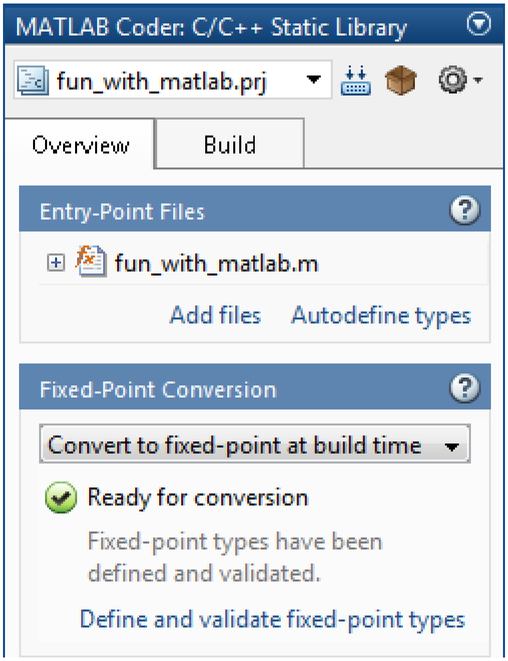
☑ Signed

Supported range **-2 : 1.9999**

To modify the proposed data types, either enter the required type into the **ProposedType** field or use the histogram controls. For more information about the histogram, see “Histogram”.

7 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a `Validation succeeded` message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.



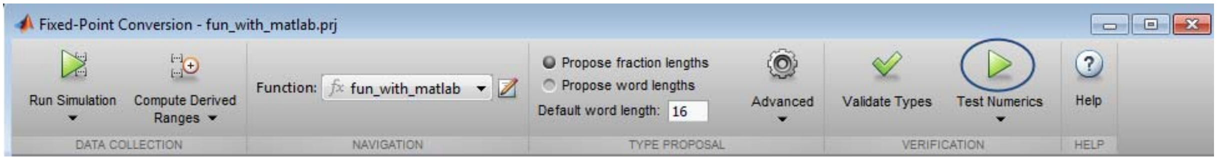
Convert to fixed-point at build time

Ready for conversion

Fixed-point types have been defined and validated.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types”.

8 Click **Test Numerics**, select `Log inputs and outputs for comparison plots`, and then click the Test Numerics button.



Function: fun_with_matlab

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

The tool 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, the tool generates an additional plot for each scalar output.

The maximum error is less than 0.03%. For the purpose of this example, this margin of error is acceptable, so you are ready to generate fixed-point C code.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics”.

9 Return to the MATLAB Coder project.

Generate Fixed-Point C Code

1 In the MATLAB Coder project, verify that the **Fixed-Point Conversion** pane displays **Ready for conversion**, and then select the **Build** tab.

2 On this tab, set the **Output type** to `C/C++ Static library`.

The default output file name is `fun_with_matlab`.

3 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/fun_with_matlab_fixpt`.

4 To view the generated code, click **View report**.

The code generation report opens and displays the generated code for `fun_with_matlab_fixpt.c`. In the generated C code, the variables are not assigned real types, they are assigned fixed-point data types.

In this case, the generated code is not optimized; it contains a number of utility functions, such as `MultiWordAdd`. MATLAB Coder generates these utility functions because the results of adding or multiplying inputs results in a sum or product that exceed 32 bits. You can optimize the generated code by modifying the word length and `fimath` settings.

Optimize Fixed-Point C Code

1 In the Fixed-Point Conversion tool, click **Advanced** to display the advanced type proposal settings.

   The fimath **Product mode** and **Sum mode** settings are both set to `FullPrecision`. In `FullPrecision` mode, the product word length grows to the sum of the word lengths of the operands.

2 Set the fimath **Product mode** and **Sum mode** to `SpecifyPrecision`.

   Selecting `SpecifyPrecision` enables the `Product word length`, `Product fraction length`, `Sum word length`, and `Sum fraction length` settings. The product word length and sum word length are both set to 32, which limits these word lengths to 32 in the generated code.

3 Click **Validate Types**.

   Because you have changed type proposal settings, you must validate the types again.

   The software validates the proposed types, displays a `Validation succeeded` message.

4 Click the Test Numerics button.

   The maximum error is still less than 0.03%, so you are ready to generate fixed-point C code.

5 Generate code again and view the generated C code for `fun_with_matlab_fixpt.c`. This time, because the word lengths in the generated code do not exceed 32 bits, the generated code does not contain utility functions.

```
void fun_with_matlab_fixpt(const short x[256], short y[256])
{
  int i0;
  int i;
  int i1;
  short b_y;
  int i2;
  int i3;
```

```
/*  [b,a] = butter(2, 0.25) */
for (i0 = 0; i0 < 256; i0++) {
  y[i0] = 0;
}

for (i = 0; i < 256; i++) {
  i0 = 25593 * x[i];
  if (i0 >= 0) {
    i1 = (int)((unsigned int)i0 >> 2);
  } else {
    i1 = ~(int)((unsigned int)~i0 >> 2);
  }

  i0 = i1 + (z[0] << 15);
  if (i0 >= 0) {
    b_y = (short)((unsigned int)i0 >> 16);
  } else {
    b_y = (short)~(int)((unsigned int)~i0 >> 16);
  }

  i0 = 25593 * x[i];
  if (i0 >= 0) {
    i2 = (int)((unsigned int)i0 >> 1);
  } else {
    i2 = ~(int)((unsigned int)~i0 >> 1);
  }

  i0 = (i2 + (z[1] << 15)) - (-30894 * b_y << 1);
  if (i0 >= 0) {
    z[0] = (short)((unsigned int)i0 >> 15);
  } else {
    z[0] = (short)~(int)((unsigned int)~i0 >> 15);
  }

  i0 = 25593 * x[i];
  if (i0 >= 0) {
    i3 = (int)((unsigned int)i0 >> 2);
  } else {
    i3 = ~(int)((unsigned int)~i0 >> 2);
    }

    i0 = i3 - 21844 * b_y;
    if (i0 >= 0) {
      z[1] = (short)((unsigned int)i0 >> 15);
    } else {
      z[1] = (short)~(int)((unsigned int)~i0 >> 15);
    }

    y[i] = b_y;
  }
}
```

Propose Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges that you specify. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

Before generating C code, you must set up the C compiler. See “Set Up C Compiler” on page 9-16.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver` .

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\coder\dti`.

2. Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
   ```

3. Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	dti.m	Entry-point MATLAB function
Test file	dti_test.m	MATLAB script that tests dti.m

The dti Function

The dti function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
 y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
```

```
 y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;

function b = subFunction(a)
b = a*a;
```

The dti_test Function

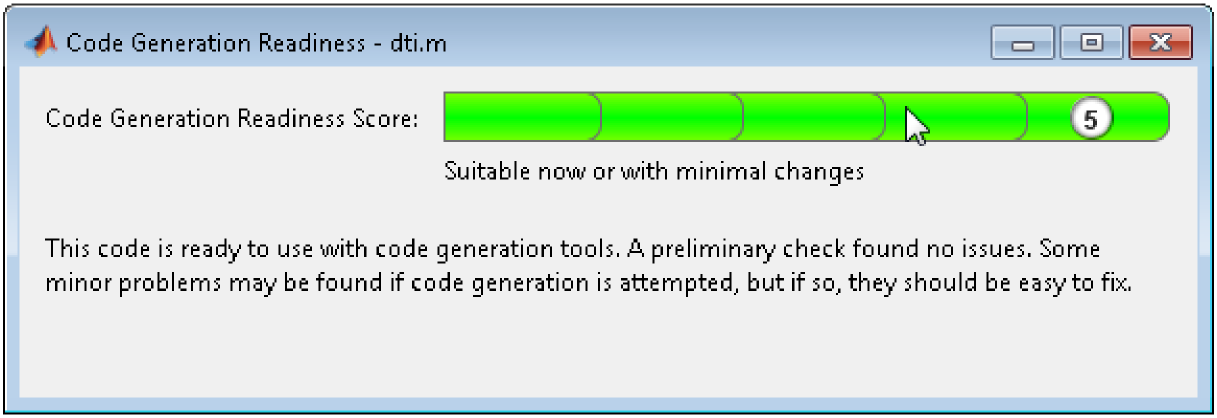
The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10);

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;
    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);

end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:len,x_in);
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2); plot(1:len,y_out);
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')

disp('Test complete.');
```

Check Code Generation Readiness

In the current working folder, right-click the `dti.m` function. From the context menu, select `Check Code Generation Readiness`.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `dti.m` function is already suitable for code generation.



Code Generation Readiness - dti.m

Code Generation Readiness Score: 5

Suitable now or with minimal changes

This code is ready to use with code generation tools. A preliminary check found no issues. Some minor problems may be found if code generation is attempted, but if so, they should be easy to fix.

If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see "Detect and Debug Code Generation Errors" on page 9-25.

Create and set up a MATLAB Coder Project

1 Navigate to the work folder that contains the file for this example.

2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `dti.prj`.

   Alternatively, at the MATLAB command line, enter

```
coder -new dti.prj
```

   By default, the project opens in the MATLAB workspace.

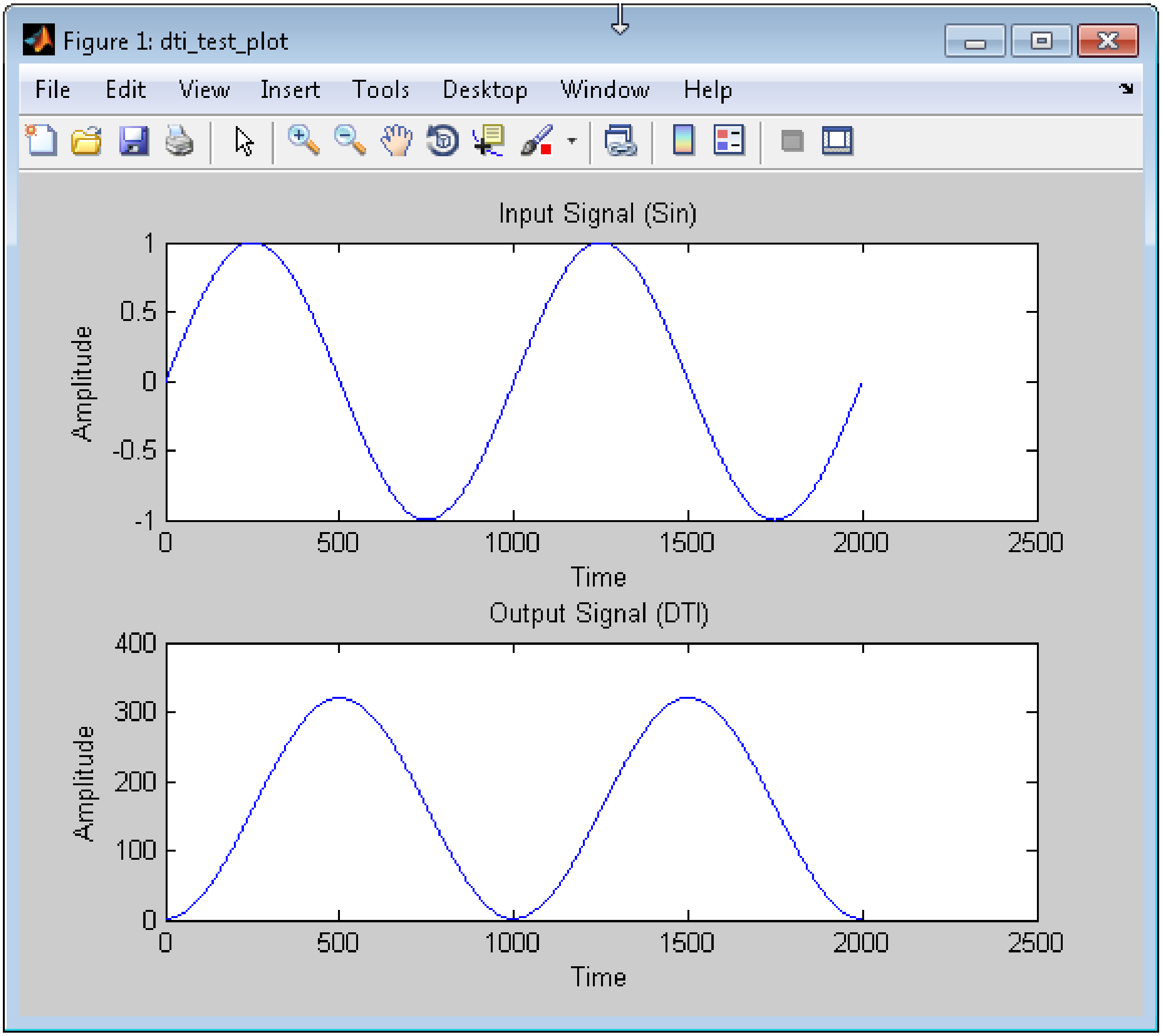
3 On the project **Overview** tab, click the **Add files** link. Browse to the file `dti.m` and then click **OK** to add the file to the project.

Define Input Types

1 On the project **Overview** tab, click the **Autodefine types** link.

2 In the Autodefine Input Types dialog box, add `dti_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



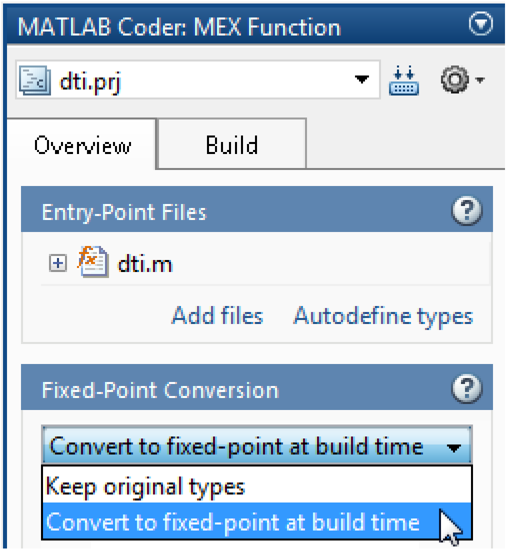
MATLAB Coder determines the input types from the test file and then displays them.

3 In the Autodefine Input Types dialog box, click **Use These Types**.

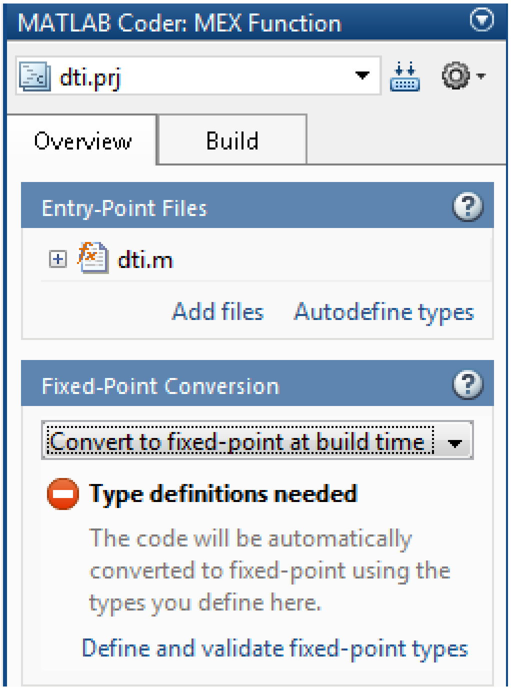
MATLAB Coder sets the type of `x` to `double(1x1)`.

Fixed-Point Conversion

1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.



The project indicates that you must first define the fixed-point data types.



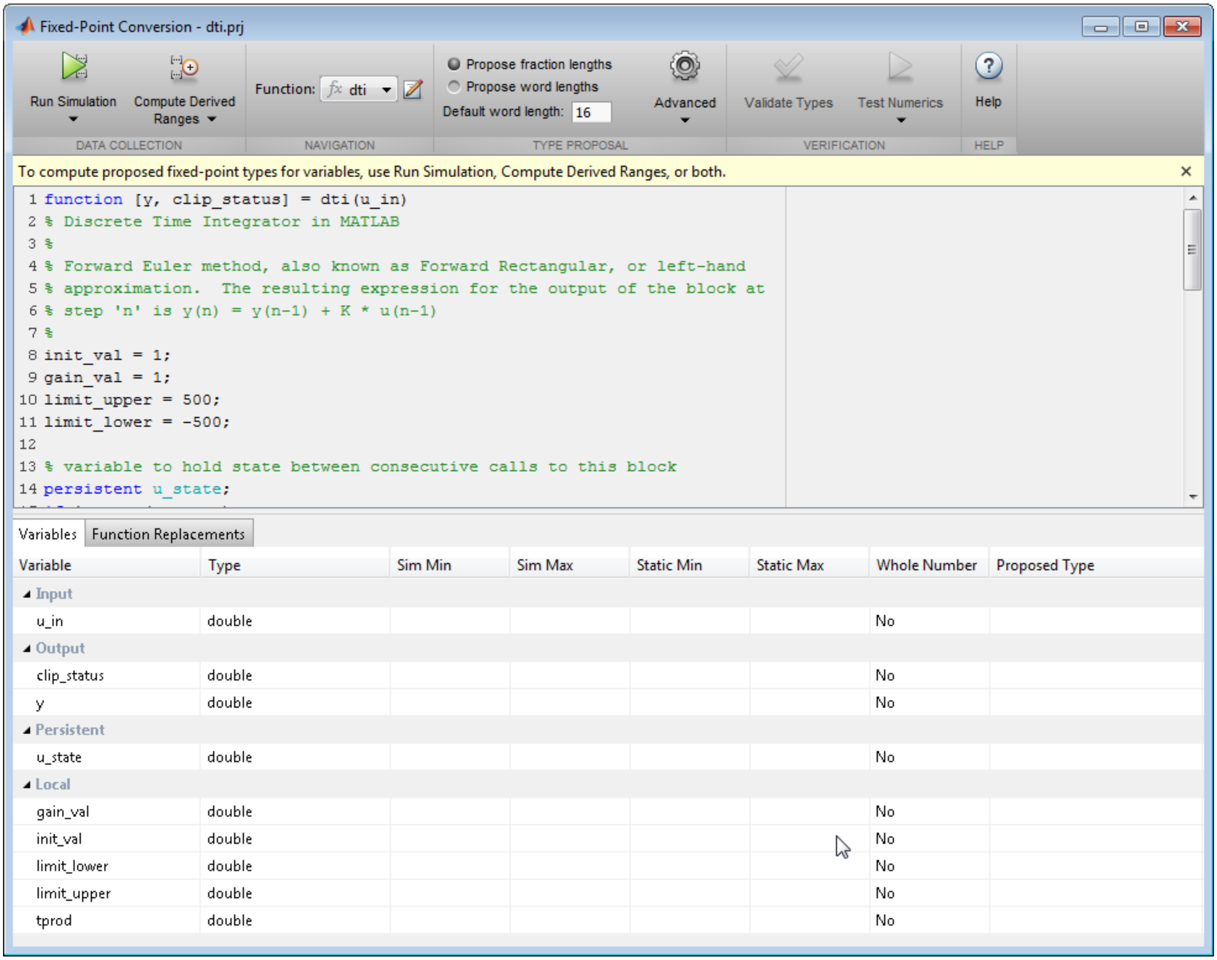
Convert to fixed-point at build time

Type definitions needed

The code will be automatically converted to fixed-point using the types you define here.

2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information”.



Function: dti

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.

```
function [y, clip_status] = dti(u_in)
% 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;
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u_in	double					No	
Output							
clip_status	double					No	
y	double					No	
Persistent							
u_state	double					No	
Local							
gain_val	double					No	
init_val	double					No	
limit_lower	double					No	
limit_upper	double					No	
tprod	double					No	

If the MEX function generation fails, the tool provides error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation”.

3 In the Fixed-Point Conversion window, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to `-1`. Then set **Static Max** to `1`.

To compute derived range information, at a minimum you must specify static minimum and maximum values for all input variables. Alternatively, if you know what data type your hardware target uses, set the proposed data type to match this type.

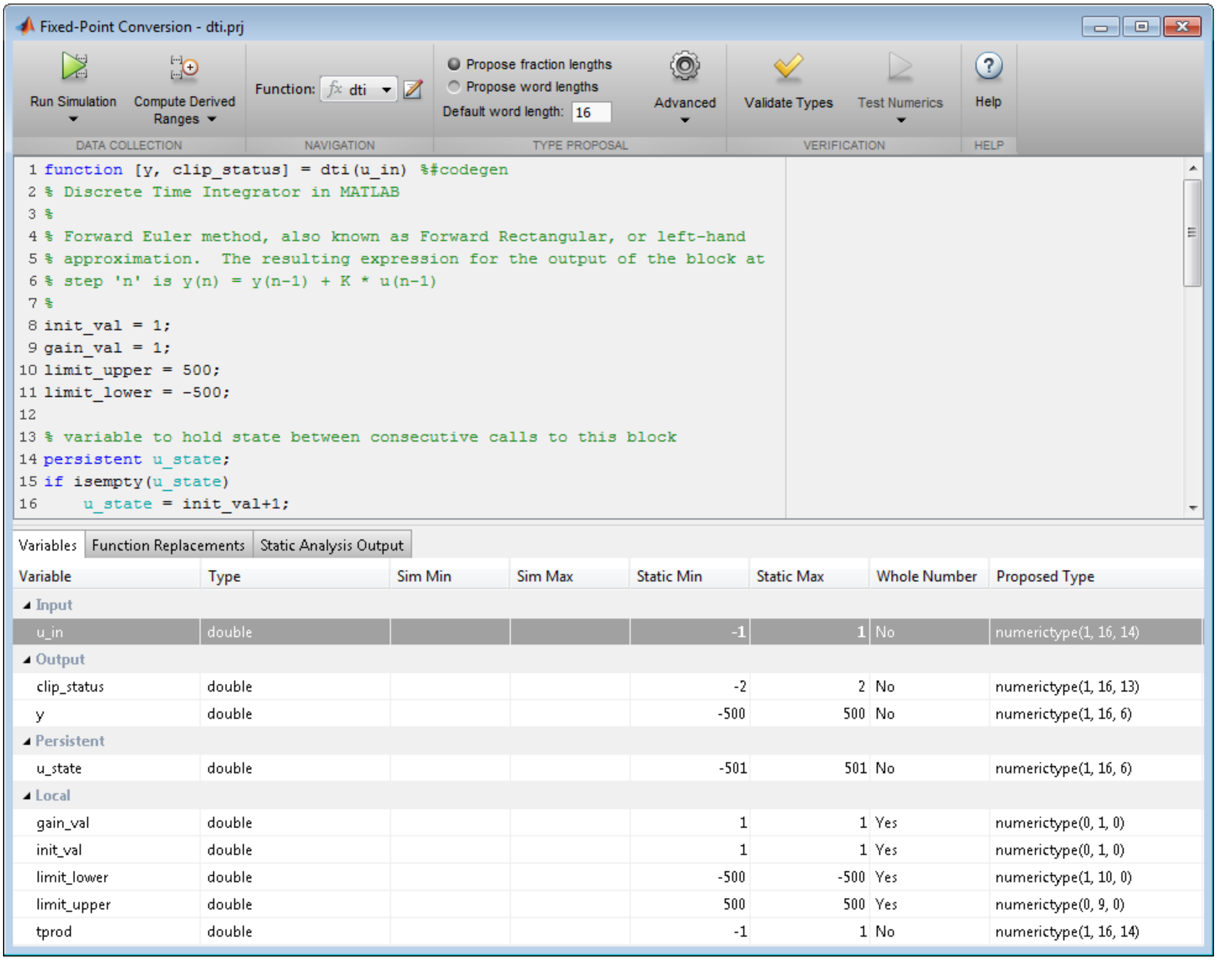
4 Click the Compute Derived Ranges button.

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, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

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 Fixed-Point Conversion tool analyses the function and computes these minimum and maximum values for clip_status.



Function: dti

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

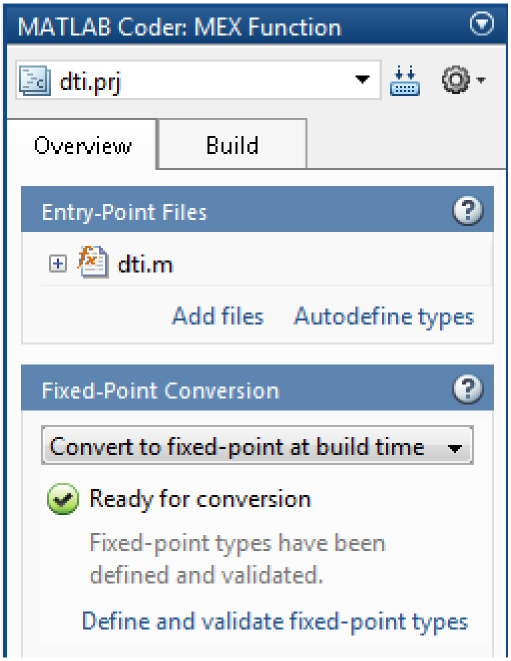
```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u_in	double			-1	1	No	numerictype(1, 16, 14)
Output							
clip_status	double			-2	2	No	numerictype(1, 16, 13)
y	double			-500	500	No	numerictype(1, 16, 6)
Persistent							
u_state	double			-501	501	No	numerictype(1, 16, 6)
Local							
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)

The tool provides a **Quick derived range analysis** option and the option to specify a timeout in case the analysis takes a very long time. For more information, see "Computing Derived Ranges"

5 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a `Validation succeeded` message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.



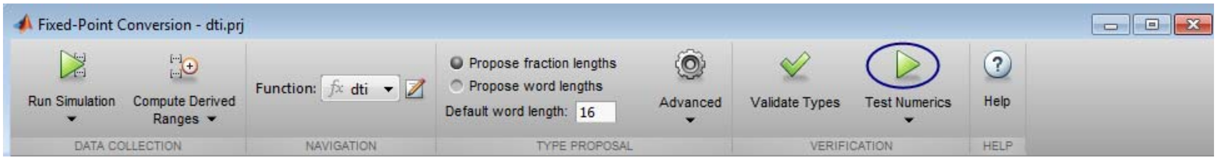
Convert to fixed-point at build time

Ready for conversion

Fixed-point types have been defined and validated.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see "Validating Types".

6 Run the test file to test the fixed-point MATLAB code. Click **Test Numerics** and select `Log inputs and outputs for comparison plots`, and then click the Test Numerics button.



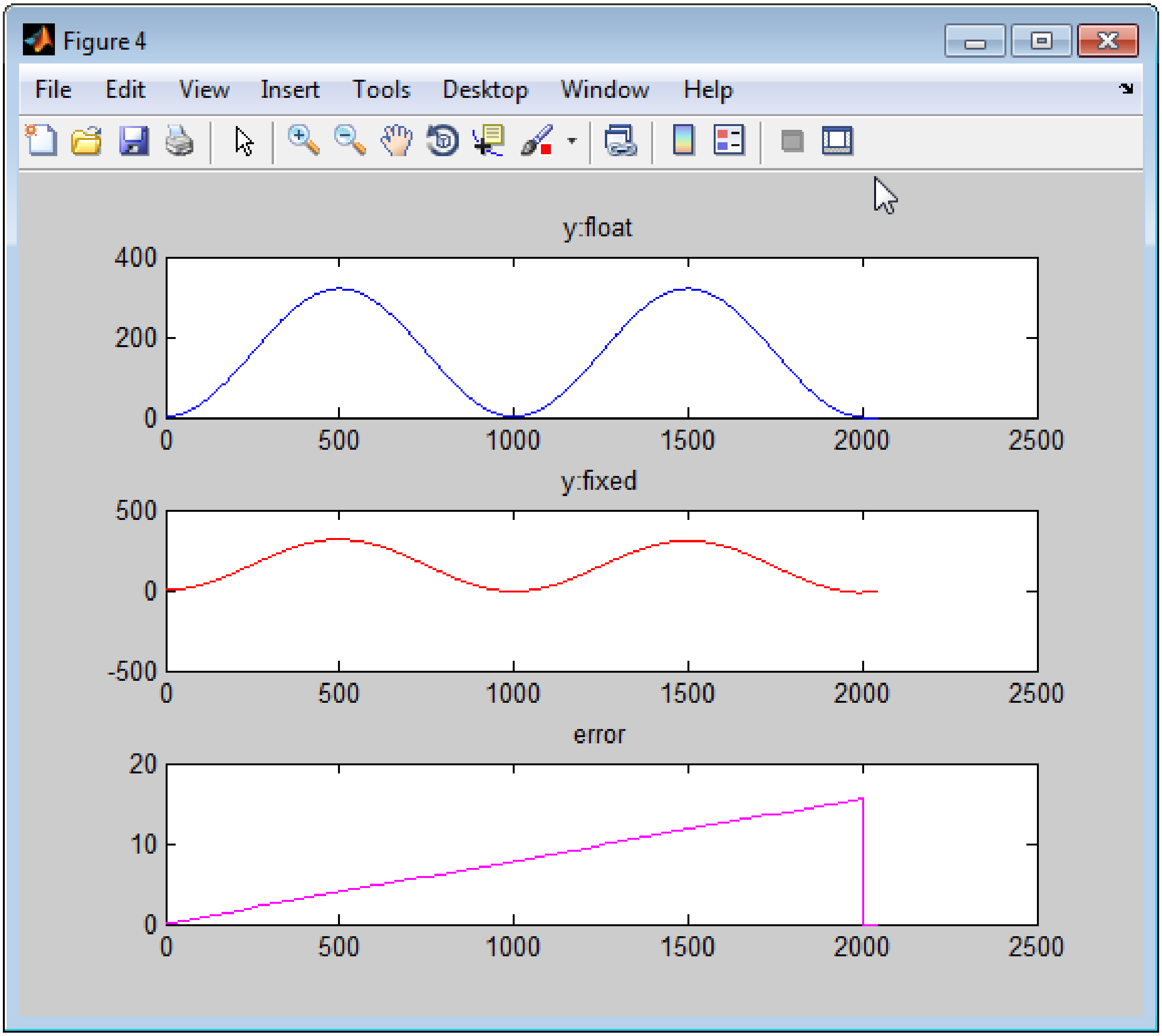
Function: dti

● Propose fraction lengths
○ Propose word lengths
Default word length: 16

The tool 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 variables `y` and `clip_status`. Because you selected to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output.

Plots are displayed for the:

- Floating-point input and output signals.
- Fixed-point input and output signals.
- Outputs `y` and `clip_status` showing the difference between the floating-point and the fixed-point runs.



The maximum difference between the floating-point and fixed-point runs for y is less than 5%. For the purpose of this example, this margin of error is acceptable, so you are ready to generate fixed-point C code.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics”.

Generate Fixed-Point C Code

1. In the MATLAB Coder project, select the **Build** tab.
2. On this tab, set the **Output type** to `C/C++ Static library`.

   The default output file name is `dti`.
3. Click **Build** to generate a library using the default project settings.

   MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/dti_FixPt`.
4. To view the generated code, click **View report**.

   The code generation report opens and displays the generated code for `dti_FixPt.c`. In the generated C code, variables are assigned fixed-point data types.

Command-Line 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.

   By default, the Fixed-Point Conversion tool shows code coverage results. Your test files should 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 verify that your test file is exercising the algorithm adequately. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. For more information, see “Code Coverage” on page 8-3.

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.

Command-Line Fixed-Point Conversion Best Practices

In this section...

"Separate data types from algorithm" on page 8-72

"Create a Test File" on page 8-72

"Entry-Point Function" on page 8-73

Separate data types from algorithm

Separating data type specifications from algorithm code allows 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.

Create a Test File

Create a test file to call your original MATLAB algorithm and fixed-point versions of the algorithm. 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.

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.

By default, the Fixed-Point Conversion tool 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 helps you verify that your test file is exercising the algorithm adequately. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. For more information, see “Code Coverage” on page 8-3.

Entry-Point Function

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.

9

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

- “Code Acceleration and Code Generation from MATLAB” on page 9-3
- “Requirements for Generating Complied C Code Files” on page 9-4
- “Functions Supported for Code Acceleration or C Code Generation” on page 9-5
- “Fixed-Point Code Acceleration and Generation Workflow” on page 9-15
- “Set Up C Compiler” on page 9-16
- “Accelerate Code Using fiaccel” on page 9-17
- “File Infrastructure and Paths Setup” on page 9-22
- “Detect and Debug Code Generation Errors” on page 9-25
- “Set Up C Code Compilation Options” on page 9-28
- “MEX Configuration Dialog Box Options” on page 9-30
- “Specify Primary Function Input Properties” on page 9-36
- “Best Practices for Accelerating Fixed-Point Code” on page 9-47
- “Use Fixed-Point Code Generation Reports” on page 9-51
- “Generate C Code from Code Containing Global Data” on page 9-57
- “Define Input Properties Programmatically in MATLAB File” on page 9-63
- “Control Run-Time Checks” on page 9-71

- “Fix Run-Time Stack Overflows” on page 9-74
- “Code Generation with MATLAB® Coder™” on page 9-75
- “Code Generation with MATLAB Function Block” on page 9-76
- “Generate Fixed-Point FIR Code Using MATLAB Function Block” on page 9-85
- “Fixed-Point FIR Code Example Parameter Values” on page 9-90
- “Accelerate Code for Variable-Size Data” on page 9-93
- “Accelerate Fixed-Point Simulation” on page 9-104
- “Propose Fixed-Point Data Types Using an Instrumented Mex Function” on page 9-107
- “Apply Fixed-Point Data Types Using an Instrumented Mex Function” on page 9-117
- “Code Generation Readiness Tool” on page 9-123
- “Check Code Using the Code Generation Readiness Tool” on page 9-130
- “Check Code Using the MATLAB Code Analyzer” on page 9-131
- “Fix Errors Detected at Code Generation Time” on page 9-132
- “Avoid Multiword Operations in Generated Code” on page 9-133

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" 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?" in the Simulink documentation

MATLAB code generation supports variable-size arrays and matrices with known upper bounds. To learn more about using variable-size signals, see "What Is Variable-Size Data?" on page 22-2.

Requirements for Generating Complied 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

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

- fipref and quantizer objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the fimath or numerictype of a given fi variable after that variable has been created.
- The boolean value of the DataTypeMode and DataType properties are not supported.
- For all SumMode property settings other than FullPrecision, the CastBeforeSum property must be set to true.
- You can use parallel for (parfor) loops in code compiled with fiaccel, but those loops are treated like regular for loops.
- When you compile code containing fi objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- The general limitations of C/C++ code generated from MATLAB apply. For more information, see “MATLAB Language Features Supported for C/C++ Code Generation”.

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	N/A
all	N/A
any	N/A
atan2	N/A

Function	Remarks/Limitations
`bitand`	Not supported for slope-bias scaled `fi` objects.
`bitandreduce`	N/A
`bitcmp`	N/A
`bitconcat`	N/A
`bitget`	N/A
`bitor`	Not supported for slope-bias scaled `fi` objects.
`bitorreduce`	N/A
`bitreplicate`	N/A
`bitrol`	N/A
`bitror`	N/A
`bitset`	N/A
`bitshift`	N/A
`bitsliceget`	N/A
`bitsll`	Generated code may not handle out of range shifting.
`bitsra`	Generated code may not handle out of range shifting.
`bitsrl`	Generated code may not handle out of range shifting.
`bitxor`	Not supported for slope-bias scaled `fi` objects.
`bitxorreduce`	N/A
`ceil`	N/A
`complex`	N/A
`conj`	N/A

Function	Remarks/Limitations
conv	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. • For variable-sized signals, you may see different results between generated code and MATLAB. - In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordiccexp	Variable-size signals are not supported.
cordiccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
cos	N/A
ctranspose	N/A
diag	If supplied, the index, *k*, must be a real and scalar integer value that is not a fi object.

Function	Remarks/Limitations
divide	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax T.divide(a,b).
double	N/A
end	N/A
eps	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and fi double signals.
eq	Not supported for fixed-point signals with different biases.
fi	• The default constructor syntax without any input arguments is not supported. • If the numerictype is not fully specified, the input to fi must be a constant, a fi, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows fi to autoscale its fraction length based on the known data type of the input. • numerictype object information must be available for nonfixed-point Simulink inputs.
filter	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
fimath	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a fimath object. You define this object in the MATLAB Function block dialog in the Model Explorer. • Use to create fimath objects in the generated code.
fix	N/A
fixed.Quantizer	N/A
floor	N/A

Function	Remarks/Limitations
`ge`	Not supported for fixed-point signals with different biases.
`get`	The syntax `structure = get(o)` is not supported.
`getlsb`	N/A
`getmsb`	N/A
`gt`	Not supported for fixed-point signals with different biases.
`hdlram`	N/A
`horzcat`	N/A
`imag`	
`int8`, `int16`, `int32`, `int64`	You cannot use `int64` in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow® chart.
`iscolumn`	N/A
`isempty`	N/A
`isequal`	N/A
`isfi`	N/A
`isfimath`	N/A
`isfimathlocal`	N/A
`isfinite`	N/A
`isinf`	N/A
`isnan`	N/A
`isnumeric`	N/A
`isnumerictype`	N/A
`isreal`	N/A
`isrow`	N/A
`isscalar`	N/A
`issigned`	N/A
`isvector`	N/A
`le`	Not supported for fixed-point signals with different biases.

Function	Remarks/Limitations
`length`	N/A
`logical`	N/A
`lowerbound`	N/A
`lsb`	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, `fi` single and double signals.
`lt`	Not supported for fixed-point signals with different biases.
`max`	N/A
`mean`	N/A
`median`	N/A
`min`	N/A
`minus`	Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`mpower`	• When the exponent `k` is a variable and the input is a scalar, the `ProductMode` property of the governing `fimath` must be `SpecifyPrecision`. • When the exponent `k` is a variable and the input is not scalar, the `SumMode` property of the governing `fimath` must be `SpecifyPrecision`. • Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `SpecifyPrecision` or `Keep LSB`. • For variable-sized signals, you may see different results between the generated code and MATLAB. - In the generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`. - In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when the first input, *a*, is nonscalar. However, when *a* is a scalar,

Function	Remarks/Limitations
	MATLAB computes the output using the ProductMode of the governing fimath.
mpy	When you provide complex inputs to the mpy function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
mrdivide	N/A
mtimes	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to SpecifyPrecision or KeepLSB. • For variable-sized signals, you may see different results between the generated code and MATLAB. - In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	numberofelements will be removed in a future release. Use numel instead.
numel	N/A

Function	Remarks/Limitations
numerictype	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a numerictype object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a nonfixed-point signal. • Use to create numerictype objects in generated code.
permute	N/A
plus	Any non-fi inputs must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
pow2	N/A
power	When the exponent k is a variable, the ProductMode property of the governing fimath must be SpecifyPrecision.
quantize	N/A
range	N/A
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterpretcast	N/A
removefimath	N/A
repmat	N/A
rescale	N/A
reshape	N/A
round	N/A
setfimath	N/A
sfi	N/A
sign	N/A
sin	N/A

Function	Remarks/Limitations
single	N/A
size	N/A
sort	N/A
sqrt	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
times	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, *k*, must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, *k*, must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32, uint64	You cannot use uint64 in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow chart.

Function	Remarks/Limitations
`uminus`	N/A
`uplus`	N/A
`upperbound`	N/A
`vertcat`	N/A

Fixed-Point Code Acceleration and Generation Workflow

Step	Action	Details
1	Set up your C compiler.	See “Set Up C Compiler” on page 9-16.
2	Set up your file infrastructure.	See “File Infrastructure and Paths Setup” on page 9-22.
3	Make your MATLAB algorithm suitable for code generation	See “Best Practices for Accelerating Fixed-Point Code” on page 9-47.
4	Set compilation options.	See “Set Up C Code Compilation Options” on page 9-28.
5	Specify properties of primary function inputs.	See “Specify Primary Function Input Properties” on page 9-36.
6	Run `fiaccel` with the appropriate command-line options.	See “Recommended Compilation Options for fiaccel” on page 9-47.

Set Up C Compiler

Set up your C compiler to generate compiled C code functions by running `mex -setup`. For more information, see `mex`. You can also use `mex` to choose and configure a different C compiler as described in "What You Need to Build MEX-Files".

For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

Accelerate Code Using fiaccel

In this section...

"Speeding Up Fixed-Point Execution with fiaccel" on page 9-17

"Running fiaccel" on page 9-17

"Generated Files and Locations" on page 9-18

"Data Type Override Using fiaccel" on page 9-21

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 9-22.
- 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 9-50.
- 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 9-18.
- 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: fiaccel/mex/*M_fcn_name*/html

You can change the name and location of generated files by using the options -o and -d when you run fiaccel.

In this example, you will use the fiaccel function to compile different parts of a simple algorithm. By comparing the run times of the two cases, you will see the benefits and best use of the fiaccel function.

Example: Comparing Run Times When Accelerating Different Algorithm Parts

The algorithm used throughout this example replicates the functionality of the MATLAB sum function, which sums the columns of a matrix. To see the algorithm, type open fi_matrix_column_sum.m at the MATLAB command line.

```
function B = fi_matrix_column_sum(A)
% Sum the columns of matrix A.
%#codegen
    [m,n] = size(A);
    w = get(A,'WordLength') + ceil(log2(m));
    f = get(A,'FractionLength');
    B = fi(zeros(1,n),true,w,f);
    for j = 1:n
        for i = 1:m
            B(j) = B(j) + A(i,j);
        end
    end
```

Trial 1: Best Performance

The best way to speed up the execution of the algorithm is to compile the entire algorithm using the fiaccel function. To evaluate the performance improvement provided by the fiaccel function when the entire algorithm is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the entire algorithm using the fiaccel function. The MATLAB tic and toc functions keep track of the run times for each method of execution.

```
% MATLAB
fipref('NumericTypeDisplay','short');
A = fi(randn(1000,10));
tic
B = fi_matrix_column_sum(A)
t_matrix_column_sum_m = toc

% fiaccel
fiaccel fi_matrix_column_sum -args {A} ...
-I [matlabroot '/toolbox/fixedpoint/fidemos']
tic
B = fi_matrix_column_sum_mex(A);
t_matrix_column_sum_mex = toc
```

Trial 2: Worst Performance

Compiling only the smallest unit of computation using the fiaccel function leads to much slower execution. In some cases, the overhead that results from calling the mex function inside a nested loop can cause even slower execution than using MATLAB functions alone. To evaluate the performance of the mex function when only the smallest unit of computation is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the smallest unit of computation with the fiaccel function, leaving the rest of the computations to MATLAB.

```
% MATLAB
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_m = toc

% fiaccel
fiaccel fi_scalar_sum -args {B(1),A(1,1)} ...
-I [matlabroot '/toolbox/fixedpoint/fidemos']
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum_mex(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_mex = toc
```

Ratio of Times

A comparison of Trial 1 and Trial 2 appears in the following table. Your computer may record different times than the ones the table shows, but the ratios should be approximately the same. There is an extreme difference in ratios between the trial where the entire algorithm was compiled using `fiaccel` (`t_matrix_column_sum_mex.m`) and where only the scalar sum was compiled (`t_scalar_sum_mex.m`). Even the file with no `fiaccel` compilation (`t_matrix_column_sum_m`) did better than when only the smallest unit of computation was compiled using `fiaccel` (`t_scalar_sum_mex`).

X (Overall Performance Rank)	Time	X/Best	X_m/X_mex
Trial 1: Best Performance			
t_matrix_column_sum_m (2)	1.99759	84.4917	84.4917
t_matrix_column_sum_mex (1)	0.0236424	1	
Trial 2: Worst Performance			
t_scalar_sum_m (4)	10.2067	431.71	2.08017
t_scalar_sum_mex (3)	4.90664	207.536	

Data Type Override Using fiaccel

Fixed-Point 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(fipref)` command in that section:

```
fipref('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 `fiaccel` command in the “Compile the M-File into a MEX File” section of the example, the resulting MEX-function uses floating-point data.

File Infrastructure and Paths Setup

In this section...

"Compile Path Search Order" on page 9-22

"Naming Conventions" on page 9-22

Compile Path Search Order

`fiaccel` resolves function calls by searching first on the code generation path and then on the MATLAB path. By default, `fiaccel` tries to compile and generate code for functions it finds on the path unless you explicitly declare the function to be extrinsic. An *extrinsic function* is a function on the MATLAB path that is dispatched to MATLAB software for execution. `fiaccel` does not compile extrinsic functions, but rather dispatches them to MATLAB for execution.

Naming Conventions

MATLAB enforces naming conventions for functions and generated files.

- "Reserved Prefixes" on page 9-22
- "Reserved Keywords" on page 9-22
- "Conventions for Naming Generated files" on page 9-24

Reserved Prefixes

MATLAB reserves the prefix `eml` for global C functions and variables in generated code. For example, run-time library function names all begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C functions or primary MATLAB functions with the prefix `eml`.

Reserved Keywords

- "C Reserved Keywords" on page 9-23
- "C++ Reserved Keywords" on page 9-23
- "Reserved Keywords for Code Generation" on page 9-24

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains any reserved keywords, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the "C++ Reserved Keywords" on page 9-23.

C Reserved Keywords.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C++ Reserved Keywords.

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t
export	private	throw	

Reserved Keywords for Code Generation.

`abs`	`fortran`	`localZCE`	`rtNaN`
`asm`	`HAVESTDIO`	`localZCSV`	`SeedFileBuffer`
`bool`	`id_t`	`matrix`	`SeedFileBufferLen`
`boolean_T`	`int_T`	`MODEL`	`single`
`byte_T`	`int8_T`	`MT`	`TID01EQ`
`char_T`	`int16_T`	`NCSTATES`	`time_T`
`cint8_T`	`int32_T`	`NULL`	`true`
`cint16_T`	`int64_T`	`NUMST`	`TRUE`
`cint32_T`	`INTEGER_CODE`	`pointer_T`	`uint_T`
`creal_T`	`LINK_DATA_BUFFER_SIZE`	`PROFILING_ENABLED`	`uint8_T`
`creal32_T`	`LINK_DATA_STREAM`	`PROFILING_NUM_SAMPLES`	`uint16_T`
`creal64_T`	`localB`	`real_T`	`uint32_T`
`cuint8_T`	`localC`	`real32_T`	`uint64_T`
`cuint16_T`	`localDWork`	`real64_T`	`UNUSED_PARAMETER`
`cuint32_T`	`localP`	`RT`	`USE_RTMODEL`
`ERT`	`localX`	`RT_MALLOC`	`VCAST_FLUSH_DATA`
`false`	`localXdis`	`rtInf`	`vector`
`FALSE`	`localXdot`	`rtMinusInf`	

Conventions for Naming Generated files

MATLAB provides platform-specific extensions for MEX files.

Platform	MEX File Extension
Linux® x86-64	`.mexa64`
Windows® (32-bit)	`.mexw32`
Windows x64	`.mexw64`

Detect and Debug Code Generation Errors

In this section...

"Debugging Strategies" on page 9-25

"Error Detection at Design Time" on page 9-26

"Error Detection at Compile Time" on page 9-26

Debugging Strategies

To prepare your algorithms for code generation, MathWorks recommends that you choose a debugging strategy for detecting and correcting violations in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. Here are two best practices:

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	**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.	• Efficient • Safe • Easy to isolate syntax violations	Requires application tests that work from the bottom up

Debugging Strategy	What to Do	Pros	Cons
Top-down verification	1 Declare all functions called by the top-level function to be extrinsic so `fiaccel` does not compile them. 2 Verify that your top-level function is suitable for code generation. 3 Work downward in the function hierarchy to: a. Remove extrinsic declarations one by one b. Compile and verify each function, ending with the leaf functions.	Lets you retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values.

Error Detection at Design Time

To detect potential issues for MEX file building as you write your MATLAB algorithm, add the `%#codegen` directive to the code that you want `fiaccel` 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 9-25.

`fiaccel` checks for all potential syntax violations at compile time. When `fiaccel` detects errors or warnings, it automatically produces a code generation report that describes the issues and provides links to the offending code. See "Use Fixed-Point Code Generation Reports" on page 9-51.

If your MATLAB code calls functions on the MATLAB path, `fiaccel` attempts to compile these functions unless you declare them to be extrinsic.

Set Up C Code Compilation Options

In this section...

"C Code Compiler Configuration Object" on page 9-28

"Compilation Options Modification at the Command Line Using Dot Notation" on page 9-28

"How fiaccel Resolves Conflicting Options" on page 9-29

C Code Compiler Configuration Object

For C code generation to a MEX file, MATLAB provides a configuration object `coder.MEXConfig` for fine-tuning the compilation. To set MEX compilation options:

1 Define the compiler configuration object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.mexconfig
```

MATLAB displays the list of compiler options and their current values in the command window.

2 Modify the compilation options as necessary. See "Compilation Options Modification at the Command Line Using Dot Notation" on page 9-28

3 Invoke `fiaccel` with the `-config` option and specify the configuration object as its argument:

```
fiaccel -config comp_cfg myMfile
```

The `-config` option instructs `fiaccel` to convert `myFile.m` to a MEX function, based on the compilation settings in `comp_cfg`.

Compilation Options Modification at the Command Line Using Dot Notation

Use dot notation to modify the value of compilation options, using this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties. For example, to change the maximum size function to inline and the stack size limit for inlined functions during MEX generation, enter this code at the command line:

```
co_cfg = coder.mexconfig
co_cfg.InlineThreshold = 25;
co_cfg.InlineStackLimit = 4096;
fiaccel -config co_cfg myFun
```

How fiaccel Resolves Conflicting Options

`fiaccel` takes the union of all options, including those specified using configuration objects, so that you can specify options in any order.

MEX Configuration Dialog Box Options

MEX Configuration Dialog Box Options

The following table describes parameters for fine-tuning the behavior of `fiaccel` for converting MATLAB files to MEX:

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Report		

Create code generation report	`GenerateReport`
	`true`, **`false`**
		Document generated code in an HTML report.

Launch report automatically	`LaunchReport`
	`true`, **`false`**
		Specify whether to automatically display HTML reports after code generation completes.
		Note Requires that you enable **Create code generation report**

Debugging		

Echo expressions without semicolons	`EchoExpressions`
	`true`, `false`
		Specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window.

Enable debug build	`EnableDebugging`
	`true`, **`false`**
		Compile the generated code in debug mode.

Language and Semantics		

Constant Folding Timeout	`ConstantFoldingTimeout`
	`integer`, **`10000`**
		Specify the maximum number of instructions to be executed by the constant folder.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description

Dynamic memory allocation	`DynamicMemoryAllocation`
	`'off'`, `'AllVariableSizeArrays'`
		Enable dynamic memory allocation for variable-size data. By default, dynamic memory allocation is disabled and `fiaccel` allocates memory statically on the stack. When you select dynamic memory allocation, `fiaccel` allocates memory for all variable-size data dynamically on the heap.
		You *must* use dynamic memory allocation for all unbounded variable-size data.

Enable variable sizing	`EnableVariableSizing`
	`true`, `false`
		Enable support for variable-size arrays.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description

Extrinsic calls	`ExtrinsicCalls`
	`true`, `false`
		Allow calls to extrinsic functions.
		When enabled (`true`), the compiler generates code for the call to a MATLAB function, but does not generate the function's internal code.
		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.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description

Global Data Synchronization Mode	`GlobalDataSyncMethod`
	`string`,**`SyncAlways`**,
	`SyncAtEntryAndExits`,
	`NoSync`
		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 `coder.extrinsic -sync:off` option to turn off synchronization for these calls.
		`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 `coder.extrinsic -sync:on` option to turn on synchronization for these calls.
		`NoSync` disables synchronization. Ensure that your generated code does not interact with MATLAB before disabling synchronization. Otherwise, inconsistencies might occur.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description

Saturate on integer overflow	`SaturateOnIntegerOverflow`
	`true`, `false`
		Add checks in the generated code to detect integer overflow or underflow.

Safety (disable for faster MEX)		

Ensure memory integrity	`IntegrityChecks`
	`true`, `false`
		Detects violations of memory integrity in code generated from MATLAB algorithms and stops execution with a diagnostic message. Setting `IntegrityChecks` to `false` also disables the run-time stack.

Ensure responsiveness	`ResponsivenessChecks`
	`true`, `false`
		Enables responsiveness checks in code generated from MATLAB algorithms.

Function Inlining and Stack Allocation		

Inline Stack Limit	`InlineStackLimit`
	`integer`, **`4000`**
		Specify the stack size limit on inlined functions.

Inline Threshold	`InlineThreshold`
	`integer`, **`10`**
		Specify the maximum size of functions to be inlined.

Inline Threshold Max	`InlineThresholdMax`
	`integer`, **`200`**
		Specify the maximum size of functions after inlining.

Stack Usage Max	`StackUsageMax`
	`integer`, **`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 `fiaccel`.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Optimizations		

Use BLAS library if possible	`EnableBLAS`
	`true`, `false`
		Speed up low-level matrix operations during simulation by calling the Basic Linear Algebra Subprograms (BLAS) library.

See Also

- "Control Run-Time Checks" on page 9-71
- "Variable-Size Data Definition for Code Generation" on page 22-3
- "Generate C Code from Code Containing Global Data" on page 9-57

Specify Primary Function Input Properties

In this section...

"Why You Must Specify Input Properties" on page 9-36

"Properties to Specify" on page 9-36

"Rules for Specifying Properties of Primary Inputs" on page 9-39

"Methods for Defining Properties of Primary Inputs" on page 9-39

"Input Properties Definition by Example at the Command Line" on page 9-40

Why You Must Specify Input Properties

To generate code in a statically typed language, `fiaccel` must determine the properties of all variables in the MATLAB code at compile time. Therefore, you must specify the class, size, and complexity of inputs to the primary function (also known as the *top-level* or *entry-point* function). If your primary function has no input parameters, `fiaccel` can compile your MATLAB algorithm without modification. You do not need to specify properties of inputs to local or external functions called by the primary function. For `fiaccel` requirements, refer to its reference page.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input:

For:	Specify Properties:				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Structure inputs*	✓	✓	✓	✓	✓ (if structure

For:	Specify Properties:				
	Class	Size	Complexity	numerictype	fimath
				(if structure field is fixed-point)	field is fixed-point)
All other inputs	✓	✓	✓		

* When a primary input is a structure, `fiaccel` treats each field as a separate input.

Default Property Values

`fiaccel` assigns the following default values for properties of primary function inputs:

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default
`fimath`	MATLAB default `fimath` object

Specifying Default Values for Structure Fields. In most cases, `fiaccel` uses defaults when you don't explicitly specify values for properties—except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you may need to specify default values for properties of structure fields. For examples, see "Example: Specifying Class and Size of Scalar Structure" on page 9-69 and "Example: Specifying Class and Size of Structure Array" on page 9-70.

Specifying Default fimath Values for MEX Functions. MEX functions generated with `fiaccel` use the MATLAB default `fimath`. The MATLAB factory default `fimath` has the following properties:

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

For more information, see "fimath Object Construction" on page 4-2.

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time error, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` will rely on the default `fimath` object at compile time. At the MATLAB prompt, generate the MEX function `text_mex` to use the factory setting of the MATLAB default `fimath`:

```
fiaccel test
% fiaccel generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex

ans =

     0

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

Supported Classes

The following table presents the class names supported by `fiaccel`:

Class Name	Description
`logical`	Logical array of true and false values
`char`	Character array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`single`	Single-precision floating-point or fixed-point number array
`double`	Double-precision floating-point or fixed-point number array
`struct`	Structure array
`embedded.fi`	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

Follow these rules when specifying the properties of primary inputs:

- For each primary function input whose class is fixed point (`fi`), you must specify the input's `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

You can use any of the following methods to define the properties of primary function inputs:

Method	Pros	Cons
"Input Properties Definition by Example at the Command Line" on page 9-40	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs	• Must be specified at the command line every time you invoke `fiaccel` (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties Programmatically in MATLAB File" on page 9-63	• Integrated with MATLAB code so you do not need to redefine properties each time you invoke `fiaccel` • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures	• Uses complex syntax

Note To specify the properties of inputs for any given primary function, use one of these methods or the other, but not both.

Input Properties Definition by Example at the Command Line

- "Command Line Option -args" on page 9-41
- "Rules for using the -args option" on page 9-42
- "Specifying Constant Inputs" on page 9-42
- "Specifying Variable-Size Inputs" on page 9-44

Command Line Option -args

fiaccel provides a command-line option -args for specifying the properties of primary function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values.

Example: Specifying Properties of Primary Inputs by Example. Consider a function that adds its two inputs:

```
function y = emcf(u,v) %#codegen
% The directive %#codegen indicates that you
% intend to generate code for this algorithm
y = u + v;
```

The following examples show how to specify different properties of the primary inputs u and v by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real, scalar, fixed-point values:

  ```
  fiaccel -o emcfx emcf ...
     -args {fi(0,1,16,15),fi(0,1,16,15)}
  ```

- Use a literal cell array of constants to specify that input u is an unsigned 16-bit, 1-by-4 vector and input v is a scalar, fixed-point value:

  ```
  fiaccel -o emcfx emcf ...
     -args {zeros(1,4,'uint16'),fi(0,1,16,15)}
  ```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

  ```
  a = fi([1;2;3;4],0,8,0)
  b = fi([5;6;7;8],0,8,0)
  ex = {a,b}
  fiaccel -o emcfx emcf -args ex
  ```

Example: Specifying Properties of Primary Fixed-Point Inputs by Example. Consider a function that calculates the square root of a fixed-point number:

```
function y = sqrtfi(x) %#codegen
y = sqrt(x);
```

To specify the properties of the primary fixed-point input `x` by example on the MATLAB command line, follow these steps:

1. Define the `numerictype` properties for `x`, as in this example:

   ```
   T = numerictype('WordLength',32,...
      'FractionLength',23,'Signed',true);
   ```

2. Define the `fimath` properties for `x`, as in this example:

   ```
   F = fimath('SumMode','SpecifyPrecision',...
       'SumWordLength',32,'SumFractionLength',23,...
       'ProductMode','SpecifyPrecision', ...
        ProductWordLength',32,'ProductFractionLength',23);
   ```

3. Create a fixed-point variable with the `numerictype` and `fimath` properties you just defined, as in this example:

   ```
   myeg = { fi(4.0,T,F) };
   ```

4. Compile the function `sqrtfi` using the `fiaccel` command, passing the variable `myeg` as the argument to the`-args` option, as in this example:

   ```
   fiaccel sqrtfi -args myeg;
   ```

Rules for using the -args option

Follow these rules when using the `-args` command-line option to define properties by example:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Specifying Constant Inputs

In cases where you know your primary inputs will not change at run time, you can specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using this command-line option:

```
-args {coder.Constant(constant_input)}
```

This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs. `fiaccel` compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function `identity` which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, type the following command at the MATLAB prompt:

```
fiaccel -o identity_mex identity...
    -args {coder.Constant(fi(0.1,1,16,15))}
```

To run the MATLAB function, supply the constant argument as follows:

```
identity(fi(0.1,1,16,15))
```

You get the following result:

```
ans =

    0.1000
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Example: Specifying a Structure as a Constant Input. Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix, as follows:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = fi(zeros(p.rows,p.cols),1,16,15) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
fiaccel rowcol ...
   -args {fi(0,1,16,15),coder.Constant(tmp)}
```

To run this code, use

```
u = fi(0.5,1,16,15)
y_m = rowcol(u,tmp)

y_mex = rowcol_mex(u)
```

Specifying Variable-Size Inputs

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation.

- *Bounded variable-size data* has fixed upper bounds; this data can be allocated statically on the stack or dynamically on the heap.
- *Unbounded variable-size data* does not have fixed upper bounds; this data must be allocated on the heap.

You can define inputs to have one or more variable-size dimensions and specify their upper bounds using the `-args` option:

Expression	Description
`-args {coder.typeof(` *`example_value`*`,` *`size_vector`*`,` *`dim_vector`*`)}`	Specifies a variable-size input with: • Same class and complexity as *`example_value`* • Same size and upper bounds as *`size_vector`* *`dim_vector`* specifies which dimensions are variable. A value of `true` or one means that the corresponding dimension is variable. A value of `false` or zero means that the corresponding dimension is fixed.

Example: Specifying a Variable-Size Vector Input.

1 Write a function that computes the sum of every `n` elements of a vector `A` and stores them in a vector `B`:

```
function B = nway(A,n) %#codegen
% Compute sum of every N elements of A and put them in B.

coder.extrinsic('error');
Tb = numerictype(1,32,24);
if ((mod(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
```

Best Practices for Accelerating Fixed-Point Code

In this section...

"Recommended Compilation Options for fiaccel" on page 9-47

"Build Scripts" on page 9-48

"Check Code Interactively Using MATLAB Code Analyzer" on page 9-49

"Separating Your Test Bench from Your Function Code" on page 9-50

"Preserving Your Code" on page 9-50

"File Naming Conventions" on page 9-50

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".

  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 9-63.

- `-report` – Generate code generation report

  Use the `-report` option to generate a report in HTML format at code generation time to help you debug your MATLAB code and verify that it is suitable for code generation. If you do not specify the `-report` option, `fiaccel` generates a report only if build errors or warnings occur.

The code generation report contains the following information:

- Summary of code generation results, including type of target and number of warnings or errors
- Target build log that records build and linking activities
- Links to generated files
- Error and warning messages (if any)

For more information, see `fiaccel`.

Build Scripts

Use build scripts to call `fiaccel` to generate MEX functions from your MATLAB function.

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors. For instance, you can use a build script to clear your workspace before each build and to specify code generation options.

This example shows a build script to run `fiaccel` to process `lms_02.m`:

```
close all;
clear all;
clc;

N = 73113;

fiaccel  -report lms_02.m ...
  -args { zeros(N,1) zeros(N,1) }
```

In this example, the following actions occur:

- `close all` deletes all figures whose handles are not hidden. See `close` in the MATLAB Graphics function reference for more information.

- `clear all` removes all variables, functions, and MEX-files from memory, leaving the workspace empty. This command also clears all breakpoints.

  Note Remove the `clear all` command from the build scripts if you want to preserve breakpoints for debugging.

- `clc` clears all input and output from the Command Window display, giving you a "clean screen."
- `N = 73113` sets the value of the variable `N`, which represents the number of samples in each of the two input parameters for the function `lms_02`
- `fiaccel -report lms_02.m -args { zeros(N,1) zeros(N,1) }` calls `fiaccel` to accelerate simulation of the file `lms_02.m` using the following options:
  - `-report` generates a code generation report
  - `-args { zeros(N,1) zeros(N,1) }` specifies the properties of the function inputs as a cell array of example values. In this case, the input parameters are N-by-1 vectors of real doubles.

Check Code Interactively Using MATLAB Code Analyzer

The code analyzer checks your code for problems and recommends modifications to maximize performance and maintainability. You can use the code analyzer to check your code continuously in the MATLAB Editor while you work.

To ensure that continuous code checking is enabled:

1. From the MATLAB menu, select **File > Preferences > Code Analyzer**.

   The list of code analyzer preferences appears.

2. Select the **Enable integrated warning and error messages** check box.

Separating Your Test Bench from Your Function Code

Separate your core algorithm from your test bench. Create a separate test script to do all the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results. See the example on the `fiaccel` reference page.

Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention, as described in "File Naming Conventions" on page 9-50. 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 9-51

"Code Generation Report Opening" on page 9-52

"Viewing Your MATLAB Code" on page 9-52

"Viewing Variables in the Variables Tab" on page 9-55

"See Also" on page 9-56

Code Generation Report Creation

When you compile your code with the `fiaccel` function or the MATLAB Coder `codegen` function, you can use the `-report` option to generate a code generation report. This report allows you to examine the data types of the variables and expressions in your code.

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
```

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".

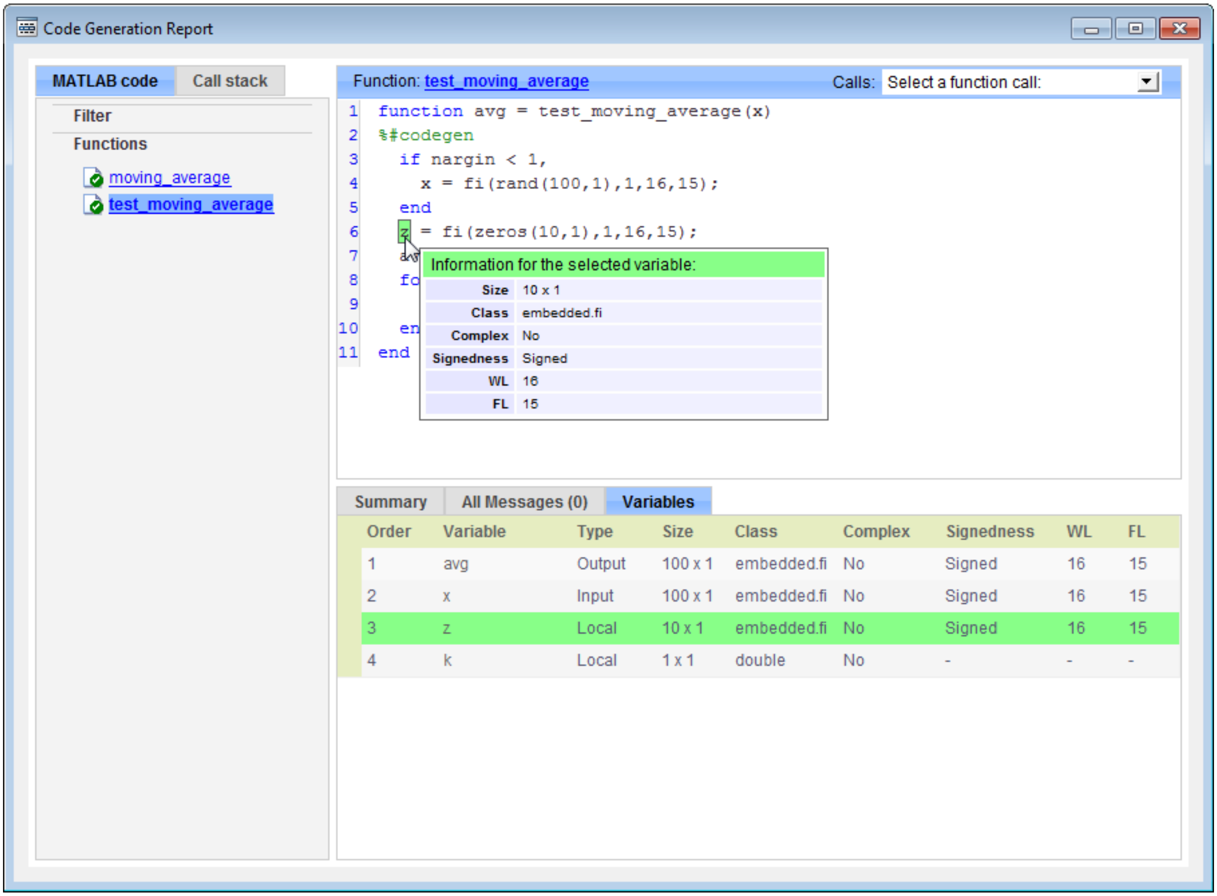
Viewing Your MATLAB Code

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.

Color	Meaning
Pink	No data type information is available for the selected variable.
Purple	Data type information is available for the selected expression at this location in the code.
Red	There is an error message associated with the selected variable or expression.

Variables in your code that have data type information available appear highlighted in green.



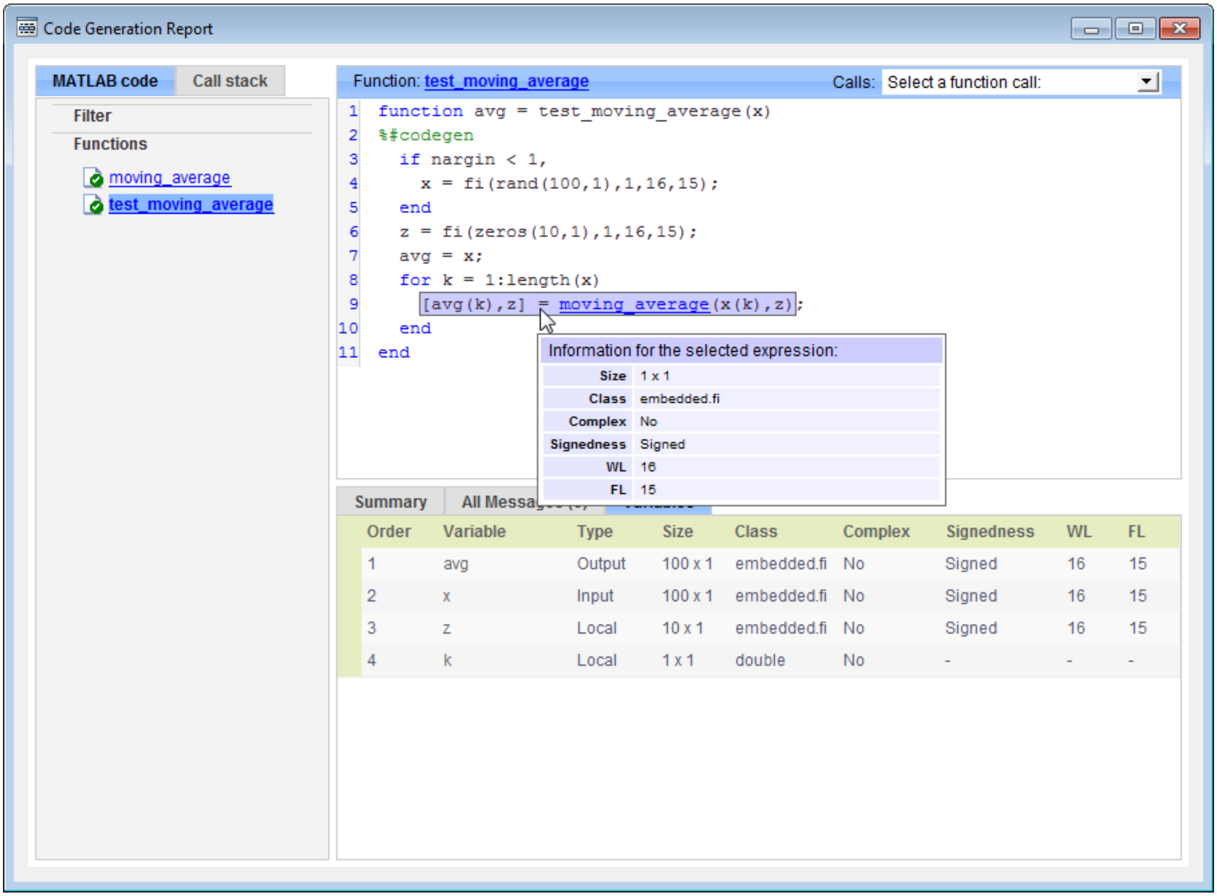
```
function avg = test_moving_average(x)
%#codegen
  if nargin < 1,
    x = fi(rand(100,1),1,16,15);
  end
  z = fi(zeros(10,1),1,16,15);
  a
  fo

  en
end
```

Information for the selected variable:

Size	10 x 1
Class	embedded.fi
Complex	No
Signedness	Signed
WL	16
FL	15

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.



```
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
```

Information for the selected expression:

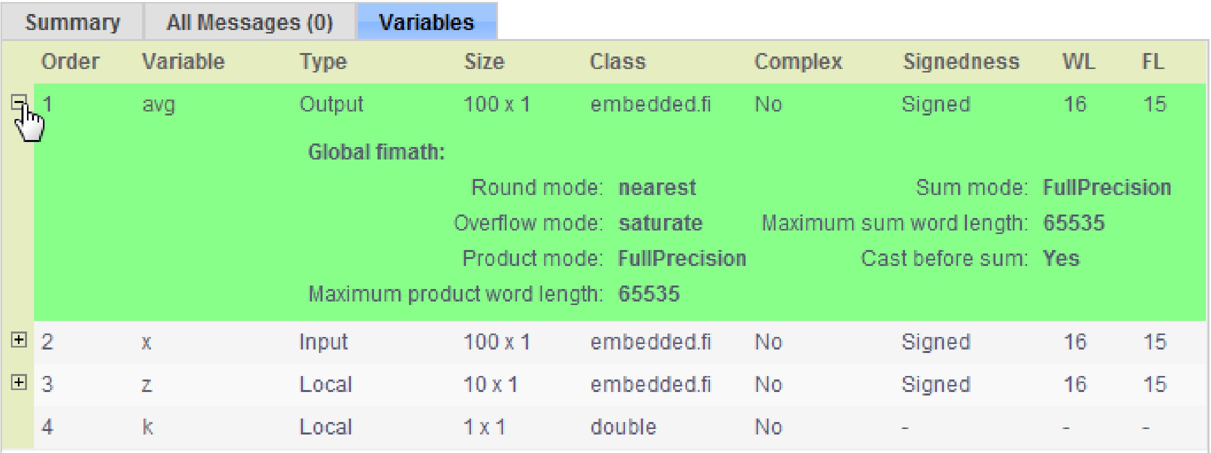
Size	1 x 1
Class	embedded.fi
Complex	No
Signedness	Signed
WL	16
FL	15

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.



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	-	-	-

Global fimath:

Round mode: **nearest**
Overflow mode: **saturate**
Product mode: **FullPrecision**
Maximum product word length: **65535**
Sum mode: **FullPrecision**
Maximum sum word length: **65535**
Cast before sum: **Yes**

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”.

Generate C Code from Code Containing Global Data

In this section...

"Workflow Overview" on page 9-57

"Declaring Global Variables" on page 9-57

"Defining Global Data" on page 9-58

"Synchronizing Global Data with MATLAB" on page 9-59

"Limitations of Using Global Data" on page 9-62

Workflow Overview

To generate MEX functions from MATLAB code that uses global data:

1. Declare the variables as global in your code.

2. Define and initialize the global data before using it.

   For more information, see "Defining Global Data" on page 9-58.

3. Compile your code using `fiaccel`.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated code. If there is no interaction between MATLAB and the generated code, it is safe to disable synchronization. Otherwise, you should enable synchronization. For more information, see "Synchronizing Global Data with MATLAB" on page 9-59.

Declaring Global Variables

For code generation, you must declare global variables before using them in your MATLAB code. Consider the `use_globals` function that uses two global variables `AR` and `B`.

```
function y = use_globals()
%#codegen
% Turn off inlining to make
% generated code easier to read
coder.inline('never');
```

```
% Declare AR and B as global variables
global AR;
global B;
AR(1) = B(1);
y = AR * 2;
```

Defining Global Data

You can define global data either in the MATLAB global workspace or at the command line. If you do not initialize global data at the command line, `fiaccel` looks for the variable in the MATLAB global workspace. If the variable does not exist, `fiaccel` generates an error.

Defining Global Data in the MATLAB Global Workspace

To compile the `use_globals` function described in "Declaring Global Variables" on page 9-57 using `fiaccel`:

1. Define the global data in the MATLAB workspace. At the MATLAB prompt, enter:

   ```
   global AR B;
   AR = fi(ones(4),1,16,14);
   B = fi([1 2 3],1,16,13);
   ```

2. Compile the function to generate a MEX file named `use_globalsx`.

   ```
   fiaccel -o use_globalsx use_globals
   ```

Defining Global Data at the Command Line

To define global data at the command line, use the `fiaccel -global` option. For example, to compile the `use_globals` function described in "Declaring Global Variables" on page 9-57, specify two global inputs `AR` and `B` at the command line.

```
fiaccel -o use_globalsx ...
   -global {'AR',fi(ones(4)),'B',fi([1 2 3])} use_globals
```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`.

Defining Variable-Sized Global Data. To provide initial values for variable-sized global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
fiaccel foo -globals {'g1',{coder.typeof(0,[2 2],1),[1 1]}}
```

For a detailed explanation of `coder.typeof` syntax, see `coder.typeof`.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The generated code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. The level of interaction determines when to synchronize global data.

When to Synchronize Global Data

By default, synchronization between global data in MATLAB and generated code occurs at MEX function entry and exit and for all *extrinsic* calls, which are calls to MATLAB functions on the MATLAB path that `fiaccel` dispatches to MATLAB for execution. This behavior ensures maximum consistency between generated code and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 9-60.

Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Ensure maximum consistency when all extrinsic calls modify global data.	`At MEX-function entry, exit and extrinsic calls` (default)	Yes. Default behavior.
Ensure maximum consistency when most extrinsic calls modify global data, but a few do not.	`At MEX-function entry, exit and extrinsic calls` (default)	Yes. Use the `coder.extrinsic -sync:off` option to turn off synchronization for the extrinsic calls that do not affect global data.
Ensure maximum consistency when most extrinsic calls do not modify global data, but a few do.	`At MEX-function entry and exit`	Yes. Use the `coder.extrinsic -sync:on` option to synchronize only the calls that modify global data.
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	`At MEX-function entry and exit`	No.
Communicate between generated 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 9-59.

You control the synchronization of global data with extrinsic functions using the coder.extrinsic -sync:on and -sync:off options.

Controlling the Global Data Synchronization Mode from the Command Line.

1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

   ```
   comp_cfg = coder.mexconfig
   ```

2 From the command line, set the GlobalDataSyncMethod property to Always, SyncAtEntryAndExits or NoSync, as applicable. For example:

   ```
   comp_cfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
   ```

3 Use the comp_cfg configuration object when compiling your code by specifying it using the -config compilation option. For example,

   ```
   fiaccel -config comp_cfg myFile
   ```

Controlling Synchronization for Extrinsic Function Calls. You can control whether synchronization between global data in MATLAB and generated code occurs before and after you call an extrinsic function. To do so, use the coder.extrinsic -sync:on and -sync:off options.

By default, global data is:

- Synchronized before and after each extrinsic call if the global data synchronization mode is At MEX-function entry, exit and extrinsic calls. If you are sure that certain extrinsic calls do not affect global data, turn off synchronization for these calls using the -sync:off option. Turning off synchronization improves performance. For example, if functions foo1 and foo2 *do not* affect global data, turn off synchronization for these functions:

  ```
  coder.extrinsic('-sync:off', 'foo1', 'foo2');
  ```

- Not synchronized if the global data synchronization mode is At MEX-function entry and exit. If the code has a few extrinsic calls that affect global data, turn on synchronization for these calls using the

-sync:on option. For example, if functions foo1 and foo2 *do* affect global data, turn on synchronization for these functions:

```
coder.extrinsic('-sync:on', 'foo1', 'foo2');
```

- Not synchronized if the global data synchronization mode is Disabled. When synchronization is disabled, you cannot control the synchronization for specific extrinsic calls. The -sync:on option has no effect.

Limitations of Using Global Data

You cannot use global data with

- The coder.cstructname function. This function does not support global variables.
- The coder.varsize function. Instead, use a coder.typeof object to define variable-sized global data as described in "Defining Variable-Sized Global Data" on page 9-59.

Define Input Properties Programmatically in MATLAB File

In this section...

"How to Use `assert`" on page 9-63

"Rules for Using assert Function" on page 9-67

"Example: Specifying Properties of Primary Fixed-Point Inputs" on page 9-68

"Example: Specifying Class and Size of Scalar Structure" on page 9-69

"Example: Specifying Class and Size of Structure Array" on page 9-70

How to Use `assert`

You can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

Specify Any Class

```
assert ( isa ( param, 'class_name') )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input `U` to a 32-bit signed integer, call:

```
...
assert(isa(U,'embedded.fi'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see "Specify numerictype of Fixed-Point Input" on page 9-66. You can also set its `fimath` properties, see "Specify fimath of Fixed-Point Input" on page 9-67.

If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U,'embedded.fi'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see "Specify numerictype of Fixed-Point Input" on page 9-66. You can also set its `fimath` properties, see "Specify fimath of Fixed-Point Input" on page 9-67.

Specify Structure Class

```
assert ( isstruct ( param ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U,'struct'));
...
```

Note If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify Any Size

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. For example, to set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *param* is real. For example, to specify that input U is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *param* is complex. For example, to specify that input U is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the numerictype properties of fi input parameter *fiparam* to the numerictype object *T*. For example, to specify the numerictype property of fixed-point input U as a signed numerictype object T with 32-bit word length and 30-bit fraction length, use the following code:

```
...
% Define the numerictype object.
```

```
T = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isequal(numerictype(U),T));
...
```

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *`fiparam`* to the `fimath` object *`F`*. For example, to specify the `fimath` property of fixed-point input `U` so that it saturates on integer overflow, use the following code:

```
...
% Define the fimath object.
F = fimath('OverflowAction','Saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

Specify Multiple Properties of Input

```
assert ( function1 ( params ) && function2 ( params ) && function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
...
assert(isa(U,'double') && ~isreal(U) && all(size(U) == [3 3]) && isa(V,'uint16'));
...
```

Rules for Using assert Function

Follow these rules when using the `assert` function to specify the properties of primary function inputs:

- Call `assert` functions at the beginning of the primary function, before any flow-control operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- If you set the class of an input parameter to `fi`:
  - You must also set its `numerictype`, see "Specify numerictype of Fixed-Point Input" on page 9-66.
  - You can also set its `fimath` properties, see "Specify fimath of Fixed-Point Input" on page 9-67.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of each field in the structure in the order in which you define the fields in the structure definition.

Example: Specifying Properties of Primary Fixed-Point Inputs

In the following example, the primary MATLAB function `emcsqrtfi` takes one fixed-point input: `x`. The code specifies the following properties for this input:

Property	Value
class	`fi`
`numerictype`	`numerictype` object `T`, as specified in the primary function
`fimath`	`fimath` object `F`, as specified in the primary function
size	`scalar` (by default)
complexity	`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);
```

Example: Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',fi(4,true,8,0));
```

This code specifies the class and size of `S` and its fields when passed as an input to your MATLAB function:

```
function y = fcn(S)

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% in the order in which you defined them.
T = numerictype('Wordlength', 8,'FractionLength', ...
   0,'signed',true);
assert(isa(S.r,'double'));
assert(isfi(S.i) && isequal(numerictype(S.i),T));

y = S;
```

Note The only way to name a field in a structure is to set at least one of its properties. Therefore in the preceding example, an `assert` function specifies that field `S.r` is of type `double`, even though `double` is the default.

Example: Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined S as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',...
   {fi(4,1,8,0), fi(5,1,8,0)});
```

The following code specifies the class and size of each field of structure input S using the first element of the array:

```
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));
T = numerictype('Wordlength', 8,'FractionLength', ...
   0,'signed',true);

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [1 2]));
assert(isa(S(1).r,'double'));
assert(isfi(S(1).i) && isequal(numerictype(S(1).i),T));

y = S;
```

Note The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S(1).r` is of type `double`, even though `double` is the default.

Control Run-Time Checks

In this section...

“Types of Run-Time Checks” on page 9-71

“When to Disable Run-Time Checks” on page 9-72

“How to Disable Run-Time Checks” on page 9-72

Types of Run-Time Checks

In simulation, the code generated for your MATLAB functions includes the following run-time checks and external function calls.

- Memory integrity checks

  These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

  Caution For safety, these checks are enabled by default. Without memory integrity checks, violations will result in unpredictable behavior.

- Responsiveness checks in code generated for MATLAB functions

  These checks enable periodic checks for Ctrl+C breaks in code generated for MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

  Caution For safety, these checks are enabled by default. Without these checks the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

  Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information about extrinsic functions, see "Declaring MATLAB Functions as Extrinsic Functions" on page 11-12.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower simulation than generating code with the checks disabled. Similarly, extrinsic calls are time consuming and have an adverse effect on performance. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation, with these caveats:

Consider disabling...	Only if...
Memory integrity checks	You are sure that your code is safe and that all array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using **Ctrl+C**.
Extrinsic calls	You are only using extrinsic calls to functions that do not affect application results.

How to Disable Run-Time Checks

To disable run-time checks:

1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.MEXConfig
```

2 From the command line set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties false, as applicable:

```
comp_cfg.IntegrityChecks = false;
comp_cfg.ExtrinsicCalls = false;
comp_cfg.ResponsivenessChecks = false;
```

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 9-76

"MATLAB Function Block with Data Type Override" on page 9-76

"Fixed-Point Data Types with MATLAB Function Block" on page 9-78

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?" in the Simulink documentation
- "Code Generation Workflow" in the MATLAB Coder documentation

MATLAB Function Block with Data Type Override

When you use the MATLAB Function block in a Simulink model that specifies data type override, the block determines the data type override equivalents of the input signal and parameter types. The block then uses these equivalent values to run the simulation. The following table shows how the MATLAB Function block determines the data type override equivalent using

- The data type of the input signal or parameter
- The data type override 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

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Applies To Setting	Override Data Type
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`
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".

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 9-5 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 9-79
- "Using fimath Objects in MATLAB Function Blocks" on page 9-80
- "Sharing Models with Fixed-Point MATLAB Function Blocks" on page 9-82

Note To simulate models using fixed-point data types in Simulink, you must have a Fixed-Point Designer license.

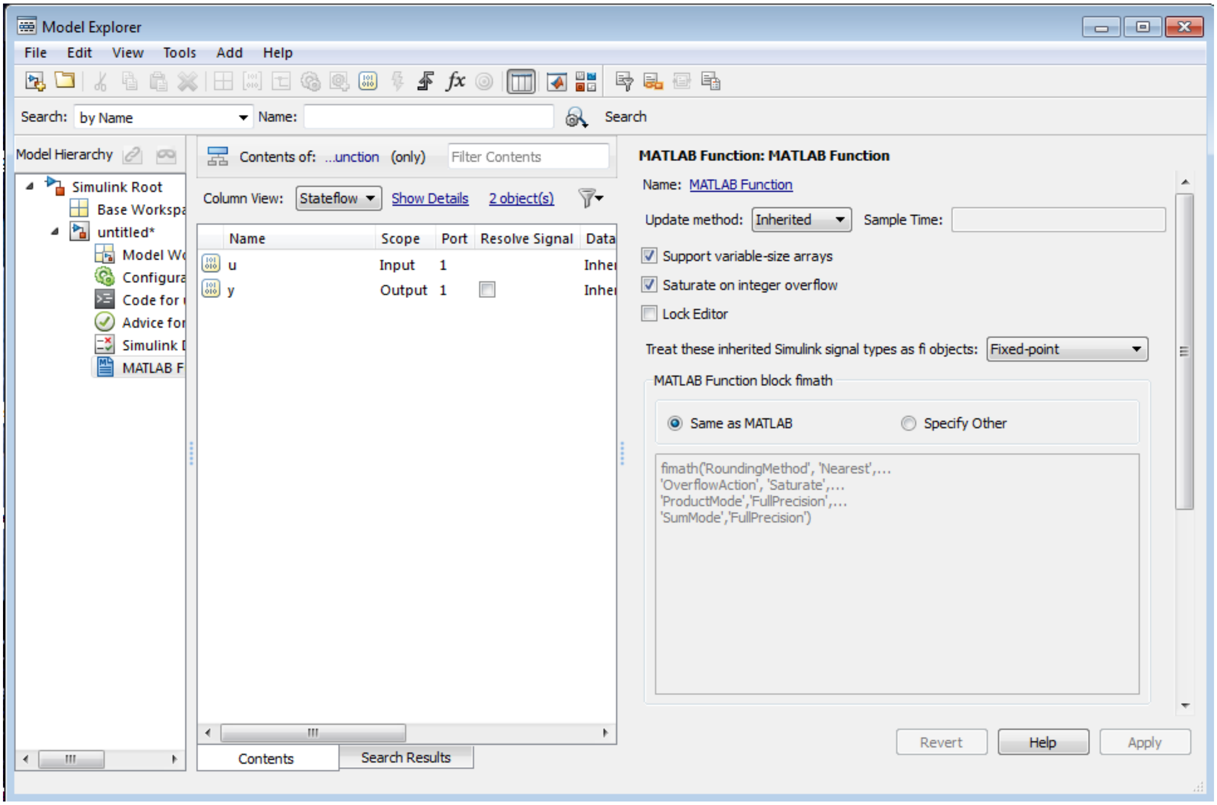
Specifying Fixed-Point Parameters in the Model Explorer

You can specify parameters for an MATLAB Function block in a fixed-point model using the Model Explorer. Try the following exercise:

1 Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.

2 Open the Model Explorer by selecting **View** > **Model Explorer** from your model.

3 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer. Then, select the **MATLAB Function** node. The Model Explorer now appears as shown in the following figure.



Name	Scope	Port	Resolve Signal	Data
u	Input	1		Inhe
y	Output	1	□	Inhe

MATLAB Function: MATLAB Function

Name: MATLAB Function

Update method: Inherited Sample Time:

☑ Support variable-size arrays

☑ Saturate on integer overflow

□ Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

● Same as MATLAB ○ Specify Other

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

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.

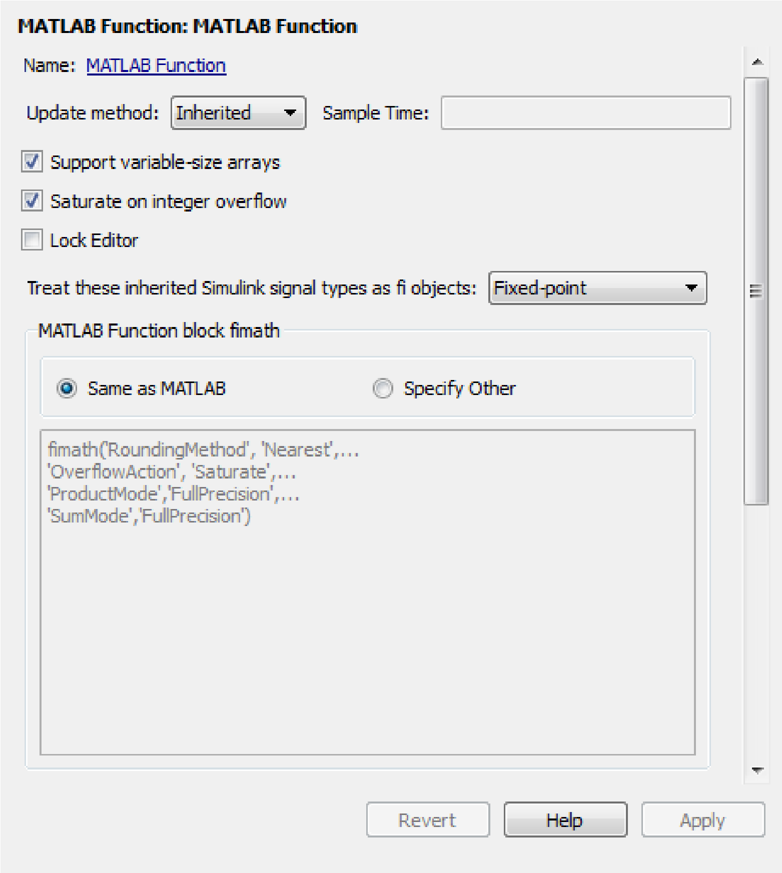
For more information on these parameters, see "Using fimath Objects in MATLAB Function Blocks" on page 9-80.

Using fimath Objects in MATLAB Function Blocks

The **MATLAB Function block fimath** parameter enables you to specify one set of `fimath` object properties for the MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the MATLAB Function block.

You can set these parameters on the following dialog box, which you can access through either the Model Explorer or the "Ports and Data Manager".



MATLAB Function: MATLAB Function

Name: MATLAB Function

Update method: Inherited   Sample Time:

☑ Support variable-size arrays

☑ Saturate on integer overflow

□ Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point

MATLAB Function block fimath

● Same as MATLAB   ○ Specify Other

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

- 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 9-82 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.

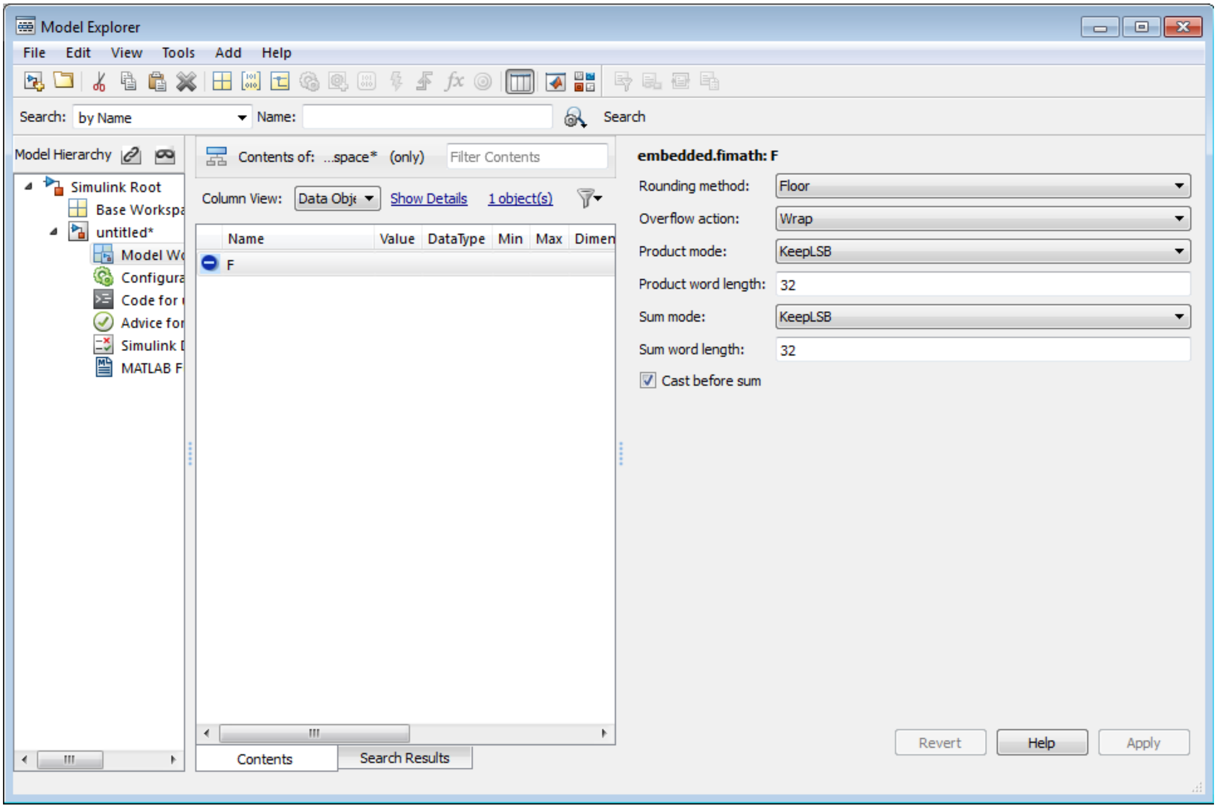
4 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer, and select the **MATLAB Function** node.

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.



embedded.fimath: F

Rounding method: Floor

Overflow action: Wrap

Product mode: KeepLSB

Product word length: 32

Sum mode: KeepLSB

Sum word length: 32

☑ Cast before sum

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 9-85

"Prepare the Inputs" on page 9-86

"Create the Model" on page 9-86

"Define the fimath Object Using the Model Explorer" on page 9-88

"Run the Simulation" on page 9-89

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;
```

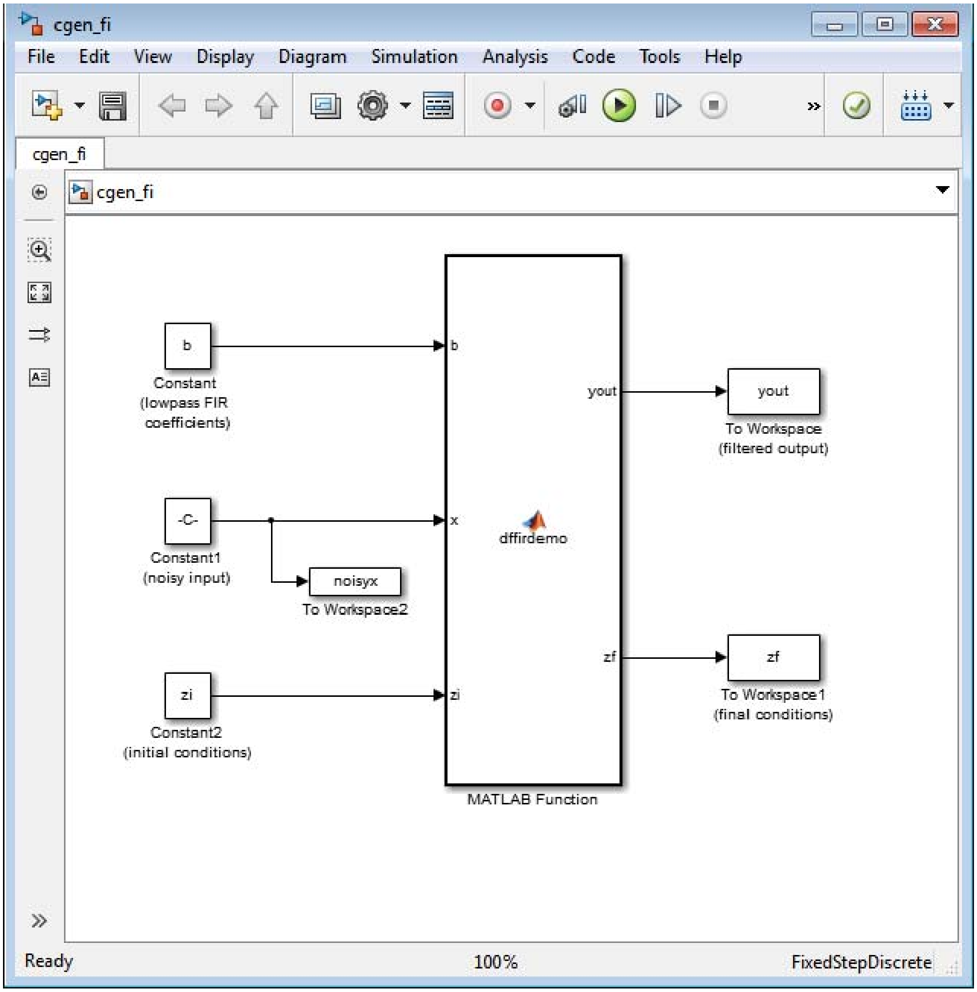
Prepare the Inputs

Define the filter coefficients *b*, noise *x*, and initial conditions *zi* 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.


cgen_fi
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
cgen_fi
cgen_fi
b
Constant
(lowpass FIR
coefficients)
-C-
Constant1
(noisy input)
noisyx
To Workspace2
zi
Constant2
(initial conditions)
b
x
zi
yout
zf
dffirdemo
MATLAB Function
yout
To Workspace
(filtered output)
zf
To Workspace1
(final conditions)
Ready
100%
FixedStepDiscrete

2 Set the block parameters in the model to these "Fixed-Point FIR Code Example Parameter Values" on page 9-90.

3 From the model menu, select **Simulation > Model Configuration Parameters** and set the following parameters.

Parameter	Value
Stop time	`0`
Type	`Fixed-step`
Solver	`discrete (no continuous states)`

Click **Apply** to save your changes.

Define the fimath Object Using the Model Explorer

1 Open the Model Explorer for the model.

2 Click the **cgen_fi** > **MATLAB Function** node in the **Model Hierarchy** pane. The dialog box for the MATLAB Function block appears in the **Dialog** pane of the Model Explorer.

3 Select **Specify other** for the **MATLAB Function block fimath** parameter on the MATLAB Function block dialog box. You can then create the following `fimath` object in the edit box:

```
fimath('RoundingMethod','Floor','OverflowAction','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32)
```

The `fimath` object you define here is associated with fixed-point inputs to the MATLAB Function block as well as the `fi` object you construct within the block.

By selecting **Specify other** for the **MATLAB Function block fimath**, you ensure that your model always uses the `fimath` properties you specified.

Run the Simulation

1. Run the simulation by selecting your model and typing **Ctrl+T**. While the simulation is running, information outputs to the MATLAB command line. You can look at the plots of the noisy signal and the filtered signal.

2. Next, build embeddable C code for your model by selecting the model and typing **Ctrl+B**. While the code is building, information outputs to the MATLAB command line. A folder called `coder_fi_grt_rtw` is created in your current working folder.

3. Navigate to `coder_fi_grt_rtw` > `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

Block	Parameter	Value
Constant2	Constant value	zi
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^-8
	Bias	0
To Workspace	Variable name	yout
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

Block	Parameter	Value
To Workspace1	Variable name	zf
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected
To Workspace2	Variable name	noisyx
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

Accelerate Code for Variable-Size Data

In this section...

"Disable Support for Variable-Size Data" on page 9-93

"Control Dynamic Memory Allocation" on page 9-94

"Accelerate Code for MATLAB Functions with Variable-Size Data" on page 9-95

"Accelerate Code for a MATLAB Function That Expands a Vector in a Loop" on page 9-97

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. By default, for MEX and C/C++ code generation, support for variable-size data is enabled and dynamic memory allocation is enabled for variable-size arrays whose size exceeds a configurable threshold.

Disable Support for Variable-Size Data

By default, for MEX and C/C++ code acceleration, support for variable-size data is enabled. You modify variable sizing settings at the command line.

1. Create a configuration object for code generation.

   ```
   cfg = coder.mexconfig;
   ```

2. Set the `EnableVariableSizing` option:

   ```
   cfg.EnableVariableSizing = false;
   ```

3. Using the `-config` option, pass the configuration object to `fiaccel` :

   ```
   fiaccel -config cfg foo
   ```

Control Dynamic Memory Allocation

By default, dynamic memory allocation is enabled for variable-size arrays whose size exceeds a configurable threshold. If you disable support for variable-size data, you also disable dynamic memory allocation. You can modify dynamic memory allocation settings at the command line.

1 Create a configuration object for code acceleration. For example, for a MEX function:

```
mexcfg = coder.mexconfig;
```

2 Set the `DynamicMemoryAllocation` option:

Setting	Action
`mexcfg.DynamicMemoryAllocation='Off';`	Dynamic memory allocation is disabled. All variable-size data is allocated statically on the stack.
`mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';`	Dynamic memory allocation is enabled for all variable-size arrays. All variable-size data is allocated dynamically on the heap.
`mexcfg.DynamicMemoryAllocation='Threshold';`	Dynamic memory allocation is enabled for all variable-size arrays whose size (in bytes) is greater than or equal to the value specified using the `Dynamic memory allocation threshold` parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

3. Optionally, if you set `Dynamic memory allocation` to `` `Threshold' ``, configure `Dynamic memory allocation threshold` to fine tune memory allocation.

4. Using the `-config` option, pass the configuration object to `fiaccel`:

   ```
   fiaccel -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 22-23.

5 Fix upper bounds errors

Cause:	**How To Fix:**	**For More Information:**
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See "Specifying Upper Bounds for Variable-Size Data" on page 22-6 and "Diagnosing and Fixing Size Mismatch Errors" on page 22-23.
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See "Control Dynamic Memory Allocation" on page 9-94

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 9-97
- "Step 1: Add Compilation Directive for Code Generation" on page 9-97
- "Step 2: Address Issues Detected by the Code Analyzer" on page 9-98
- "Step 3: Generate MEX Code" on page 9-98
- "Step 4: Fix the Size Mismatch Error" on page 9-100
- "Step 5: Compare Execution Speed of MEX Function to Original Code" on page 9-102

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`, `abs(B(i) - B(j))` > `tol` for all `i` and `j`. Initially, assume input vector `A` can store up to 100 elements.

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

Step 1: Add Compilation Directive for Code Generation

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

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
```

```
   if abs(A(k) - A(i)) > tol
      B = [B A(i)];
      k = i;
   end
end
```

Step 2: Address Issues Detected by the Code Analyzer

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

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

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

Step 3: Generate MEX Code

To generate MEX code, use the fiaccel function.

1 Generate a MEX function for uniquetol:

```
T = numerictype(1, 16, 15);
fiaccel -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 uniquetol:

- 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”.

- 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.

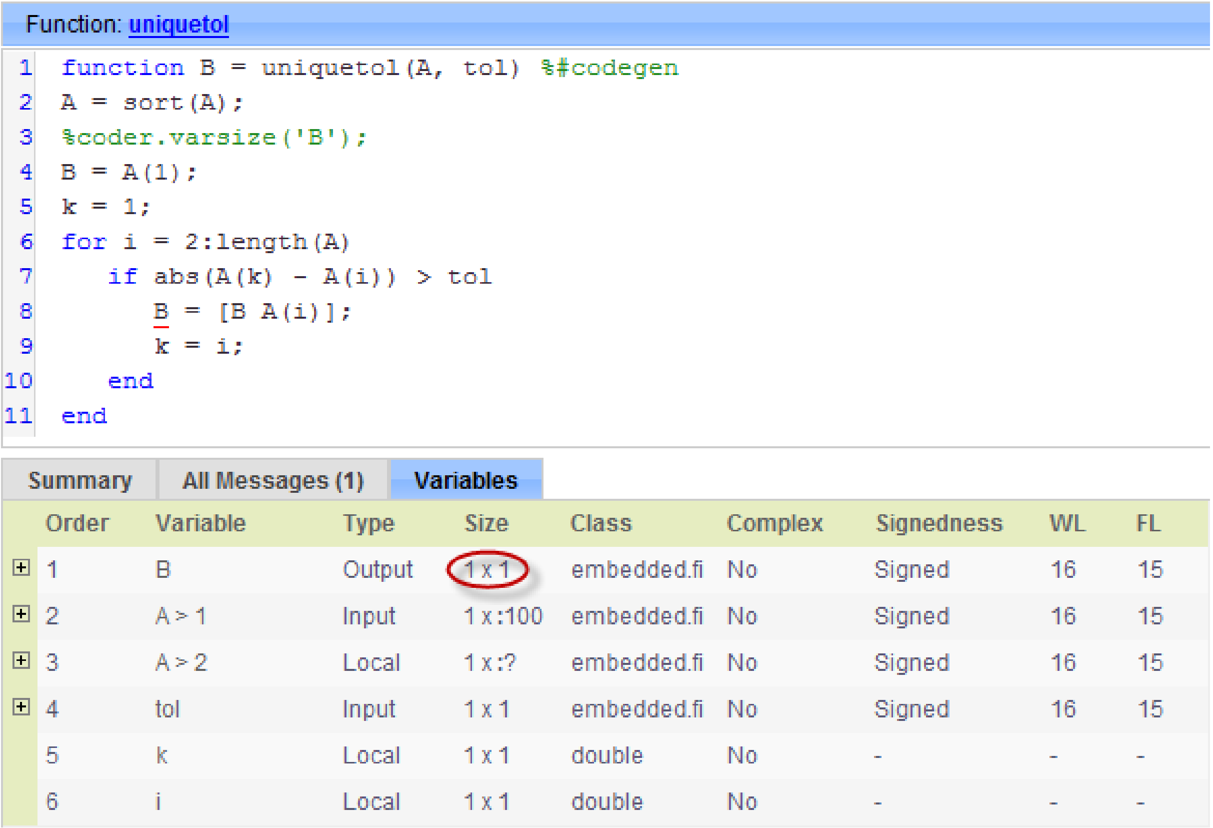
The `-report` option instructs `fiaccel` to generate a code generation report, even if no errors or warnings occur.

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

```
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
```

	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.

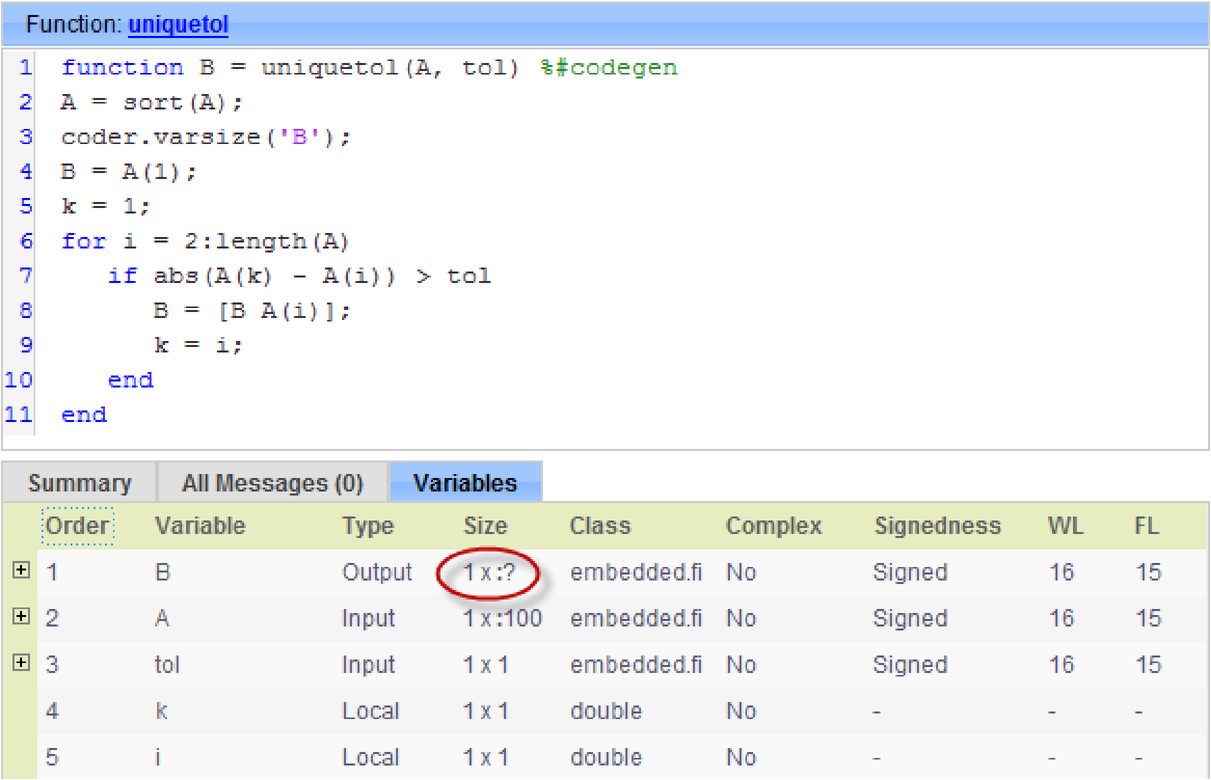
2 Generate code again using the same command:

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

In the current folder, fiaccel 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

```
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
```

	Order	Variable	Type	Size	Class	Complex	Signedness	WL	FL
⊞	1	B	Output	1 x :?	embedded.fi	No	Signed	16	15
⊞	2	A	Input	1 x :100	embedded.fi	No	Signed	16	15
⊞	3	tol	Input	1 x 1	embedded.fi	No	Signed	16	15
	4	k	Local	1 x 1	double	No	-	-	-
	5	i	Local	1 x 1	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
   ```

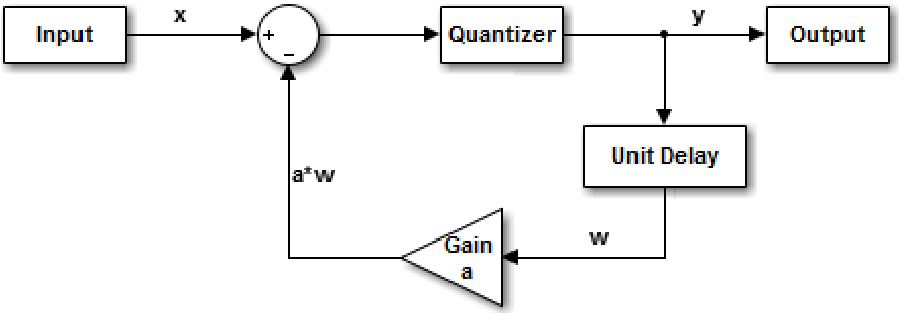
   This example shows that generating a MEX function using `fiaccel` greatly accelerates the execution of the fixed-point algorithm.

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');
fiacceldir = tempdirObj.tempDir;
fiaccelsrc = ...
    fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','fiaccel
copyfile(fiaccelsrc,fiacceldir,'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(fiacceldir,'fiaccelFeedback.m'))
```

```
function [y,w] = fiaccelFeedback(x,a,y,w)
%FIACCELFEEDBACK Quantizer and feedback loop used in EMLMEXBASICSDEMO.
% Copyright 1984-2012 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 =

  122.5921
```

Clean up Temporary Files

```
clear fiaccelFeedback_mex;
tempdirObj.cleanUp;
```

Propose Fixed-Point Data Types Using an Instrumented Mex Function

This example shows how to propose fixed-point data types using an instrumented MEX function.

This capability is not compatible with automatic fixed-point conversion. If you select `Convert to fixed point at build time`, you cannot build instrumented MEX functions.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

Before generating C code, you must set up the C compiler. See “Set Up C Compiler” on page 9-16.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver` .

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\coder\fun_with_matlab`.
2. Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
   ```

3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	`fun_with_matlab.m`	Entry-point MATLAB function
Test file	`fun_with_matlab_test.m`	MATLAB script that tests `fun_with_matlab.m`

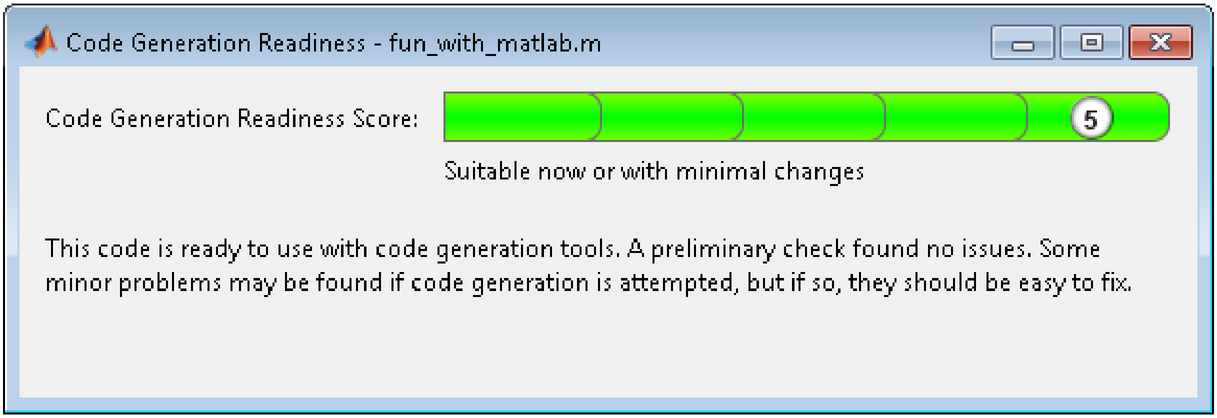
The fun_with_matlab Function

```
function y = fun_with_matlab(x) %#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
```

Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select `Check Code Generation Readiness`.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.


Code Generation Readiness - fun_with_matlab.m
Code Generation Readiness Score:
5
Suitable now or with minimal changes

This code is ready to use with code generation tools. A preliminary check found no issues. Some minor problems may be found if code generation is attempted, but if so, they should be easy to fix.

If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see "Detect and Debug Code Generation Errors" on page 9-25.

Create and set up a MATLAB Coder Project

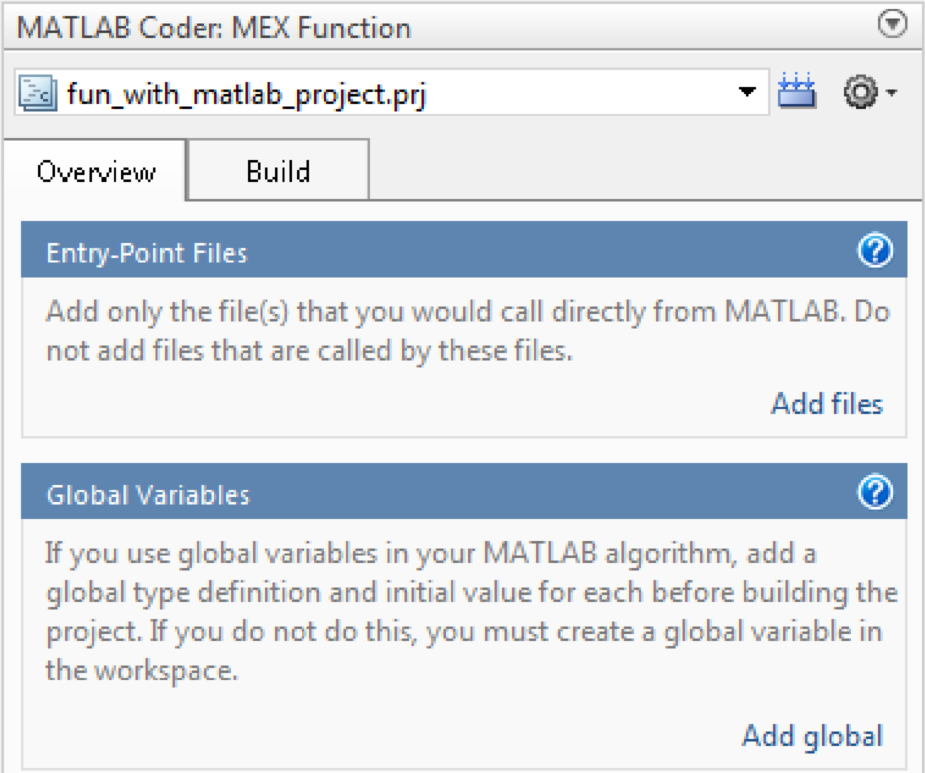
1 Navigate to the work folder that contains the file for this tutorial.

2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_matlab_project.prj`.

   Alternatively, at the MATLAB command line, enter

   ```
   coder -new fun_with_matlab_project.prj
   ```

   By default, the project opens in the MATLAB workspace.


MATLAB Coder: MEX Function

fun_with_matlab_project.prj

Entry-Point Files

Add only the file(s) that you would call directly from MATLAB. Do not add files that are called by these files.

Global Variables

If you use global variables in your MATLAB algorithm, add a global type definition and initial value for each before building the project. If you do not do this, you must create a global variable in the workspace.

3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

About the fun_with_matlab_test Script

The test script runs the fun_with_matlab function with three input signals: chirp, step, and impulse. The script then plots the results.

Contents of fun_with_matlab_test

```
% fun_with_matlab_test
%
% Define representative inputs
N = 256;                    % Number of points
t = linspace(0,1,N);        % Time vector from 0 to 1 second
```

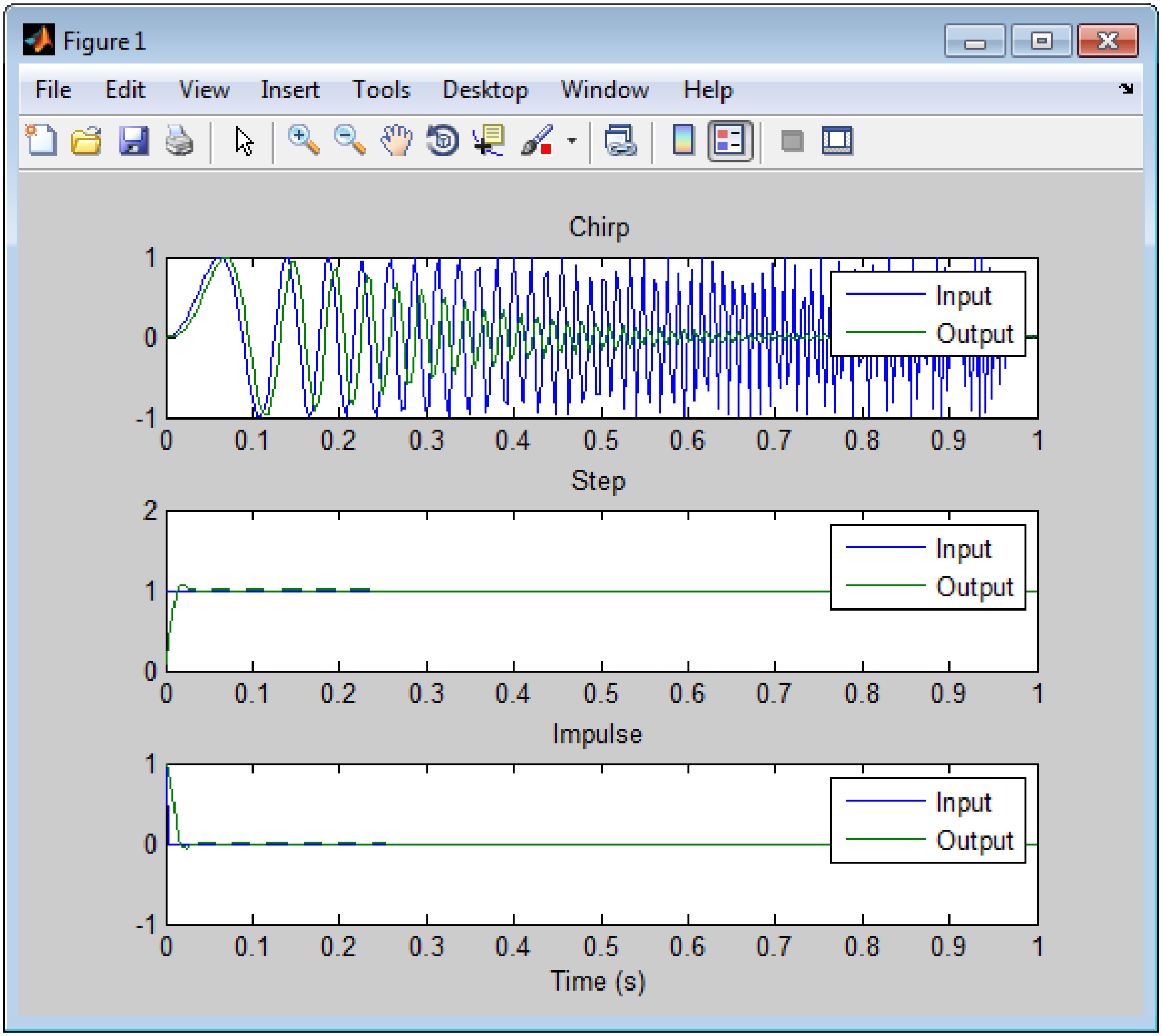
```
f1 = N/2;                    % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);          % Step
x_impulse = zeros(1,N);      % Impulse
x_impulse(1)=1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i=1:size(x,1)
  y(i,:) = fun_with_matlab(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i=1:size(x,1)
  subplot(size(x,1),1,i);
  plot(t,x(i,:),t,y(i,:));
  title(titles{i})
  legend('Input','Output');
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.');
```

Define Input Types

1 On the project **Overview** tab, click the **Autodefine types** link.

2 In the Autodefine Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

   The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them in the Autodefine Input Types dialog box.

3 In this dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

Build Instrumented MEX Function

1. In the project, click the **Build** tab.

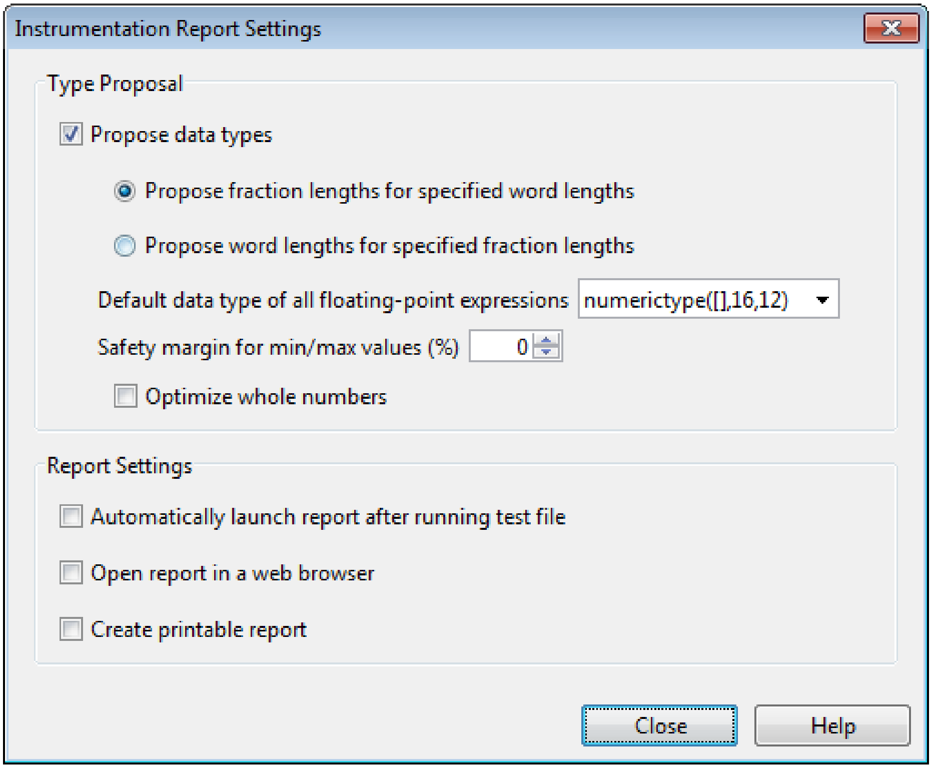
2. On the **Build** tab, set the **Output type** to `Instrumented MEX Function`.

3. Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function `fun_with_matlab_mex` in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of variables in your MATLAB code.

View Data Type Proposal Settings

1. On the **Show Instrumentation Results** pane, click the **Data type proposal and report settings** link.



Instrumentation Report Settings

Type Proposal

☑ Propose data types

● Propose fraction lengths for specified word lengths

○ Propose word lengths for specified fraction lengths

Default data type of all floating-point expressions: numerictype([],16,12)

Safety margin for min/max values (%): 0

□ Optimize whole numbers

Report Settings

□ Automatically launch report after running test file

□ Open report in a web browser

□ Create printable report

This example uses the default data type proposal settings which propose fraction lengths for the specified word lengths. Because the MATLAB code is floating-point, the word length is specified by the **Default data type of all floating-point expressions** field. You can specify the `numerictype` signedness, word length and fraction length. Specifying `[]` for signedness instructs MATLAB Coder to choose the signedness based on simulation values. The default word length is `16`. The default fraction length is `12`.

2 Close the dialog box.

Run Simulation

1 On the **Run Simulation** pane, verify that the test file is set to `fun_with_matlab_test` and that **Redirect entry-point calls to MEX**

function is selected. That way, each call to `fun_with_matlab` is replaced with a call to the instrumented MEX function `fun_with_matlab_mex`.

2 On the **Run Simulation** pane, click **Run**.

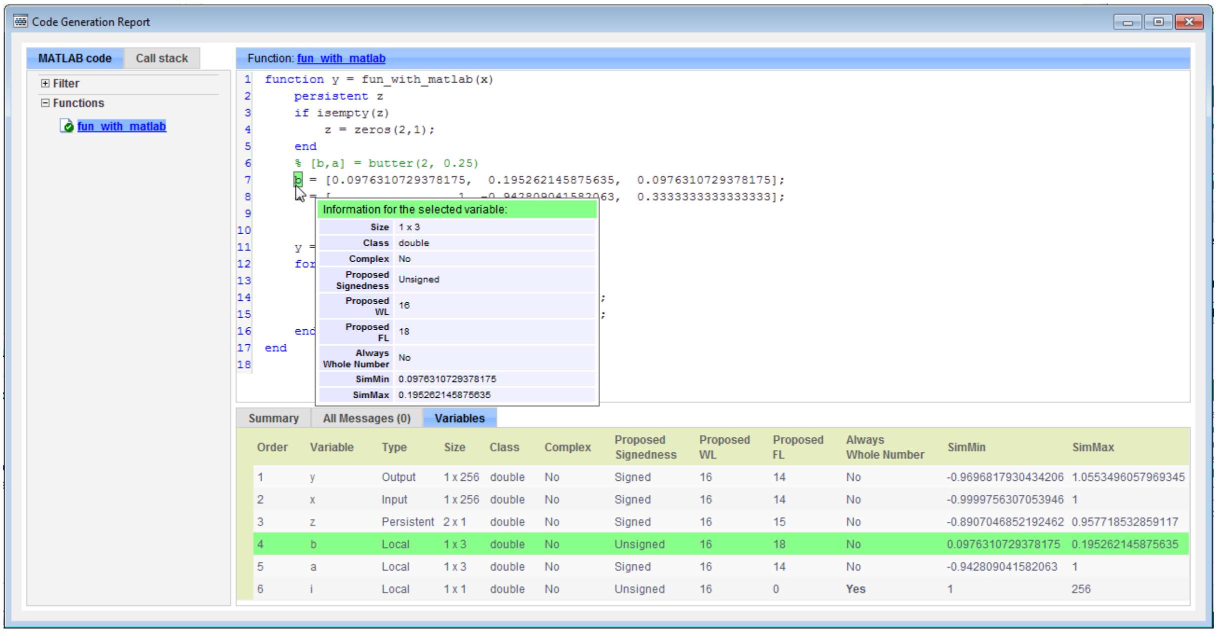
   The `fun_with_matlab_test` file runs and calls `fun_with_matlab_mex`. The outputs of the filters are displayed as before.

View Code Generation Report

1 On the **Show Instrumentation Results** pane, click **View Report**.

2 In the **Code Generation Report**, click the **Variables** tab.

   The report displays the simulation minimum and maximum values and the proposed data types.



```
function y = fun_with_matlab(x)
    persistent z
    if isempty(z)
        z = zeros(2,1);
    end
    % [b,a] = butter(2, 0.25)
    b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
     = [                  1  -0.942809041582063,  0.3333333333333333];


    y =
    for
                                                   ;
                                                   ;
    end
end
```

Information for the selected variable:

Size	1 x 3
Class	double
Complex	No
Proposed Signedness	Unsigned
Proposed WL	16
Proposed FL	18
Always Whole Number	No
SimMin	0.0976310729378175
SimMax	0.195262145875635

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax
1	y	Output	1 x 256	double	No	Signed	16	14	No	-0.9696817930434206	1.0553496057969345
2	x	Input	1 x 256	double	No	Signed	16	14	No	-0.9999756307053946	1
3	z	Persistent	2 x 1	double	No	Signed	16	15	No	-0.8907046852192462	0.957718532859117
4	b	Local	1 x 3	double	No	Unsigned	16	18	No	0.0976310729378175	0.195262145875635
5	a	Local	1 x 3	double	No	Signed	16	14	No	-0.942809041582063	1
6	i	Local	1 x 1	double	No	Unsigned	16	0	Yes	1	256

MATLAB Coder proposes data types with word length of 16 and fraction length optimized to avoid overflows.

Next Steps

To learn how to apply the proposed data types to your entry-point MATLAB function and verify that the fixed-point version of your algorithm is functionally equivalent to your original MATLAB algorithm, see “Apply Fixed-Point Data Types Using an Instrumented Mex Function” on page 9-117.

Apply Fixed-Point Data Types Using an Instrumented Mex Function

This example shows how to write a fixed-point version of your entry-point function using the data types proposed in “Propose Fixed-Point Data Types Using an Instrumented Mex Function” on page 9-107.

This capability is not compatible with automatic fixed-point conversion. If you select `Convert to fixed point at build time`, you cannot build instrumented MEX functions.

You will learn how to:

- Use the proposed data types to create a fixed-point version of your entry-point function.
- Update your test file to call the fixed-point entry-point function.
- Verify that the fixed-point function is functionally equivalent to the original MATLAB algorithm.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

Before generating C code, you must set up the C compiler. See “Set Up C Compiler” on page 9-16.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\coder\fun_with_matlab`.
2. Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
   ```

3. Copy the following files to your local working folder.

Type	Name	Description
Function code	`fun_with_matlab.m`	Entry-point MATLAB function
Test file	`fun_with_matlab_test.m`	MATLAB script that tests `fun_with_matlab.m`
Function code	`fun_with_fi.m`	Entry-point MATLAB function — fixed-point version of `fun_with_matlab` that uses data types proposed in "Propose Fixed-Point Data Types Using an Instrumented Mex Function" on page 9-107
Test file	`fun_with_fi_test.m`	MATLAB script that runs both `fun_with_matlab` and `fun_with_fi` and compares the results

The `fun_with_fi` Function

The `fun_with_fi` is a fixed-point version of the `fun_with_matlab` function that uses the data types proposed in "Propose Fixed-Point Data Types Using an Instrumented Mex Function" on page 9-107.

Variable	Proposed Signedness	Proposed Word Length	Proposed Fraction Length
y	Signed	16	14
x	Signed	16	14
z	Signed	16	15
a	Unsigned	16	18
b	Signed	16	14
i	Unsigned	16	0

For example, in `fun_with_matlab`, variable `y` is defined as `y = zeros(size(x));`. In `fun_with_fi`, to specify that it is a signed fixed-point data type with a word length of 16 and a fraction length of 14:

```
y = fi(zeros(size(x)),1,16,14,'OverflowAction','Wrap','RoundingMethod','Floor');
```

For more information, see `fi`.

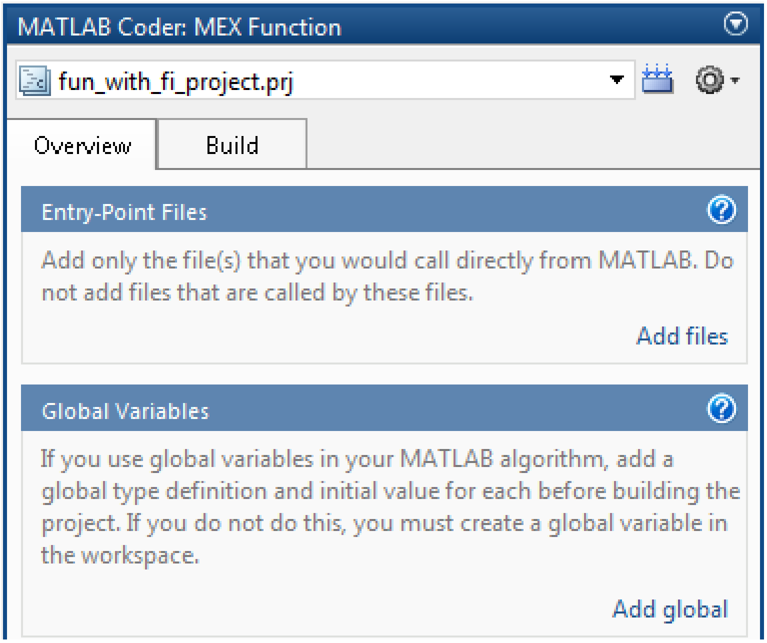
Create and set up a MATLAB Coder Project

1. Navigate to the work folder that contains the file for this tutorial.
2. On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_fi_project.prj`.

   Alternatively, at the MATLAB command line, enter

   ```
   coder -new fun_with_fi_project.prj
   ```

   By default, the project opens in the MATLAB workspace.



MATLAB Coder: MEX Function

fun_with_fi_project.prj

Entry-Point Files

Add only the file(s) that you would call directly from MATLAB. Do not add files that are called by these files.

Global Variables

If you use global variables in your MATLAB algorithm, add a global type definition and initial value for each before building the project. If you do not do this, you must create a global variable in the workspace.

3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_fi.m`, and then click **OK** to add the file to the project.

Define Input Types

1 On the project **Overview** tab, click the **Autodefine types** link.

2 In the Autodefine Input Types dialog box, add `fun_with_fi_test` as a test file, and then click **Run**.

   The test file runs and plots the outputs of the filter. MATLAB Coder determines the input types from the test file and then displays them.

3 In the Autodefine Input Types dialog box, click **Use These Types** to accept the autodefined input type.

   MATLAB Coder sets the type of `x` to `double(1x256)`.

The fun_with_fi_test Script

The `fun_with_fi_test` script runs the original floating-point MATLAB algorithm, `fun_with_matlab`, then runs the fixed-point version of the algorithm, `fun_with_fi`. The script then plots the outputs for the floating-point and fixed-point algorithms and the difference in results.

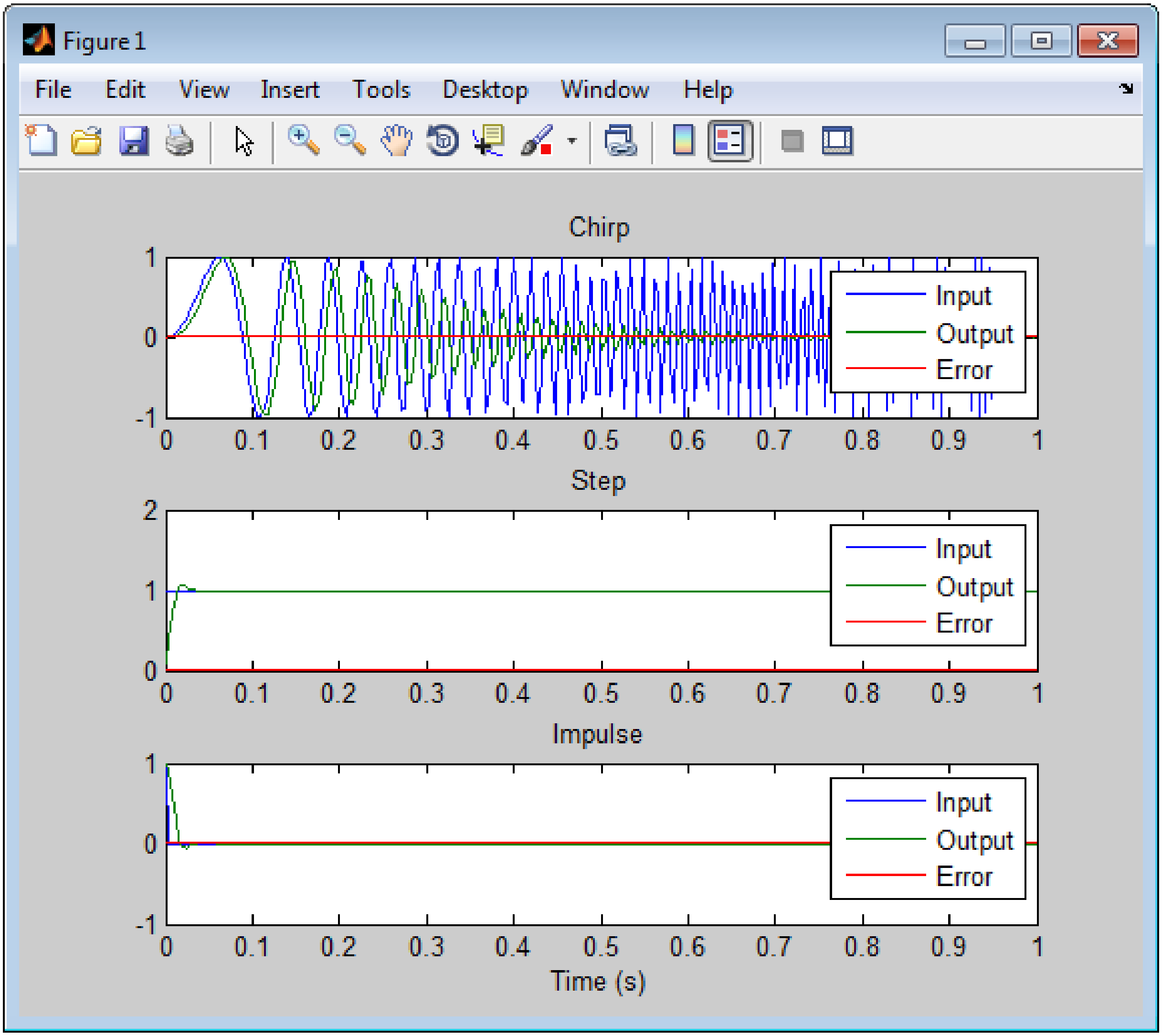
Run Simulation

1. In the project, click the **Build** tab.

2. On the **Verification** pane, verify that the test file is set to `fun_with_fi_test`. Clear **Redirect entry-point calls to MEX function** so that the test file calls the MATLAB versions of the original and fixed-point algorithms.

3. On **Verification** pane, click **Run**.

   The `fun_with_fi_test` file runs. The test file runs the original MATLAB algorithm and the fixed-point version, and plots the difference in their outputs.



4 Optionally, zoom in on each plot in turn to view the error (difference between the two versions of the algorithm). In this example, the errors are very small, on the order of 10^{-3}. If the error is unacceptably large, refine the fixed-point data types.

Code Generation Readiness Tool

In this section...

"What Information Does the Code Generation Readiness Tool Provide?" on page 9-123

"Summary Tab" on page 9-124

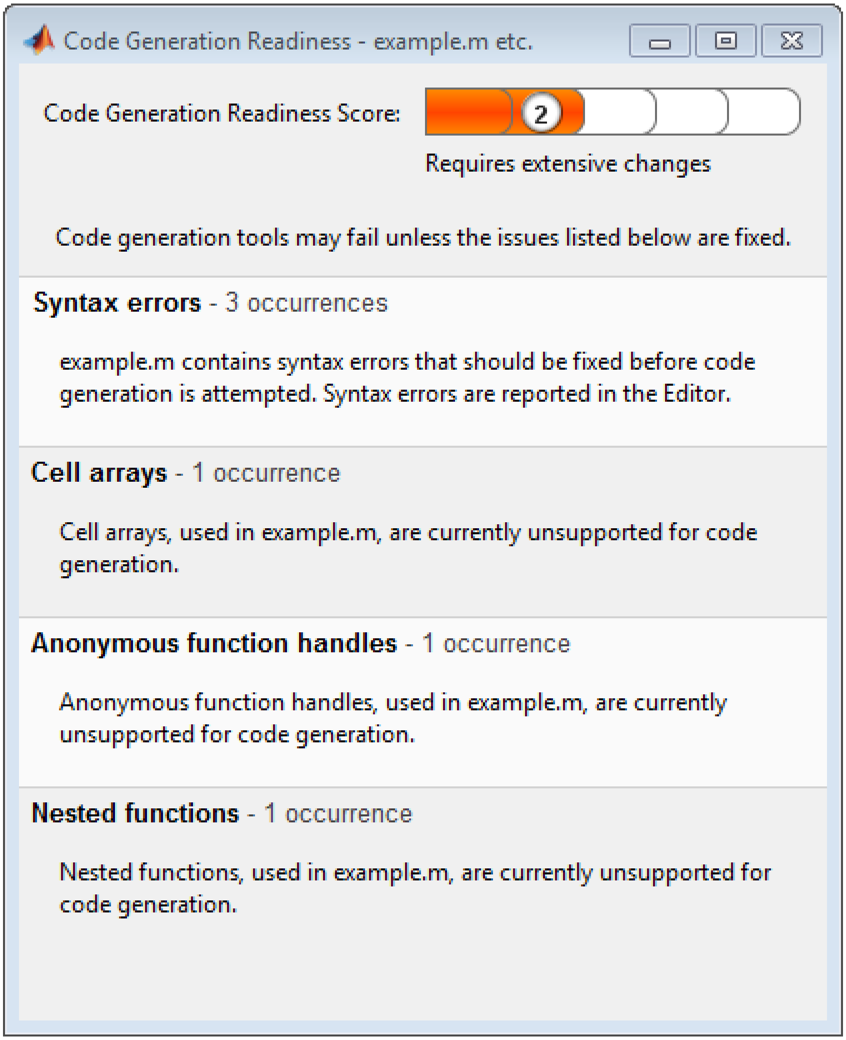
"Code Structure Tab" on page 9-126

"See Also" on page 9-129

What Information Does the Code Generation Readiness Tool Provide?

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code suitable for code generation. The tool might not detect all code generation issues. Under certain circumstances, it might report false errors. Because the tool might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.

Summary Tab



Code Generation Readiness - example.m etc.

Code Generation Readiness Score: 2

Requires extensive changes

Code generation tools may fail unless the issues listed below are fixed.

Syntax errors - 3 occurrences

example.m contains syntax errors that should be fixed before code generation is attempted. Syntax errors are reported in the Editor.

Cell arrays - 1 occurrence

Cell arrays, used in example.m, are currently unsupported for code generation.

Anonymous function handles - 1 occurrence

Anonymous function handles, used in example.m, are currently unsupported for code generation.

Nested functions - 1 occurrence

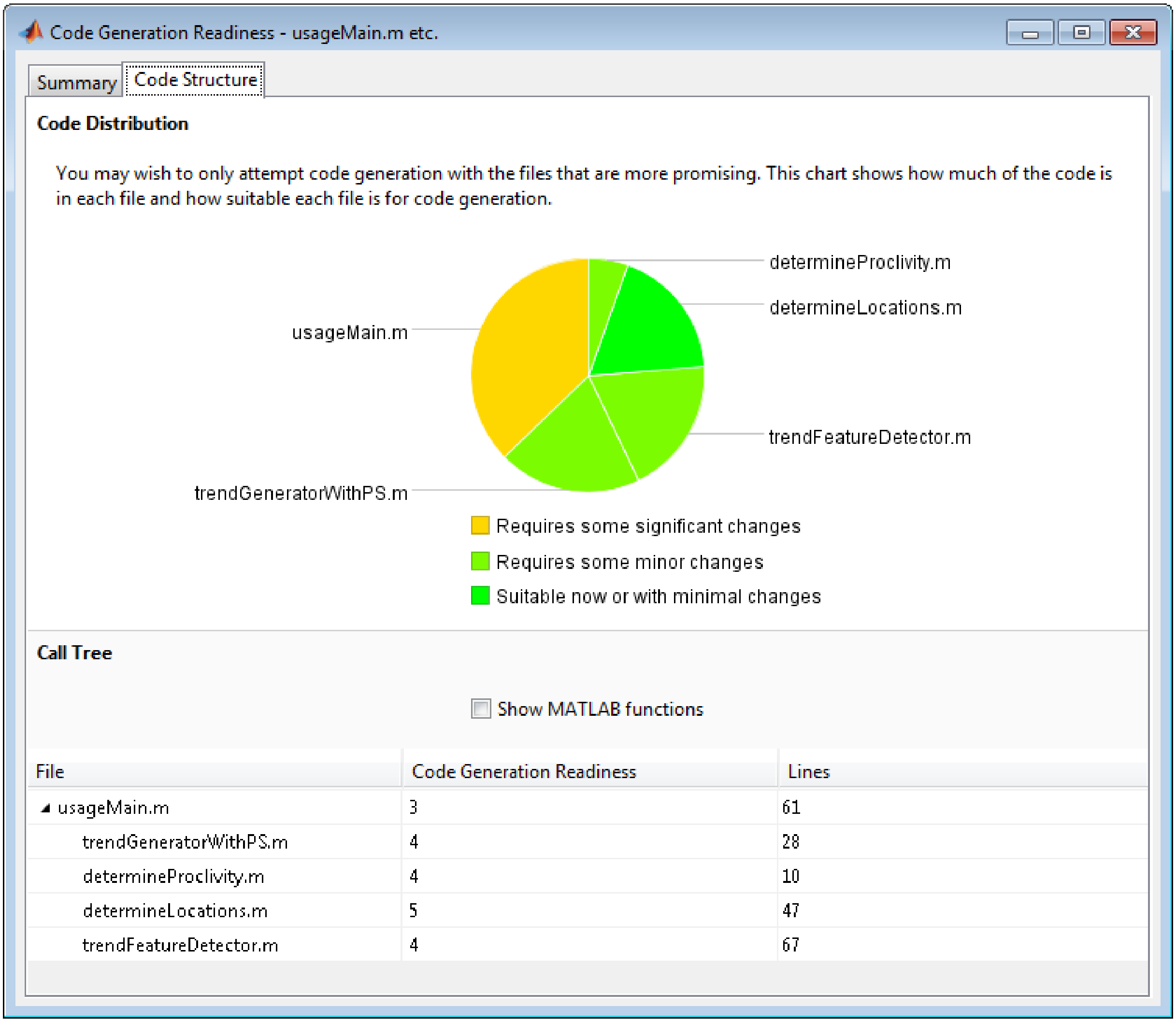
Nested functions, used in example.m, are currently unsupported for code generation.

The **Summary** tab provides a **Code Generation Readiness Score** which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes.

On this tab, the tool also provides information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. Use the code analyzer to learn more about the issues and how to fix them.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as recursion, cell arrays, nested functions, and function handles.
- Unsupported data types.

Code Structure Tab



Code Generation Readiness - usageMain.m etc.
Summary
Code Structure

Code Distribution

You may wish to only attempt code generation with the files that are more promising. This chart shows how much of the code is in each file and how suitable each file is for code generation.

determineProclivity.m
determineLocations.m
usageMain.m
trendFeatureDetector.m
trendGeneratorWithPS.m
🟨 Requires some significant changes
🟩 Requires some minor changes
🟩 Suitable now or with minimal changes

Call Tree

□ Show MATLAB functions

File	Code Generation Readiness	Lines
◢ usageMain.m	3	61
trendGeneratorWithPS.m	4	28
determineProclivity.m	4	10
determineLocations.m	5	47
trendFeatureDetector.m	4	67

If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab provides information about the relative size of each file and how suitable each file is for code generation.

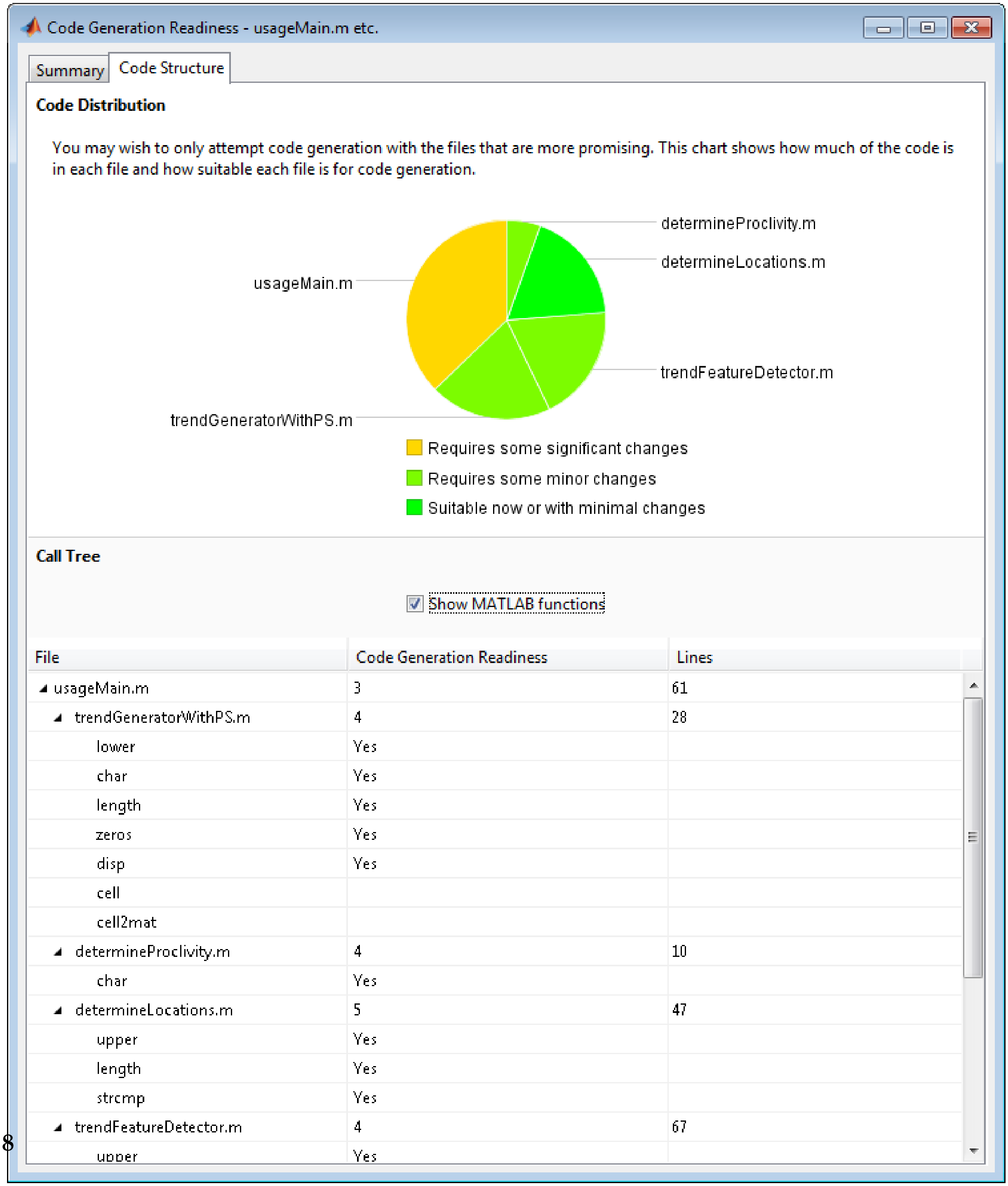
Code Distribution

The **Code Distribution** pane provides a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. This information is useful during the planning phase of a project for estimation and scheduling purposes. If the report indicates that there are multiple files not yet suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane provides information on the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions. If you select **Show MATLAB Functions**, the report also lists the MATLAB functions called by your function code. For each of these MATLAB functions, if the function is supported for code generation, the report sets **Code Generation Readiness** to Yes.



Code Distribution

You may wish to only attempt code generation with the files that are more promising. This chart shows how much of the code is in each file and how suitable each file is for code generation.

🟨 Requires some significant changes

🟩 Requires some minor changes

🟩 Suitable now or with minimal changes

Call Tree

☑ Show MATLAB functions

File	Code Generation Readiness	Lines
◢ usageMain.m	3	61
◢ trendGeneratorWithPS.m	4	28
lower	Yes	
char	Yes	
length	Yes	
zeros	Yes	
disp	Yes	
cell		
cell2mat		
◢ determineProclivity.m	4	10
char	Yes	
◢ determineLocations.m	5	47
upper	Yes	
length	Yes	
strcmp	Yes	
◢ trendFeatureDetector.m	4	67
upper	Yes	

See Also

- "Check Code Using the Code Generation Readiness Tool" on page 9-130

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 9-123

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 generation software detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “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 9-25.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generation software determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls in MATLAB Generated Code” on page 11-2. 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 9-51
- “Why Test MEX Functions in MATLAB?”
- “When to Generate Code from MATLAB Algorithms” on page 16-2
- “Debugging Strategies” on page 9-25
- “Declaring MATLAB Functions as Extrinsic Functions” on page 11-12

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.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.

```
int64m_T my_add1(int a, unsigned int b)
{
  int64m_T y;
  int64m_T r0;
  int64m_T r1;
  int64m_T r2;
  int96m_T r3;
  int96m_T r4;
  int64m_T r5;
  sLong2MultiWord(a, &r0.chunks[0U], 2);
  sMultiWordShl(&r0.chunks[0U], 2, 29U, &r1.chunks[0U], 2);
  MultiWordSignedWrap(&r1.chunks[0U], 2, 2U, &r2.chunks[0U]);
  uLong2MultiWord(b, &r0.chunks[0U], 2);
  MultiWordSignedWrap(&r0.chunks[0U], 2, 2U, &r1.chunks[0U]);
  MultiWordAdd(&r2.chunks[0U], &r1.chunks[0U], &y.chunks[0U], 2);
  sMultiWord2MultiWord(&y.chunks[0U], 2, &r3.chunks[0U], 3);
  sMultiWordShl(&r3.chunks[0U], 3, 2U, &r4.chunks[0U], 3);
  sMultiWord2sMultiWordSat(&r4.chunks[0U], 3, &r5.chunks[0U], 2);
  sMultiWordShr(&r5.chunks[0U], 2, 2U, &y.chunks[0U], 2);
   return y;
}
```

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.

Interoperability with Other Products

- “fi Objects with Simulink” on page 10-2
- “fi Objects with DSP System Toolbox” on page 10-8
- “Ways to Generate Code” on page 10-13

fi Objects with Simulink

In this section...

"Reading Fixed-Point Data from the Workspace" on page 10-2

"Writing Fixed-Point Data to the Workspace" on page 10-2

"Setting the Value and Data Type of Block Parameters" on page 10-6

"Logging Fixed-Point Signals" on page 10-6

"Accessing Fixed-Point Block Data During Simulation" on page 10-7

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]'
```

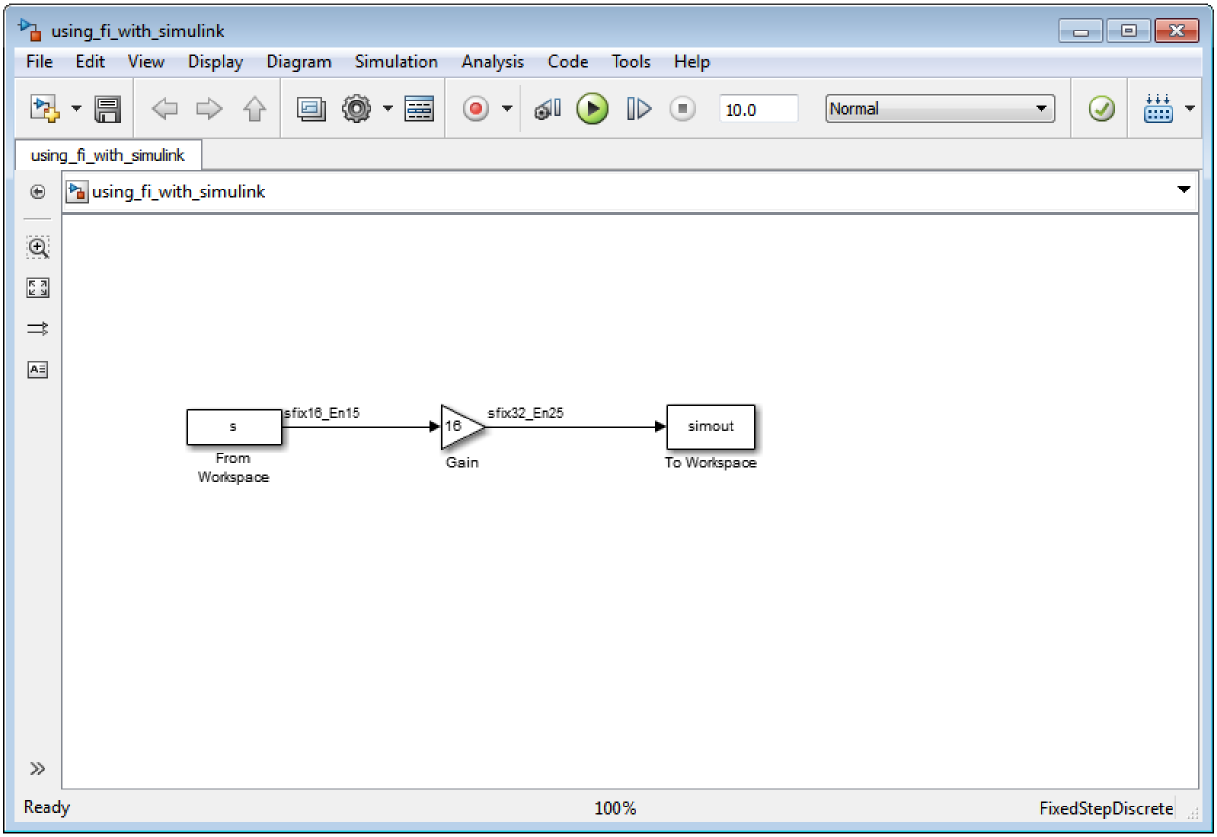
```
s =

    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. Refer to “Block Support for Data and Signal Types” in the Simulink documentation for more information.

Logging Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as `fi` objects.

To enable signal logging for a signal:

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” 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 10-8

"Writing Fixed-Point Signals to the Workspace" on page 10-8

"fi Objects with dfilt Objects" on page 10-12

Reading Fixed-Point Signals from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model using the Signal From Workspace and Triggered Signal From Workspace blocks from DSP System Toolbox™ software. Enter the name of the defined `fi` variable in the **Signal** parameter of the Signal From Workspace or Triggered Signal From Workspace block.

Writing Fixed-Point Signals to the Workspace

Fixed-point output from a model can be written to the MATLAB workspace via the Signal To Workspace or Triggered To Workspace block from the blockset. The fixed-point data is always written as a 2-D or 3-D array.

Note To write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace or Triggered To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

For example, you can use the following code to create a `fi` object in the MATLAB workspace. You can then use the Signal From Workspace block to bring the data into a Simulink model.

```
a = fi([sin(0:10)' sin(10:-1:0)'])

a =

         0   -0.5440
    0.8415    0.4121
    0.9093    0.9893
    0.1411    0.6570
   -0.7568   -0.2794
   -0.9589   -0.9589
   -0.2794   -0.7568
    0.6570    0.1411
    0.9893    0.9093
    0.4121    0.8415
   -0.5440         0


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

The Signal From Workspace block in the following model has these settings:

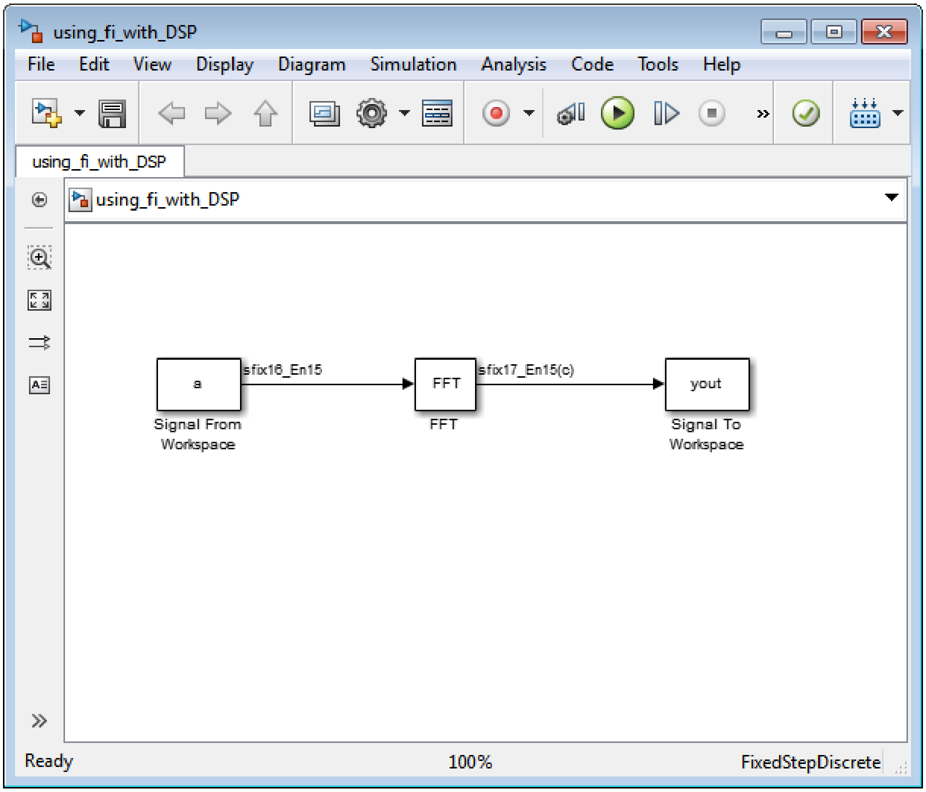
- **Signal** — `a`
- **Sample time** — `1`
- **Samples per frame** — `2`
- **Form output after final data value by** — `Setting to zero`

The following parameters in the **Solver** pane of the **Model Configuration Parameters** dialog have these settings:

- **Start time** — `0.0`

- **Stop time** — `10.0`
- **Type** — `Fixed-step`
- **Solver** — `Discrete (no continuous states)`
- **Fixed step size (fundamental sample time)** — `1.0`

Remember, to write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.



The Signal To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` object.

```
yout =


(:,:,1) =

    0.8415   -0.1319
   -0.8415   -0.9561


(:,:,2) =

    1.0504    1.6463
    0.7682    0.3324


(:,:,3) =

   -1.7157   -1.2383
    0.2021    0.6795


(:,:,4) =

    0.3776   -0.6157
   -0.9364   -0.8979


(:,:,5) =

    1.4015    1.7508
    0.5772    0.0678


(:,:,6) =

   -0.5440         0
   -0.5440         0
```

```
    DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 17
  FractionLength: 15
```

fi Objects with dfilt Objects

When the `Arithmetic` property is set to `'fixed'`, you can use an existing `fi` object as the input, states, or coefficients of a `dfilt` object in DSP System Toolbox software. Also, fixed-point filters in the toolbox return `fi` objects as outputs. Refer to the DSP System Toolbox software documentation for more information.

Ways to Generate Code

There are several ways to use Fixed-Point Designer software to generate code:

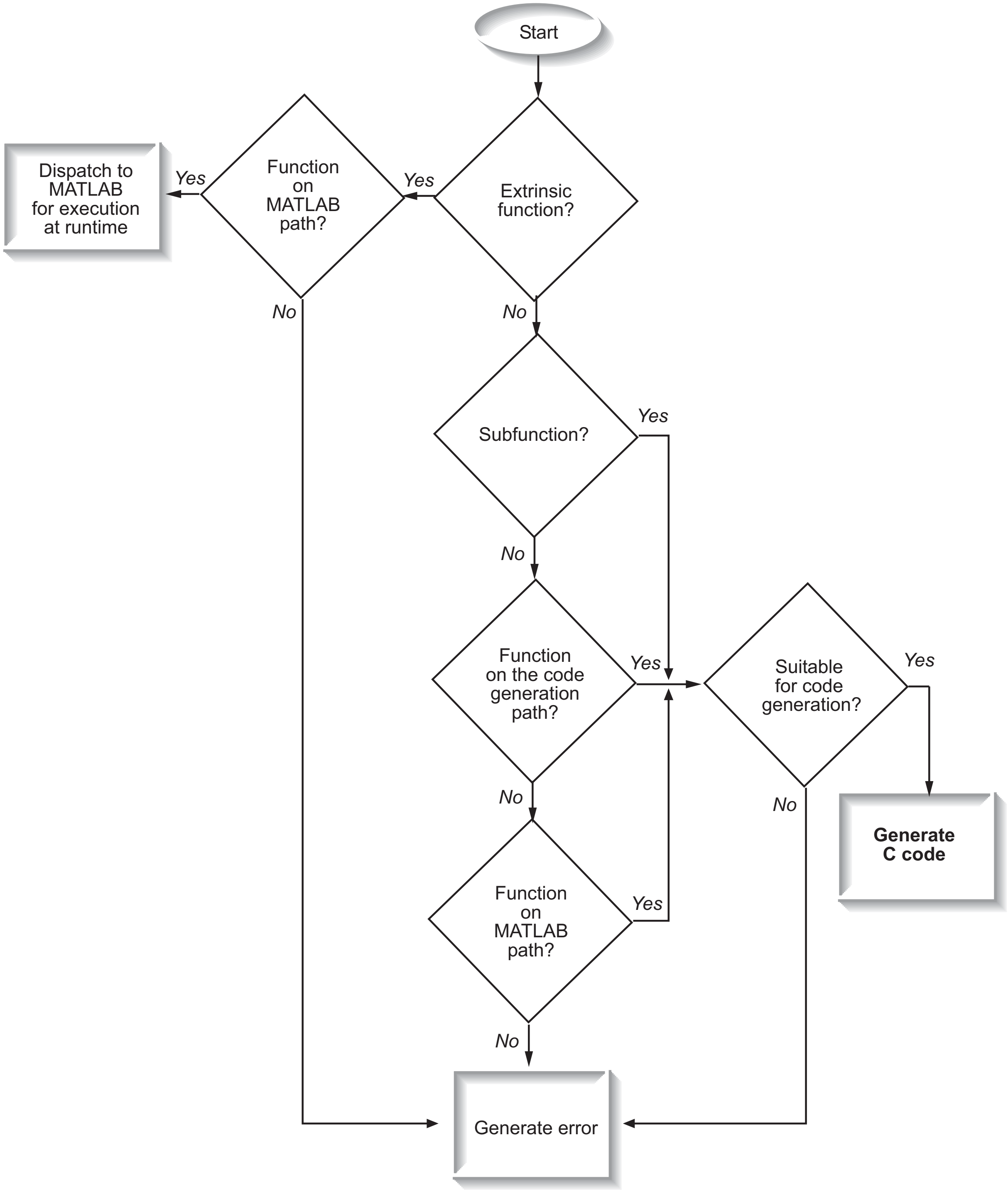
- The Fixed-Point Designer `fiaccel` function converts your fixed-point MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.
- The MATLAB Coder `codegen` function automatically converts MATLAB code to C/C++ code. Using the MATLAB Coder software 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" 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 in MATLAB Generated Code” on page 11-2
- “Resolution of File Types on Code Generation Path” on page 11-6
- “Compilation Directive %#codegen” on page 11-8
- “Call Local Functions” on page 11-9
- “Call Supported Toolbox Functions” on page 11-10
- “Call MATLAB Functions” on page 11-11

Resolution of Function Calls in MATLAB Generated Code

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:


Start
Extrinsic function?
Yes
Function on MATLAB path?
Yes
Dispatch to MATLAB for execution at runtime
No
No
Subfunction?
Yes
No
Function on the code generation path?
Yes
No
Function on MATLAB path?
Yes
No
Suitable for code generation?
Yes
No
Generate C code
Generate error

Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path

  See “Compile Path Search Order” on page 11-4.

- Attempts to compile functions unless the code generation software 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 11-12. During simulation, the code generation software generates code for the call to an extrinsic function, but does not generate the function’s internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

  The code generation 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 the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 11-6

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1. Code generation path

   MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.

2. MATLAB path

   If the function is not on the code generation path, MATLAB searches this path.

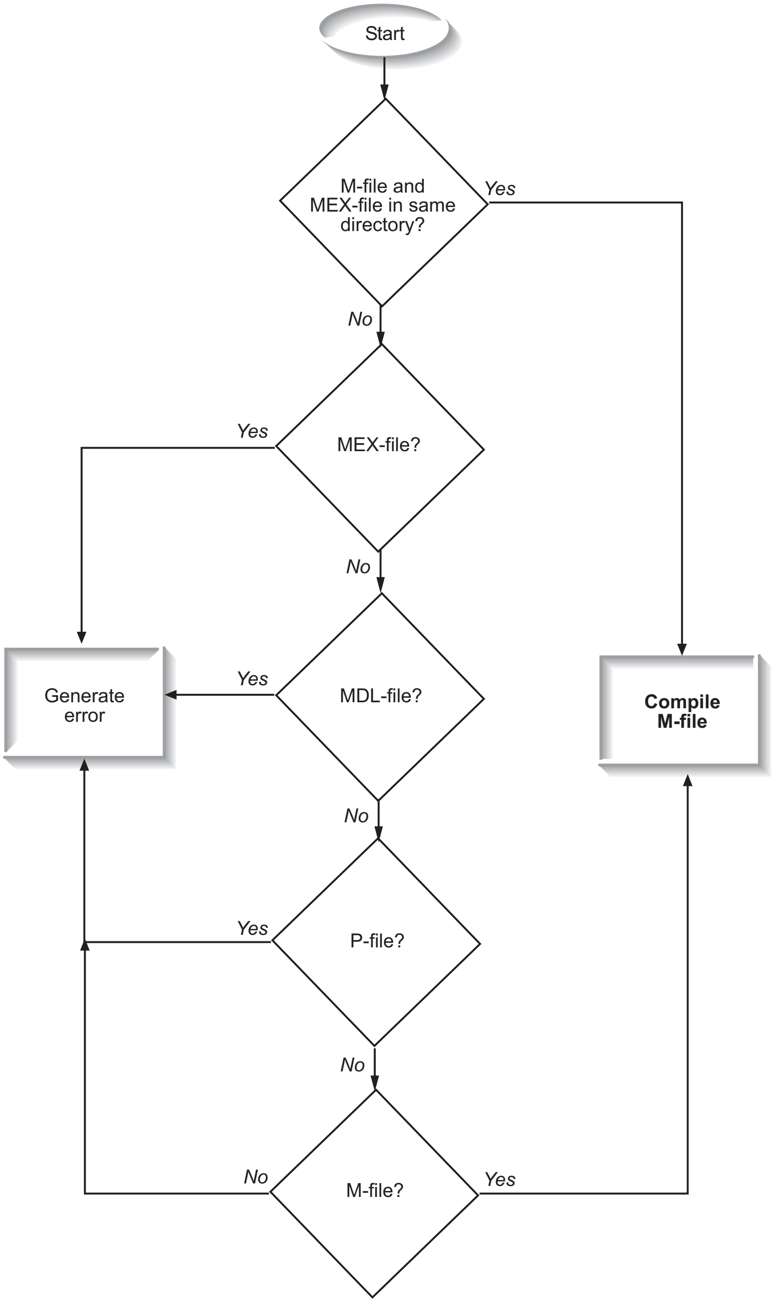
MATLAB applies the same dispatcher rules when searching each path (see "Function Precedence Order").

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:


Start
M-file and MEX-file in same directory?
Yes
No
MEX-file?
Yes
No
MDL-file?
Yes
No
P-file?
Yes
No
M-file?
No
Yes
Generate error
Compile M-file

Compilation Directive %#codegen

Add the `%#codegen` directive (or pragma) to your function to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during code generation.

Call Local Functions

Local functions are functions defined in the body of MATLAB function. They work the same way for code generation as they do when executing your algorithm in the MATLAB environment.

The following example illustrates how to define and call a local function mean:

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

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

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

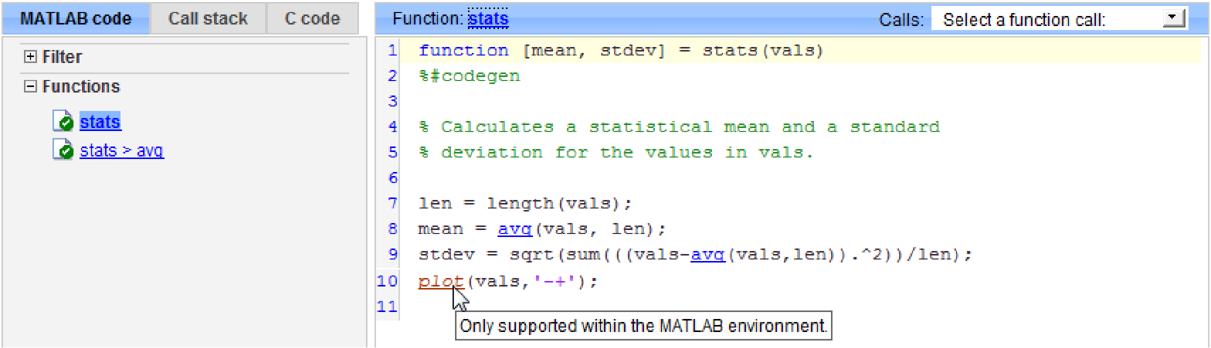
Call Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 21-2.

Call MATLAB Functions

The code generation software 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 generation software generates code for these functions, but does not generate their internal code. During standalone code generation, MATLAB 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, MATLAB 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 generation software dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.



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

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

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');
```

Only supported within the MATLAB environment.

For unsupported functions other than common visualization functions, you must declare the functions (like `pause`) to be extrinsic (see "Resolution of Function Calls in MATLAB Generated Code" on page 11-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 11-16).

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

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 11-12).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using feval” on page 11-16).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

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

Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`:

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
%  and displays the triangle.

c = sqrt(a^2 + b^2);
create_plot(a, b, color);


function create_plot(a, b, color)
%Declare patch and axis as extrinsic

coder.extrinsic('patch');

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

The code generation software detects that axis is not supported for code generation and automatically treats it as an extrinsic function. The compiler does not generate code for patch and axis, but instead dispatches them to MATLAB for execution.

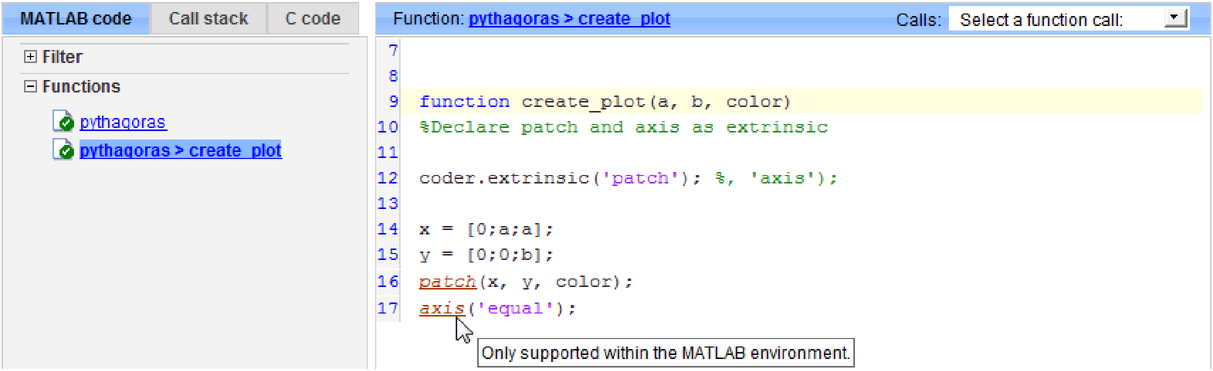
To test the function, follow these steps:

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

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

2 Click the link to the code generation report and then, in the report, view the MATLAB code for create_plot.

The report highlights the patch and axis functions to indicate that they are supported only within the MATLAB environment.



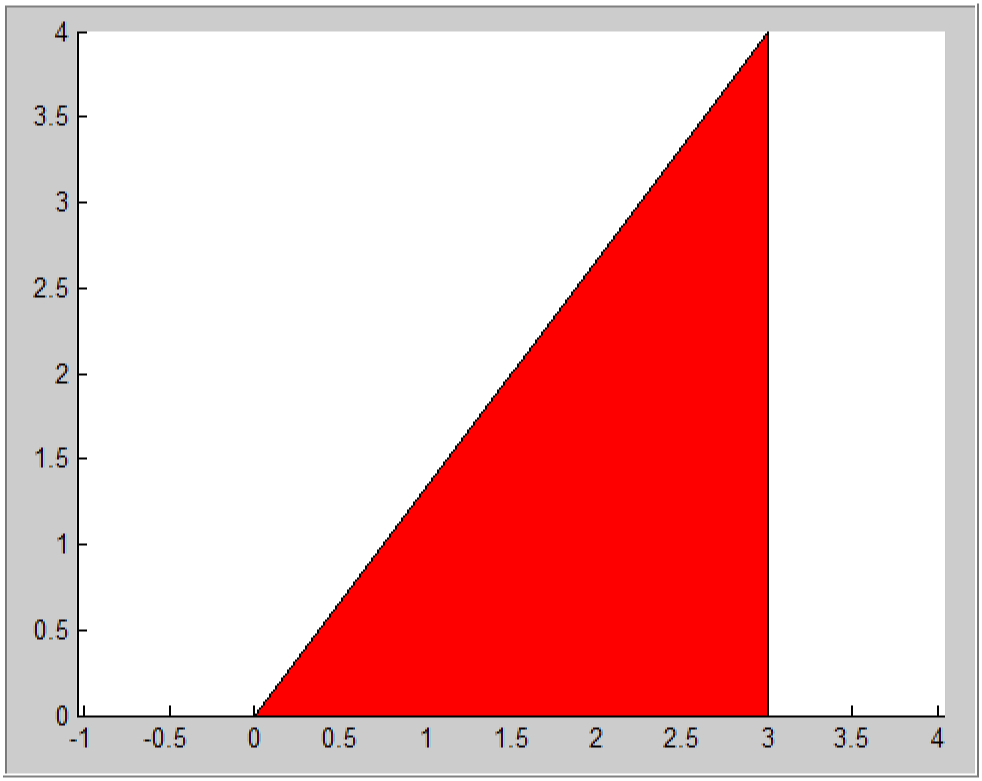
```


function create_plot(a, b, color)
%Declare patch and axis as extrinsic

coder.extrinsic('patch'); %, 'axis');

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

Only supported within the MATLAB environment.

3 Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the coder.extrinsic Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output — such as `pause` — during simulation, without generating unnecessary code (see "How MATLAB Resolves Extrinsic Functions During Simulation" on page 11-16).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see "Working with mxArrays" on page 11-17).
- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see "Scope of Extrinsic Function Declarations" on page 11-15).

- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see "Scope of Extrinsic Function Declarations" on page 11-15). To narrow the scope, use `feval` (see "Calling MATLAB Functions Using feval" on page 11-16).

Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` as treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

  ```
  function y = foo %#codegen
  coder.extrinsic('rat');
  [N D] = rat(pi);
  y = 0;
  y = mymin(N, D);

  function y = mymin(a,b)
  coder.extrinsic('min');
  y = min(a,b);
  ```

Here, the function rat is extrinsic every time it is called inside the main function foo, but the function min is extrinsic only when called inside the local function mymin.

- Call the MATLAB function using feval, as described in "Calling MATLAB Functions Using feval" on page 11-16.

Calling MATLAB Functions Using feval

The function feval is automatically interpreted as an extrinsic function during code generation. Therefore, you can use feval to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

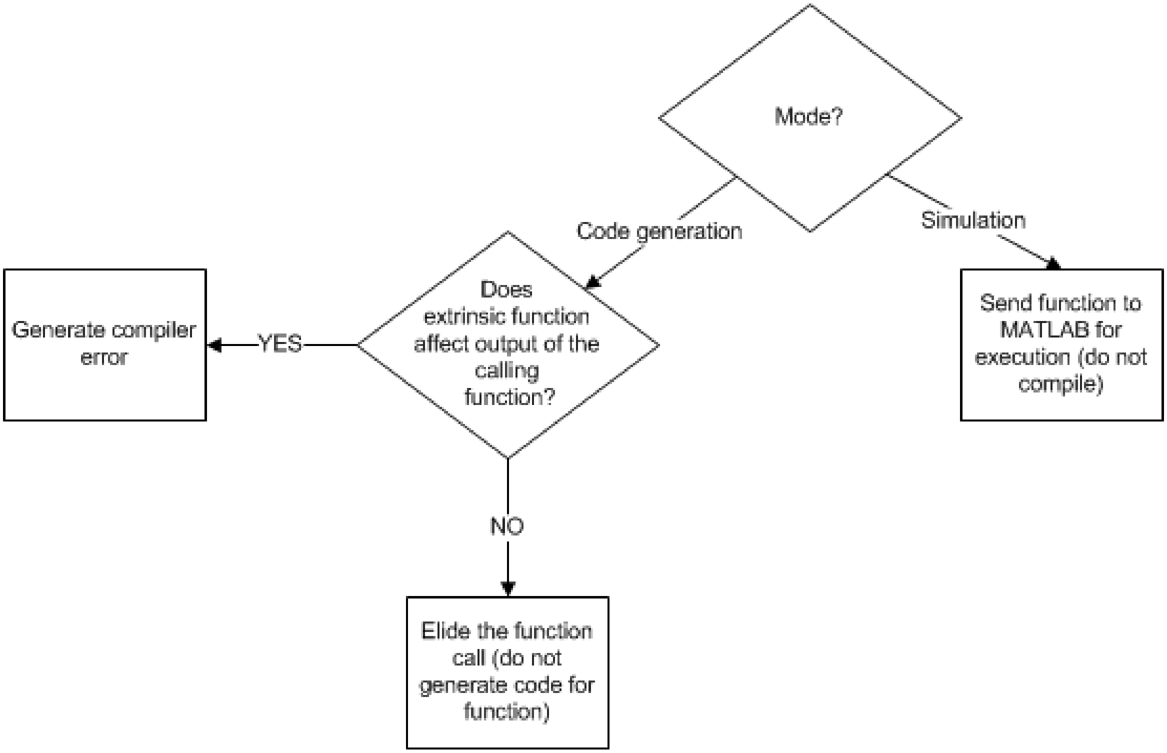
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because feval is extrinsic, the statement feval('min', N, D) is evaluated by MATLAB — not compiled — which has the same result as declaring the function min extrinsic for just this one call. By contrast, the function rat is extrinsic throughout the function foo.

How MATLAB Resolves Extrinsic Functions During Simulation

MATLAB resolves calls to extrinsic functions — functions that do not support code generation — as follows:


Mode?
Code generation
Simulation
Does extrinsic function affect output of the calling function?
YES
Generate compiler error
NO
Elide the function call (do not generate code for function)
Send function to MATLAB for execution (do not compile)

During simulation, MATLAB generates code for the call to an extrinsic function, but does not generate the function's internal code. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see "Working with mxArrays" on page 11-17). Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, MATLAB issues a compiler error.

Working with mxArrays

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing mxArrays in variables
- Passing mxArrays to functions and returning them from functions
- Converting mxArrays to known types at run time

To use mxArrays returned by extrinsic functions in other operations, you must first convert them to known types, as described in "Converting mxArrays to Known Types" on page 11-18.

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's 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 may 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 MATLAB Classes

- "MATLAB Classes Definition for Code Generation" on page 12-2
- "Classes That Support Code Generation" on page 12-8
- "Generate Code for MATLAB Value Classes" on page 12-9
- "Generate Code for MATLAB Handle Classes and System Objects" on page 12-15
- "MATLAB Classes in Code Generation Reports" on page 12-18
- "Troubleshooting Issues with MATLAB Classes" on page 12-21

MATLAB Classes Definition for Code Generation

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

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an @-folder.	"Creating a Single, Self-Contained Class Definition File"
Restricted set of language features.	"Language Limitations" on page 12-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 12-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 12-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 12-15
Calls to base class constructor.	"Calls to Base Class Constructor" on page 12-6
Global variables containing MATLAB objects are not supported for code generation.	N/A

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`

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

  For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

  ```
  codegen ClassNameA
  ```

- If an entry-point MATLAB function has an input or output that is a MATLAB class, you cannot generate code for this function.

  For example, if function `foo` takes one input, `a`, that is a MATLAB object, you cannot generate code for `foo` by executing:

```
codegen foo -args {a}
```

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

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

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

- 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 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.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you normally would when running your code in the MATLAB environment:

- 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 generation software 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 generation software assumes that it is undefined at the beginning of the constructor. The code generation software does not assign an empty matrix as the default.

  - If the property does not have an initial value and the code generation software 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;
    ```

  - If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.

  - `coder.varsize` is not supported for class properties.

- MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns `true (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
```

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 12-9
Handle classes including user-defined System objects	"Generate Code for MATLAB Handle Classes and System Objects" on page 12-15

For more information, see:

- "Classes in the MATLAB Language"
- "MATLAB Classes Definition for Code Generation" on page 12-2

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

1 In a writable folder, create a MATLAB value class, Shape. Save the code as Shape.m.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out =  obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    methods(Abstract = true)
        getarea(obj);
    end
    methods(Static)
        function d = distanceBetweenShapes(shape1,shape2)
            xDist = abs(shape1.centerX - shape2.centerX);
            yDist = abs(shape1.centerY - shape2.centerY);
            d = sqrt(xDist^2 + yDist^2);
        end
    end
end
```

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

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

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

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

4 Write a function that uses this class.

```
function [TotalArea, Distance] =   use_shape
%#codegen
s = Square(2,1,2);
r = Rhombus(3,4,7,10);
TotalArea  = s.area + r.area;
Distance = Shape.distanceBetweenShapes(s,r);
```

5 Generate a static library for use_shape and generate a code generation report.

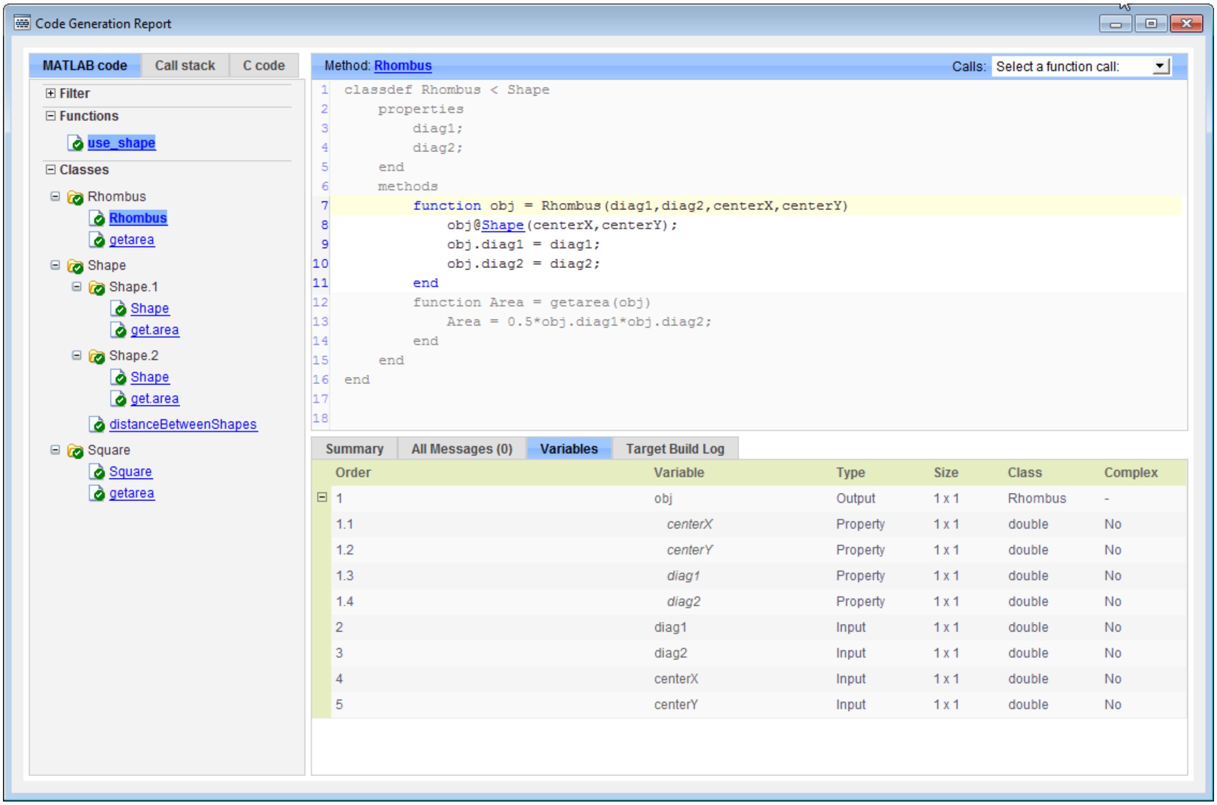
```
codegen -config:lib -report use_shape
```

codegen generates a C static library with the default name, use_shape, and supporting files in the default folder, codegen/lib/use_shape.

6 Click the **View report** link.

7 In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the complete list of properties, expand the list as shown for obj.

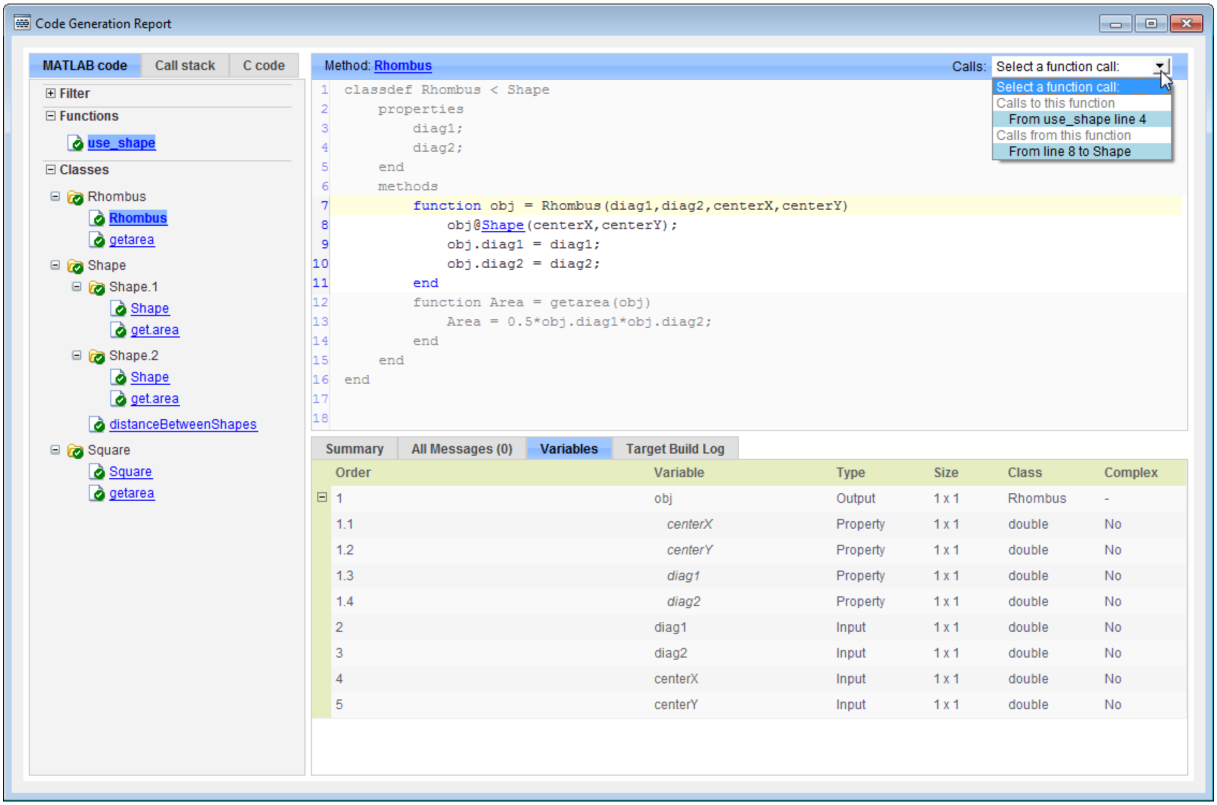


```
classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)
            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	*centerX*	Property	1 x 1	double	No
1.2	*centerY*	Property	1 x 1	double	No
1.3	*diag1*	Property	1 x 1	double	No
1.4	*diag2*	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

8 At the top right side of the report, expand the **Calls** list.

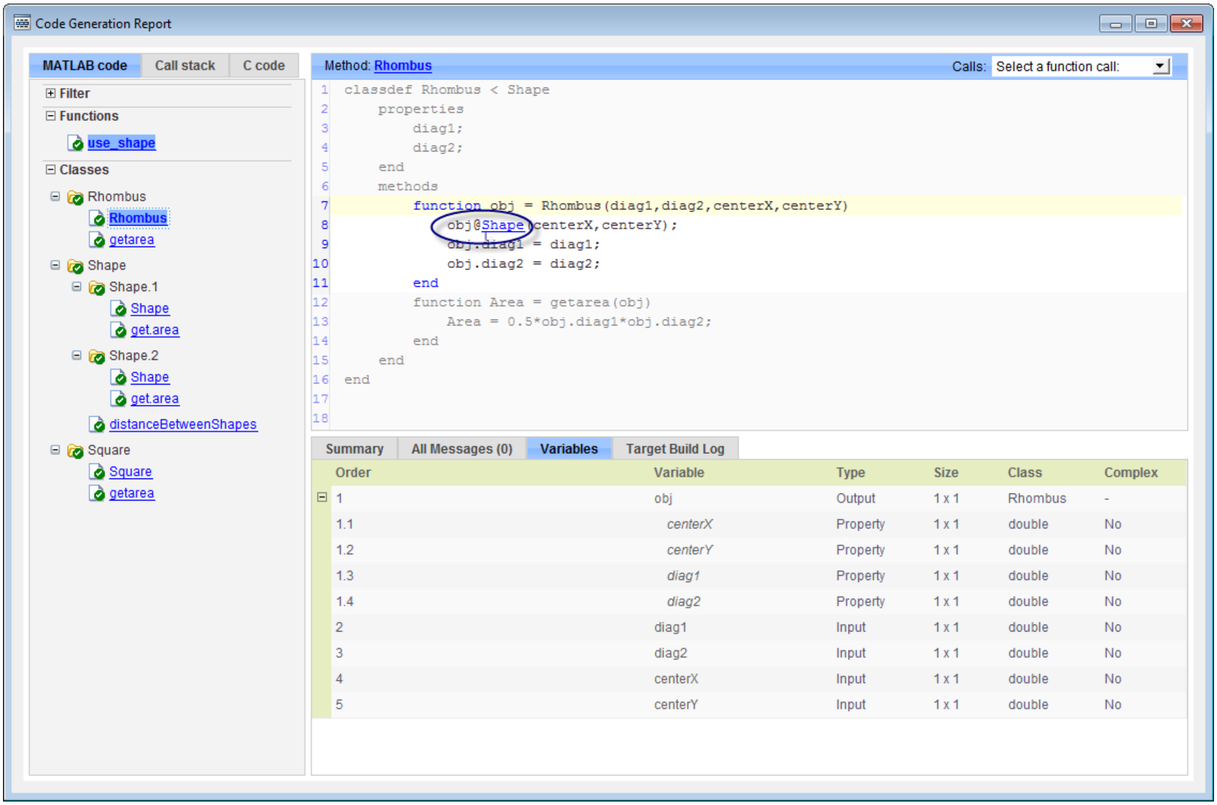
The **Calls** list shows that there is a call to the `Rhombus` constructor from `use_shape` and that this constructor calls the `Shape` constructor.



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

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1	*centerX*	Property	1 x 1	double	No
1.2	*centerY*	Property	1 x 1	double	No
1.3	*diag1*	Property	1 x 1	double	No
1.4	*diag2*	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

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



```
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	*centerX*	Property	1 x 1	double	No
1.2	*centerY*	Property	1 x 1	double	No
1.3	*diag1*	Property	1 x 1	double	No
1.4	*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 link takes you to the Shape method in the Shape class definition.

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

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.

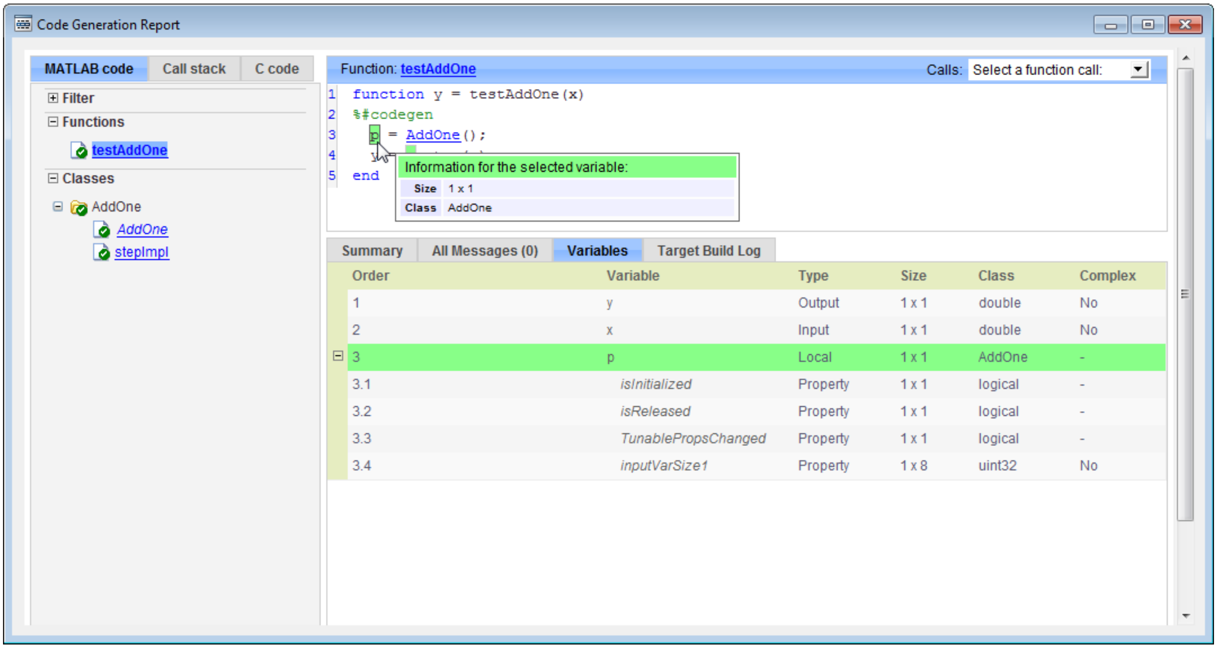
```
codegen -report testAddOne -args {0}
```

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.

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

4 Click the **View report** link.

5 In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.



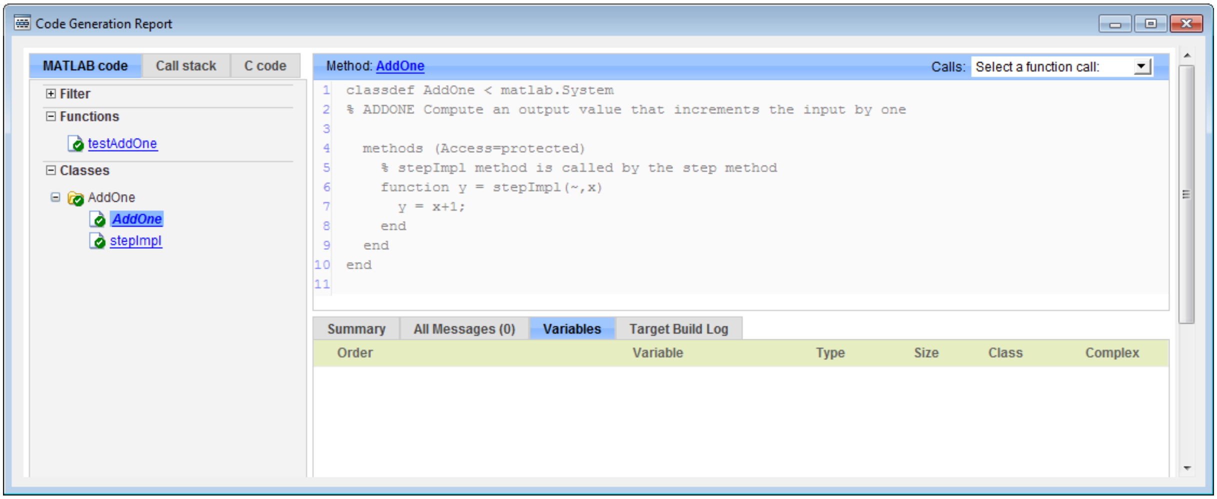
```
function y = testAddOne(x)
%#codegen
  p = AddOne();
  y
end
```

Information for the selected variable:

Size	1 x 1
Class	AddOne

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	x	Input	1 x 1	double	No
⊟ 3	p	Local	1 x 1	AddOne	-
3.1	*isInitialized*	Property	1 x 1	logical	-
3.2	*isReleased*	Property	1 x 1	logical	-
3.3	*TunablePropsChanged*	Property	1 x 1	logical	-
3.4	*inputVarSize1*	Property	1 x 8	uint32	No

6 To view the class definition, on the **Classes** panel, click `AddOne`.



Method: AddOne

Calls: Select a function call:

```
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
```

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

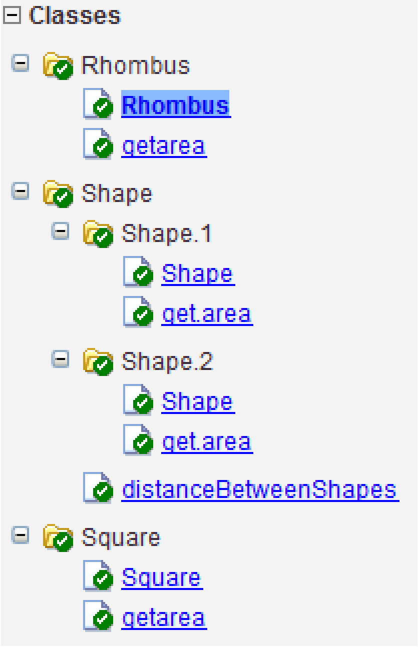
The report displays an alphabetical hierarchical list of the classes used in the your MATLAB code. For each class, you can:

- 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`.



- ⊟ Classes
  - ⊟ Rhombus
    - Rhombus
    - getarea
  - ⊟ Shape
    - ⊟ Shape.1
      - Shape
      - get.area
    - ⊟ Shape.2
      - Shape
      - get.area
    - distanceBetweenShapes
  - ⊟ Square
    - Square
    - getarea

Packages. If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see "Packages Create Namespaces".

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. Click the + symbol next to the object name to open the list.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a

MATLAB object with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

How to Generate a Code Generation Report

Add the `-report` option to your `codegen` command (requires a MATLAB Coder license)

Troubleshooting Issues with MATLAB Classes

Class *class* does not have a property with name *name*

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
  properties
    myprop
  end
  methods
    function this = MyClass
      this.myprop = MyClass2;
    end
    function y = mymethod(this)
      y = this.myprop;
    end
  end
end
```

```
classdef MyClass2 < handle
  properties
    aa
  end
end
```

You cannot generate code for function `foo`.

```
function foo

h = MyClass;

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

In this function, h.mymethod() returns a handle object of type MyClass2. In MATLAB, the assignment h.mymethod().aa = 12; changes the property of that object. Code generation does not support this assignment.

Workaround

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;
```

Defining Data for Code Generation

- "Data Definition for Code Generation" on page 13-2
- "Code Generation for Complex Data" on page 13-4
- "Code Generation for Characters" on page 13-7

Data Definition for Code Generation

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

Data	What's Different	More Information
Complex numbers	• Complexity of variables must be set at time of assignment and before first use • Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero **Note** Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic	"Code Generation for Complex Data" on page 13-4
Characters	Restricted to 8 bits of precision	"Code Generation for Characters" on page 13-7

Data	What's Different	More Information
Enumerated data	• Supports integer-based enumerated types only • Restricted use in `switch` statements and `for`-loops	"Enumerated Data"
Function handles	• Function handles must be scalar values • Same bound variable cannot reference different function handles • Cannot pass function handles to or from primary or extrinsic functions • Cannot view function handles from the debugger	"Function Handles"

Code Generation for Complex Data

In this section...

"Restrictions When Defining Complex Variables" on page 13-4

"Expressions With Complex Operands Yield Complex Results" on page 13-5

Restrictions When Defining Complex Variables

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

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

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

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

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

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

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

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

Expressions With Complex Operands Yield Complex Results

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 example:

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

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

Exceptions to this behavior are:

- Values returned by MEX functions are real when the imaginary part of the value is zero.

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

  The MEX function `foo_mex` returns the real value `1`.

  ```
  z = foo_mex
  ```

- Complex arguments to extrinsic functions are real when the imaginary part of the argument is zero.

  ```
  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 Characters

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

Defining Functions for Code Generation

- "Specify Variable Numbers of Arguments" on page 14-2
- "Supported Index Expressions" on page 14-3
- "Apply Operations to a Variable Number of Arguments" on page 14-4
- "Implement Wrapper Functions" on page 14-7
- "Pass Property/Value Pairs" on page 14-8
- "Variable Length Argument Lists for Code Generation" on page 14-10

Specify Variable Numbers of Arguments

You can use `varargin` in a function definition to specify that the function accepts a variable number of input arguments for a given input argument. You can use `varargout` in a function definition to specify that the function returns a variable number of arguments for a given output argument.

When you use `varargin` and `varargout` for code generation, there are the following limitations:

- You cannot use `varargout` in the function definition for a top-level function.
- You cannot use `varargin` in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use `varargin` to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

Common applications of `varargin` and `varargout` for code generation are to:

- "Apply Operations to a Variable Number of Arguments" on page 14-4
- "Implement Wrapper Functions" on page 14-7
- "Pass Property/Value Pairs" on page 14-8

Code generation relies on loop unrolling to produce simple and efficient code for `varargin` and `varargout`. This technique permits most common uses of `varargin` and `varargout`, but some uses are not allowed (see "Variable Length Argument Lists for Code Generation" on page 14-10).

For more information about using `varargin` and `varargout` in MATLAB functions, see Passing Variable Numbers of Arguments.

Supported Index Expressions

In MATLAB, `varargin` and `varargout` are cell arrays. Generated code does not support cell arrays, but does allow you to use the most common syntax — curly braces `{}` — for indexing into `varargin` and `varargout` arrays, as in this example:

```
%#codegen
function [x,y,z] = fcn(a,b,c)
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:length(varargin)
   varargout{i} = varargin{i};
end
```

You can use the following index expressions. The *exp* arguments must be constant expressions or depend on a loop index variable.

Expression		**Description**
`varargin` *(read only)*	`varargin{`*exp*`}`	Read the value of element *exp*
	`varargin{`*exp1*`:` *exp2*`}`	Read the values of elements *exp1* through *exp2*
	`varargin{:}`	Read the values of all elements
`varargout` *(read and write)*	`varargout{`*exp*`}`	Read or write the value of element *exp*

Note The use of `()` is not supported for indexing into `varargin` and `varargout` arrays.

Apply Operations to a Variable Number of Arguments

You can use varargin and varargout in for-loops to apply operations to a variable number of arguments. To index into varargin and varargout arrays in generated code, the value of the loop index variable must be known at compile time. Therefore, during code generation, the compiler attempts to automatically unroll these for-loops. Unrolling eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. For example, the following function automatically unrolls its for-loop in the generated code:

```
%#codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
   varargout{i} = varargin{i} * 2.54;
end
```

When to Force Loop Unrolling

To automatically unroll for-loops containing varargin and varargout expressions, the relationship between the loop index expression and the index variable must be determined at compile time.

In the following example, the function fcn cannot detect a logical relationship between the index expression j and the index variable i:

```
%#codegen
function [x,y,z] = fcn(a,b,c)

[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;
for i = 1:length(varargin)
    j = j+1;
    varargout{j} = varargin{j};
```

```
end
```

As a result, the function does not unroll the loop and generates a compilation error:

```
Nonconstant expression or empty matrix.
This expression must be constant because
its value determines the size or class of some expression.
```

To fix the problem, you can force loop unrolling by wrapping the loop header in the function `coder.unroll`, as follows:

```
%#codegen
function [x,y,z] = fcn(a,b,c)
  [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
  j = 0;
  for i = coder.unroll(1:length(varargin))
      j = j + 1;
      varargout{j} = varargin{j};
  end;
```

Using Variable Numbers of Arguments in a for-Loop

The following example multiplies a variable number of input dimensions in inches by 2.54 to convert them to centimeters:

```
%#codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
   varargout{i} = varargin{i} * 2.54;
end
```

Key Points About the Example

- `varargin` and `varargout` appear in the local function `inch_2_cm`, not in the top-level function `conv_2_metric`.
- The index into `varargin` and `varargout` is a for-loop variable

For more information, see "Variable Length Argument Lists for Code Generation" on page 14-10.

Implement Wrapper Functions

You can use varargin and varargout to write wrapper functions that accept up to 64 inputs and pass them directly to another function.

Passing Variable Numbers of Arguments from One Function to Another

The following example passes a variable number of inputs to different optimization functions, based on a specified input method:

```
%#codegen
function answer = fcn(method,a,b,c)
answer = optimize(method,a,b,c);

function answer = optimize(method,varargin)
  if strcmp(method,'simple')
    answer = simple_optimization(varargin{:});
  else
    answer = complex_optimization(varargin{:});
  end
...
```

Key Points About the Example

- You can use {:} to read all elements of varargin and pass them to another function.
- You can mix variable and fixed numbers of arguments.

For more information, see “Variable Length Argument Lists for Code Generation” on page 14-10.

Pass Property/Value Pairs

You can use varargin to pass property/value pairs in functions. However, for code generation, you must take precautions to avoid type mismatch errors when evaluating varargin array elements in a for-loop:

If	Do This:
You assign varargin array elements to local variables in the for-loop	Verify that for all pairs, the size, type, and complexity are the same for each property and the same for each value
Properties or values have different sizes, types, or complexity	Do not assign varargin array elements to local variables in a for-loop; reference the elements directly

For example, in the following function test1, the sizes of the property strings and numeric values are not the same in each pair:

```
%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        name = varargin{i};
        value = varargin{i+1};
        switch name
            case 'size'
                v = set_size(v, value);
            case 'rgb'
                v = set_color(v, value);
            otherwise
        end
    end
end
```

```
...
```

Generated code determines the size, type, and complexity of a local variable based on its first assignment. In this example, the first assignments occur in the first iteration of the for-loop:

- Defines local variable `name` with size equal to 4
- Defines local variable `value` with a size of scalar

However, in the second iteration, the size of the property string changes to 3 and the size of the numeric value changes to a vector, resulting in a type mismatch error. To avoid such errors, reference `varargin` array values directly, not through local variables, as highlighted in this code:

```
%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
            switch varargin{i}
            case 'size'
                v = set_size(v, varargin{i+1});
            case 'rgb'
                v = set_color(v, varargin{i+1});
            otherwise
        end
    end
end
...
```

Variable Length Argument Lists for Code Generation

Use variable length argument lists in top-level functions according to guidelines

When you use varargin and varargout for code generation, there are the following limitations:

- You cannot use varargout in the function definition for a top-level function.
- You cannot use varargin in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use varargin to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

A *top-level function* is:

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

For example, the following code generates compilation errors:

```
%#codegen
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
   varargout{i} = varargin{i} * 2.54;
end
```

To fix the problem, write a top-level function that specifies a fixed number of inputs and outputs. Then call inch_2_cm as an external function or local function, as in this example:

```
%#codegen
function [cmL, cmW, cmH] = conv_2_metric(inL, inW, inH)
[cmL, cmW, cmH] = inch_2_cm(inL, inW, inH);
```

```
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
   varargout{i} = varargin{i} * 2.54;
end
```

Use curly braces {} to index into the argument list

For code generation, you can use curly braces `{}`, but not parentheses `()`, to index into `varargin` and `varargout` arrays. For more information, see "Supported Index Expressions" on page 14-3.

Verify that indices can be computed at compile time

If you use an expression to index into `varargin` or `varargout`, make sure that the value of the expression can be computed at compile time. For examples, see "Apply Operations to a Variable Number of Arguments" on page 14-4.

Do not write to varargin

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

Defining MATLAB Variables for C/C++ Code Generation

- "Variables Definition for Code Generation" on page 15-2
- "Best Practices for Defining Variables for C/C++ Code Generation" on page 15-3
- "Eliminate Redundant Copies of Variables in Generated Code" on page 15-7
- "Reassignment of Variable Properties" on page 15-9
- "Define and Initialize Persistent Variables" on page 15-10
- "Reuse the Same Variable with Different Properties" on page 15-11
- "Avoid Overflows in for-Loops" on page 15-16
- "Supported Variable Types" on page 15-18

Variables Definition for Code Generation

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

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

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

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

For more information, see "Best Practices for Defining Variables for C/C++ Code Generation" on page 15-3.

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

In this section...

"Define Variables By Assignment Before Using Them" on page 15-3

"Use Caution When Reassigning Variables" on page 15-6

"Use Type Cast Operators in Variable Definitions" on page 15-6

"Define Matrices Before Assigning Indexed Variables" on page 15-6

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

Define properties this way so that the variable is defined on the required execution paths during C/C++ code generation (see Defining a Variable for Multiple Execution Paths on page 15-4).

The data that you assign to a variable can be a scalar, matrix, or structure. If your variable is a structure, define the properties of each field explicitly (see Defining Fields in a Structure on page 15-5).

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in "Eliminate Redundant Copies of Variables in Generated Code" on page 15-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see "Define and Initialize Persistent Variables" on page 15-10.

Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

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

Here, *x* is assigned only if `c > 0` and used only when `c > 0`. This code works in MATLAB, but generates a compilation error during code generation because it detects that *x* is undefined on some execution paths (when `c <= 0`),.

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

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

Defining Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
  s.a = 11;
  disp(s);
else
  s.a = 12;
  s.b = 12;
end
% Try to use s
use(s);
...
```

Here, the first part of the `if` statement uses only the field *a*, and the `else` clause uses fields *a* and *b*. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see "Structure Definition for Code Generation" on page 20-2.

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 15-9.

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 22-33.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

"When Redundant Copies Occur" on page 15-7

"How to Eliminate Redundant Copies by Defining Uninitialized Variables" on page 15-7

"Defining Uninitialized Variables" on page 15-8

When Redundant Copies Occur

During C/C++ code generation, MATLAB checks for statements that attempt to access uninitialized memory. If it detects execution paths where a variable is used but is potentially not defined, it generates a compile-time error. To prevent these errors, define 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 15-7.

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 15-11.

Define and Initialize Persistent Variables

Persistent variables are local to the function in which they are defined, but they retain their values in memory between calls to the function. To define persistent variables for C/C++ code generation, use the `persistent` statement, as in this example:

```
persistent PROD_X;
```

The definition should appear at the top of the function body, after the header and comments, but before the first use of the variable. During code generation, the value of the persistent variable is initialized to an empty matrix by default. You can assign your own value after the definition by using the `isempty` statement, as in this example:

```
function findProduct(inputvalue) %#codegen
persistent PROD_X

if isempty(PROD_X)
   PROD_X = 1;
end
PROD_X = PROD_X * inputvalue;
end
```

Reuse the Same Variable with Different Properties

In this section...

"When You Can Reuse the Same Variable with Different Properties" on page 15-11

"When You Cannot Reuse Variables" on page 15-12

"Limitations of Variable Reuse" on page 15-14

When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if MATLAB can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report (see "Use Fixed-Point Code Generation Reports" on page 9-51).

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

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

To compile this example and see how MATLAB renames the reused variable *t*, see Variable Reuse in an if Statement on page 15-12.

When You Cannot Reuse Variables

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

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

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

Variable Reuse in an if Statement

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

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

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

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

2 Compile `example1`.

For example, to generate a MEX function, enter:

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

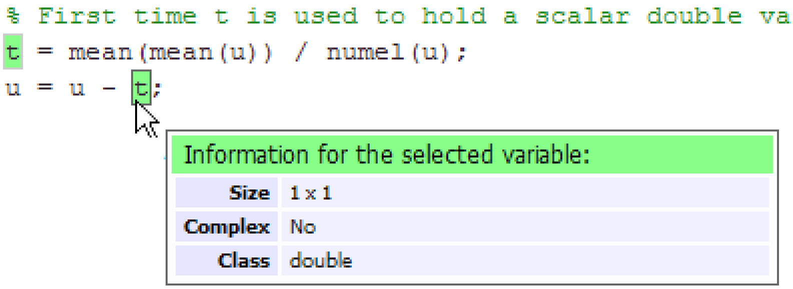
Note `codegen` requires a MATLAB Coder license.

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

3 Open the code generation report.

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

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



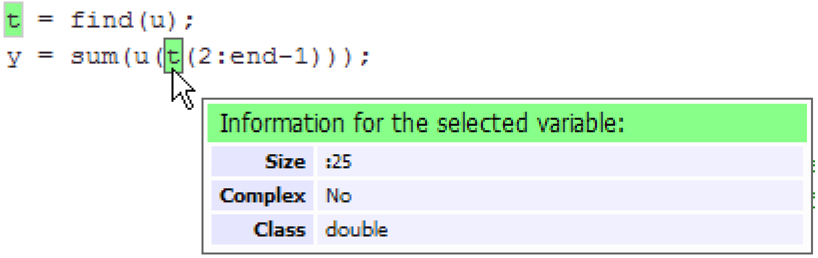
```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

Information for the selected variable:

Size	1 x 1
Complex	No
Class	double

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

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



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

Information for the selected variable:

Size	:25
Complex	No
Class	double

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

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

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

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

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

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

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

- Persistent variables.
- Global variables.
- Variables passed to C code using `coder.ref`, `coder.rref`, `coder.wref`.
- Variables whose size is set using `coder.varsize`.
- Variables whose names are controlled using `coder.cstructname`.
- The index variable of a `for`-loop when it is used inside the loop body.

- The block outputs of a MATLAB Function block in a Simulink model.
- Chart-owned variables of a MATLAB function in a Stateflow chart.

Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generation software detects that a loop variable might overflow on the last iteration of the `for`-loop, it reports an error.

To avoid this error, use the workarounds provided in the following table.

Loop conditions causing the error	Workaround

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

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

Loop conditions causing the error	Workaround

• The loop counter increments or decrements by 1
• The start value equals the minimum or maximum value of the integer type
• The end value equals the maximum or minimum value of the integer type
The loop covers the full range of the integer type.
	Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double For example, rewrite:
	`M= intmin('int16');`
	`N= intmax('int16');`
	`for k=M:N`
	` % Loop body`
	`end`
	to
	`M= intmin('int16');`
	`N= intmax('int16');`
	`for k=int32(M):int32(N)`
	` % Loop body`
	`end`

• The loop counter increments or decrements by a value not equal to 1 • On last loop iteration, the loop variable value is not equal to the end value **Note** The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.	Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.

Supported Variable Types

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

Type	Description
`char`	Character array (string)
`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 16-2
- "Which Code Generation Feature to Use" on page 16-4
- "Prerequisites for C/C++ Code Generation from MATLAB" on page 16-6
- "MATLAB Code Design Considerations for Code Generation" on page 16-7
- "Expected Differences in Behavior After Compiling MATLAB Code" on page 16-9
- "MATLAB Language Features Supported for C/C++ Code Generation" on page 16-13

When to Generate Code from MATLAB Algorithms

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

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

When Not to Generate Code from MATLAB Algorithms

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

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

To:	Use:
Deploy web-based or Windows applications	• MATLAB Builder NE • MATLAB Builder JA
Interface C code with MATLAB	MATLAB `mex` function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	`codegen` function	MATLAB Coder	Try this in "MEX Function Generation at the Command Line".
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder user interface	MATLAB Coder	Try this in "C Code Generation Using the Project Interface".
	`codegen` function	MATLAB Coder	Try this in "C Code Generation at the Command Line".
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder user interface	MATLAB Coder	See "Accelerate MATLAB Algorithms".
	`codegen` function	MATLAB Coder	
Integrate MATLAB code into Simulink	MATLAB Function block	Simulink	Try this in "Track Object Using MATLAB Code".
Speed up fixed-point MATLAB code	`fiaccel` function	Fixed-Point Designer	Learn more in "Code Acceleration and Code Generation from MATLAB" on page 9-3.
Integrate custom C code into MATLAB and generate efficient, readable code	`codegen` function	MATLAB Coder	Learn more in "Specify External File Locations".

To...	Use...	Required Product	To Explore Further...
Integrate custom C code into code generated from MATLAB	`coder.ceval` function	MATLAB Coder	Learn more in `coder.ceval`.
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder™	Learn more at `www.mathworks.com/products/slhdlcoder`.

Prerequisites for C/C++ Code Generation from MATLAB

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

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

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

- Data types

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

- Array sizing

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

- Memory

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

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

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

- Speed

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

  To improve the speed of the generated code:

  - Choose a suitable C/C++ compiler. Do not use the default compiler that MathWorks supplies with MATLAB for Windows 32-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”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” on page 22-4
- “Control Dynamic Memory Allocation” on page 9-94
- “Control Run-Time Checks” on page 9-71

Expected Differences in Behavior After Compiling MATLAB Code

In this section...

“Why Are There Differences?” on page 16-9

“Character Size” on page 16-9

“Order of Evaluation in Expressions” on page 16-9

“Termination Behavior” on page 16-10

“Size of Variable-Size N-D Arrays” on page 16-10

“Size of Empty Arrays” on page 16-11

“Floating-Point Numerical Results” on page 16-11

“NaN and Infinity Patterns” on page 16-12

“Code Generation Target” on page 16-12

“MATLAB Class Initial Values” on page 16-12

“Variable-Size Support for Code Generation” on page 16-12

Why Are There Differences?

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

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Characters” on page 13-7.

Order of Evaluation in Expressions

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

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

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

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

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

as

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

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

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they do not have side effects. As a result, the generated code may terminate even though the corresponding MATLAB code does not.

Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code, but always returns `[4 2]` in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 22-29.

Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. See “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 22-30.

Floating-Point Numerical Results

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

When computer hardware uses extended precision registers

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

For certain advanced library functions

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

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

For implementation of BLAS library functions

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

NaN and Infinity Patterns

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

Code Generation Target

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

MATLAB Class Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see "Defining Class Properties for Code Generation" on page 12-4.

Variable-Size Support for Code Generation

For incompatibilities with MATLAB in variable-size support for code generation, see:

- "Incompatibility with MATLAB for Scalar Expansion"
- "Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays"
- "Incompatibility with MATLAB in Determining Size of Empty Arrays"
- "Incompatibility with MATLAB in Vector-Vector Indexing"
- "Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation"

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

MATLAB supports the following language features in generated code:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see "Variable-Size Data Definition for Code Generation" on page 22-3)
- Subscripting (see "Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" on page 22-33)
- Complex numbers (see "Code Generation for Complex Data" on page 13-4)
- Numeric classes (see "Supported Variable Types" on page 15-18)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see "Code Acceleration and Code Generation from MATLAB" on page 9-3)
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see "Define and Initialize Persistent Variables" on page 15-10)
- Global variables.
- Structures
- Characters (see "Code Generation for Characters" on page 13-7)
- Function handles
- Frames
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions
- MATLAB classes

- Ability to call functions (see "Resolution of Function Calls in MATLAB Generated Code" on page 11-2)

MATLAB Language Features Not Supported for C/C++ Code Generation

MATLAB does not support the following features in generated code:

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

Code Generation for Enumerated Data

- "Enumerated Data Definition for Code Generation" on page 17-2
- "Enumerated Types Supported for Code Generation" on page 17-3
- "When to Use Enumerated Data for Code Generation" on page 17-5
- "Generate Code for Enumerated Data from MATLAB Algorithms" on page 17-6
- "Define Enumerated Data for Code Generation" on page 17-8
- "Instantiate Enumerated Types for Code Generation" on page 17-10
- "Operations on Enumerated Data Allowed for Code Generation" on page 17-11
- "Include Enumerated Data in Control Flow Statements" on page 17-14
- "Customize Enumerated Types Based on int32" on page 17-20
- "Control Names of Enumerated Type Values in Generated Code" on page 17-26
- "Change and Reload Enumerated Data Types" on page 17-28
- "Restrictions on Use of Enumerated Data in `for`-Loops" on page 17-29
- "Toolbox Functions That Support Enumerated Types for Code Generation" on page 17-30

Enumerated Data Definition for Code Generation

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

What's Different	More Information
Supports integer-based enumerated types only	"Enumerated Types Supported for Code Generation" on page 17-3
Name of each enumerated data type must be unique	"Naming Enumerated Types for Code Generation" on page 17-9
Each enumerated data type must be defined in a separate file on the MATLAB path	"Define Enumerated Data for Code Generation" on page 17-8 and "How to Generate Code for Enumerated Data" on page 17-6
Restricted set of operations	"Operations on Enumerated Data Allowed for Code Generation" on page 17-11
Restricted use in `for`-loops	"Restrictions on Use of Enumerated Data in `for`-Loops" on page 17-29

Enumerated Types Supported for Code Generation

Enumerated Type Based on int32

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

Syntax

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

Example

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

In this example, the statement classdef(Enumeration) PrimaryColors < int32 means that the enumerated type PrimaryColors is based on the built-in type int32. As such, PrimaryColors inherits the characteristics of the int32 type, as well as defining its own unique characteristics. For example, PrimaryColors is restricted to three enumerated values:

Enumerated Value	Enumerated Name	Underlying Numeric Value
Red(1)	Red	1
Blue(2)	Blue	2
Yellow(4)	Yellow	4

How to Use

Define enumerated data in MATLAB code and compile the source file. For example, to generate C/C++ code from your MATLAB source, you can use codegen, as described in “Generate Code for Enumerated Data from MATLAB Algorithms” on page 17-6.

Note codegen requires a MATLAB Coder license.

When to Use Enumerated Data for Code Generation

You can use enumerated types to represent program states and to control program logic, especially when you need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow more robust error checking than integers or strings

  For example, if you mistype the name of an element in the enumerated type, you get a compile-time error that the element does not belong to the set of allowable values.
- Produce more efficient code than strings

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

Generate Code for Enumerated Data from MATLAB Algorithms

Step	Action	How?
1	Define an enumerated data type that inherits from `int32`.	See "Define Enumerated Data for Code Generation" on page 17-8.
2	Instantiate the enumerated type in your MATLAB algorithm.	See "Instantiate Enumerated Types for Code Generation" on page 17-10.
3	Compile the function with `codegen`.	See "How to Generate Code for Enumerated Data" on page 17-6.

This workflow requires a MATLAB Coder license.

How to Generate Code for Enumerated Data

Use the command `codegen` to generate MEX, C, or C++ code from the MATLAB algorithm that contains the enumerated data (requires a MATLAB Coder license). Each enumerated data type must be defined on the MATLAB path in a separate file as a class derived from the built-in type `int32`. See "Define Enumerated Data for Code Generation" on page 17-8.

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

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

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

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

You should get the following result:

```
ans =

    RED
```

Define Enumerated Data for Code Generation

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

1. Create a class definition file.

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

2. Enter the class definition as follows:

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

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

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

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

3. Define enumerated values in an `enumeration` section as follows:

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

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

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

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

4 Save the file on the MATLAB path.

The name of the file must match the name of the enumerated data type. The match is case sensitive.

To add a folder to the MATLAB search path, type `addpath` *pathname* at the MATLAB command prompt. For more information, see "What Is the MATLAB Search Path?", `addpath`, and `savepath`.

For examples of enumerated data type definitions, see "Define Enumerated Data for Code Generation" on page 17-8.

Naming Enumerated Types for Code Generation

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

For example, you cannot name an enumerated data type `mode` because MATLAB for code generation provides a toolbox function of the same name.

For a list of toolbox functions supported for code generation, see "Functions Supported for C/C++ Code Generation — Alphabetical List" on page 21-2.

Instantiate Enumerated Types for Code Generation

To instantiate an enumerated type for code generation from MATLAB algorithms, use dot notation to specify `ClassName.EnumName`. For an example, see "Include Enumerated Data in Control Flow Statements" on page 17-14.

Operations on Enumerated Data Allowed for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples are based on the definitions of the enumeration type `LEDcolor` described in "Class Definition: LEDcolor" on page 17-14.

Assignment Operator, =

Example	Result

`xon = LEDcolor.GREEN`
`xoff = LEDcolor.RED`
	`xon =`
	`GREEN`
	`xoff =`
	`RED`

Relational Operators, < > <= >= == ~=

Example	Result

`xon == xoff`	`ans =`
	`0`

`xon <= xoff`	`ans =`
	`1`

`xon > xoff`	`ans =`
	`0`

Cast Operation

Example	Result
double(LEDcolor.RED)	ans = 2

z = 2
y = LEDcolor(z)
	z =
	
	2
	
	y =
	
	RED

Indexing Operation

Example	Result

m = [1 2]
n = LEDcolor(m)
p = n(LEDcolor.GREEN)
	m =
	
	1 2
	
	n =
	
	GREEN RED
	
	p =
	
	GREEN

Control Flow Statements: if, switch, while

Statement	Example	Executable Example

`if`	`if state == sysMode.ON`
	`    led = LEDcolor.GREEN;`
	`else`
	`    led = LEDcolor.RED;`
	`end`
		“`if` Statement with Enumerated Data Types” on page 17-14

`switch`	`switch button`
	`    case VCRButton.Stop`
	`        state = VCRState.Stop;`
	`    case VCRButton.PlayOrPause`
	`        state = VCRState.Play;`
	`    case VCRButton.Next`
	`        state = VCRState.Forward;`
	`    case VCRButton.Previous`
	`        state = VCRState.Rewind;`
	`    otherwise`
	`         state = VCRState.Stop;`
	`end`
		“`switch` Statement with Enumerated Data Types” on page 17-15

`while`	`while state ~= State.Ready`
	`    switch state`
	`        case State.Standby`
	`            initialize();`
	`            state = State.Boot;`
	`        case State.Boot`
	`            boot();`
	`            state = State.Ready;`
	`    end`
	`end`
		“`while` Statement with Enumerated Data Types” on page 17-18

Include Enumerated Data in Control Flow Statements

The following control statements work with enumerated operands in generated code. However, there are restrictions (see "Restrictions on Use of Enumerated Data in for-Loops" on page 17-29).

if Statement with Enumerated Data Types

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

Class Definition: sysMode

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

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

Class Definition: LEDcolor

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

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

MATLAB Function: displayState

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

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

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

Build and Test a MEX Function for `displayState`

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

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

2. Test the function. For example,

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

   You should get the following result:

   ```
   ans =

       RED
   ```

`switch` Statement with Enumerated Data Types

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

Class Definition: VCRState

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
        Play(2),
        Forward(3),
```

```
        Rewind(4)
    end
end
```

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

Class Definition: VCRButton

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

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

MATLAB Function: VCR

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

```
function s = VCR(button)
%#codegen

persistent state

if isempty(state)
    state = VCRState.Stop;
end

switch state
    case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}
        state = handleDefault(button);
    case VCRState.Play
        switch button
```

```
            case VCRButton.PlayOrPause, state = VCRState.Pause;
            otherwise, state = handleDefault(button);
        end
    case VCRState.Pause
        switch button
            case VCRButton.PlayOrPause, state = VCRState.Play;
            otherwise, state = handleDefault(button);
        end
end
s = state;

function state = handleDefault(button)
switch button
    case VCRButton.Stop, state = VCRState.Stop;
    case VCRButton.PlayOrPause, state = VCRState.Play;
    case VCRButton.Next, state = VCRState.Forward;
    case VCRButton.Previous, state = VCRState.Rewind;
    otherwise, state = VCRState.Stop;
end
```

Build and Test a MEX Function for VCR

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

   ```
   codegen -args {VCRButton.Stop}  VCR
   ```

2. Test the function. For example,

   ```
   s = VCR(VCRButton.Stop)
   ```

   You should get the following result:

   ```
   s =

       Stop
   ```

while Statement with Enumerated Data Types

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

Class Definition: State

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

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

MATLAB Function: Setup

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

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

state = initState;

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

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

```
s = state;

function initialize()
% Perform initialization.

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

Build and Test a MEX Executable for `Setup`

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

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

2 Test the function. For example,

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

   You should get the following result:

   ```
   s =

       Ready
   ```

Customize Enumerated Types Based on int32

About Customizing Enumerated Types

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

Enumerated class definitions can include an optional methods section. You can override the following methods to customize the behavior of an enumerated type. To override a method, include a customized version of the method in the methods section in the enumerated class definition. If you do not want to override the inherited methods, omit the methods section.

Method	Description	Default Value Returned or Specified	When to Use
`addClassNameToEnumNames`	Specifies whether the class name becomes a prefix in the generated code.	`true` — prefix is used	If you do not want the class name to become a prefix in the generated code, override this method to set the return value to `false`. See "Control Names of Enumerated Type Values in Generated Code" on page 17-26.
`getDefaultValue`	Returns the default enumerated value.	`''`	If you want the default value for the enumerated type to be something other than the first value listed in the enumerated class definition, override this method to specify a default value. See "Specify a Default Enumerated Value" on page 17-22.

Method	Description	Default Value Returned or Specified	When to Use
getHeaderFile	Specifies the file in which the enumerated class is defined for code generation.	''	If you want to use an enumerated class definition that is specified in a custom header file, override this method to return the path to this header file. In this case, the code generation software does not generate the class definition. See "Specify a Header File" on page 17-23

Specify a Default Enumerated Value

The code generation software and related generated code use an enumerated data type's default value when you do not provide an initial value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumerated class definition. To specify a different default value, add your own getDefaultValue method to the methods section. The following code shows a shell for the getDefaultValue method:

```
function retVal = getDefaultValue()
% GETDEFAULTVALUE  Returns the default enumerated value.
%   This value must be an instance of the enumerated class.
%   If this method is not defined, the first enumerated value is used.
  retVal = ThisClass.EnumName;
end
```

To customize this method, provide a value for ThisClass.EnumName that specifies the desired default.ThisClass must be the name of the class within which the method exists. EnumName must be the name of an enumerated value defined in that class. For example:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
    methods (Static)
    function y = getDefaultValue()
      y = LEDcolor.RED;
    end
  end
end
```

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

Specify a Header File

To prevent the declaration of an enumerated type from being embedded in the generated code, allowing you to provide the declaration in an external file, include the following method in the enumerated class's methods section:

```
function y = getHeaderFile()
% GETHEADERFILE  File where type is defined for generated code.
%   If specified, this file is #included where required in the code.
%   Otherwise, the type is written out in the generated code.
y = 'filename';
end
```

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

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

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

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

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

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

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

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

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

3 Generate a library for the function displayState that takes one input of enumerated data type sysMode.

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

codegen generates a C static library with the default name, displayState, and supporting files in the default folder, codegen/lib/displayState.

4 Click the **View Report** link.

5 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The header file contains a `#include` statement for the external header file.

```
#include "my_LEDcolor.h"
```

It does not include a declaration for the enumerated class.

Control Names of Enumerated Type Values in Generated Code

This example shows how to control the name of enumerated type values in code generated by MATLAB Coder. (Requires a MATLAB Coder license.) The example uses the enumerated data type definitions and function displayState described in "Include Enumerated Data in Control Flow Statements" on page 17-14.

1 Generate a library for the function displayState that takes one input of enumerated data type sysMode.

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

codegen generates a C static library with the default name, displayState, and supporting files in the default folder, codegen/lib/displayState.

2 Click the **View Report** link.

3 In the report, on the **C Code** tab, click the link to the displayState_types.h file.

The report displays the header file containing the enumerated data type definition.

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

The enumerated value names include the class name prefix LEDcolor_.

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

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

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

5 Clear existing class instances:

```
clear classes
```

6 Generate code again.

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

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

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

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

For more information, see:

- `codegen`
- "Include Enumerated Data in Control Flow Statements" on page 17-14 for a description of the example function `displayState` and its enumerated type definitions

Change and Reload Enumerated Data Types

You can change the definition of an enumerated data type by editing and saving the file that contains the definition. You do not need to inform MATLAB that a class definition has changed. MATLAB automatically reads the modified definition when you save the file. However, the class definition changes do not take full effect if class instances (enumerated values) exist that reflect the previous class definition. Such instances might exist in the base workspace or might be cached. The following table explains options for removing instances of an enumerated data type from the base workspace and cache.

If In Base Workspace...	If In Cache...
Do one of the following: • Locate and delete specific obsolete instances. • Delete the classes from the workspace by using the `clear classes` command. For more information, see `clear`.	• Clear MEX functions that are caching instances of the class.

Restrictions on Use of Enumerated Data in for-Loops

Do not use enumerated data as the loop counter variable in for-loops

To iterate over a range of enumerated data with consecutive values, you can cast the enumerated data to int32 in the loop counter.

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

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

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

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

Toolbox Functions That Support Enumerated Types for Code Generation

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

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

- `squeeze`

Code Generation for Function Handles

- “Function Handle Definition for Code Generation” on page 18-2
- “Define and Pass Function Handles for Code Acceleration” on page 18-3
- “Function Handle Limitations for Code Generation” on page 18-5

Function Handle Definition for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently. You can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported for code generation (see "Functions Supported for C/C++ Code Generation — Alphabetical List" on page 21-2)

  Note You cannot define handles that reference extrinsic MATLAB functions.

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

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in "Function Handle Limitations for Code Generation" on page 18-5

Define and Pass Function Handles for Code Acceleration

The following code example shows how to define and call function handles for code acceleration.

```
function [y1, y2] = addval(m)
%#codegen

 disp(m);

  % Pass function handle to addone
  %  to add one to each element of m
  y1 = map(@addone, m);
  disp(y1);

  % Pass function handle to addtwo
  %  to add two to each element of m
  y2 = map(@addtwo, m);
  disp(y2);

  function y = map(f,m)
    y = m;
    for i = 1:numel(y)
       y(i) = f(y(i));
    end

  function y = addone(u)
  y = u + 1;

  function y = addtwo(u)
  y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

You can use the function `fiaccel` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

1. At the MATLAB command prompt, define and initialize a 3-by-3 matrix:

   ```
   m = fi(magic(3));
   ```

2. Use `fiaccel` to compile the function to a MEX executable:

   ```
   fiaccel addval -args {m}
   ```

3. Execute the function:

   ```
   [y1, y2] = addval_mex(m);
        8     1     6
        3     5     7
        4     9     2

             DataTypeMode: Fixed-point: binary point scaling
               Signedness: Signed
               WordLength: 16
           FractionLength: 11
        9     2     7
        4     6     8
        5    10     3

             DataTypeMode: Fixed-point: binary point scaling
               Signedness: Signed
               WordLength: 16
           FractionLength: 11
       10     3     8
        5     7     9
        6    11     4

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

Function Handle Limitations for Code Generation

Function handles must be scalar values.

You cannot store function handles in matrices or structures.

You cannot use the same bound variable to reference different function handles.

After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example

```
%Incorrect code
...
x = @plus;
x = @minus;
...
```

This code produces a compilation error.

You cannot pass function handles to or from extrinsic functions.

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions. For more information, see “Declaring MATLAB Functions as Extrinsic Functions” on page 11-12

You cannot pass function handles to or from primary functions.

You cannot pass function handles as inputs to or outputs from primary functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as primary inputs. `plotFcn` attempts to call the function referenced by the fhandle with the input data and plot the results. However, this code generates a compilation error, indicating that the function isa does not recognize 'function_handle' as a class name when called inside a MATLAB function to specify properties of primary inputs.

You cannot view function handles from the debugger

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

Generate Efficient and Reusable Code

- “Optimization Strategies” on page 19-3
- “Modularize MATLAB Code” on page 19-6
- “Eliminate Redundant Copies of Function Inputs” on page 19-7
- “Inline Code” on page 19-9
- “Control Inlining Using Configuration Object” on page 19-11
- “Fold Function Calls into Constants” on page 19-14
- “Control Stack Space Usage” on page 19-16
- “Stack Allocation and Performance” on page 19-17
- “Rewrite Logical Array Indexing as a Loop” on page 19-18
- “Dynamic Memory Allocation and Performance” on page 19-19
- “Minimize Dynamic Memory Allocation” on page 19-21
- “Provide Maximum Size for Variable-size Arrays” on page 19-22
- “Disable Dynamic Memory Allocation During Code Generation” on page 19-29
- “Set Dynamic Memory Allocation Threshold” on page 19-30
- “Excluding Unused Paths from Generated Code” on page 19-33
- “Prevent Code Generation for Unused Execution Paths” on page 19-34
- “Generate Code with Parallel for-loops (parfor)” on page 19-37
- “Minimize Redundant Operations in Loops” on page 19-39

- “Unroll for-Loops” on page 19-41
- “Support for Integer Overflow and Non-Finites” on page 19-44
- “Integrate Custom Code” on page 19-46
- “MATLAB® Coder™ Optimizations in Generated Code” on page 19-52
- “Generate Reusable Code” on page 19-56

Optimization Strategies

MATLAB Coder introduces certain optimizations when generating `C/C++` code or `MEX` functions from your MATLAB code. For more information, see "MATLAB Coder Optimizations in Generated Code".

To optimize your generated code further, you can:

- Adapt your MATLAB code.
- Control code generation using the configuration object from the command-line or the Project Settings dialog box.

To optimize the execution speed of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have `for`-loops whose iterations are independent of each other.	"Generate Code with Parallel for-loops (parfor)"
You have variable-size arrays in your MATLAB code.	"Minimize Dynamic Memory Allocation"
You have multiple variable-size arrays in your MATLAB code. You want dynamical memory allocation for larger arrays and static allocation for smaller ones.	"Set Dynamic Memory Allocation Threshold"
You want your generated function to be called by reference.	"Eliminate Redundant Copies of Function Inputs"
You are calling small functions in your MATLAB code.	"Inline Code"
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining Using Configuration Object"
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants"

Condition	Action
You have loop operations in your MATLAB code that do not depend on the loop index.	"Minimize Redundant Operations in Loops"
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	"Disable Support for Integer Overflow"
You know beforehand that `Inf`-s and `NaN`-s will not occur during execution of your generated code.	"Disable Support for Non-Finites"
You have `for`-loops with few iterations.	"Unroll for-Loops"
You already have legacy C/C++ code optimized for your target environment.	"Integrate Custom Code"

To optimize the memory usage of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have `if/else/elseif` statements or `switch/case/otherwise` statements in your MATLAB code. You do not require some branches of the statements in your generated code.	"Prevent Code Generation for Unused Execution Paths"
You have logical array indexing in your MATLAB code. For more information, see "Using Logicals in Array Indexing".	"Rewrite Logical Array Indexing as a Loop"
You want your generated function to be called by reference.	"Eliminate Redundant Copies of Function Inputs"
You have limited stack space for your generated code.	"Control Stack Space Usage"
You are calling small functions in your MATLAB code.	"Inline Code"

Condition	Action
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining Using Configuration Object"
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants"
You have loop operations in your MATLAB code that do not depend on the loop index.	"Minimize Redundant Operations in Loops"
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	"Disable Support for Integer Overflow"
You know beforehand that `Inf`-s and `NaN`-s will not occur during execution of your generated code.	"Disable Support for Non-Finites"

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++ codes for the individual sections. You can reuse these codes 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, MATLAB passes the variable by reference in the generated code 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 you rewrite function `foo` without the optimization:

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

MATLAB generates code that 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
```

Inline Code

MATLAB uses internal heuristics to determine whether or not to inline functions in the generated code. You can use the coder.inline directive to fine-tune these heuristics for individual functions. For more information, see coder.inline.

In this section...

"Prevent Function Inlining" on page 19-9

"Use Inlining in Control Flow Statements" on page 19-9

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 Using Configuration Object"

Control Inlining Using Configuration Object

This example shows how to control inlining behavior using the `codegen` configuration object. Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at a large number of 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 19-11

"Control Size of Functions After Inlining" on page 19-12

"Control Stack Size Limit on Inlined Functions" on page 19-12

Control Size of Functions Inlined

You can control the maximum size of functions that can be inlined from the Project Settings dialog box or the command line. 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.

- 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 control the maximum size of functions after inlining from the Project Settings dialog box or the command line. 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.

- 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 can be accomodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can control the stack size limit on inlined functions from the Project Settings dialog box or the command line.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline stack limit**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

  ```
  cfg = coder.config('lib');
  cfg.InlineStackLimit = 2000;
  ```

Generate code using this configuration object.

Related Examples

- “Inline Code”

Fold Function Calls into Constants

This example shows how to specify constants in generated code using `coder.const`. The code generation software 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 generation software generates 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 generation software 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 used by the generated code. Set the maximum stack usage when:

- You have limited stack space, for instance, in case of embedded targets.
- Your C compiler reports a run-time stack overflow.

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 can be accommodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Control Stack Space Usage Using Project Interface

1. On the **Build** tab **Settings** pane, set the **Output type** to `C/C++ Static Library`, `C/C++ Dynamic Library`, or `C/C++ Executable` (depending on your requirements).

2. Click the **More settings** link to open the **Project Settings** dialog box.

3. On the **Memory** tab, set the field, **Stack usage max**, to the value that you want.

Control Stack Space Usage from 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;
   ```

Concepts

- “Stack Allocation and Performance”

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”.

Rewrite Logical Array Indexing as a Loop

Rewriting logical array indexing as a loop can optimize the generated code for both speed and readability. For more information on logical array indexing, see "Using Logicals in Array Indexing".

For example, the MATLAB function, `foo`, uses logical array indexing.

```
function x = foo(x,N)  %#codegen
assert(all(size(x) == [1 100]))
x(x>N) = N;
```

The generated C code for this function is not very efficient. Rewrite the MATLAB code to use a loop instead of logical indexing:

```
function x = foo_rewrite(x,N)  %#codegen
assert(all(size(x) == [1 100]))
for ii=1:numel(x)
    if x(ii) > N
        x(ii) = N;
    end
end
```

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. For more information on dynamic memory allocation, see "Bounded Versus Unbounded Variable-Size Data".

When Dynamic Memory Allocation Occurs

Dynamic memory allocation occurs when the code generation software 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 might 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 possible 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 19-21.

Minimize Dynamic Memory Allocation

When possible, you should minimize dynamic memory allocation since it leads to slower execution of generated code. Dynamic memory allocation occurs when the code generation software cannot find upper bounds for variable-size arrays.

You can avoid dynamic memory allocation of a variable-size array if you know its maximum possible size. To do so, follow these steps:

1. "Provide Maximum Size for Variable-size Arrays" on page 19-22.

2. Depending on your requirements, do one of the following:
   - "Disable Dynamic Memory Allocation During Code Generation" on page 19-29.
   - "Set Dynamic Memory Allocation Threshold"

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.

Concepts

- "Dynamic Memory Allocation and Performance"

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 generation software to determine a maximum size for the array. An array of size equal to this maximum size is then defined in the generated code (static memory allocation).

  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 ensures that `y` is assigned an array of size 25 in the generated code. 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 size of an array, `y`.

The following code restricts the product of M and N to 25. It then uses this product to specify 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 ensures that y is assigned an array of size 25 in the generated code.

Alternatively, if you restrict M and N individually, it leads to dynamic memory allocation:

```
function y = array_init_from_prod (M,N)
   assert(M < 5);
   assert(N < 5);
   size=M*N;
   y=zeros(1,size);
```

This code causes dynamic memory allocation because M and N can both have unbounded negative values. Therefore, their product can be unbounded and positive even though, individually, their positive values are bounded.

Tip Place the assert statement on a variable immediately before it is used to specify array size.

Tip You can use assert statements to restrict array sizes in most cases. When expanding an array inside a loop, this strategy does not work if the number of loop runs is known only at run time.

-

Restrict Concatenations in a Loop Using coder.varsize with Upper Bounds

You can expand arrays beyond their initial size by concatenation. When you concatenate additional elements inside a loop, there are two syntax rules for expanding arrays.

1

Array size during initialization is not a compile-time constant

If the size of an array during initialization is not a compile-time constant, you can expand it by concatenating additional elements:

```
function out=ExpandArray(in) % Expand an array by five elements
  out = zeros(1,in);
  for i=1:5
     out = [out 0];
  end
```

2

Array size during initialization is a compile-time constant

Before concatenating elements, you have to declare the array as variable-size using `coder.varsize`:

```
function out=ExpandArray() % Expand an array by five elements
  out = zeros(1,5);
  coder.varsize('out');
  for i=1:5
     out = [out 0];
  end
```

Either case leads to dynamic memory allocation. To prevent dynamic memory allocation in such cases, use `coder.varsize` with explicit upper bounds. This example shows how to use `coder.varsize` with explicit upper bounds:

Restrict Concatenations Using coder.varsize with Upper Bounds

1 Define a function, `RunningAverage`, that calculates the running average of an `N`-element subset of an array:

```
  function avg=RunningAverage(N)

% Array whose elements are to be averaged
  NumArray=[1 6 8 2 5 3];

% Initialize average:
% These will also be the first two elements of the function output
  avg=[0 0];

% Place a bound on the argument
  coder.varsize('avg',[1 8]);

% Loop to calculate running average
  for i=1:N
    s=0;
    s=s+sum(NumArray(1:i));
    avg=[avg s/i];
  % Increase the size of avg as required by concatenation
  end
```

The output, `avg`, is an array that you can expand as required to accommodate the running averages. As a new running average is calculated, it is added to the array `avg` through concatenation, thereby expanding the array.

Because the maximum possible number of running averages is equal to the number of elements in `NumArray`, you can supply an explicit upper bound for `avg` in the `coder.varsize` statement. In this example, the upper bound is 8 (the two initial elements plus the six elements of `NumArray`).

2 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, avg is assigned an array of size 8 (static memory allocation). The function definition for RunningAverage appears as follows (using built-in C types):

```
void RunningAverage (double N, double avg_data[8], int avg_size[2])
```

3. By contrast, if you remove the explicit upper bound, the generated code dynamically allocates avg.

   Replace the statement

   ```
   coder.varsize('avg',[1 8]);
   ```

   with:

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

4. Generate code for RunningAverage with input argument of type double:

   ```
   codegen -config:lib -report RunningAverage -args 2
   ```

   In the generated code, avg is assigned a pointer to an array, thereby allowing dynamic memory allocation. The function definition for RunningAverage appears as follows (using built-in C types):

   ```
   void Test(double N, emxArray_real_T *avg)
   ```

   Note Dynamic memory allocation also occurs if you precede coder.varsize('avg') with the following assert statement:

   ```
   assert(N < 6);
   ```

   The assert statement does not restrict the number of concatenations within the loop.

-

Constrain Array Size When Rearranging a Matrix

The statement `out = reshape(in,m,n,...)` takes an array, `in`, as an argument and returns array, `out`, having the same elements as `in`, but reshaped as an `m`-by-`n`-by-`...` matrix. If one of the size variables `m,n,....` is not a compile-time constant, then dynamic memory allocation of `out` takes place.

To avoid dynamic memory allocation, use an `assert` statement before the `reshape` statement to restrict the size variables `m,n,...` to `numel(in)`. This example shows how to use an `assert` statement before a `reshape` statement:

Rearrange a Matrix into Given Number of Rows

1 Define a function, `ReshapeMatrix`, which takes an input variable, `N`, and reshapes a matrix, `mat`, to have `N` rows:

```
  function [out1,out2] = ReshapeMatrix(N)

  mat = [1 2 3 4 5; 4 5 6 7 8]
% Since mat has 10 elements, N must be a factor of 10
% to pass as argument to reshape

  out1 = reshape(mat,N,[]);
% N is not restricted

  assert(N < numel(mat));
% N is restricted to number of elements in mat
  out2 = reshape(mat,N,[]);
```

2 Generate code for `ReshapeArray` using the `codegen` command (the input argument does not have to be a factor of 10):

```
codegen -config:lib -report ReshapeArray -args 3
```

While `out1` is dynamically allocated, `out2` is assigned an array with size 100 (=10 X 10) in the generated code.

Tip If your system has limited memory, do not use the `assert` statement in this way. For an `n`-element matrix, the `assert` statement creates an `n`-by-`n` matrix, which might be large.

Related Examples

- "Minimize Dynamic Memory Allocation"
- "Disable Dynamic Memory Allocation During Code Generation"
- "Set Dynamic Memory Allocation Threshold"

Concepts

- "Dynamic Memory Allocation and Performance"

Disable Dynamic Memory Allocation During Code Generation

Disabling dynamic memory allocation during code generation leads to faster execution of generated code. You can disable dynamic memory allocation explicitly from the project settings dialog box or the command line.

To disable dynamic memory allocation in the Project Settings box :

1. On the MATLAB Coder project **Build** tab, click **More settings**.
2. In the **Project Settings** dialog box **Memory** tab, under **Enable variable-sizing**, set **Dynamic memory allocation** to `Never`.

To disable dynamic memory allocation from 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';
   ```

Disabling dynamic memory allocation leads to a code generation error if a variable-size array in the MATLAB code does not have a maximum upper bound. Therefore, you can also use this feature to identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the ones that are dynamically allocated in the generated code.

Related Examples

- “Minimize Dynamic Memory Allocation”
- “Provide Maximum Size for Variable-size Arrays”
- “Set Dynamic Memory Allocation Threshold”

Concepts

- “Dynamic Memory Allocation and Performance”

Set Dynamic Memory Allocation Threshold

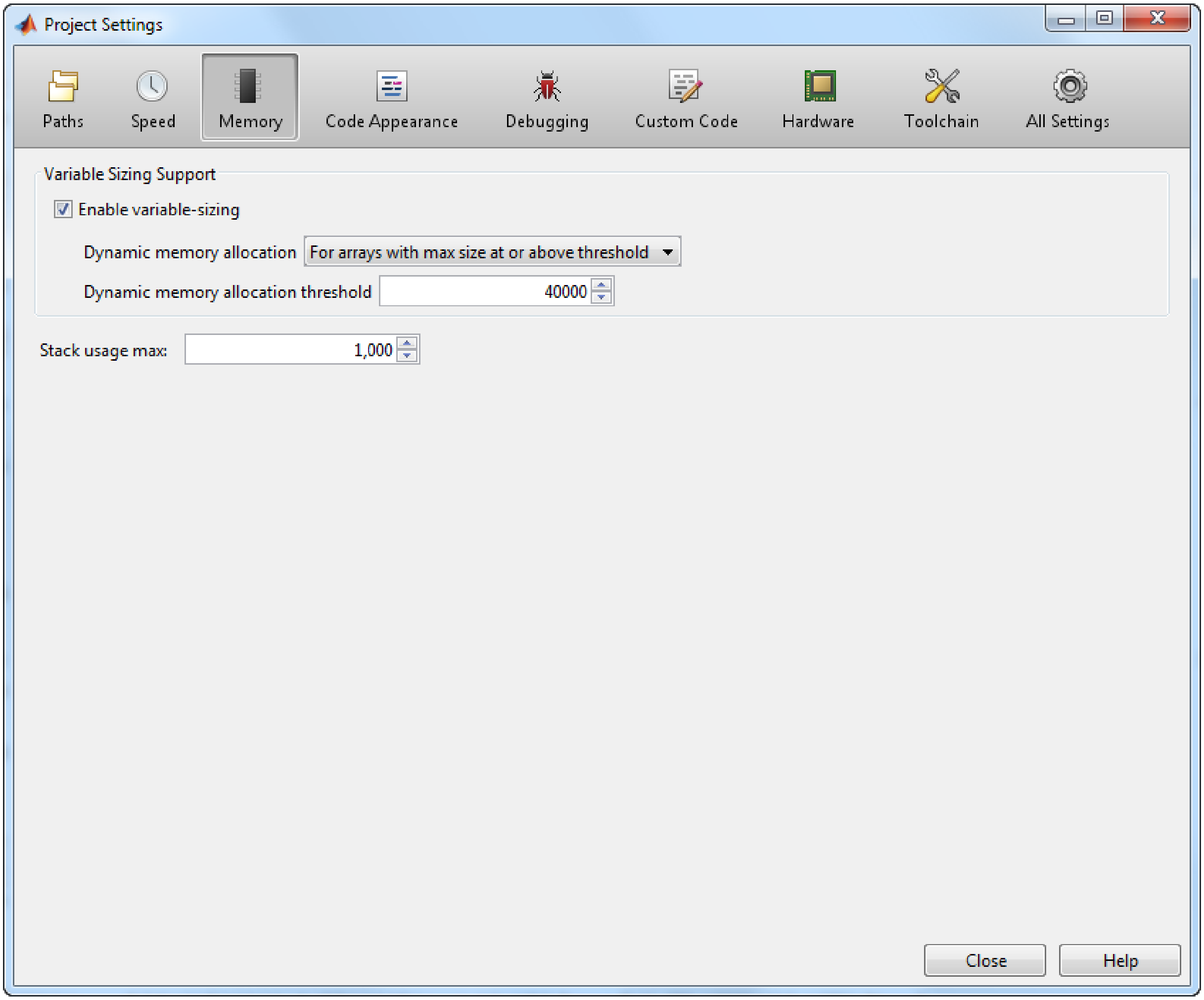
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-sizearrays, you can disable it only for arrays below a certain size. Set a dynamic memory allocation threshold to disable dynamic memory allocation for array size below the threshold and enable it for array size at or above the threshold.

Use this strategy when you want to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, it can be more efficient to speed up generated code by allocating memory statically. Though static memory allocation can lead to unused storage space, it might not be a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, you can reduce storage requirements significantly using dynamic memory allocation.

Set Dynamic Memory Allocation Threshold Using Project Interface

1. On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
2. On the **Memory** tab, select **Enable variable-sizing**.
3. On the same tab, select the **For arrays with max size at or above threshold** option in the **Dynamic memory allocation** list.
4. Set the **Dynamic memory allocation threshold** to the value that you want.



Variable Sizing Support

☑ Enable variable-sizing

Dynamic memory allocation: For arrays with max size at or above threshold

Dynamic memory allocation threshold: 40000

Stack usage max: 1,000

The **Dynamic memory allocation threshold** value is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that can be accommodated by a certain value of `DynamicMemoryAllocationThreshold`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Set Dynamic Memory Allocation Threshold from 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, 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 can be accommodated by a certain value of DynamicMemoryAllocationThreshold. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Related Examples

- "Minimize Dynamic Memory Allocation"
- "Provide Maximum Size for Variable-size Arrays"
- "Disable Dynamic Memory Allocation During Code Generation"

Concepts

- "Dynamic Memory Allocation and Performance"

Excluding Unused Paths from Generated Code

In certain situations, you do not need some branches of an `if/elseif/if` statement or a `switch/case/otherwise` statement in your generated code. For instance :

- You have a MATLAB function that performs mutiple 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 generation software generates code only for the branch that is chosen by the control-flow variable.

Related Examples

- "Prevent Code Generation for Unused Execution Paths"

Prevent Code Generation for Unused Execution Paths

In this section...

"Prevent Code Generation When Local Variable Controls Flow" on page 19-34

"Prevent Code Generation When Input Variable Controls Flow" on page 19-35

If a variable controls the flow of an `if/elseif/if` 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 19-34.
- 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 19-35.

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 `ch`.

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.

Concepts

- "Excluding Unused Paths from Generated Code"

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.

Concepts

- “Algorithm Acceleration Using Parallel for-loops (parfor)”
- “Classification of Variables in parfor-loops”
- “Reduction Assignments in parfor-loops”

Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix `B` is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
     C=C + inv(B)*A^i*B;
  end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

1 Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of `n` terms in the following power series expansion:

$$C = 1 + B^{-1}AB + B^{-1}A^2B + \ldots$$

```
  function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
  C=zeros(size(A));

% Perform the series sum
  for i=1:n
     C=C+inv(B)*A^i*B;
  end
```

2 Generate code for `SeriesFunc` with 4-by-4 matrices passed as input arguments for `A` and `B`:

```
X = coder.typeof(zeros(4));
codegen -config:lib -launchreport SeriesFunc -args {X,X,10}
```

In the generated code, the inversion of B is performed n times inside the loop. It is more economical to perform the inversion operation once outside the loop because it does not depend on the loop index.

3 Modify SeriesFunc as follows:

```
  function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
  C=zeros(size(A));

% Perform the inversion outside the loop
  inv_B=inv(B);

% Perform the series sum
  for i=1:n
     C=C+inv_B*A^i*B;
  end
```

This procedure performs the inversion of B only once, leading to faster execution of the generated code.

Unroll for-Loops

Unrolling for-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant.

You can also force loop unrolling for individual functions by wrapping the loop header in a coder.unroll function. For more information, see coder.unroll.

Limit Copying the for-loop Body in Generated Code

To limit the number of times that you copy the body of a for-loop in generated code:

1 Write a MATLAB function getrand(n) that uses a for-loop to generate a vector of length n and assign random numbers to specific elements. Add a test function test_unroll. This function calls getrand(n) with n equal to values both less than and greater than the threshold for copying the for-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
  % Calling getrand 8 times triggers unroll
  y1 = getrand(8);
  % Calling getrand 50 times does not trigger unroll
  y2 = getrand(50);

function y = getrand(n)
  % Turn off inlining to make
  % generated code easier to read
  coder.inline('never');

  % Set flag variable dounroll to repeat loop body
  % only for fewer than 10 iterations
  dounroll = n < 10;
  % Declare size, class, and complexity
  % of variable y by assignment
  y = zeros(n, 1);
  % Loop body begins
  for i = coder.unroll(1:2:n, dounroll)
      if (i > 2) && (i < n-2)
          y(i) = rand();
      end;
  end;
  % Loop body ends
```

2 In the default output folder, codegen/lib/test_unroll, generate C static library code for test_unroll :

```
codegen -config:lib test_unroll
```

In test_unroll.c, the generated C code for getrand(8) repeats the body of the for-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void getrand(double y[8])
{
  /*  Turn off inlining to make  */
  /*  generated code easier to read */
  /*  Set flag variable dounroll to repeat loop body */
  /*  only for fewer than 10 iterations */
  /*  Declare size, class, and complexity */
  /*  of variable y by assignment */
  memset(&y[0], 0, sizeof(double) << 3);

  /*  Loop body begins */
  y[2] = b_rand();
  y[4] = b_rand();

  /*  Loop body ends */
}
```

The generated C code for getrand(50) does not unroll the for-loop because the number of iterations is greater than 10:

```
static void b_getrand(double y[50])
{
  int i;
  int b_i;
```

```
/*  Turn off inlining to make  */
/*  generated code easier to read */
/*  Set flag variable dounroll to repeat loop body */
/*  only for fewer than 10 iterations */
/*  Declare size, class, and complexity */
/*  of variable y by assignment */
memset(&y[0], 0, 50U * sizeof(double));

/*  Loop body begins */
for (i = 0; i < 25; i++) {
  b_i = (i << 1) + 1;
  if ((b_i > 2) && (b_i < 48)) {
    y[b_i - 1] = b_rand();
  }
}
```

Support for Integer Overflow and Non-Finites

In addition to code generated for your MATLAB function, the code-generation software generates 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.
- Non-finite values (`inf` and `NaN`) are generated from an operation. The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c` and `rtGetNaN.c` (with corresponding header files).

You can suppress generation of the supporting code if you know beforehand that such situations will not arise. This action reduces the size and increases the speed of generated code at the cost of potentially producing results that do not match simulation in case the situations arise.

Disable Support for Integer Overflow

You can disable support for integer overflow in the project settings dialog box or at the command line. On disabling this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.

- In the project settings dialog box:
  1. On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
  2. To disable support for integer overflow, 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. To disable support for integer overflow, set the `SaturateOnIntegerOverflow` property to `false`.

```
cfg.SaturateOnIntegerOverflow = false;
```

Disable Support for Non-Finites

You can disable support for non-finites (`inf` and `NaN`) in the project settings dialog box or at the command line.

- In the project settings dialog box:

  1. On the **Build** tab **Settings** pane, set the **Output type** to `C/C++ Static Library`, `C/C++ Dynamic Library`, or `C/C++ Executable` (depending on your requirements).
  2. Click the **More settings** link to open the **Project Settings** dialog box.
  3. To disable support for integer overflow, on the **Speed** tab, clear`Support non-finite numbers`.

- At the command line:

  1. Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`,`'dll'` or `'exe'` (depending on your requirements). For example:

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

  2. To disable support for integer overflow, set the `SupportNonFinite` property to `false`.

     ```
     cfg.SupportNonFinite = false;
     ```

Integrate Custom Code

This example shows how to integrate custom code to enhance performance of generated code. Although MATLAB Coder generates optimized code for most applications, you might have legacy code optimized for your specific requirements. For example:

- You have custom libraries optimized for your target environment.
- You have custom libraries for functions not supported by MATLAB Coder.
- You have custom libraries that meet standards set by your company.

In such cases, you can integrate your custom code with the code generated by MATLAB Coder.

This example illustrates how to integrate the function `cublasSgemm` from the NVIDIA® CUDA® Basic Linear Algebra Subroutines (CUBLAS) library in generated code. This function performs matrix multiplication on a Graphics Processing Unit (GPU).

1. Define a class `ExternalLib_API` that derives from the class `coder.ExternalDependency`. `ExternalLib_API` defines an interface to the `CUBLAS` library through the following methods:

   - `getDescriptiveName`: Returns a descriptive name for `ExternalLib_API` to be used for error messages.
   - `isSupportedContext`: Determines if the build context supports the `CUBLAS` library.
   - `updateBuildInfo`: Adds header file paths and link files to the build information.
   - `GPU_MatrixMultiply`: Defines the interface to the `CUBLAS` library function `cublasSgemm`.

   ExternalLib_API.m

   ```
   classdef ExternalLib_API < coder.ExternalDependency
       %#codegen

       methods (Static)
   ```

```
function bName = getDescriptiveName(~)
    bName = 'ExternalLib_API';
end

function tf = isSupportedContext(ctx)
    if  ctx.isMatlabHostTarget()
        tf = true;
    else
        error('CUBLAS library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, ~, ~] = ctx.getStdLibInfo();

    % Include header file path
    % Include header files later using coder.cinclude
    hdrFilePath = 'C:\My_Includes';
    buildInfo.addIncludePaths(hdrFilePath);

    % Include link files
    linkFiles = strcat('libcublas', linkLibExt);
    linkPath = 'C:\My_Libs';
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    linkFiles = strcat('libcudart', linkLibExt);
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

end

%API for library function 'cuda_MatrixMultiply'
function C = GPU_MatrixMultiply(A, B)
    assert(isa(A,'single'), 'A must be single.');
    assert(isa(B,'single'), 'B must be single.');

    if(coder.target('MATLAB'))
        C=A*B;
    else

        % Include header files
        %     for external functions and typedefs
        % Header path included earlier using updateBuildInfo
        coder.cinclude('"cuda_runtime.h"');
        coder.cinclude('"cublas_v2.h"');

        % Compute dimensions of input matrices
        m = int32(size(A, 1));
        k = int32(size(A, 2));
        n = int32(size(B, 2));

        % Declare pointers to matrices on destination GPU
        d_A = coder.opaque('float*');
        d_B = coder.opaque('float*');
        d_C = coder.opaque('float*');

        % Compute memory to be allocated for matrices
        % Single = 4 bytes
        size_A = m*k*4;
        size_B = k*n*4;
        size_C = m*n*4;

        % Define error variables
        error = coder.opaque('cudaError_t');
        cudaSuccessV = coder.opaque('cudaError_t', ...
            'cudaSuccess');

        % Assign memory on destination GPU
        error = coder.ceval('cudaMalloc', ...
            coder.wref(d_A), size_A);
        assert(error == cudaSuccessV, ...
            'cudaMalloc(A) failed');
        error = coder.ceval('cudaMalloc', ...
            coder.wref(d_B), size_B);
```

```
assert(error == cudaSuccessV, ...
    'cudaMalloc(B) failed');
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_C), size_C);
assert(error == cudaSuccessV, ...
    'cudaMalloc(C) failed');

% Define direction of copying
hostToDevice = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyHostToDevice');

% Copy matrices to destination GPU
error = coder.ceval('cudaMemcpy',  ...
    d_A, coder.rref(A), size_A, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(A) failed');

error = coder.ceval('cudaMemcpy',  ...
    d_B, coder.rref(B), size_B, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(B) failed');

% Define type and size for result
C = zeros(m, n, 'single');

error = coder.ceval('cudaMemcpy', ...
    d_C, coder.rref(C), size_C, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

% Define handle variables for external library
handle = coder.opaque('cublasHandle_t');
blasSuccess = coder.opaque('cublasStatus_t', ...
    'CUBLAS_STATUS_SUCCESS');

% Initialize external library
ret = coder.opaque('cublasStatus_t');
ret = coder.ceval('cublasCreate', coder.wref(handle));
assert(ret == blasSuccess, 'cublasCreate failed');


TRANSA = coder.opaque('cublasOperation_t', ...
    'CUBLAS_OP_N');
```

```
                alpha = single(1);
                beta = single(0);

                % Multiply matrices on GPU
                ret = coder.ceval('cublasSgemm', handle, ...
                    TRANSA,TRANSA,m,n,k, ...
                    coder.rref(alpha),d_A,m, ...
                    d_B,k, ...
                    coder.rref(beta),d_C,k);

                assert(ret == blasSuccess, 'cublasSgemm failed');

                % Copy result back to local host
                deviceToHost = coder.opaque('cudaMemcpyKind', ...
                    'cudaMemcpyDeviceToHost');
                error = coder.ceval('cudaMemcpy', coder.wref(C), ...
                    d_C, size_C, deviceToHost);
                assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

            end
        end
    end
end
```

2 To perform the matrix multiplication using the interface defined in method GPU_MatrixMultiply and the build information in ExternalLib_API, include the following line in your MATLAB code:

   ```
   C= ExternalLib_API.GPU_MatrixMultiply(A,B);
   ```

   For instance, you can define a MATLAB function Matrix_Multiply that solely performs this matrix multiplication.

   ```
   function C = Matrix_Multiply(A, B) %#codegen
    C= ExternalLib_API.GPU_MatrixMultiply(A,B);
   ```

3 Define a MEX configuration object using coder.config. For using the CUBLAS libraries, set the target language for code generation to C++.

   ```
   cfg=coder.config('mex');
   cfg.TargetLang='C++';
   ```

4 Generate code for Matrix_Multiply using cfg as the configuration object and two 2 X 2 matrices of type single as arguments. Since cublasSgemm supports matrix multiplication for data type float, the corresponding MATLAB matrices must have type single.

```
codegen -config cfg Matrix_Multiply ...
            -args {ones(2,'single'),ones(2,'single')}
```

5 Test the generated MEX function Matrix_Multiply_mex using two 2 X 2 identity matrices of type single.

```
Matrix_Multiply_mex(eye(2,'single'),eye(2,'single'))
```

The output is also a 2 X 2 identity matrix.

See Also

coder.ceval | coder.opaque | coder.rref | coder.wref | assertcoder.ExternalDependency | coder.BuildConfig |

Related Examples

- "Encapsulate Interface to an External C Library"

Concepts

- "Encapsulating the Interface to External Code"

MATLAB Coder Optimizations in Generated Code

In this section...

"Constant Folding" on page 19-52

"Loop Fusion" on page 19-53

"Successive Matrix Operations Combined" on page 19-54

"Unreachable Code Elimination" on page 19-54

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 generation software 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 generation software 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”.

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;
  ```

- 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 generation software 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 generation software 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 generation software 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 in MATLAB Generated Code".

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.

20

Code Generation for MATLAB Structures

- “Structure Definition for Code Generation” on page 20-2
- “Structure Operations Allowed for Code Generation” on page 20-3
- “Define Scalar Structures for Code Generation” on page 20-4
- “Define Arrays of Structures for Code Generation” on page 20-7
- “Make Structures Persistent” on page 20-9
- “Index Substructures and Fields” on page 20-10
- “Assign Values to Structures and Fields” on page 20-12
- “Pass Large Structures as Input Parameters” on page 20-14

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 20-3
Observe restrictions on properties and values of scalar structures.	"Define Scalar Structures for Code Generation" on page 20-4
Make structures uniform in arrays.	"Define Arrays of Structures for Code Generation" on page 20-7
Reference structure fields individually during indexing.	"Index Substructures and Fields" on page 20-10
Avoid type mismatch when assigning values to structures and fields.	"Assign Values to Structures and Fields" on page 20-12

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...

“Restriction When Using struct” on page 20-4

“Restrictions When Defining Scalar Structures by Assignment” on page 20-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 20-4

“Restriction on Adding New Fields After First Use” on page 20-5

Restriction When Using struct

When you use the `struct` function to create scalar structures for code generation, you cannot create structures of cell arrays.

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

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

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

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

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

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

Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform 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 20-7

"Using repmat to Define an Array of Structures with Consistent Field Properties" on page 20-7

"Defining an Array of Structures Using Concatenation" on page 20-8

Ensuring Consistency of Fields

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

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

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

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

1. Create a scalar structure, as described in "Define Scalar Structures for Code Generation" on page 20-4.

2. Call `repmat`, passing the scalar structure and the dimensions of the array.

3. Assign values to each structure using standard array indexing and structure dot notation.

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

```
...
```

```
s.a = 0;
s.b = 0;
X = repmat(s,1,3);
X(1).a = 1;
X(2).a = 2;
X(3).a = 3;
X(1).b = 4;
X(2).b = 5;
X(3).b = 6;
...
```

Defining an Array of Structures Using Concatenation

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

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

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

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

Make Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the isempty statement, as described in “Define and Initialize Persistent Variables” on page 15-10.

For example, the following function defines structure X to be persistent and initializes its fields a and b:

```
function f(u)  %#codegen
persistent X

if isempty(X)
   X.a = 1;
   X.b = 2;
end
```

Index Substructures and Fields

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

Reference substructure field values individually using dot notation

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

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

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

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

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

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

Reference field values individually in structure arrays

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

```
...
```

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

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

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

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 20-7 for more information.

Do not reference fields dynamically

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

Assign Values to Structures and Fields

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

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

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

Do not use field values as constants

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

```
...
Y.a = 3;
Y.b = 5;
X = zeros(Y.a,Y.b); % Generates an error
```

In this example, even though you set fields `a` and `b` of structure `Y` to the values 3 and 5 respectively, `Y.a` and `Y.b` are not constants in generated code. Therefore, they are not valid arguments to pass to the function `zeros`.

Note An exception to this behavior occurs if the structure is declared completely using the `struct` function

```
...
Y = struct('a',3,'b',5);
X = zeros(Y.a,Y.b); % Generates a fixed-size 3 X 5 matrix
```

Do not assign mxArrays to structures

You cannot assign `mxArrays` to structure elements; convert `mxArrays` to known types before code generation (see "Working with mxArrays" on page 11-17).

Pass Large Structures as Input Parameters

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

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

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

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

21

Functions Supported for Code Generation

- “Functions Supported for C/C++ Code Generation — Alphabetical List” on page 21-2
- “Functions Supported for C/C++ Code Generation — Categorical List” on page 21-88

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

You can generate efficient C/C++ code for a subset of MATLAB and toolbox functions that you call from MATLAB code. In generated code, each supported function has the same name, arguments, and functionality as its MATLAB or toolbox counterparts. However, to generate code for these functions, you must adhere to certain limitations when calling them from your MATLAB source code. These limitations appear in the list below.

To find supported functions by MATLAB category or toolbox, see "Functions Supported for C/C++ Code Generation — Categorical List" on page 21-88.

Note For more information on code generation for fixed-point algorithms, refer to "Code Acceleration and Code Generation from MATLAB" on page 9-3.

Function	Product	Remarks and Limitations
`abs`	MATLAB	—
`abs`	Fixed-Point Designer	—
`accumneg`	Fixed-Point Designer	—
`accumpos`	Fixed-Point Designer	—
`acos`	MATLAB	• Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`acosd`	MATLAB	—

Function	Product	Remarks and Limitations
acosh	MATLAB	• Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Designer	—
aictest	Phased Array System Toolbox™	Does not support variable-size inputs.
albersheim	Phased Array System Toolbox	Does not support variable-size inputs.
all	MATLAB	—
all	Fixed-Point Designer	—
ambgfun	Phased Array System Toolbox	Does not support variable-size inputs.
and	MATLAB	—
any	MATLAB	—
any	Fixed-Point Designer	—
aperture2gain	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
`asec`	MATLAB	—
`asecd`	MATLAB	—
`asech`	MATLAB	—
`asin`	MATLAB	• Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`asind`	MATLAB	—
`asinh`	MATLAB	—
`assert`	MATLAB	• Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time. • For standalone code generation, excluded from the generated code. • See "Rules for Using assert Function" on page 23-13.
`atan`	MATLAB	—
`atan2`	MATLAB	—
`atan2`	Fixed-Point Designer	—
`atan2d`	MATLAB	—
`atand`	MATLAB	—

Function	Product	Remarks and Limitations
`atanh`	MATLAB	• Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`az2broadside`	Phased Array System Toolbox	Does not support variable-size inputs.
`azel2phitheta`	Phased Array System Toolbox	Does not support variable-size inputs.
`azel2phithetapat`	Phased Array System Toolbox	Does not support variable-size inputs.
`azel2uv`	Phased Array System Toolbox	Does not support variable-size inputs.
`azel2uvpat`	Phased Array System Toolbox	Does not support variable-size inputs.
`azelaxes`	Phased Array System Toolbox	Does not support variable-size inputs.
`barthannwin`	Signal Processing Toolbox™	• Does not support variable-size inputs. • Window length must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`bartlett`	Signal Processing Toolbox	- Does not support variable-size inputs. - Window length must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`beat2range`	Phased Array System Toolbox	Does not support variable-size inputs.
`besselap`	Signal Processing Toolbox	- Does not support variable-size inputs. - Filter order must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`beta`	MATLAB	—
`betacdf`	Statistics Toolbox™	—
`betainc`	MATLAB	—
`betaincinv`	MATLAB	—
`betainv`	Statistics Toolbox	—
`betaln`	MATLAB	—
`betapdf`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
`betarnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`betastat`	Statistics Toolbox	—
`bi2de`	Communications System Toolbox™	—
`billingsleyicm`	Phased Array System Toolbox	Does not support variable-size inputs.
`bin2dec`	MATLAB	• Does not match MATLAB when the input is empty.
`binaryFeatures`	Computer Vision System Toolbox™	—
`binocdf`	Statistics Toolbox	—
`binoinv`	Statistics Toolbox	—
`binopdf`	Statistics Toolbox	—
`binornd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`binostat`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
bitand	MATLAB	• Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Designer	• Not supported for slope-bias scaled fi objects.
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	• Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—
bitget	MATLAB	• Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
bitget	Fixed-Point Designer	—
bitmax	MATLAB	—
bitor	MATLAB	• Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
bitor	Fixed-Point Designer	• Not supported for slope-bias scaled fi objects.
bitorreduce	Fixed-Point Designer	—
bitreplicate	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
`bitrevorder`	Signal Processing Toolbox	- Does not support variable-size inputs. - Computation performed at run time.
`bitrol`	Fixed-Point Designer	—
`bitror`	Fixed-Point Designer	—
`bitset`	MATLAB	- Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
`bitset`	Fixed-Point Designer	—
`bitshift`	MATLAB	- Does not support floating-point input for the first argument. The first argument must belong to an unsigned integer class.
`bitshift`	Fixed-Point Designer	—
`bitsliceget`	Fixed-Point Designer	—
`bitsll`	Fixed-Point Designer	- Generated code may not handle out of range shifting.
`bitsra`	Fixed-Point Designer	- Generated code may not handle out of range shifting.
`bitsrl`	Fixed-Point Designer	- Generated code may not handle out of range shifting.
`bitxor`	MATLAB	- Does not support floating-point inputs. The arguments must belong to an unsigned integer class.
`bitxor`	Fixed-Point Designer	- Not supported for slope-bias scaled `fi` objects.
`bitxorreduce`	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
`blackman`	Signal Processing Toolbox	- Does not support variable-size inputs. - Window length must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`blackmanharris`	Signal Processing Toolbox	- Does not support variable-size inputs. - Window length must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`blanks`	MATLAB	—
`blkdiag`	MATLAB	—
`bohmanwin`	Signal Processing Toolbox	- Does not support variable-size inputs. - Window length must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`broadside2az`	Phased Array System Toolbox	Does not support variable-size inputs.
`bsxfun`	MATLAB	—

Function	Product	Remarks and Limitations
buttap	Signal Processing Toolbox	- Does not support variable-size inputs. - Filter order must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
butter	Signal Processing Toolbox	- Does not support variable-size inputs. - Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
buttord	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
bwlookup	Image Processing Toolbox™	- For best results, specify an input image of class logical.

Function	Product	Remarks and Limitations
`bwmorph`	Image Processing Toolbox	• The text string specifying the operation must be a constant and, for best results, specify an input image of class `logical`.
`cart2pol`	MATLAB	—
`cart2sph`	MATLAB	—
`cart2sphvec`	Phased Array System Toolbox	Does not support variable-size inputs.
`cast`	MATLAB	—
`cat`	MATLAB	—
`cbfweights`	Phased Array System Toolbox	Does not support variable-size inputs.
`cdf`	Statistics Toolbox	—
`ceil`	MATLAB	—
`ceil`	Fixed-Point Designer	—
`cfirpm`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`char`	MATLAB	—

Function	Product	Remarks and Limitations
`cheb1ap`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`cheb1ord`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`cheb2ap`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`cheb2ord`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`chebwin`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`cheby1`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`cheby2`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`chi2cdf`	Statistics Toolbox	—
`chi2inv`	Statistics Toolbox	—
`chi2pdf`	Statistics Toolbox	—
`chi2rnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`chi2stat`	Statistics Toolbox	—

`chol`	MATLAB	• When there are two output arguments, either make the input matrix variable-size in both dimensions, or, if the input matrix must be fixed size, copy the input matrix to a variable-size matrix before calling `chol`.
		`coder.varsize('B');`
		`B = A;`
		`[B,p] = chol(B);`

Function	Product	Remarks and Limitations
`circpol2pol`	Phased Array System Toolbox	Does not support variable-size inputs.
`circshift`	MATLAB	—
`class`	MATLAB	—
`compan`	MATLAB	—
`complex`	MATLAB	—
`complex`	Fixed-Point Designer	—
`computer`	MATLAB	• Information about the computer on which the code generation software is running. • Use only when the code generation target is S-function (Simulation) or MEX-function.
`cond`	MATLAB	—
`conj`	MATLAB	—
`conj`	Fixed-Point Designer	—
`conndef`	Image Processing Toolbox	All input arguments must be compile-time constants.
`conv`	MATLAB	—
`conv`	Fixed-Point Designer	• Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`. • For variable-sized signals, you may see different results between MATLAB and the generated code. - In generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`.

Function	Product	Remarks and Limitations
		- In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.
`conv2`	MATLAB	—
`convergent`	Fixed-Point Designer	—
`convn`	MATLAB	—
`cordicabs`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicangle`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicatan2`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordiccart2pol`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordiccexp`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordiccos`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicpol2cart`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicrotate`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicsin`	Fixed-Point Designer	• Variable-size signals are not supported.
`cordicsincos`	Fixed-Point Designer	• Variable-size signals are not supported.

Function	Product	Remarks and Limitations
`cornerPoints`	Computer Vision System Toolbox	—
`corrcoef`	MATLAB	• Row-vector input is only supported when the first two inputs are vectors and nonscalar.
`cos`	MATLAB	—
`cos`	Fixed-Point Designer	—
`cosd`	MATLAB	—
`cosh`	MATLAB	—
`cot`	MATLAB	—
`cotd`	MATLAB	—
`coth`	MATLAB	—
`cov`	MATLAB	—
`cross`	MATLAB	• If supplied, `dim` must be a constant.
`csc`	MATLAB	—
`cscd`	MATLAB	—
`csch`	MATLAB	—
`ctranspose`	MATLAB	—
`ctranspose`	Fixed-Point Designer	—
`cumprod`	MATLAB	• Logical inputs are not supported. Cast input to `double` first.
`cumsum`	MATLAB	• Logical inputs are not supported. Cast input to `double` first.
`cumtrapz`	MATLAB	—

Function	Product	Remarks and Limitations
dct	Signal Processing Toolbox	• Does not support variable-size inputs. • Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
de2bi	Communications System Toolbox	—
deal	MATLAB	—
deblank	MATLAB	• Supports only inputs from the char class. • Input values must be in the range 0-127.
dec2bin	MATLAB	• If input d is double, d must be less than 2^52. • If input d is single, d must be less than 2^23. • Unless you specify input n to be constant and n is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, n must be at least 52 for double, 23 for single, 16 for char, 32 for int32, 16 for int16, and so on.

Function	Product	Remarks and Limitations
dec2hex	MATLAB	• If input d is double, d must be less than 2^52. • If input d is single, d must be less than 2^23. • Unless you specify input n to be constant and n is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, n must be at least 13 for double, 6 for single, 4 for char, 8 for int32, 4 for int16, and so on.
dechirp	Phased Array System Toolbox	Does not support variable-size inputs.
deconv	MATLAB	—
del2	MATLAB	—
delayseq	Phased Array System Toolbox	Does not support variable-size inputs.
depressionang	Phased Array System Toolbox	Does not support variable-size inputs.
det	MATLAB	—
detectFASTFeatures	Computer Vision System Toolbox	—
detectMSERFeatures	Computer Vision System Toolbox	—
detectSURFFeatures	Computer Vision System Toolbox	—

Function	Product	Remarks and Limitations
`detrend`	MATLAB	• If supplied and not empty, the input argument `bp` must satisfy the following requirements: - Be real. - Be sorted in ascending order. - Restrict elements to integers in the interval `[1, n-2]`. `n` is the number of elements in a column of input argument `X` , or the number of elements in `X` when `X` is a row vector. - Contain all unique values.
`diag`	MATLAB	• If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value. • For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), `diag`: - Treats the input as a vector input. - Returns a matrix with the given vector along the specified diagonal. • For variable-size inputs that are not variable-length vectors, `diag`: - Treats the input as a matrix. - Does not support inputs that are vectors at run time. - Returns a variable-length vector. If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, the output is 0-by-1 not 0-by-0. However, if the input is a constant size 0-by-0, the output is `[]`.

Function	Product	Remarks and Limitations
		• For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), `diag` treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force `diag` to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use: - `diag(x(:))` instead of `diag(x)` - `diag(x(:),k)` instead of `diag(x,k)`
`diag`	Fixed-Point Designer	• If supplied, the index, *k*, must be a real and scalar integer value that is not a `fi` object.
`diff`	MATLAB	• If supplied, the arguments representing the number of times to apply `diff` and the dimension along which to calculate the difference must be constants.
`disparity`	Computer Vision System Toolbox	—
`divide`	Fixed-Point Designer	• Any non-`fi` input must be constant. Its value must be known at compile time so that it can be cast to a `fi` object. • Complex and imaginary divisors are not supported. • The syntax `T.divide(a,b)` is not supported.
`dop2speed`	Phased Array System Toolbox	Does not support variable-size inputs.
`dopsteeringvec`	Phased Array System Toolbox	Does not support variable-size inputs.
`dot`	MATLAB	—
`double`	MATLAB	—

Function	Product	Remarks and Limitations
`double`	Fixed-Point Designer	—
`downsample`	Signal Processing Toolbox	- Does not support variable-size inputs.
`dpss`	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`effearthradius`	Phased Array System Toolbox	Does not support variable-size inputs.
`eig`	MATLAB	- `QZ` algorithm used in all cases, whereas MATLAB might use different algorithms for different inputs. Consequently, `V` might represent a different basis of eigenvectors, and the eigenvalues in `D` might not be in the same order as in MATLAB. - With one input, `[V,D] = eig(A)`, the results will be similar to those obtained using `[V,D] = eig(A,eye(size(A)),'qz')` in MATLAB, except that for code generation, the columns of `V` are normalized. - Options `'balance'`, `'nobalance'` are not supported for the standard eigenvalue problem, and `'chol'` is not supported for the symmetric generalized eigenvalue problem. - Outputs are of complex type.

Function	Product	Remarks and Limitations
`ellip`	Signal Processing Toolbox	• Does not support variable-size inputs. • Inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`ellipap`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`ellipke`	MATLAB	—
`ellipord`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`end`	Fixed-Point Designer	—
`epipolarLine`	Computer Vision System Toolbox	—

Function	Product	Remarks and Limitations
eps	MATLAB	—
eps	Fixed-Point Designer	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and fi double signals.
eq	MATLAB	—
eq	Fixed-Point Designer	Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
error	MATLAB	For standalone code generation, excluded from the generated code.
espritdoa	Phased Array System Toolbox	Does not support variable-size inputs.
estimate Fundamental Matrix	Computer Vision System Toolbox	—
estimate Uncalibrated Rectification	Computer Vision System Toolbox	—
extractFeatures	Computer Vision System Toolbox	—
evcdf	Statistics Toolbox	—
evinv	Statistics Toolbox	—
evpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
evrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
evstat	Statistics Toolbox	—
exp	MATLAB	—
expcdf	Statistics Toolbox	—
expint	MATLAB	—
expinv	Statistics Toolbox	—
expm	MATLAB	—
expm1	MATLAB	—
exppdf	Statistics Toolbox	—
exprnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
expstat	Statistics Toolbox	—
extractFeatures	Computer Vision System Toolbox	—

Function	Product	Remarks and Limitations
eye	MATLAB	• classname must be a built-in MATLAB numeric type. Does not invoke the static eye method for other classes. For example, eye(m, n, 'myclass') does not invoke myclass.eye(m,n).
factor	MATLAB	• The maximum double precision input is 2^33. • The maximum single precision input is 2^25. • The input n cannot have type int64 or uint64.
factorial	MATLAB	—
false	MATLAB	• Dimensions must be real, nonnegative, integers.
fcdf	Statistics Toolbox	—
fclose	MATLAB	—
fft	MATLAB	• Length of input vector must be a power of 2.
fft2	MATLAB	• Length of input matrix dimensions must each be a power of 2.
fftn	MATLAB	• Length of input matrix dimensions must each be a power of 2.
fftshift	MATLAB	—

Function	Product	Remarks and Limitations
fi	Fixed-Point Designer	• Use to create a fixed-point constant or variable. • The default constructor syntax without input arguments is not supported. • The syntax `fi('PropertyName',PropertyValue...)` is not supported. To use property name/property value pairs, you must first specify the value `v` of the `fi` object as in `fi(v,'PropertyName',PropertyValue...)`. • If the input value is not known at compile time, you must provide complete `numerictype` information. • `numerictype` object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter	Fixed-Point Designer	• Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`.
filter2	MATLAB	—
filtfilt	Signal Processing Toolbox	• Does not support variable-size inputs. • Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`fimath`	Fixed-Point Designer	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the `fimath` object defined in the MATLAB Function dialog in the Model Explorer. • Use to create `fimath` objects in generated code.
`find`	MATLAB	• Issues an error if a variable-sized input becomes a row vector at run time. **Note** This limitation does not apply when the input is scalar or a variable-length row vector. • For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.
`finv`	Statistics Toolbox	—
`fir1`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`fir2`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`fircls`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`fircls1`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`firls`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`firpm`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`firpmord`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
firrcos	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
fix	MATLAB	—
fix	Fixed-Point Designer	—
fixed.Quantizer	Fixed-Point Designer	—
flattopwin	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
flintmax	MATLAB	—
flipdim	MATLAB	—
fliplr	MATLAB	—
flipud	MATLAB	—
floor	MATLAB	—
floor	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
fopen	MATLAB	• Does not support: - machineformat, encoding, or fileID inputs - message output - fopen(`all') • If you disable extrinsic calls, you cannot return fileIDs created with fopen to MATLAB or extrinsic functions. You can only use such fileIDs internally. • You can open up to 20 files when generating C/C++ executables, static libraries, or dynamic libraries.
fpdf	Statistics Toolbox	—
fprintf	MATLAB	• Does not support: - b and t subtypes on %u, %o %x, and %X formats - $ flag for reusing input arguments - printing arrays • There is no automatic casting. Input arguments must match their format types for predictable results. • Escaped characters are limited to the decimal range of 0–127. • A call to fprintf with fileID equal to 1 or 2 becomes printf in the generated C/C++ code in the following cases: - The fprintf call is inside a parfor loop. - Extrinsic calls are disabled.

Function	Product	Remarks and Limitations
		- When the MATLAB behavior differs from the C compiler behavior, fprintf matches the C compiler behavior in the following cases: - The format specifier has a corresponding C format specifier, for example, %e or %E. - The fprintf call is inside a parfor loop. - Extrinsic calls are disabled. - When you call fprintf with the format specifier %s, do not put a null character in the middle of the input string. Use fprintf(fid, '%c', char(0)) to write a null character. - When you call fprintf with an integer format specifier, the type of the integer argument must be a type that the target hardware can represent as a native C type. For example, if you call fprintf('%d', int64(n)), the target hardware must have a native C type that supports a 64-bit integer.
freqspace	MATLAB	—
freqz	Signal Processing Toolbox	- Does not support variable-size inputs. - freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See "freqz With No Output Arguments".
frnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: - The output is nonscalar. - An input parameter is invalid for the distribution.

Function	Product	Remarks and Limitations
`fspecial`	Image Processing Toolbox	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
`fspl`	Phased Array System Toolbox	Does not support variable-size inputs.
`fstat`	Statistics Toolbox	—
`full`	MATLAB	—

`fzero`	MATLAB	• The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument.
		• Supports up to three output arguments. Does not support the fourth output argument (the `output` structure).
		• Only supports the `TolX` and `FunValCheck` fields of an `options` input structure. Ignores other options in an `options` input structure. You cannot use the `optimset` function to create the `options` structure. Create this structure directly, for example,
		`opt.TolX = tol;`
		`opt.FunValCheck = 'on';`
		The input structure field names must match exactly.

`gain2aperture`	Phased Array System Toolbox	Does not support variable-size inputs.
`gamcdf`	Statistics Toolbox	—
`gaminv`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
`gamma`	MATLAB	—
`gammainc`	MATLAB	—
`gammaincinv`	MATLAB	—
`gammaln`	MATLAB	—
`gampdf`	Statistics Toolbox	—
`gamrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`gamstat`	Statistics Toolbox	—
`gaussfir`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`gausswin`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .

Function	Product	Remarks and Limitations
`gcd`	MATLAB	—
`ge`	MATLAB	—
`ge`	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.
`geocdf`	Statistics Toolbox	—
`geoinv`	Statistics Toolbox	—
`geomean`	Statistics Toolbox	—
`geopdf`	Statistics Toolbox	—
`geornd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`geostat`	Statistics Toolbox	—
`get`	Fixed-Point Designer	• The syntax `structure = get(o)` is not supported.
`getlsb`	Fixed-Point Designer	—
`getmsb`	Fixed-Point Designer	—
`gevcdf`	Statistics Toolbox	—
`gevinv`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
gevpdf	Statistics Toolbox	—
gevrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
gevstat	Statistics Toolbox	—
global2localcoord	Phased Array System Toolbox	Does not support variable-size inputs.
gpcdf	Statistics Toolbox	—
gpinv	Statistics Toolbox	—
gppdf	Statistics Toolbox	—
gprnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
gpstat	Statistics Toolbox	—
gradient	MATLAB	—
grazingang	Phased Array System Toolbox	Does not support variable-size inputs.
gt	MATLAB	—

Function	Product	Remarks and Limitations
`gt`	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.
`hadamard`	MATLAB	—
`hamming`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`hankel`	MATLAB	—
`hann`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`harmmean`	Statistics Toolbox	—
`hdlram`	Fixed-Point Designer	—
`hex2dec`	MATLAB	—
`hex2num`	MATLAB	• For `n = hex2num(S)`, `size(S,2) <= length(num2hex(0))`
`hilb`	MATLAB	—

Function	Product	Remarks and Limitations
hist	MATLAB	• Histogram bar plotting not supported; call with at least one output argument. • If supplied, the second argument x must be a scalar constant. • Inputs must be real.
histc	MATLAB	• The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.
horizonrange	Phased Array System Toolbox	Does not support variable-size inputs.
horzcat	Fixed-Point Designer	—
hygecdf	Statistics Toolbox	—
hygeinv	Statistics Toolbox	—
hygepdf	Statistics Toolbox	—
hygernd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
hygestat	Statistics Toolbox	—
hypot	MATLAB	—
icdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
`idct`	Signal Processing Toolbox	• Does not support variable-size inputs. • Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`idivide`	MATLAB	• For efficient generated code, MATLAB rules for divide by zero are supported only for the `'round'` option.
`ifft`	MATLAB	• Length of input vector must be a power of 2. • Output of `ifft` block is complex. • Does not support the `'symmetric'` option.
`ifft2`	MATLAB	• Length of input matrix dimensions must each be a power of 2. • Does not support the `'symmetric'` option.
`ifftn`	MATLAB	• Length of input matrix dimensions must each be a power of 2. • Does not support the `'symmetric'` option.
`ifftshift`	MATLAB	—
`imag`	MATLAB	—
`imag`	Fixed-Point Designer	—
`imcomplement`	Image Processing Toolbox	Does not support `int64` and `uint64` data types.

Function	Product	Remarks and Limitations
imfill	Image Processing Toolbox	The optional input connectivity, conn, and the string 'holes' must be compile-time constants. Supports only up to 3-D inputs. (No N-D support.) The interactive mode to select points, imfill(BW,0,CONN), is not supported in code generation. locations can be a *P*-by-1 vector, in which case it contains the linear indices of the starting locations. locations can also be a *P*-by-ndims(I) matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile time, you cannot change it at run time. However, the number of points in locations can be varied at run time. Generated code for this function uses a precompiled platform-specific shared library.
imhmax	Image Processing Toolbox	The optional third input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imhmin	Image Processing Toolbox	The optional third input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imreconstruct	Image Processing Toolbox	The optional third input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.

Function	Product	Remarks and Limitations
`imregionalmax`	Image Processing Toolbox	The optional second input argument, `conn`, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
`imregionalmin`	Image Processing Toolbox	The optional second input argument, `conn`, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
`ind2sub`	MATLAB	• The first argument should be a valid size vector. Size vectors for arrays with more than `intmax` elements are not supported.
`inf`	MATLAB	• Dimensions must be real, nonnegative, integers.
`insertMarker`	Computer Vision System Toolbox	• `'Color'` input cannot be a cell array. • `position` input cannot be a `cornerPoints` object. • `marker` input must be constant. • `marker` input cannot be `'s'`.
`insertShape`	Computer Vision System Toolbox	• `position` input cannot be a cell array. • `'Color'` input cannot be a cell array. • `shape` input must be constant.
`int8`, `int16`, `int32`, `int64`	MATLAB	You cannot use `int64` in a : • MATLAB Function block in a Simulink model • MATLAB function in a Stateflow chart

Function	Product	Remarks and Limitations
int8, int16, int32, int64	Fixed-Point Designer	You cannot use int64 in a : - MATLAB Function block in a Simulink model - MATLAB function in a Stateflow chart
integralImage	Computer Vision System Toolbox	—
interp1	MATLAB	- Supports only linear and nearest interpolation methods. - Does not handle evenly spaced X indices separately. - X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.
interp2	MATLAB	- Supports only 5 <= nargin <= 7. - XI and YI must be the same size. - Supports only 'linear' and 'nearest' methods. - For best results, supply X and Y as vectors. - When the X or Y inputs are not vectors, interp2 references only the first row of X and first column of Y. Supports "plaid" input for X and Y but does not verify that the input data is "plaid". - X and Y must contain monotonically increasing values. If your application provides monotonically decreasing values, first use fliplr and flipud to change X, Y, and Z to monotonically increasing form before calling interp2.

Function	Product	Remarks and Limitations
`intersect`	MATLAB	• When you do not specify the `'rows'` option: – Inputs `A` and `B` must be vectors. If you specify the `'legacy'` option, inputs `A` and `B` must be row vectors. – The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. – The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example, `zeros(1,0)`, to represent the empty set. – If you specify the `'legacy'` option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output `C` is 0-by-0. • When the `setOrder` is `'sorted'` or when you specify the `'legacy'` option, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order. • Complex inputs must be `single` or `double`. • When one input is complex and the other input is real, do one of the following: – Set `setOrder` to `'stable'`. – Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.

Function	Product	Remarks and Limitations
`intfilt`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`intmax`	MATLAB	—
`intmin`	MATLAB	—
`inv`	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
`invhilb`	MATLAB	—
`ipermute`	MATLAB	—
`iptcheckconn`	Image Processing Toolbox	All input arguments must be compile-time constants.
`iqr`	Statistics Toolbox	—
`isa`	MATLAB	—
`iscell`	MATLAB	—
`ischar`	MATLAB	—
`iscolumn`	MATLAB	—
`iscolumn`	Fixed-Point Designer	—
`isdeployed`	MATLAB Compiler	• Returns true and false as appropriate for MEX and SIM targets • Returns false for other targets
`isempty`	MATLAB	—

Function	Product	Remarks and Limitations
isempty	Fixed-Point Designer	—
isEpipoleInImage	Computer Vision System Toolbox	—
isequal	MATLAB	—
isequal	Fixed-Point Designer	—
isequaln	MATLAB	—
isfi	Fixed-Point Designer	—
isfield	MATLAB	• Does not support cell input for second argument
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isfinite	Fixed-Point Designer	—
isfloat	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Designer	—
isinteger	MATLAB	—
isletter	MATLAB	• Input values from the char class must be in the range 0-127
islogical	MATLAB	—

Function	Product	Remarks and Limitations
ismac	MATLAB	• Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
ismatrix	MATLAB	—
ismcc	MATLAB Compiler	• Returns true and false as appropriate for MEX and SIM targets. • Returns false for other targets.
ismember	MATLAB	• The second input, B, must be sorted in ascending order. • Complex inputs must be single or double.
isnan	MATLAB	—
isnan	Fixed-Point Designer	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Designer	—
isnumerictype	Fixed-Point Designer	—
ispc	MATLAB	• Returns true or false based on the MATLAB version you use for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
isprime	MATLAB	• The maximum double precision input is 2^33. • The maximum single precision input is 2^25. • The input X cannot have type int64 or uint64.
isreal	MATLAB	—

Function	Product	Remarks and Limitations
isreal	Fixed-Point Designer	—
isrow	MATLAB	—
isrow	Fixed-Point Designer	—
isscalar	MATLAB	—
isscalar	Fixed-Point Designer	—
issigned	Fixed-Point Designer	—
issorted	MATLAB	—
isspace	MATLAB	• Input values from the char class must be in the range 0–127.
issparse	MATLAB	—
isstrprop	MATLAB	• Supports only inputs from char and integer classes. • Input values must be in the range 0-127.
isstruct	MATLAB	—
istrellis	Communications System Toolbox	—
isunix	MATLAB	• Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
isvector	MATLAB	—
isvector	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
`kaiser`	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`kaiserord`	Signal Processing Toolbox	- Does not support variable-size inputs. - Computation performed at run time.
`kron`	MATLAB	—
`kurtosis`	Statistics Toolbox	—
`label2rgb`	Image Processing Toolbox	Referring to the standard syntax: `RGB = label2rgb(L,map,zerocolor,order)` - Submit at least two input arguments: the label matrix, `L`, and the colormap matrix, `map`. - `map` must be an `n-by-3`, `double`, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. - If you set the boundary color `zerocolor` to the same color as one of the regions, `label2rgb` will not issue a warning. - If you supply a value for `order`, it must be `'noshuffle'`.
`lcm`	MATLAB	—

Function	Product	Remarks and Limitations
lcmvweights	Phased Array System Toolbox	Does not support variable-size inputs.
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Designer	—
levinson	Signal Processing Toolbox	• Does not support variable-size inputs. • If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
lineToBorderPoints	Computer Vision System Toolbox	—
linsolve	MATLAB	• The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: - `UT` - `LT` - `UHESS` = `true` (the `TRANSA` can be either `true` or `false`) - `SYM` = `true` and `POSDEF` = `true`

Function	Product	Remarks and Limitations
		Other options are equivalent to using `mldivide`.
`linspace`	MATLAB	—
`load`	MATLAB	• Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use `coder.load`. • Does not support use of the function without assignment to a structure or array. For example, use `S = load(filename)`, not `load(filename)`. • The output `S` must be the name of a structure or array without any subscripting. For example, `S[i] = load('myFile.mat')` is not allowed. • Arguments to `load` must be compile-time constant strings. • Does not support loading objects. • If the MAT-file contains unsupported constructs, use `load(filename,variables)` to load only the supported constructs. • You cannot use `save` in a function intended for code generation. The code generation software does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save the workspace data to a MAT-file. You must use `coder.varsize` to explicitly declare variable-size data loaded using the `load` function.
`local2globalcoord`	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
log	MATLAB	• Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Designer	—
logncdf	Statistics Toolbox	—
logninv	Statistics Toolbox	—
lognpdf	Statistics Toolbox	—
lognrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
lognstat	Statistics Toolbox	—
logspace	MATLAB	—
lower	MATLAB	• Supports only char inputs. • Input values must be in the range 0-127.

Function	Product	Remarks and Limitations
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and double signals.
lt	MATLAB	—
lt	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.
lu	MATLAB	—
mad	Statistics Toolbox	Input dim cannot be empty.
magic	MATLAB	—
matchFeatures	Computer Vision System Toolbox	—
max	MATLAB	—
max	Fixed-Point Designer	—
maxflat	Signal Processing Toolbox	• Does not support variable-size inputs. • Inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
mdltest	Phased Array System Toolbox	Does not support variable-size inputs.
mean	MATLAB	—

Function	Product	Remarks and Limitations
`mean`	Fixed-Point Designer	—
`median`	MATLAB	—
`median`	Fixed-Point Designer	—
`meshgrid`	MATLAB	—
`mfilename`	MATLAB	—
`min`	MATLAB	—
`min`	Fixed-Point Designer	—
`minus`	MATLAB	—
`minus`	Fixed-Point Designer	• Any non-`fi` input must be constant. Its value must be known at compile time so that it can be cast to a `fi` object.
`mldivide`	MATLAB	—
`mnpdf`	Statistics Toolbox	—
`mod`	MATLAB	• Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. If one of the inputs has type `int64` or `uint64`, then both inputs must have the same type.
`mode`	MATLAB	• Does not support third output argument `C` (cell array).
`moment`	Statistics Toolbox	If `order` is nonintegral and `X` is real, use `moment(complex(X), order)`.
`mpower`	MATLAB	—

Function	Product	Remarks and Limitations
mpower	Fixed-Point Designer	• The exponent input, *k*, must be constant; that is, its value must be known at compile time. • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. • For variable-sized signals, you may see different results between MATLAB and the generated code. - In generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
mpy	Fixed-Point Designer	• When you provide complex inputs to the mpy function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
mrdivide	MATLAB	—
mrdivide	Fixed-Point Designer	—
MSERRegions	Computer Vision System Toolbox	—
mtimes	MATLAB	—

Function	Product	Remarks and Limitations
`mtimes`	Fixed-Point Designer	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object. • Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`. • For variable-sized signals, you may see different results between MATLAB and the generated code. - In generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`. - In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.
`mvdrweights`	Phased Array System Toolbox	Does not support variable-size inputs.
NaN or `nan`	MATLAB	• Dimensions must be real, nonnegative, integers.
`nancov`	Statistics Toolbox	If the input is variable-size and is `[]` at run time, returns `[]` not `NaN`.
`nanmax`	Statistics Toolbox	—
`nanmean`	Statistics Toolbox	—
`nanmedian`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
nanmin	Statistics Toolbox	—
nanstd	Statistics Toolbox	—
nansum	Statistics Toolbox	—
nanvar	Statistics Toolbox	—
nargchk	MATLAB	• Output structure does not include stack information. **Note** nargchk will be removed in a future release.
nargin	MATLAB	—
narginchk	MATLAB	—
nargout	MATLAB	• For a function with no output arguments, returns 1 if called without a terminating semicolon. **Note** This behavior also affects extrinsic calls with no terminating semicolon. nargout is 1 for the called function in MATLAB.
nargoutchk	MATLAB	—
nbincdf	Statistics Toolbox	—
nbininv	Statistics Toolbox	—
nbinpdf	Statistics Toolbox	—

Function	Product	Remarks and Limitations
`nbinrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`nbinstat`	Statistics Toolbox	—
`ncfcdf`	Statistics Toolbox	—
`ncfinv`	Statistics Toolbox	—
`ncfpdf`	Statistics Toolbox	—
`ncfrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`ncfstat`	Statistics Toolbox	—
`nchoosek`	MATLAB	• When the first input, `x`, is a scalar, `nchoosek` returns a binomial coefficient. In this case, `x` must be a nonnegative integer. It cannot have type `int64` or `uint64`. • When the first input, `x`, is a vector, `nchoosek` treats it as a set. In this case, `x` can have type `int64` or `uint64`. • The second input, `k`, cannot have type `int64` or `uint64`.

Function	Product	Remarks and Limitations
`nctcdf`	Statistics Toolbox	—
`nctinv`	Statistics Toolbox	—
`nctpdf`	Statistics Toolbox	—
`nctrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`nctstat`	Statistics Toolbox	—
`ncx2cdf`	Statistics Toolbox	—
`ncx2rnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`ncx2stat`	Statistics Toolbox	—
`ndgrid`	MATLAB	—
`ndims`	MATLAB	—
`ndims`	Fixed-Point Designer	—
`ne`	MATLAB	—
`ne`	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.

Function	Product	Remarks and Limitations
`nearest`	Fixed-Point Designer	—
`nextpow2`	MATLAB	—
`nnz`	MATLAB	—
`noisepow`	Phased Array System Toolbox	Does not support variable-size inputs.
`nonzeros`	MATLAB	—
`norm`	MATLAB	—
`normcdf`	Statistics Toolbox	—
`normest`	MATLAB	—
`norminv`	Statistics Toolbox	—
`normpdf`	Statistics Toolbox	—
`normrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`normstat`	Statistics Toolbox	—
`not`	MATLAB	—
`npwgnthresh`	Phased Array System Toolbox	Does not support variable-size inputs.
`nthroot`	MATLAB	—
`null`	MATLAB	• Might return a different basis than MATLAB • Does not support rational basis option (second input)

Function	Product	Remarks and Limitations
num2hex	MATLAB	—
numberofelements	Fixed-Point Designer	numberofelements will be removed in a future release. Use numel instead.
numel	MATLAB	—
numel	Fixed-Point Designer	—
numerictype	Fixed-Point Designer	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a numerictype object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a non-fixed-point signal. • Use to create numerictype objects in the generated code.
nuttallwin	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
ones	MATLAB	• Dimensions must be real, nonnegative, integers.
or	MATLAB	—
orth	MATLAB	• Might return a different basis than MATLAB
padarray	Image Processing Toolbox	Supports only up to 3-D inputs. Input arguments, padval and direction, are expected to be compile-time constants.

Function	Product	Remarks and Limitations
parzenwin	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for fiaccel, use coder.Constant. For more information, see .
pascal	MATLAB	—
pdf	Statistics Toolbox	—
permute	MATLAB	—
permute	Fixed-Point Designer	—
phitheta2azel	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2azelpat	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uv	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uvpat	Phased Array System Toolbox	Does not support variable-size inputs.
physconst	Phased Array System Toolbox	Does not support variable-size inputs.
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—

Function	Product	Remarks and Limitations
`plus`	Fixed-Point Designer	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`poisscdf`	Statistics Toolbox	—
`poissinv`	Statistics Toolbox	—
`poisspdf`	Statistics Toolbox	—
`poissrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`poisstat`	Statistics Toolbox	—
`pol2cart`	MATLAB	—
`pol2circpol`	Phased Array System Toolbox	Does not support variable-size inputs.
`polellip`	Phased Array System Toolbox	Does not support variable-size inputs.
`polloss`	Phased Array System Toolbox	Does not support variable-size inputs.
`polratio`	Phased Array System Toolbox	Does not support variable-size inputs.
`polsignature`	Phased Array System Toolbox	Does not support variable-size inputs.
`poly`	MATLAB	• Does not discard nonfinite input values • Complex input produces complex output

Function	Product	Remarks and Limitations
`poly2trellis`	Communications System Toolbox	—
`polyfit`	MATLAB	—
`polyval`	MATLAB	—
`pow2`	Fixed-Point Designer	—
`power`	MATLAB	• Generates an error during simulation. Returns `NaN` in generated code when both `X` and `Y` are real, but `power(X,Y)` is complex. To get the complex result, make the input value `X` complex by passing in `complex(X)`. For example, `power(complex(X),Y)`. • Generates an error during simulation. Returns `NaN` in generated code when both `X` and `Y` are real, but `X .^ Y` is complex. To get the complex result, make the input value `X` complex by using `complex(X)`. For example, `complex(X).^Y`.
`power`	Fixed-Point Designer	• The exponent input, `k`, must be constant. Its value must be known at compile time.
`prctile`	Statistics Toolbox	• "Automatic dimension restriction" • If the output `Y` is a vector, the orientation of `Y` differs from MATLAB when all of the following are true: - You do not supply the `dim` input. - `X` is a variable-size array. - `X` is not a variable-length vector. - X is a vector at run time. - The orientation of the vector `X` does not match the orientation of the vector `p`.

Function	Product	Remarks and Limitations
		In this case, the output Y matches the orientation of X not the orientation of p.
primes	MATLAB	• The maximum double precision input is 2^32. • The maximum single precision input is 2^24. • The input n cannot have type int64 or uint64.
prod	MATLAB	—
psi	MATLAB	—
pulsint	Phased Array System Toolbox	Does not support variable-size inputs.
qr	MATLAB	—
quad2d	MATLAB	• Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.
quadgk	MATLAB	—
quantile	Statistics Toolbox	—
quantize	Fixed-Point Designer	—
quatconj	Aerospace Toolbox	—
quatdivide	Aerospace Toolbox	—
quatinv	Aerospace Toolbox	—

Function	Product	Remarks and Limitations
quatmod	Aerospace Toolbox	—
quatmultiply	Aerospace Toolbox	—
quatnorm	Aerospace Toolbox	—
quatnormalize	Aerospace Toolbox	—
radareqpow	Phased Array System Toolbox	Does not support variable-size inputs.
radareqrng	Phased Array System Toolbox	Does not support variable-size inputs.
radareqsnr	Phased Array System Toolbox	Does not support variable-size inputs.
radarvcd	Phased Array System Toolbox	Does not support variable-size inputs.
radialspeed	Phased Array System Toolbox	Does not support variable-size inputs.
rand	MATLAB	—
randg	Statistics Toolbox	—
randi	MATLAB	—
randn	MATLAB	—
random	Statistics Toolbox	—
randperm	MATLAB	—
range	Fixed-Point Designer	—
range2beat	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
range2bw	Phased Array System Toolbox	Does not support variable-size inputs.
range2time	Phased Array System Toolbox	Does not support variable-size inputs.
rangeangle	Phased Array System Toolbox	Does not support variable-size inputs.
rank	MATLAB	—
raylcdf	Statistics Toolbox	—
raylinv	Statistics Toolbox	—
raylpdf	Statistics Toolbox	—
raylrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
raylstat	Statistics Toolbox	—
rcond	MATLAB	—
rdcoupling	Phased Array System Toolbox	Does not support variable-size inputs.
rdivide	MATLAB	—
rdivide	Fixed-Point Designer	—
real	MATLAB	—

Function	Product	Remarks and Limitations
`real`	Fixed-Point Designer	—
`reallog`	MATLAB	—
`realmax`	MATLAB	—
`realmax`	Fixed-Point Designer	—
`realmin`	MATLAB	—
`realmin`	Fixed-Point Designer	—
`realpow`	MATLAB	—
`realsqrt`	MATLAB	—
`rectwin`	Signal Processing Toolbox	• Does not support variable-size inputs. • All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`reinterpretcast`	Fixed-Point Designer	—
`rem`	MATLAB	• Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. • If one of the inputs has type `int64` or `uint64`, then both inputs must have the same type.
`removefimath`	Fixed-Point Designer	—
`repmat`	MATLAB	—

Function	Product	Remarks and Limitations
`repmat`	Fixed-Point Designer	—
`resample`	Signal Processing Toolbox	• Does not support variable-size inputs. • The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`rescale`	Fixed-Point Designer	—
`reshape`	MATLAB	—
`reshape`	Fixed-Point Designer	—
`rng`	MATLAB	• For library and executable code generation targets, and for MEX targets when extrinsic calls are disabled, supports only the `'default'` input and these `generator` inputs: - `'twister'` - `'v4'` - `'v5normal'` For these targets, the output of `s=rng` in the generated code differs from the MATLAB output. You cannot return the output of `s=rng` from the generated code and pass it to `rng` in MATLAB.

Function	Product	Remarks and Limitations
		• For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by `rng`.
`rocpfa`	Phased Array System Toolbox	• Does not support variable-size inputs. • Does not support `NonfluctuatingNoncoherent` signal type.
`rocsnr`	Phased Array System Toolbox	• Does not support variable-size inputs. • Does not support `NonfluctuatingNoncoherent` signal type.
`rootmusicdoa`	Phased Array System Toolbox	Does not support variable-size inputs.
`roots`	MATLAB	• Output is variable size • Output is complex • Roots may not be in the same order as MATLAB • Roots of poorly conditioned polynomials may not match MATLAB
`rosser`	MATLAB	—
`rot90`	MATLAB	—
`rotx`	Phased Array System Toolbox	Does not support variable-size inputs.
`roty`	Phased Array System Toolbox	Does not support variable-size inputs.
`rotz`	Phased Array System Toolbox	Does not support variable-size inputs.
`round`	MATLAB	—
`round`	Fixed-Point Designer	—
`rsf2csf`	MATLAB	—

Function	Product	Remarks and Limitations
`schur`	MATLAB	Might sometimes return a different Schur decomposition in generated code than in MATLAB.
`sec`	MATLAB	—
`secd`	MATLAB	—
`sech`	MATLAB	—
`sensorcov`	Phased Array System Toolbox	Does not support variable-size inputs.
`sensorsig`	Phased Array System Toolbox	Does not support variable-size inputs.
`setdiff`	MATLAB	• When you do not specify the `'rows'` option: - Inputs `A` and `B` must be vectors. If you specify the `'legacy'` option, inputs `A` and `B` must be row vectors. - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. - Do not use `[]` to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, `zeros(1,0)`, to represent the empty set. - If you specify the `'legacy'` option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the `'legacy'` and `'rows'` options, the output `ia` is a column vector. If `ia` is empty, it is 0-by-1, never 0-by-0, even if the output `C` is 0-by-0. • When the `setOrder` is `'sorted'` or when you specify the `'legacy'` option, the inputs must

Function	Product	Remarks and Limitations
		already be sorted in ascending order. The first output, `C`, is sorted in ascending order. • Complex inputs must be `single` or `double`. • When one input is complex and the other input is real, do one of the following: – Set `setOrder` to `'stable'`. – Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
`setfimath`	Fixed-Point Designer	—
`setxor`	MATLAB	• When you do not specify the `'rows'` option: – Inputs `A` and `B` must be vectors with the same orientation. If you specify the `'legacy'` option, inputs `A` and `B` must be row vectors. – The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. – The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example , `zeros(1,0)`, to represent the empty set. – If you specify the `'legacy'` option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output `C` is 0-by-0.

Function	Product	Remarks and Limitations
		- When the `setOrder` is `'sorted'` or when you specify the `'legacy'` flag, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order. - Complex inputs must be `single` or `double`. - When one input is complex and the other input is real, do one of the following: - Set `setOrder` to `'stable'`. - Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
`sfi`	Fixed-Point Designer	—
`sgolay`	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`shiftdim`	MATLAB	Second argument must be a constant.
`shnidman`	Phased Array System Toolbox	Does not support variable-size inputs.
`sign`	MATLAB	—
`sign`	Fixed-Point Designer	—
`sin`	MATLAB	—

Function	Product	Remarks and Limitations
`sin`	Fixed-Point Designer	—
`sind`	MATLAB	—
`single`	MATLAB	—
`single`	Fixed-Point Designer	—
`sinh`	MATLAB	—
`size`	MATLAB	—
`size`	Fixed-Point Designer	—
`skewness`	Statistics Toolbox	—
`sort`	MATLAB	If the input is a complex type, `sort` orders the output according to absolute value. When `x` is a complex type that has all zero imaginary parts, use `sort(real(x))` to compute the sort order for real types. See "Code Generation for Complex Data" on page 13-4.
`sort`	Fixed-Point Designer	—
`sortrows`	MATLAB	If the input is a complex type, `sortrows` orders the output according to absolute value. When `x` is a complex type that has all zero imaginary parts, use `sortrows(real(x))` to compute the sort order for real types. See "Code Generation for Complex Data" on page 13-4.
`sosfilt`	Signal Processing Toolbox	• Does not support variable-size inputs. • Computation performed at run time.
`speed2dop`	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
sph2cart	MATLAB	—
sph2cartvec	Phased Array System Toolbox	Does not support variable-size inputs.
spsmooth	Phased Array System Toolbox	Does not support variable-size inputs.
squeeze	MATLAB	—
sqrt	MATLAB	• Generates an error during simulation and returns NaN in generated code when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
sqrt	Fixed-Point Designer	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sqrtm	MATLAB	—
std	MATLAB	—
steervec	Phased Array System Toolbox	Does not support variable-size inputs.
stokes	Phased Array System Toolbox	Does not support variable-size inputs.
storedInteger	Fixed-Point Designer	—
storedIntegerToDouble	Fixed-Point Designer	—
str2func	MATLAB	• String must be constant/known at compile time
strcmp	MATLAB	—
strcmpi	MATLAB	• Input values from the char class must be in the range 0-127.

Function	Product	Remarks and Limitations
stretchfreq2rng	Phased Array System Toolbox	Does not support variable-size inputs.
strfind	MATLAB	• Does not support cell arrays. • If pattern does not exist in str, returns zeros(1,0) not []. To check for an empty return, use isempty. • Inputs must be character row vectors.
strjust	MATLAB	—
strncmp	MATLAB	—
strncmpi	MATLAB	• Input values from the char class must be in the range 0-127.
strrep	MATLAB	• Does not support cell arrays. • If oldSubstr does not exist in origStr, returns blanks(0). To check for an empty return, use isempty. • Inputs must be character row vectors.
strtok	MATLAB	—
strtrim	MATLAB	• Supports only inputs from the char class. • Input values must be in the range 0-127.
struct	MATLAB	—
structfun	MATLAB	• Does not support the ErrorHandler option. • The number of outputs must be less than or equal to three.
sub	Fixed-Point Designer	—
sub2ind	MATLAB	• The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported.

Function	Product	Remarks and Limitations
subsasgn	Fixed-Point Designer	—
subspace	MATLAB	—
subsref	Fixed-Point Designer	—
sum	MATLAB	—
sum	Fixed-Point Designer	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
surfacegamma	Phased Array System Toolbox	Does not support variable-size inputs.
surfclutterrcs	Phased Array System Toolbox	Does not support variable-size inputs.
SURFPoints	Computer Vision System Toolbox	—
svd	MATLAB	Uses a different SVD implementation than MATLAB. As the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
swapbytes	MATLAB	Inheritance of the class of the input to swapbytes in a MATLAB Function block is supported only when the class of the input is double. For non-double inputs, the input port data types must be specified, not inherited.
systemp	Phased Array System Toolbox	Does not support variable-size inputs.
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—

Function	Product	Remarks and Limitations
`taylorwin`	Signal Processing Toolbox	• Does not support variable-size inputs. • Inputs must be constant **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`tcdf`	Statistics Toolbox	—
`time2range`	Phased Array System Toolbox	Does not support variable-size inputs.
`times`	MATLAB	—
`times`	Fixed-Point Designer	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object. • When you provide complex inputs to the `times` function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to `On`.
`tinv`	Statistics Toolbox	—
`toeplitz`	MATLAB	—
`tpdf`	Statistics Toolbox	—
`trace`	MATLAB	—
`trapz`	MATLAB	—
`transpose`	MATLAB	—

Function	Product	Remarks and Limitations
`transpose`	Fixed-Point Designer	—
`triang`	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`tril`	MATLAB	- If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
`tril`	Fixed-Point Designer	- If supplied, the index, *k*, must be a real and scalar integer value that is not a `fi` object.
`triu`	MATLAB	- If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
`triu`	Fixed-Point Designer	- If supplied, the index, *k*, must be a real and scalar integer value that is not a `fi` object.
`trnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: - The output is nonscalar. - An input parameter is invalid for the distribution.
`true`	MATLAB	- Dimensions must be real, nonnegative, integers.
`tstat`	Statistics Toolbox	—

Function	Product	Remarks and Limitations
`tukeywin`	Signal Processing Toolbox	- Does not support variable-size inputs. - All inputs must be constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`typecast`	MATLAB	- Value of string input argument `type` must be lower case - You might receive a size error when you use `typecast` with inheritance of input port data types in MATLAB Function blocks. To avoid this error, specify the block's input port data types explicitly.
`ufi`	Fixed-Point Designer	—
`uint8`, `uint16`, `uint32`, `uint64`	MATLAB	You cannot use `uint64` in a : - MATLAB Function block in a Simulink model - MATLAB function in a Stateflow chart
`uint8`, `uint16`, `uint32`, `uint64`	Fixed-Point Designer	You cannot use `uint64` in a : - MATLAB Function block in a Simulink model - MATLAB function in a Stateflow chart
`uminus`	MATLAB	—
`uminus`	Fixed-Point Designer	—

Function	Product	Remarks and Limitations
`unidcdf`	Statistics Toolbox	—
`unidinv`	Statistics Toolbox	—
`unidpdf`	Statistics Toolbox	—
`unidrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`unidstat`	Statistics Toolbox	—
`unifcdf`	Statistics Toolbox	—
`unifinv`	Statistics Toolbox	—
`unifpdf`	Statistics Toolbox	—
`unifrnd`	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
`unifstat`	Statistics Toolbox	—
`unigrid`	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
`union`	MATLAB	- When you do not specify the `'rows'` option: – Inputs `A` and `B` must be vectors with the same orientation. If you specify the `'legacy'` option, inputs `A` and `B` must be row vectors. – The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. – The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example , `zeros(1,0)`, to represent the empty set. – If you specify the `'legacy'` option, empty outputs are row vectors, 1-by-0, never 0-by-0. - When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output `C` is 0-by-0. - When the `setOrder` is `'sorted'` or when you specify the `'legacy'` option, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order. - Complex inputs must be `single` or `double`. - When one input is complex and the other input is real, do one of the following: – Set `setOrder` to `'stable'`. – Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.

Function	Product	Remarks and Limitations
`unique`	MATLAB	• When you do not specify the`'rows'` option: - The input `A` must be a vector. If you specify the `'legacy'` option, the input `A` must be a row vector. - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1. - The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example , `zeros(1,0)`, to represent the empty set. - If you specify the `'legacy'` option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the `'rows'` option and the `'legacy'`option, outputs `ia` and `ic` are column vectors. If these outputs are empty, they are 0-by-1, even if the output `C` is 0-by-0. • When the `setOrder` is `'sorted'` or when you specify the `'legacy'` option, the input `A` must already be sorted in ascending order. The first output, `C`, is sorted in ascending order. • Complex inputs must be `single` or `double`.
`unwrap`	MATLAB	• Row vector input is only supported when the first two inputs are vectors and nonscalar • Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors

Function	Product	Remarks and Limitations
`upfirdn`	Signal Processing Toolbox	- Does not support variable-size inputs. - Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
`uplus`	MATLAB	—
`uplus`	Fixed-Point Designer	—
`upper`	MATLAB	- Supports only `char` inputs. - Input values must be in the range 0-127.
`upperbound`	Fixed-Point Designer	—
`upsample`	Signal Processing Toolbox	- Does not support variable-size inputs. - Either declare input `n` as constant, or use the `assert` function in the calling function to set upper bounds for `n`. For example, `assert(n<10)`
`uv2azel`	Phased Array System Toolbox	Does not support variable-size inputs.
`uv2azelpat`	Phased Array System Toolbox	Does not support variable-size inputs.

Function	Product	Remarks and Limitations
uv2phitheta	Phased Array System Toolbox	Does not support variable-size inputs.
uv2phithetapat	Phased Array System Toolbox	Does not support variable-size inputs.
val2ind	Phased Array System Toolbox	Does not support variable-size inputs.
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Designer	—
wblcdf	Statistics Toolbox	—
wblinv	Statistics Toolbox	—
wblpdf	Statistics Toolbox	—
wblrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: • The output is nonscalar. • An input parameter is invalid for the distribution.
wblstat	Statistics Toolbox	—
wilkinson	MATLAB	—

Function	Product	Remarks and Limitations
xcorr	Signal Processing Toolbox	- Does not support variable-size inputs. - Does not support the case where A is a matrix. - Does not support partial (abbreviated) strings of `biased`, `unbiased`, `coeff`, or `none`. - Computation performed at run time.
xor	MATLAB	—
yulewalk	Signal Processing Toolbox	- Does not support variable-size inputs. - If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. **Specifying constants** To specify a constant input for `fiaccel`, use `coder.Constant`. For more information, see .
zeros	MATLAB	- Dimensions must be real, nonnegative, integers.
zp2tf	MATLAB	—
zscore	Statistics Toolbox	—

Functions Supported for C/C++ Code Generation — Categorical List

In this section...

"Aerospace Toolbox Functions" on page 21-89

"Arithmetic Operator Functions" on page 21-90

"Bit-Wise Operation Functions" on page 21-90

"Casting Functions" on page 21-91

"Communications System Toolbox Functions" on page 21-91

"Complex Number Functions" on page 21-91

"Computer Vision System Toolbox Functions" on page 21-92

"Data and File Management Functions" on page 21-94

"Data Type Functions" on page 21-94

"Derivative and Integral Functions" on page 21-95

"Discrete Math Functions" on page 21-95

"Error Handling Functions" on page 21-96

"Exponential Functions" on page 21-96

"Filtering and Convolution Functions" on page 21-97

"Fixed-Point Designer Functions" on page 21-97

"Histogram Functions" on page 21-106

"Image Processing Toolbox Functions" on page 21-106

"Input and Output Functions" on page 21-108

"Interpolation and Computational Geometry Functions" on page 21-109

"Linear Algebra" on page 21-109

"Logical Operator Functions" on page 21-109

"MATLAB® Compiler™ Functions" on page 21-110

"MATLAB Desktop Environment Functions" on page 21-110

"Matrix and Array Functions" on page 21-110

In this section...

"Nonlinear Numerical Methods" on page 21-114

"Phased Array System Toolbox Functions" on page 21-115

"Polynomial Functions" on page 21-118

"Programming Utilities" on page 21-118

"Relational Operator Functions" on page 21-118

"Rounding and Remainder Functions" on page 21-119

"Set Functions" on page 21-119

"Signal Processing Functions in MATLAB" on page 21-120

"Signal Processing Toolbox Functions" on page 21-120

"Special Values" on page 21-125

"Specialized Math" on page 21-125

"Statistical Functions" on page 21-126

"Statistics Toolbox Functions" on page 21-126

"String Functions" on page 21-132

"Structure Functions" on page 21-133

"Trigonometric Functions" on page 21-133

Aerospace Toolbox Functions

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

Arithmetic Operator Functions

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

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

Bit-Wise Operation Functions

Function	Description
`flintmax`	Largest consecutive integer in floating-point format
`swapbytes`	Swap byte ordering

Casting Functions

Data Type	Description
cast	Cast variable to different data type
char	Create character array (string)
class	Query class of object argument
double	Convert to double-precision floating point
int8, int16, int32, int64	Convert to signed integer data type
logical	Convert to Boolean true or false data type
single	Convert to single-precision floating point
typecast	Convert data types without changing underlying data
uint8, uint16, uint32, uint64	Convert to unsigned integer data type

Communications System Toolbox Functions

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

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components
conj	Return the conjugate of a complex number
imag	Return the imaginary part of a complex number
isnumeric	Return true for numeric arrays

Function	Description
`isreal`	Return `false` (0) for a complex number
`isscalar`	Return `true` if array is a scalar
`real`	Return the real part of a complex number
`unwrap`	Correct phase angles to produce smoother phase plots

Computer Vision System Toolbox Functions

Function	Description
`binaryFeatures`	Object for storing binary feature vectors
`cornerPoints`	Object for storing corner points
`detectFASTFeatures`	Find corners using FAST algorithm
`detectMSERFeatures`	Detect MSER features
`detectSURFFeatures`	Detect SURF features
`disparity`	Disparity map between stereo images

Function	Description
`epipolarLine`	Compute epipolar lines for stereo images
`estimateFundamentalMatrix`	Estimate fundamental matrix from corresponding points in stereo image
`estimateUncalibratedRectification`	Uncalibrated stereo rectification
`extractFeatures`	Extract interest point descriptors
`insertMarker`	Insert markers in image or video
`insertShape`	Insert shapes in image or video
`integralImage`	Compute integral image
`isEpipoleInImage`	Determine whether image contains epipole
`lineToBorderPoints`	Intersection points of lines in image and image border

Function	Description
matchFeatures	Find matching image features
MSERRegions	Object for storing MSER regions
SURFPoints	Object for storing SURF interest points
vision.KalmanFilter	Kalman filter for object tracking

Data and File Management Functions

Function	Description
computer	Information about computer on which MATLAB software is running
fclose	Close one or all open files
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
load	Load data from MAT-file into workspace

Data Type Functions

Function	Description
deal	Distribute inputs to outputs
iscell	Determine whether input is cell array

Function	Description
`nargchk`	Validate number of input arguments **Note** `nargchk` will be removed in a future release.
`narginchk`	Validate number of input arguments
`nargoutchk`	Validate number of output arguments
`str2func`	Construct function handle from function name string
`structfun`	Apply function to each field of scalar structure

Derivative and Integral Functions

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

Discrete Math Functions

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

Error Handling Functions

Function	Description
`assert`	Generate error when condition is violated
`error`	Display message and abort function

Exponential Functions

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

Filtering and Convolution Functions

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

Fixed-Point Designer Functions

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

- `fipref` and `quantizer` objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numerictype` of a given `fi` variable after that variable has been created.
- The `boolean` value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- You can use parallel for (`parfor`) loops in code compiled with `fiaccel`, but those loops are treated like regular `for` loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- The general limitations of C/C++ code generated from MATLAB apply. For more information, see “MATLAB Language Features Supported for C/C++ Code Generation”.

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	N/A
all	N/A
any	N/A
atan2	N/A
bitand	Not supported for slope-bias scaled fi objects.
bitandreduce	N/A
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled fi objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	Generated code may not handle out of range shifting.
bitsra	Generated code may not handle out of range shifting.
bitsrl	Generated code may not handle out of range shifting.
bitxor	Not supported for slope-bias scaled fi objects.
bitxorreduce	N/A
ceil	N/A

Function	Remarks/Limitations
`complex`	N/A
`conj`	N/A
`conv`	• Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`. • For variable-sized signals, you may see different results between generated code and MATLAB. – In the generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`. – In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.
`convergent`	N/A
`cordicabs`	Variable-size signals are not supported.
`cordicangle`	Variable-size signals are not supported.
`cordicatan2`	Variable-size signals are not supported.
`cordiccart2pol`	Variable-size signals are not supported.
`cordiccexp`	Variable-size signals are not supported.
`cordiccos`	Variable-size signals are not supported.
`cordicpol2cart`	Variable-size signals are not supported.
`cordicrotate`	Variable-size signals are not supported.
`cordicsin`	Variable-size signals are not supported.
`cordicsincos`	Variable-size signals are not supported.
`cos`	N/A
`ctranspose`	N/A
`diag`	If supplied, the index, *k*, must be a real and scalar integer value that is not a `fi` object.

Function	Remarks/Limitations
divide	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • Complex and imaginary divisors are not supported. • Code generation in MATLAB does not support the syntax T.divide(a,b).
double	N/A
end	N/A
eps	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and fi double signals.
eq	Not supported for fixed-point signals with different biases.
fi	• The default constructor syntax without any input arguments is not supported. • If the numerictype is not fully specified, the input to fi must be a constant, a fi, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows fi to autoscale its fraction length based on the known data type of the input. • numerictype object information must be available for nonfixed-point Simulink inputs.
filter	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
fimath	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a fimath object. You define this object in the MATLAB Function block dialog in the Model Explorer. • Use to create fimath objects in the generated code.
fix	N/A
fixed.Quantizer	N/A
floor	N/A

Function	Remarks/Limitations
ge	Not supported for fixed-point signals with different biases.
get	The syntax structure = get(o) is not supported.
getlsb	N/A
getmsb	N/A
gt	Not supported for fixed-point signals with different biases.
hdlram	N/A
horzcat	N/A
imag	
int8, int16, int32, int64	You cannot use int64 in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow chart.
iscolumn	N/A
isempty	N/A
isequal	N/A
isfi	N/A
isfimath	N/A
isfimathlocal	N/A
isfinite	N/A
isinf	N/A
isnan	N/A
isnumeric	N/A
isnumerictype	N/A
isreal	N/A
isrow	N/A
isscalar	N/A
issigned	N/A
isvector	N/A
le	Not supported for fixed-point signals with different biases.

Function	Remarks/Limitations
length	N/A
logical	N/A
lowerbound	N/A
lsb	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and double signals.
lt	Not supported for fixed-point signals with different biases.
max	N/A
mean	N/A
median	N/A
min	N/A
minus	Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
mpower	• When the exponent k is a variable and the input is a scalar, the ProductMode property of the governing fimath must be SpecifyPrecision. • When the exponent k is a variable and the input is not scalar, the SumMode property of the governing fimath must be SpecifyPrecision. • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to SpecifyPrecision or Keep LSB. • For variable-sized signals, you may see different results between the generated code and MATLAB. - In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when the first input, *a*, is nonscalar. However, when *a* is a scalar,

Function	Remarks/Limitations
	MATLAB computes the output using the ProductMode of the governing fimath.
mpy	When you provide complex inputs to the mpy function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
mrdivide	N/A
mtimes	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to SpecifyPrecision or KeepLSB. • For variable-sized signals, you may see different results between the generated code and MATLAB. - In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath. - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	numberofelements will be removed in a future release. Use numel instead.
numel	N/A

Function	Remarks/Limitations
`numerictype`	• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a `numerictype` object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a nonfixed-point signal. • Use to create `numerictype` objects in generated code.
`permute`	N/A
`plus`	Any non-`fi` inputs must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`pow2`	N/A
`power`	When the exponent `k` is a variable, the `ProductMode` property of the governing `fimath` must be `SpecifyPrecision`.
`quantize`	N/A
`range`	N/A
`rdivide`	N/A
`real`	N/A
`realmax`	N/A
`realmin`	N/A
`reinterpretcast`	N/A
`removefimath`	N/A
`repmat`	N/A
`rescale`	N/A
`reshape`	N/A
`round`	N/A
`setfimath`	N/A
`sfi`	N/A
`sign`	N/A
`sin`	N/A

Function	Remarks/Limitations
single	N/A
size	N/A
sort	N/A
sqrt	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
times	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • When you provide complex inputs to the times function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
transpose	N/A
tril	If supplied, the index, *k*, must be a real and scalar integer value that is not a fi object.
triu	If supplied, the index, *k*, must be a real and scalar integer value that is not a fi object.
ufi	N/A
uint8, uint16, uint32, uint64	You cannot use uint64 in a MATLAB Function block in a Simulink model or in a MATLAB function in a Stateflow chart.

Function	Remarks/Limitations
`uminus`	N/A
`uplus`	N/A
`upperbound`	N/A
`vertcat`	N/A

Histogram Functions

Function	Description
`hist`	Non-graphical histogram
`histc`	Histogram count

Image Processing Toolbox Functions

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

Function	Remarks/Limitations
`bwlookup`	For best results, specify an input image of class `logical`.
`bwmorph`	The text string specifying the operation must be a constant and, for best results, specify an input image of class `logical`.
`conndef`	All input arguments must be compile-time constants.
`fspecial`	Allinputs must be compile-time constants. Expressions or variables are allowed if their values do not change.
`imcomplement`	Does not support `int64` and `uint64` data types.

Function	Remarks/Limitations
imfill	The optional input connectivity, conn and the string 'holes' must be a compile time constants. Supports only up to 3-D inputs. (No N-D support.) The interactive mode to select points, imfill(BW,0,CONN) is not supported in code generation. locations can be a *P*-by-1 vector, in which case it contains the linear indices of the starting locations. locations can also be a *P*-by-ndims(I) matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run-time. However, the number of points in locations can be varied at run-time. Generated code for this function uses a precompiled platform-specific shared library.
imhmax	The optional third input argument, conn, must be a compile-time constant Generated code for this function uses a precompiled platform-specific shared library.
imhmin	The optional third input argument, conn, must be a compile-time constant Generated code for this function uses a precompiled platform-specific shared library.
imreconstruct	The optional third input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
imregionalmax	The optional second input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.

Function	Remarks/Limitations
imregionalmin	The optional second input argument, conn, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.
iptcheckconn	All input arguments must be compile-time constants.
label2rgb	Referring to the standard syntax: RGB = label2rgb(L, map, zerocolor, order) • Submit at least two input arguments: the label matrix, L, and the colormap matrix, map. • map must be an n-by-3, double, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function. • If you set the boundary color zerocolor to the same color as one of the regions, label2rgb will not issue a warning. • If you supply a value for order, it must be 'noshuffle'.
padarray	Support only up to 3-D inputs. Input arguments, padval and direction are expected to be compile-time constants.

Input and Output Functions

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

Interpolation and Computational Geometry Functions

Function	Description
cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
meshgrid	Generate X and Y arrays for 3-D plots
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Linear Algebra

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

Logical Operator Functions

Function	Description
and	Logical AND (&&)
bitand	Bitwise AND
bitcmp	Bitwise complement

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

MATLAB Compiler Functions

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

MATLAB Desktop Environment Functions

Function	Description
ismac	Determine if version is for Mac OS X platform
ispc	Determine if version is for Windows (PC) platform
isunix	Determine if version is for UNIX® platform

Matrix and Array Functions

Function	Description
abs	Return absolute value and complex magnitude of an array
all	Test if all elements are nonzero

Function	Description
`angle`	Phase angle
`any`	Test for any nonzero elements
`blkdiag`	Construct block diagonal matrix from input arguments
`bsxfun`	Applies element-by-element binary operation to two arrays with singleton expansion enabled
`cat`	Concatenate arrays along specified dimension
`circshift`	Shift array circularly
`compan`	Companion matrix
`cond`	Condition number of a matrix with respect to inversion
`cov`	Covariance matrix
`cross`	Vector cross product
`cumprod`	Cumulative product of array elements
`cumsum`	Cumulative sum of array elements
`det`	Matrix determinant
`diag`	Return a matrix formed around the specified diagonal vector and the specified diagonal (0, 1, 2,...) it occupies
`diff`	Differences and approximate derivatives
`dot`	Vector dot product
`eig`	Eigenvalues and eigenvectors
`eye`	Identity matrix
`false`	Return an array of 0s for the specified dimensions
`find`	Find indices and values of nonzero elements
`flipdim`	Flip array along specified dimension
`fliplr`	Flip matrix left to right
`flipud`	Flip matrix up to down
`full`	Convert sparse matrix to full matrix
`hadamard`	Hadamard matrix

Function	Description
hankel	Hankel matrix
hilb	Hilbert matrix
ind2sub	Subscripts from linear index
inv	Inverse of a square matrix
invhilb	Inverse of Hilbert matrix
ipermute	Inverse permute dimensions of array
iscolumn	True if input is a column vector
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequaln	Test arrays for equality, treating NaNs as equal
isfinite	Detect finite elements of an array
isfloat	Determine if input is floating-point array
isinf	Detect infinite elements of an array
isinteger	Determine if input is integer array
islogical	Determine if input is logical array
ismatrix	True if input is a matrix
isnan	Detect NaN elements of an array
isrow	True if input is a row vector
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
kron	Kronecker tensor product
length	Return the length of a matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square

Function	Description
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndgrid	Generate arrays for N-D functions and interpolation
ndims	Number of dimensions
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
norm	Vector and matrix norms
normest	2-norm estimate
numel	Number of elements in array or subscripted array
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed random numbers
randperm	Random permutation
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array
reshape	Reshape one array into the dimensions of another
rng	Control random number generation
rosser	Classic symmetric eigenvalue test problem

Function	Description
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Nonlinear Numerical Methods

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

Phased Array System Toolbox Functions

Function	Description
aictest	Dimension of signal subspace
albersheim	Required SNR using Albersheim's equation
ambgfun	Ambiguity function
aperture2gain	Convert effective aperture to gain
az2broadside	Convert azimuth angle to broadside angle
azel2phitheta	Convert angles from azimuth/elevation form to phi/theta form
azel2phithetapat	Convert radiation pattern from azimuth/elevation to phi/theta form
azel2uv	Convert azimuth/elevation angles to u/v coordinates
azel2uvpat	Convert radiation pattern from azimuth/elevation form to u/v form
azelaxes	Spherical basis vectors in 3-by-3 matrix form
beat2range	Convert beat frequency to range
billingsleyicm	Billingsley's intrinsic clutter motion (ICM) model
broadside2az	Convert broadside angle to azimuth angle
cart2sphvec	Convert vector from Cartesian components to spherical representation
cbfweights	Conventional beamformer weights
circpol2pol	Convert circular component representation of field to linear component representation
dechirp	Perform dechirp operation on FMCW signal
delayseq	Delay or advance sequence
depressionang	Depression angle of surface target
dop2speed	Convert Doppler shift to speed
dopsteeringvec	Doppler steering vector
effearthradius	Effective earth radius
espritdoa	Direction of arrival using TLS ESPRIT
fspl	Free space path loss
gain2aperture	Convert gain to effective aperture

Function	Description
global2localcoord	Convert global to local coordinates
grazingang	Grazing angle of surface target
horizonrange	Horizon range
lcmvweights	Narrowband linearly constrained minimum variance (LCMV) beamformer weights
local2globalcoord	Convert local to global coordinates
mdltest	Dimension of signal subspace
mvdrweights	Minimum variance distortionless response (MVDR) beamformer weights
noisepow	Receiver noise power
npwgnthresh	Detection SNR threshold for signal in white Gaussian noise
phitheta2azel	Convert angles from phi/theta form to azimuth/elevation form
phitheta2azelpat	Convert radiation pattern from phi/theta form to azimuth/elevation form
phitheta2uv	Convert phi/theta angles to u/v coordinates
phitheta2uvpat	Convert radiation pattern from phi/theta form to u/v form
physconst	Physical constants
pol2circpol	Convert linear component representation of field to circular component representation
polellip	Parameters of ellipse traced out by tip of a polarized field vector
polloss	Polarization loss
polratio	Ratio of vertical to horizontal linear polarization components of a field
polsignature	Copolarization and cross-polarization signatures
pulsint	Pulse integration
radareqpow	Peak power estimate from radar equation
radareqrng	Maximum theoretical range estimate
radareqsnr	SNR estimate from radar equation

Function	Description
radarvcd	Vertical coverage diagram
radialspeed	Relative radial speed
range2beat	Convert range to beat frequency
range2bw	Convert range resolution to required bandwidth
range2time	Convert propagation distance to propagation time
rangeangle	Range and angle calculation
rdcoupling	Range Doppler coupling
rocpfa	Receiver operating characteristic curves by false-alarm probability
rocsnr	Receiver operating characteristic curves by SNR
rootmusicdoa	Direction of arrival using Root MUSIC
rotx	Rotation matrix for rotations around x-axis
roty	Rotation matrix for rotations around y-axis
rotz	Rotation matrix for rotations around z-axis
sensorcov	Sensor spatial covariance matrix
sensorsig	Simulate received signal at sensor array
shnidman	Required SNR using Shnidman's equation
speed2dop	Convert speed to Doppler shift
sph2cartvec	Convert vector from spherical basis components to Cartesian components
spsmooth	Spatial smoothing
steervec	Steering vector
stokes	Stokes parameters of polarized field
stretchfreq2rng	Convert frequency offset to range
surfacegamma	Gamma value for different terrains
surfclutterrcs	Surface clutter radar cross section (RCS)
systemp	Receiver system-noise temperature
time2range	Convert propagation time to propagation distance

Function	Description
unigrid	Uniform grid
uv2azel	Convert u/v coordinates to azimuth/elevation angles
uv2azelpat	Convert radiation pattern from u/v form to azimuth/elevation form
uv2phitheta	Convert u/v coordinates to phi/theta angles
uv2phithetapat	Convert radiation pattern from u/v form to phi/theta form
val2ind	Uniform grid index

Polynomial Functions

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

Programming Utilities

Function	Description
mfilename	File name of currently running function

Relational Operator Functions

Function	Description
eq	Equal (==)
ge	Greater than or equal to (>=)
gt	Greater than (>)
le	Less than or equal to (<=)

Function	Description
`lt`	Less than (<)
`ne`	Not equal (~=)

Rounding and Remainder Functions

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

Set Functions

Function	Description
`intersect`	Set intersection of two arrays
`ismember`	Array elements that are members of set array
`issorted`	Determine whether set elements are in sorted order
`setdiff`	Set difference of two arrays

Function	Description
`setxor`	Set exclusive OR of two arrays
`union`	Set union of two arrays
`unique`	Unique values in array

Signal Processing Functions in MATLAB

Function	Description
`chol`	Cholesky factorization
`conv`	Convolution and polynomial multiplication
`fft`	Discrete Fourier transform
`fft2`	2-D discrete Fourier transform
`fftn`	N-D discrete Fourier transform
`fftshift`	Shift zero-frequency component to center of spectrum
`filter`	Filter a data sequence using a digital filter that works for both real and complex inputs
`freqspace`	Frequency spacing for frequency response
`ifft`	Inverse discrete Fourier transform
`ifft2`	2-D inverse discrete Fourier transform
`ifftn`	N-D inverse discrete Fourier transform
`ifftshift`	Inverse discrete Fourier transform shift
`svd`	Singular value decomposition
`zp2tf`	Convert zero-pole-gain filter parameters to transfer function form

Signal Processing Toolbox Functions

These functions do not support variable-size inputs, you must define the size and type of the function inputs. For more information, see “Specifying Inputs in Code Generation from MATLAB ”.

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

Function	Remarks/Limitations
`barthannwin`	Window length must be a constant. Expressions or variables are allowed if their values do not change.
`bartlett`	Window length must be a constant. Expressions or variables are allowed if their values do not change.
`besselap`	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
`bitrevorder`	—
`blackman`	Window length must be a constant. Expressions or variables are allowed if their values do not change.
`blackmanharris`	Window length must be a constant. Expressions or variables are allowed if their values do not change.
`bohmanwin`	Window length must be a constant. Expressions or variables are allowed if their values do not change.
`buttap`	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
`butter`	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
`buttord`	All inputs must be constants. Expressions or variables are allowed if their values do not change.
`cfirpm`	All inputs must be constants. Expressions or variables are allowed if their values do not change.
`cheb1ap`	All inputs must be constants. Expressions or variables are allowed if their values do not change.
`cheb2ap`	All inputs must be constants. Expressions or variables are allowed if their values do not change.
`cheb1ord`	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
dct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
firls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	All inputs must be constants. Expressions or variables are allowed if their values do not change.
flattopwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See "freqz With No Output Arguments".
gaussfir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
gausswin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hamming	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hann	All inputs must be constant. Expressions or variables are allowed if their values do not change.
idct	Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
intfilt	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiser	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiserord	—
levinson	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
`maxflat`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`nuttallwin`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`parzenwin`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`rectwin`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`resample`	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
`sgolay`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`sosfilt`	—
`taylorwin`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`triang`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`tukeywin`	All inputs must be constant. Expressions or variables are allowed if their values do not change.
`upfirdn`	• Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. • Variable-size inputs are not supported.
`upsample`	Either declare input `n` as constant, or use the `assert` function in the calling function to set upper bounds for `n`. For example, `assert(n<10)`
`xcorr`	—
`yulewalk`	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Special Values

Symbol	Description
`eps`	Floating-point relative accuracy
`inf`	IEEE® arithmetic representation for positive infinity
`intmax`	Largest possible value of specified integer type
`intmin`	Smallest possible value of specified integer type
NaN or `nan`	Not a number
`pi`	Ratio of the circumference to the diameter for a circle
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

Specialized Math

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

Symbol	Description
gammaincinv	Inverse incomplete gamma function
gammaln	Logarithm of the gamma function
psi	Psi (polygamma) function

Statistical Functions

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

Statistics Toolbox Functions

Function	Description
betacdf	Beta cumulative distributive function
betainv	Beta inverse cumulative distribution function
betapdf	Beta probability density function
betarnd	Beta random numbers
betastat	Beta mean and variance
binocdf	Binomial cumulative distribution function
binoinv	Binomial inverse cumulative distribution function
binopdf	Binomial probability density function
binornd	Binomial random numbers
binostat	Binomial mean and variance

Function	Description
`cdf`	Cumulative distribution function of probability distribution object
`chi2cdf`	Chi-square cumulative distribution function
`chi2inv`	Chi-square inverse cumulative distribution function
`chi2pdf`	Chi-square probability density function
`chi2rnd`	Chi-square random numbers
`chi2stat`	Chi-square mean and variance
`evcdf`	Extreme value cumulative distribution function
`evinv`	Extreme value inverse cumulative distribution
`evpdf`	Extreme value probability density function
`evrnd`	Extreme value random numbers
`evstat`	Extreme value mean and variance
`expcdf`	Exponential cumulative distribution function
`expinv`	Exponential inverse cumulative distribution function
`exppdf`	Exponential probability density function
`exprnd`	Exponential random numbers
`expstat`	Exponential mean and variance
`fcdf`	F cumulative distribution function
`finv`	F inverse cumulative distribution function
`fpdf`	F probability density function
`frnd`	F random numbers
`fstat`	F mean and variance
`gamcdf`	Gamma cumulative distribution function
`gaminv`	Gamma inverse cumulative distribution function
`gampdf`	Gamma probability density function
`gamrnd`	Gamma random numbers
`gamstat`	Gamma mean and variance

Function	Description
`geocdf`	Geometric cumulative distribution function
`geoinv`	Geometric inverse cumulative distribution function
`geomean`	Geometric mean
`geopdf`	Geometric probability density function
`geornd`	Geometric random numbers
`geostat`	Geometric mean and variance
`gevcdf`	Generalized extreme value cumulative distribution function
`gevinv`	Generalized extreme value inverse cumulative distribution function
`gevpdf`	Generalized extreme value probability density function
`gevrnd`	Generalized extreme value random numbers
`gevstat`	Generalized extreme value mean and variance
`gpcdf`	Generalized Pareto cumulative distribution function
`gpinv`	Generalized Pareto inverse cumulative distribution function
`gppdf`	Generalized Pareto probability density function
`gprnd`	Generalized Pareto random numbers
`gpstat`	Generalized Pareto mean and variance
`harmmean`	Harmonic mean
`hygecdf`	Hypergeometric cumulative distribution function
`hygeinv`	Hypergeometric inverse cumulative distribution function
`hygepdf`	Hypergeometric probability density function
`hygernd`	Hypergeometric random numbers
`hygestat`	Hypergeometric mean and variance
`icdf`	Inverse cumulative distribution function of probability distribution object
`iqr`	Interquartile range of probability distribution object

Function	Description
kurtosis	Kurtosis
logncdf	Lognormal cumulative distribution function
logninv	Lognormal inverse cumulative distribution function
lognpdf	Lognormal probability density function
lognrnd	Lognormal random numbers
lognstat	Lognormal mean and variance
mad	Mean or median absolute deviation
mnpdf	Multinomial probability density function
moment	Central moments
nancov	Covariance ignoring NaN values
nanmax	Maximum ignoring NaN values
nanmean	Mean ignoring NaN values
nanmedian	Median ignoring NaN values
nanmin	Minimum ignoring NaN values
nanstd	Standard deviation ignoring NaN values
nansum	Sum ignoring NaN values
nanvar	Variance, ignoring NaN values
nbincdf	Negative binomial cumulative distribution function
nbininv	Negative binomial inverse cumulative distribution function
nbinpdf	Negative binomial probability density function
nbinrnd	Negative binomial random numbers
nbinstat	Negative binomial mean and variance
ncfcdf	Noncentral F cumulative distribution function
ncfinv	Noncentral F inverse cumulative distribution function
ncfpdf	Noncentral F probability density function
ncfrnd	Noncentral F random numbers

Function	Description
`ncfstat`	Noncentral F mean and variance
`nctcdf`	Noncentral t cumulative distribution function
`nctinv`	Noncentral t inverse cumulative distribution function
`nctpdf`	Noncentral t probability density function
`nctrnd`	Noncentral t random numbers
`nctstat`	Noncentral t mean and variance
`ncx2cdf`	Noncentral chi-square cumulative distribution function
`ncx2rnd`	Noncentral chi-square random numbers
`ncx2stat`	Noncentral chi-square mean and variance
`normcdf`	Normal cumulative distribution function
`norminv`	Normal inverse cumulative distribution function
`normpdf`	Normal probability density function
`normrnd`	Normal random numbers
`normstat`	Normal mean and variance
`pdf`	Probability density function of probability distribution object
`poisscdf`	Poisson cumulative distribution function
`poissinv`	Poisson inverse cumulative distribution function
`poisspdf`	Poisson probability density function
`poissrnd`	Poisson random numbers
`poisstat`	Poisson mean and variance
`prctile`	Percentiles of a data set
`quantile`	Quantiles of a data set
`randg`	Gamma random numbers with unit scale
`random`	Random numbers
`raylcdf`	Rayleigh cumulative distribution function
`raylinv`	Rayleigh inverse cumulative distribution function

Function	Description
raylpdf	Rayleigh probability density function
raylrnd	Rayleigh random numbers
raylstat	Rayleigh mean and variance
skewness	Skewness
tcdf	Student's *t* cumulative distribution function
tinv	Student's *t* inverse cumulative distribution function
tpdf	Student's *t* probability density function
trnd	Student's *t* random numbers
tstat	Student's *t* mean and variance
unidcdf	Discrete uniform cumulative distribution function
unidinv	Discrete uniform inverse cumulative distribution function
unidpdf	Discrete uniform probability density function
unidrnd	Discrete uniform random numbers
unidstat	Discrete uniform mean and variance
unifcdf	Continuous uniform cumulative distribution function
unifinv	Continuous uniform inverse cumulative distribution function
unifpdf	Continuous uniform probability density function
unifrnd	Continuous uniform random numbers
unifstat	Continuous uniform mean and variance
wblcdf	Weibull cumulative distribution function
wblinv	Weibull inverse cumulative distribution function
wblpdf	Weibull probability density function
wblrnd	Weibull random numbers
wblstat	Weibull mean and variance
zscore	Standardized *z*-scores

String Functions

Function	Description
bin2dec	Convert binary number string to decimal number
bitmax	Maximum double-precision floating-point integer
blanks	Create string of blank characters
char	Create character array (string)
deblank	Strip trailing blanks from end of string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
ischar	True for character array (string)
isletter	Array elements that are alphabetic letters
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
lower	Convert string to lowercase
num2hex	Convert singles and doubles to IEEE hexadecimal strings
strcmp	Compare strings (case sensitive)
strcmpi	Compare strings (case insensitive)
strfind	Find one string within another
strjust	Justify character array
strncmp	Compare first n characters of strings (case sensitive)
strncmpi	Compare first n characters of strings (case insensitive)
strrep	Find and replace substring
strtok	Selected parts of string
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

Structure Functions

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

Trigonometric Functions

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atan2d	Four-quadrant inverse tangent, result in degrees
atand	Inverse tangent; result in degrees

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

22

Code Generation for Variable-Size Data

- “What Is Variable-Size Data?” on page 22-2
- “Variable-Size Data Definition for Code Generation” on page 22-3
- “Bounded Versus Unbounded Variable-Size Data” on page 22-4
- “Control Memory Allocation of Variable-Size Data” on page 22-5
- “Specify Variable-Size Data Without Dynamic Memory Allocation” on page 22-6
- “Variable-Size Data in Code Generation Reports” on page 22-10
- “Define Variable-Size Data for Code Generation” on page 22-12
- “C Code Interface for Arrays” on page 22-19
- “Diagnose and Fix Variable-Size Data Errors” on page 22-23
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 22-27
- “Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation” on page 22-36

What Is Variable-Size Data?

Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.

For example, in the following MATLAB function nway, B is a variable-size array; its length is not known at compile time.

```
function B = nway(A,n)
% Compute average of every N elements of A and put them in B.
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
         B(i) = mean(A(k + (0:n-1)));
         k = k + n;
    end
else
    error('n <= 0 or does not divide number of elements evenly');
end
```

Variable-Size Data Definition for Code Generation

In the MATLAB language, data can vary in size. By contrast, the semantics of generated code constrains the class, complexity, and shape of every expression, variable, and structure field. Therefore, for code generation, you must use each variable consistently. Each variable must:

- Be either complex or real (determined at first assignment)
- Have a consistent shape

  For fixed-size data, the shape is the same as the size returned in MATLAB. For example, if `size(A) == [4 5]`, the shape of variable `A` is 4 x 5. For variable-size data, the shape can be abstract. That is, one or more dimensions can be unknown (such as `4x?` or `?x?`).

By default, the compiler detects code logic that attempts to change these fixed attributes after initial assignments, and flags these occurrences as errors during code generation. However, you can override this behavior by defining variables or structure fields as variable-size data.

For more information, see “Bounded Versus Unbounded Variable-Size Data” on page 22-4

Bounded Versus Unbounded Variable-Size Data

You can generate code for bounded and unbounded variable-size data. *Bounded variable-size data* has fixed upper bounds; this data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds; this data *must* be allocated on the heap. If you use unbounded data, you must use dynamic memory allocation so that the compiler:

- Does not check for upper bounds
- Allocates memory on the heap instead of the stack

You can control the memory allocation of variable-size data. For more information, see "Control Memory Allocation of Variable-Size Data" on page 22-5.

Control Memory Allocation of Variable-Size Data

Data whose size (in bytes) is greater than or equal to the dynamic memory allocation threshold is allocated on the heap. The default dynamic memory allocation threshold is 64 kilobytes. Data whose size is less than this threshold is allocated on the stack.

Dynamic memory allocation is an expensive operation; the performance cost might be too high for small data sets. If you use small variable-size data sets or data that does not change size at run time, disable dynamic memory allocation. See “Control Dynamic Memory Allocation” on page 9-94.

. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 22-6.

Specify Variable-Size Data Without Dynamic Memory Allocation

In this section…

“Fixing Upper Bounds Errors” on page 22-6

“Specifying Upper Bounds for Variable-Size Data” on page 22-6

Fixing Upper Bounds Errors

If MATLAB cannot determine or compute the upper bound, you must specify an upper bound. See “Specifying Upper Bounds for Variable-Size Data” on page 22-6 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 22-25

Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 22-6
- “Specifying Upper Bounds on the Command Line for Variable-Size Inputs” on page 22-6
- “Specifying Unknown Upper Bounds for Variable-Size Inputs” on page 22-7
- “Specifying Upper Bounds for Local Variable-Size Data” on page 22-7
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 22-8

When to Specify Upper Bounds for Variable-Size Data

When using static allocation on the stack during code generation, MATLAB must be able to determine upper bounds for variable-size data. Specify the upper bounds explicitly for variable-size data from external sources, such as inputs and outputs.

Specifying Upper Bounds on the Command Line for Variable-Size Inputs

Use the `coder.typeof` construct with the `-args` option on the `codegen` command line (requires a MATLAB Coder license). For example:

```
codegen foo -args {coder.typeof(double(0),[3 100],1)}
```

This command specifies that the input to function foo is a matrix of real doubles with two variable dimensions. The upper bound for the first dimension is 3; the upper bound for the second dimension is 100. For a detailed explanation of this syntax, see coder.typeof.

Specifying Unknown Upper Bounds for Variable-Size Inputs

If you use dynamic memory allocation, you can specify that you don't know the upper bounds of inputs. To specify an unknown upper bound, use the infinity constant Inf in place of a numeric value. For example:

```
codegen foo -args {coder.typeof(double(0), [1 Inf])}
```

In this example, the input to function foo is a vector of real doubles without an upper bound.

Specifying Upper Bounds for Local Variable-Size Data

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

You do not need to specify upper bounds when using dynamic allocation on the heap. In this case, MATLAB assumes variable-size data is unbounded and does not attempt to determine upper bounds.

Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data. Use the assert function with relational operators to constrain the value of variables that specify the dimensions of variable-size data. For example:

```
function y = dim_need_bound(n) %#codegen
assert (n <= 5);
L= ones(n,n);
M = zeros(n,n);
M = [L; M];
y = M;
```

This assert statement constrains input n to a maximum size of 5, defining L and M as variable-sized matrices with upper bounds of 5 for each dimension.

Specifying the Upper Bounds for All Instances of a Local Variable. Use the coder.varsize function to specify the upper bounds for all instances of a local variable in a function. For example:

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

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

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

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

Using a Matrix Constructor with Nonconstant Dimensions

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

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

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

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

Variable-Size Data in Code Generation Reports

In this section...

"What Reports Tell You About Size" on page 22-10

"How Size Appears in Code Generation Reports" on page 22-11

"How to Generate a Code Generation Report" on page 22-11

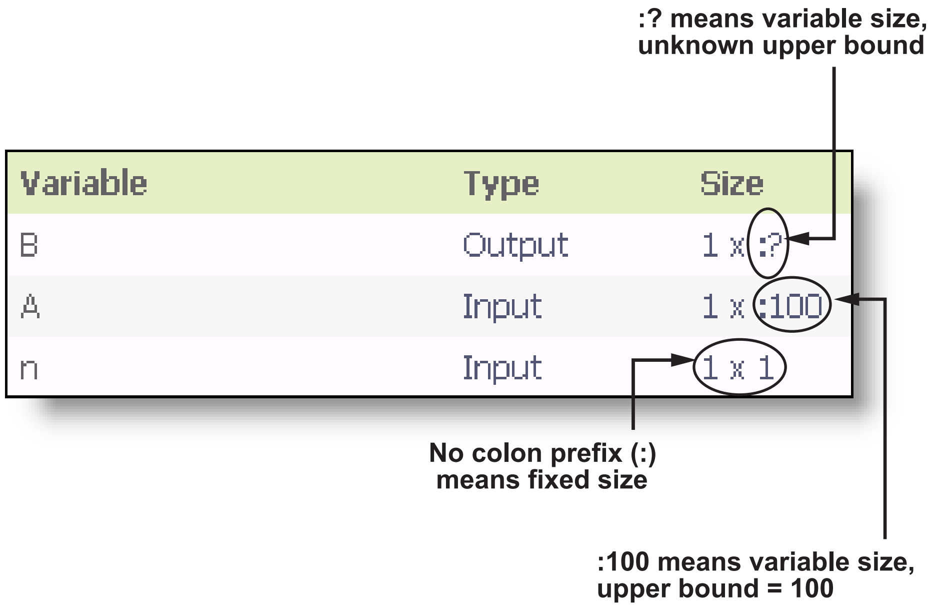
What Reports Tell You About Size

Code generation reports:

- Differentiate fixed-size from variable-size data
- Identify variable-size data with unknown upper bounds
- Identify variable-size data with fixed dimensions

  If you define a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends `*` to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions does not change during execution.
- Provide guidance on how to fix size mismatch and upper bounds errors.

How Size Appears in Code Generation Reports

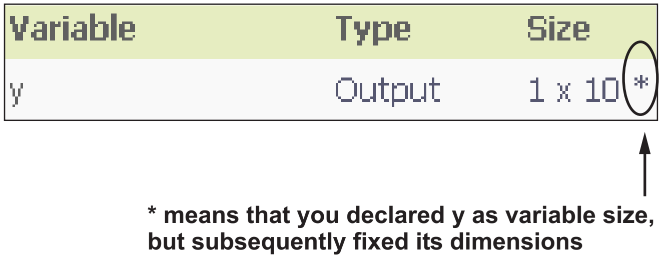


Variable	Type	Size
B	Output	1 x :?
A	Input	1 x :100
n	Input	1 x 1

:? means variable size, unknown upper bound

No colon prefix (:) means fixed size

:100 means variable size, upper bound = 100



Variable	Type	Size
y	Output	1 x 10*

* means that you declared y as variable size, but subsequently fixed its dimensions

How to Generate a Code Generation Report

Add the `-report` option to your `fiaccel` command.

Define Variable-Size Data for Code Generation

In this section...

"When to Define Variable-Size Data Explicitly" on page 22-12

"Using a Matrix Constructor with Nonconstant Dimensions" on page 22-13

"Inferring Variable Size from Multiple Assignments" on page 22-13

"Defining Variable-Size Data Explicitly Using coder.varsize" on page 22-15

When to Define Variable-Size Data Explicitly

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

Method	See
Assign the data from a variable-size matrix constructor such as • `ones` • `zeros` • `repmat`	"Using a Matrix Constructor with Nonconstant Dimensions" on page 22-13
Assign multiple, constant sizes to the same variable before using (reading) the variable.	"Inferring Variable Size from Multiple Assignments" on page 22-13
Define all instances of a variable to be variable sized	"Defining Variable-Size Data Explicitly Using coder.varsize" on page 22-15

Using a Matrix Constructor with Nonconstant Dimensions

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

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

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

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

Inferring Variable Size from Multiple Assignments

You can define variable-size data by assigning multiple, constant sizes to the same variable before you use (read) the variable in your code. When MATLAB uses static allocation on the stack for code generation, it infers the upper bounds from the largest size specified for each dimension. When you assign the same size to a given dimension across all assignments, MATLAB assumes that the dimension is fixed at that size. The assignments can specify different shapes as well as sizes.

When dynamic memory allocation is used, MATLAB does not check for upper bounds; it assumes variable-size data is unbounded.

Inferring Upper Bounds from Multiple Definitions with Different Shapes

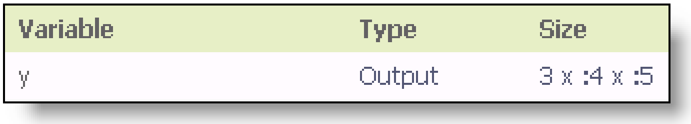
```
function y = var_by_multiassign(u) %#codegen
```

```
if (u > 0)
   y = ones(3,4,5);
else
   y = zeros(3,1);
end
```

When static allocation is used, this function infers that y is a matrix with three dimensions, where:

- First dimension is fixed at size 3
- Second dimension is variable with an upper bound of 4
- Third dimension is variable with an upper bound of 5

The code generation report represents the size of matrix y like this:

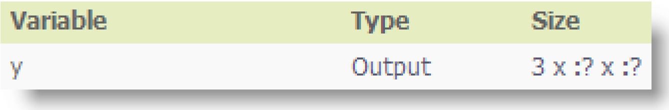


Variable	Type	Size
y	Output	3 x :4 x :5

When dynamic allocation is used, the function analyzes the dimensions of y differently:

- First dimension is fixed at size 3
- Second and third dimensions are unbounded

In this case, the code generation report represents the size of matrix y like this:



Variable	Type	Size
y	Output	3 x :? x :?

Defining Variable-Size Data Explicitly Using coder.varsize

Use the function `coder.varsize` to define one or more variables or structure fields as variable-size data. Optionally, you can also specify which dimensions vary along with their upper bounds (see "Specifying Which Dimensions Vary" on page 22-15). For example:

- Define `B` as a variable-size 2-by-2 matrix, where each dimension has an upper bound of 64:

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

- Define `B` as a variable-size matrix:

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

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

For more information, see the `coder.varsize` reference page.

Specifying Which Dimensions Vary

You can use the function `coder.varsize` to specify which dimensions vary. For example, the following statement defines `B` as a row vector whose first dimension is fixed at 2, but whose second dimension can grow to an upper bound of 16:

```
coder.varsize('B', [2, 16], [0 1])
```

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size; dimensions that correspond to ones or `true` vary in size. `coder.varsize` usually treats dimensions of size 1 as fixed (see "Defining Variable-Size Matrices with Singleton Dimensions" on page 22-16).

For more information about the syntax, see the `coder.varsize` reference page.

Allowing a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix `Y` with fixed 2-by-2 dimensions before first use (where the statement `Y = Y + u` reads from `Y`). However, `coder.varsize` defines `Y` as a variable-size matrix, allowing it to change size based on decision logic in the `else` clause:

```
function Y = var_by_if(u) %#codegen
if (u > 0)
   Y = zeros(2,2);
   coder.varsize('Y');
   if (u < 10)
       Y = Y + u;
   end
else
   Y = zeros(5,5);
end
```

Without `coder.varsize`, MATLAB infers `Y` to be a fixed-size, 2-by-2 matrix and generates a size mismatch error during code generation.

Defining Variable-Size Matrices with Singleton Dimensions

A singleton dimension is 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 using matrix constructor expressions or matrix functions.

  For example, in this function, both X and Y behave like vectors where only their second dimensions are variable sized:

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

You can override this behavior by using coder.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. For more information, see the coder.varsize reference page.

Defining Variable-Size Structure Fields

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

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

  d = struct('values', zeros(1,0), 'color', 0);
```

```
data = repmat(d, [3 3]);
coder.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 sized.

Here are other examples:

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

  In this example, `data` is a scalar variable that contains matrix `A`. Each element of matrix `A` contains a variable-size field `B`.

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

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

C Code Interface for Arrays

In this section...

"C Code Interface for Statically Allocated Arrays" on page 22-19

"C Code Interface for Dynamically Allocated Arrays" on page 22-20

"Utility Functions for Creating emxArray Data Structures" on page 22-21

C Code Interface for Statically Allocated Arrays

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

For example, consider the MATLAB function `uniquetol`:

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

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

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

In the generated code, the function declaration is:

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

There are two pieces of information about A:

- double A_data[100]: the maximum size of input A (where 100 is the maximum size specified using coder.typeof).
- int A_size[2]: the actual size of the input.

C Code Interface for Dynamically Allocated Arrays

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

emxArray Structure Definition

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

where baseTypedef is the predefined type in rtwtypes.h corresponding to baseType. For example, here's 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;
```

Note that 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".

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

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

C Code Interface for Structure Fields

Field	Description
*data	Pointer to data of type <*baseType*>
*size	Pointer to first element of size vector. Length of the vector equals the number of dimensions.
allocatedSize	Number of elements currently allocated for the array. If the size changes, MATLAB reallocates memory based on the new size.
numDimensions	Number of dimensions of the size vector, that is, the number of dimensions you can access without crossing into unallocated or unused memory
canFreeData	Boolean flag indicating how to deallocate memory: • true – MATLAB deallocates memory automatically • false – Calling program determines when to deallocate memory

Utility Functions for Creating emxArray Data Structures

When you generate code that uses variable-size data, the code generation software exports a set of utility functions that you can use to create and interact with emxArrays in your generated code. To call these functions in your main C function, include the generated header file. For example, when you generate code for function foo, include foo_emxAPI.h in your main C function. .

Function	Arguments	Description

`emxArray_<baseType> *emxCreateWrapper_<baseType> (...)`	`*data`
	`num_rows`
	`num_cols`
		Creates a new 2-dimensional `emxArray`, but does not allocate it on the heap. Instead uses memory provided by the user and sets `canFreeData` to `false` so it does not inadvertently free user memory, such as the stack.

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

`emxArray_<baseType> *emxCreate_<baseType> (...)`	`num_rows` `num_cols`	Creates a new two-dimensional `emxArray` on the heap, initialized to zero. All data elements have the data type specified by *baseTypeName*.
`emxArray_<baseType> *emxCreateND_<baseType> (...)`	`numDimensions` `*size`	Same as `emxCreate`, except it creates a new N-dimensional `emxArray` on the heap.
`emxArray_<baseType> *emxDestroyArray_<baseType> (...)`	`*emxArray`	Frees dynamic memory allocated by `*emxCreate` and `*emxCreateND` functions.

Diagnose and Fix Variable-Size Data Errors

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 22-23

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 22-25

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

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

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

Compiling this function produces this error:

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

There are several ways to fix this error:

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

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

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

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

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

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

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

Empty Matrix Reshaped to Match Variable-Size Specification

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

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

```
If u < 0
   Y = [Y u];
end
```

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

Performing Binary Operations on Fixed and Variable-Size Operands

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

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

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

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

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

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

There are several ways to fix the problem:

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

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

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

Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

"Incompatibility with MATLAB for Scalar Expansion" on page 22-27

"Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays" on page 22-29

"Incompatibility with MATLAB in Determining Size of Empty Arrays" on page 22-30

"Incompatibility with MATLAB in Determining Class of Empty Arrays" on page 22-31

"Incompatibility with MATLAB in Vector-Vector Indexing" on page 22-32

"Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" on page 22-33

"Incompatibility with MATLAB in Concatenating Variable-Size Matrices" on page 22-34

"Dynamic Memory Allocation Not Supported for MATLAB Function Blocks" on page 22-35

Incompatibility with MATLAB for Scalar Expansion

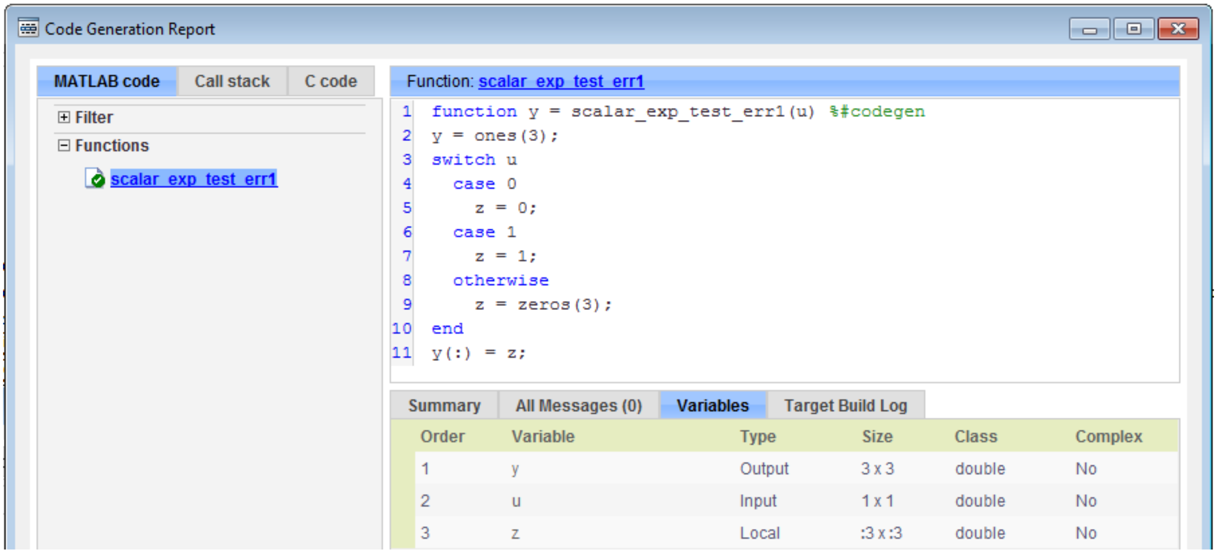
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

During code generation, the standard MATLAB scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Instead, it generates a size mismatch error at run time for MEX functions. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior.

For example, in the following function, `z` is scalar for the `switch` statement `case 0` and `case 1`. MATLAB applies scalar expansion when evaluating `y(:)  = z;` for these two cases.

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

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



Function: scalar_exp_test_err1

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

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

If you run the MEX function with u equal to zero or one, even though z is scalar at run time, the generated code does not perform scalar expansion and a run-time error occurs.

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

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

Workaround

Use indexing to force z to be a scalar value:

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

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

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

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

- Three-element vector in generated code

- Two-element vector in MATLAB code

Workarounds

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

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

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

- Rewrite `size(A)`:

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

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

Incompatibility with MATLAB in Determining Size of Empty Arrays

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

For example, consider the following code:

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

```
end
```

Concatenation requires its operands to match on the size of the dimension that is not being concatenated. In the preceding concatenation the scalar value has size 1x1 and x has size 0x0. To support this use case, the code generation software determines the size for x as [1 x :?]. Because there is another assignment x = [] after the concatenation, the size of x in the generated code is 1x0 instead of 0x0.

Workaround

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

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

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

Incompatibility with MATLAB in 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 generation software considers all execution paths. It determines that based on the statement x = ['a', x], the class of x is char.

Workaround

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

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

Incompatibility with MATLAB in Vector-Vector Indexing

In vector-vector indexing, you use one vector as an index into another vector. When either vector is variable sized, you might get a run-time error during code generation. Consider the index expression A(B). The general rule for indexing is that size(A(B)) == size(B). However, when both A and B are vectors, MATLAB applies a special rule: use the orientation of A as the orientation of the output. For example, if size(A) == [1 5] and size(B) == [3 1], then size(A(B)) == [1 3].

In this situation, if the code generation software detects that both A and B are vectors at compile time, it applies the special rule and gives the same result as MATLAB. However, if either `A` or `B` is a variable-size matrix (has shape `?x?`) at compile time, the code generation software applies only the general indexing rule. Then, if both `A` and `B` become vectors at run time, the code generation software reports a run-time error when you run the MEX function. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior. It is best practice to generate and test a MEX function before generating C code.

Workaround

Force your data to be a vector by using the colon operator for indexing: `A(B(:))`. For example, suppose your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing:

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
 else
    D = A(B);
 end
...
```

The indexing in the first branch specifies that `C` and `B(:)` are compile-time vectors. As a result, the code generation software applies the standard vector-vector indexing rule.

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

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

- Initialization of the following style:

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

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

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

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

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

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

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

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

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

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

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

Dynamic Memory Allocation Not Supported for MATLAB Function Blocks

You cannot use dynamic memory allocation for variable-size data in MATLAB Function blocks. Use bounded instead of unbounded variable-size data.

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation

In this section...
"Common Restrictions" on page 22-36
"Toolbox Functions with Variable Sizing Restrictions" on page 22-37

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in "Toolbox Functions with Variable Sizing Restrictions" on page 22-37.

Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape `1x:n` or `:nx1` (one dimension is variable sized and the other is fixed at size 1). Other shapes are not permitted, even if they are vectors at run time.

Automatic dimension restriction

When the function selects the working dimension automatically, it bases the selection on the upper bounds for the dimension sizes. In the case of the `sum` function, `sum(X)` selects its working dimension automatically, while `sum(X, dim)` uses `dim` as the explicit working dimension.

For example, if `X` is a variable-size matrix with dimensions `1x:3x:5`, `sum(x)` behaves like `sum(X,2)` in generated code. In MATLAB, it behaves like `sum(X,2)` provided `size(X,2)` is not 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`. Consequently, you get a run-time error if an automatically selected working dimension assumes a length of 1 at run time.

To avoid the issue, specify the intended working dimension explicitly as a constant value.

Array-to-vector restriction

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

Array-to-scalar restriction

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

Toolbox Functions with Variable Sizing Restrictions

The following restrictions apply to specific toolbox functions, but only for code generation.

Function	Restrictions with Variable-Size Data
`all`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`any`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`bsxfun`	• Dimensions expand only where one input array or the other has a fixed length of 1.
`cat`	• Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.

Function	Restrictions with Variable-Size Data
`conv`	• See "Variable-length vector restriction" on page 22-36. • Input vectors must have the same orientation, either both row vectors or both column vectors.
`cov`	• For `cov(X)`, see"Array-to-vector restriction" on page 22-37.
`cross`	• Variable-size array inputs that become vectors at run time must have the same orientation.
`deconv`	• For both arguments, see"Variable-length vector restriction" on page 22-36.
`detrend`	• For first argument for row vectors only, see "Array-to-vector restriction" on page 22-37 .
`diag`	• See "Array-to-vector restriction" on page 22-37 .
`diff`	• See "Automatic dimension restriction" on page 22-36. • Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, `diff(x,2,1)` works but `diff(x,5,1)` generates a run-time error.
`fft`	• See "Automatic dimension restriction" on page 22-36.

Function	Restrictions with Variable-Size Data
`filter`	• For first and second arguments, see "Variable-length vector restriction" on page 22-36. • See "Automatic dimension restriction" on page 22-36.
`hist`	• For second argument, see "Variable-length vector restriction" on page 22-36. • For second input argument, see"Array-to-scalar restriction" on page 22-37.
`histc`	• See "Automatic dimension restriction" on page 22-36.
`ifft`	• See "Automatic dimension restriction" on page 22-36.
`ind2sub`	• First input (the size vector input) must be fixed size.
`interp1`	• For the `Y` input and `xi` input, see"Array-to-vector restriction" on page 22-37. • `Y` input can become a column vector dynamically. • A run-time error occurs if `Y` input is not a variable-length vector and becomes a row vector at run time.
`ipermute`	• Order input must be fixed size.
`issorted`	• For optional rows input, see "Variable-length vector restriction" on page 22-36.

Function	Restrictions with Variable-Size Data
`magic`	• Argument must be a constant. • Output can be fixed-size matrices only.
`max`	• See "Automatic dimension restriction" on page 22-36.
`mean`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`median`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`min`	• See "Automatic dimension restriction" on page 22-36.
`mode`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
mtimes	• When an input is variable-size, MATLAB determines whether to generate code for a general matrix*matrix multiplication or a scalar*matrix multiplication, based on whether one of the arguments is a fixed-size scalar. If neither argument is a fixed-size scalar, the inner dimensions of the two arguments must agree even if a variable-size matrix input is a scalar at run time.
nchoosek	• The second input, k, must be a fixed-size scalar. • The second input, k, must be a constant for static allocation. If you enable dynamic allocation, the second input can be a variable. • You cannot create a variable-size array by passing in a variable, k, unless you enable dynamic allocation.
permute	• Order input must be fixed-size.
planerot	• Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.
poly	• See "Variable-length vector restriction" on page 22-36.
polyfit	• For first and second arguments, see "Variable-length vector restriction" on page 22-36.

Function	Restrictions with Variable-Size Data
`prod`	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`rand`	• For an upper-bounded variable `N`, `rand(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. • For an upper-bounded variable `N`, `rand([1,N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`randn`	• For an upper-bounded variable `N`, `randn(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. • For an upper-bounded variable `N`, `randn([1,N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`reshape`	• When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.
`roots`	• See "Variable-length vector restriction" on page 22-36.

Function	Restrictions with Variable-Size Data
shiftdim	• If you do not supply the second argument, the number of shifts is determined at compilation time by the upper bounds of the dimension sizes. Consequently, at run time the number of shifts is constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must have the same number of dimensions when you supply a positive number of shifts.
std	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
sub2ind	• First input (the size vector input) must be fixed size.
sum	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
trapz	• See "Automatic dimension restriction" on page 22-36. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
`typecast`	• See “Variable-length vector restriction” on page 22-36 on first argument.
`var`	• See “Automatic dimension restriction” on page 22-36. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

23

Primary Functions

Primary Function Input Specification

In this section...

“Why You Must Specify Input Properties” on page 23-2

“Properties to Specify” on page 23-2

“Rules for Specifying Properties of Primary Inputs” on page 23-4

“Methods for Defining Properties of Primary Inputs” on page 23-5

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	✓	✓	✓	✓	✓

For...	Specify properties...				
Each field in a structure input	**Specify properties for each field according to its class** When a primary input is a structure, the code generation software treats each field as a separate input. Therefore, you must specify properties for allfields of a primary structure input in the order that they appear in the structure definition: • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify `numerictype`, and `fimath`.				
Other inputs	✓	✓	✓		

Default Property Values

Fixed-Point Designer assigns the following default values for properties of primary function inputs.

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default
`fimath`	MATLAB default `fimath` object

Supported Classes

The following table presents the class names supported by Fixed-Point Designer.

Class Name	Description
`logical`	Logical array of true and false values
`char`	Character array
`int8`	8-bit signed integer array

Class Name	Description
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64–bit unsigned integer array
`single`	Single-precision floating-point or fixed-point number array
`double`	Double-precision floating-point or fixed-point number array
`struct`	Structure array
`embedded.fi`	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules.

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (`fi`), you must specify the input `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
Note If you define input properties programmatically in the MATLAB file, you cannot use this method	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs	• Must be specified at the command line every time you invoke `codegen` (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties Programmatically in the MATLAB File"	• 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	• 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 Programmatically in the MATLAB File

With MATLAB Coder, you use the MATLAB assert function to define properties of primary function inputs directly in your MATLAB file.

In this section...

"How to Use assert with MATLAB® Coder™" on page 23-6

"Rules for Using assert Function" on page 23-13

"Specifying General Properties of Primary Inputs" on page 23-14

"Specifying Properties of Primary Fixed-Point Inputs" on page 23-15

"Specifying Class and Size of Scalar Structure" on page 23-15

"Specifying Class and Size of Structure Array" on page 23-16

How to Use assert with MATLAB Coder

Use the assert function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

You must use one of the following methods when specifying input properties using the assert function. Use the exact syntax that is provided; do not modify it.

- "Specify Any Class" on page 23-7
- "Specify fi Class" on page 23-7
- "Specify Structure Class" on page 23-8
- "Specify Fixed Size" on page 23-8
- "Specify Scalar Size" on page 23-9
- "Specify Upper Bounds for Variable-Size Inputs" on page 23-9
- "Specify Inputs with Fixed- and Variable-Size Dimensions" on page 23-9
- "Specify Size of Individual Dimensions" on page 23-10
- "Specify Real Input" on page 23-11

- "Specify Complex Input" on page 23-11
- "Specify numerictype of Fixed-Point Input" on page 23-11
- "Specify fimath of Fixed-Point Input" on page 23-12
- "Specify Multiple Properties of Input" on page 23-12

Specify Any Class

```
assert ( isa ( param, 'class_name') )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input U to a 32-bit signed integer, call:

```
...
assert(isa(U,'int32'));
...
```

If you set the class of an input parameter to fi, you must also set its numerictype, see "Specify numerictype of Fixed-Point Input" on page 23-11. You can also set its fimath properties, see "Specify fimath of Fixed-Point Input" on page 23-12. If you do not set the fimath properties, codegen uses the MATLAB default fimath value.

If you set the class of an input parameter to struct, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class fi (fixed-point numeric object). For example, to set the class of input U to fi, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U,'embedded.fi'));
...
```

If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify numerictype of Fixed-Point Input” on page 23-11. You can also set its `fimath` properties, see “Specify fimath of Fixed-Point Input” on page 23-12. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Structure Class

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter *`param`* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U, 'struct'));
...
```

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Fixed Size

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *`param`* to the size specified by dimensions *`dims`*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. To set the size of input U to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Upper Bounds for Variable-Size Inputs

```
assert ( all(size(param)<=[N0 N1 ...]));
assert ( all(size(param)<[N0 N1 ...]));
```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input U to be less than or equal to a 3-by-2 matrix, call:

```
assert(all(size(U)<=[3 2]));
```

Note You can also specify upper bounds for variable-size inputs using coder.varsize.

Specify Inputs with Fixed- and Variable-Size Dimensions

```
assert ( all(size(param)>=[M0 M1 ...]));
assert ( all(size(param)<=[N0 N1 ...]));
```

When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, `k`, the lower-bound `Mk` must be less than or equal to the upper-bound `Nk`.
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be non-negative.

To fix the size of the first dimension of input `U` to 3 and set the second dimension as variable size with upper-bound of 2, call:

```
assert(all(size(U)>=[3 0]));
assert(all(size(U)<=[3 2]));
```

Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);
assert (size(param, k)<=Nk);
assert (size(param, k)<Nk);
```

You can specify individual dimensions as well as specifying all dimensions simultaneously or instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension `k` of input parameter *`param`*. To set the upper-bound size of the first dimension of input `U` to 3, call:

```
assert(size(U,1)<=3)
```

To fix the size of the second dimension of input `U` to 2, call:

```
assert(size(U,2)==2)
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *`param`* is real. To specify that input `U` is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *`param`* is complex. To specify that input `U` is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *`fiparam`* to the `numerictype` object *`T`*. For example, to specify the `numerictype` property of fixed-point input `U` as a signed `numerictype` object `T` with 32-bit word length and 30-bit fraction length, use the following code:

```
%#codegen
...
% Define the numerictype object.
```

```
T = 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:

```
%#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.
- Use the `assert` function with MATLAB Coder only for specifying properties of primary function inputs before converting your MATLAB code to C/C++ code.
- If you set the class of an input parameter to `fi`, you must also set its `numerictype`. See “Specify numerictype of Fixed-Point Input” on page 23-11. You can also set its `fimath` properties. See “Specify fimath of Fixed-Point Input” on page 23-12. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(`*`param`*`)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
  - You must also specify an upper-bound size for each dimension of the input parameter.
  - For each dimension, `k`, the lower-bound `Mk` must be less than or equal to the upper-bound `Nk`.
  - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
  - Bounds must be non-negative.
- If you specify individual dimensions, the following rules apply:
  - You must specify the size of each dimension at least once.

- The last dimension specification takes precedence over earlier specifications.

Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
`pennywhistle`	class	`int16`
	size	220500-by-1 vector
	complexity	`real` (by default)
`win`	class	`double`
	size	1024-by-1 vector
	complexity	`real` (by default)

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...
```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...
```

Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Designer software.

In the following example, the primary MATLAB function `mcsqrtfi` takes one fixed-point input `x`. The code specifies the following properties for this input.

Property	Value
class	`fi`
`numerictype`	`numerictype` object `T`, as specified in the primary function
`fimath`	`fimath` object `F`, as specified in the primary function
size	`scalar`
complexity	`real` (by default)

```
function y = mcsqrtfi(x) %#codegen
T = numerictype('WordLength',32,'FractionLength',23,...
                'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
           'SumWordLength',32,'SumFractionLength',23,...
           'ProductMode','SpecifyPrecision',...
           'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of `S` and its fields when passed as an input to your MATLAB function:

```
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 don't explicitly specify values for properties, MATLAB Coder uses defaults — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of a structure field

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 2-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

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

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [2 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of all fields.

System Objects Supported for Code Generation

System Objects Supported for C/C++ Code Generation

In this section...

"Code Generation for System Objects" on page 24-2
"Computer Vision System Toolbox System Objects" on page 24-2
"Communications System Toolbox System Objects" on page 24-7
"DSP System Toolbox System Objects" on page 24-13
"Phased Array System Toolbox System Objects" on page 24-19
"Image Acquisition Toolbox System Objects" on page 24-23

Code Generation for System Objects

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

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. For general information on MATLAB objects, see "Begin Using Object-Oriented Programming".

Computer Vision System Toolbox System Objects

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

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
`vision.BoundaryTracer`	Trace object boundaries in binary images.
`vision.ContrastAdjuster`	Adjust image contrast by linear scaling.
`vision.Deinterlacer`	Remove motion artifacts by deinterlacing input video signal.
`vision.EdgeDetector`	Find edges of objects in images.
`vision.ForegroundDetector`	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
`vision.HistogramBasedTracker`	Track object in video based on histogram. This object supports tunable properties in code generation.
`vision.HistogramEqualizer`	Enhance contrast of images using histogram equalization.
`vision.TemplateMatcher`	Perform template matching by shifting template over image.
Conversions	
`vision.Autothresholder`	Convert intensity image to binary image.
`vision.ChromaResampler`	Downsample or upsample chrominance components of images.
`vision.ColorSpaceConverter`	Convert color information between color spaces.
`vision.DemosaicInterpolator`	Demosaic Bayer's format images.
`vision.GammaCorrector`	Apply or remove gamma correction from images or video streams.
`vision.ImageComplementer`	Compute complement of pixel values in binary, intensity, or RGB images.
`vision.ImageDataTypeConverter`	Convert and scale input image to specified output data type.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
Feature Detection, Extraction, and Matching	
`vision.CornerDetector`	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
`vision.Convolver`	Compute 2-D discrete convolution of two input matrices.
`vision.ImageFilter`	Perform 2-D FIR filtering of input matrix.
`vision.MedianFilter`	2D median filtering.
Geometric Transformations	
`vision.GeometricRotator`	Rotate image by specified angle.
`vision.GeometricScaler`	Enlarge or shrink image size.
`vision.GeometricShearer`	Shift rows or columns of image by linearly varying offset.
`vision.GeometricTransformer`	Apply projective or affine transformation to an image.
`vision.GeometricTransformEstimator`	Estimate geometric transformation from matching point pairs.
`vision.GeometricTranslator`	Translate image in two-dimensional plane using displacement vector.
Morphological Operations	
`vision.ConnectedComponentLabeler`	Label and count the connected regions in a binary image.
`vision.MorphologicalClose`	Perform morphological closing on image.
`vision.MorphologicalDilate`	Perform morphological dilation on an image.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.MorphologicalErode`	Perform morphological erosion on an image.
`vision.MorphologicalOpen`	Perform morphological opening on an image.
Object Detection	
`vision.CascadeObjectDetector`	Detect objects using the Viola-Jones algorithm.
`vision.HistogramBasedTracker`	Track object in video based on histogram. This object supports tunable properties in code generation.
`vision.PointTracker`	Track points in video using Kanade-Lucas-Tomasi (KLT) algorithm.
`vision.PeopleDetector`	Detect upright people using HOG features.
Sinks	
`vision.VideoPlayer`	Send video data to computer screen. This System object does not generate code. It is automatically declared as an *extrinsic* variable using the `coder.extrinsic` function.
`vision.DeployableVideoPlayer`	Send video data to computer screen.
`vision.VideoFileWriter`	Write video frames and audio samples to multimedia file.
Sources	
`vision.VideoFileReader`	Read video frames and audio samples from compressed multimedia file.
Statistics	
`vision.Autocorrelator`	Compute 2-D autocorrelation of input matrix.
`vision.BlobAnalysis`	Compute statistics for connected regions in a binary image.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.Crosscorrelator`	Compute 2-D cross-correlation of two input matrices.
`vision.Histogram`	Generate histogram of each input matrix. This object has no tunable properties.
`vision.LocalMaximaFinder`	Find local maxima in matrices.
`vision.Maximum`	Find maximum values in input or sequence of inputs.
`vision.Mean`	Find mean value of input or sequence of inputs.
`vision.Median`	Find median values in an input.
`vision.Minimum`	Find minimum values in input or sequence of inputs.
`vision.PSNR`	Compute peak signal-to-noise ratio (PSNR) between images.
`vision.StandardDeviation`	Find standard deviation of input or sequence of inputs.
`vision.Variance`	Find variance values in an input or sequence of inputs.
Text & Graphics	
`vision.AlphaBlender`	Combine images, overlay images, or highlight selected pixels.
`vision.MarkerInserter`	Draw markers on output image.
`vision.ShapeInserter`	Draw rectangles, lines, polygons, or circles on images.
`vision.TextInserter`	Draw text on image or video stream.
Transforms	
`vision.DCT`	Compute 2-D discrete cosine transform.
`vision.FFT`	Two-dimensional discrete Fourier transform.

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.HoughLines`	Find Cartesian coordinates of lines that are described by rho and theta pairs.
`vision.HoughTransform`	Find lines in images via Hough transform.
`vision.IDCT`	Compute 2-D inverse discrete cosine transform.
`vision.IFFT`	Two–dimensional inverse discrete Fourier transform.
`vision.Pyramid`	Perform Gaussian pyramid decomposition.
Utilities	
`vision.ImagePadder`	Pad or crop input image along its rows, columns, or both.

Communications System Toolbox System Objects

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

Supported Communications System Toolbox System Objects

Object	Description
Source Coding	
`comm.DifferentialDecoder`	Decode binary signal using differential decoding
`comm.DifferentialEncoder`	Encode binary signal using differential coding
Channels	
`comm.AWGNChannel`	Add white Gaussian noise to input signal
`comm.LTEMIMOChannel`	Filter input signal through LTE MIMO multipath fading channel

Supported Communications System Toolbox System Objects (Continued)

Object	Description
`comm.MIMOChannel`	Filter input signal through MIMO multipath fading channel
`comm.BinarySymmetricChannel`	Introduce binary errors
Equalizers	
`comm.MLSEEqualizer`	Equalize using maximum likelihood sequence estimation
Filters	
`comm.IntegrateAndDumpFilter`	Integrate discrete-time signal with periodic resets
Measurements	
`comm.ACPR`	Measure adjacent channel power ratio
`comm.CCDF`	Measure complementary cumulative distribution function
`comm.EVM`	Measure error vector magnitude
`comm.MER`	Measure modulation error ratio
Sources	
`comm.BarkerCode`	Generate Barker code
`comm.HadamardCode`	Generate Hadamard code
`comm.KasamiSequence`	Generate a Kasami sequence
`comm.OVSFCode`	Generate OVSF code
`comm.PNSequence`	Generate a pseudo-noise (PN) sequence
`comm.WalshCode`	Generate Walsh code from orthogonal set of codes
Error Detection and Correction – Block Coding	
`comm.BCHDecoder`	Decode data using BCH decoder
`comm.BCHEncoder`	Encode data using BCH encoder
`comm.LDPCDecoder`	Decode binary low-density parity-check code
`comm.LDPCEncoder`	Encode binary low-density parity-check code

Supported Communications System Toolbox System Objects (Continued)

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

Supported Communications System Toolbox System Objects (Continued)

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

Supported Communications System Toolbox System Objects (Continued)

Object	Description
`comm.DBPSKDemodulator`	Demodulate using M-ary DPSK method
`comm.QPSKDemodulator`	Demodulate using quadrature PSK method
`comm.QPSKModulator`	Modulate using quadrature PSK method
`comm.PSKDemodulator`	Demodulate using M-ary PSK method
`comm.PSKModulator`	Modulate using M-ary PSK method
`comm.OQPSKDemodulator`	Demodulate offset quadrature PSK modulated data
`comm.OQPSKModulator`	Modulate using offset quadrature PSK method
Digital Baseband Modulation – Amplitude	
`comm.GeneralQAMDemodulator`	Demodulate using arbitrary QAM constellation. This object has no tunable properties in code generation.
`comm.GeneralQAMModulator`	Modulate using arbitrary QAM constellation
`comm.PAMDemodulator`	Demodulate using M-ary PAM method
`comm.PAMModulator`	Modulate using M-ary PAM method
`comm.RectangularQAMDemodulator`	Demodulate using rectangular QAM method
`comm.RectangularQAMModulator`	Modulate using rectangular QAM method
Digital Baseband Modulation – Frequency	
`comm.FSKDemodulator`	Demodulate using M-ary FSK method
`comm.FSKModulator`	Modulate using M-ary FSK method
Digital Baseband Modulation – Trelllis Coded	
`comm.GeneralQAMTCMDemodulator`	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation
`comm.GeneralQAMTCMModulator`	Convolutionally encode binary data and map using arbitrary QAM constellation
`comm.PSKTCMDemodulator`	Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Supported Communications System Toolbox System Objects (Continued)

Object	Description
`comm.PSKTCMModulator`	Convolutionally encode binary data and map using M-ary PSK constellation
`comm.RectangularQAMTCMDemodulator`	Demodulate convolutionally encoded data mapped to rectangular QAM constellation
`comm.RectangularQAMTCMModulator`	Convolutionally encode binary data and map using rectangular QAM constellation
Digital Baseband Modulation – Continuous Phase	
`comm.CPFSKDemodulator`	Demodulate using CPFSK method and Viterbi algorithm
`comm.CPFSKModulator`	Modulate using CPFSK method
`comm.CPMDemodulator`	Demodulate using CPM method and Viterbi algorithm
`comm.CPMModulator`	Modulate using CPM method
`comm.GMSKDemodulator`	Demodulate using GMSK method and the Viterbi algorithm
`comm.GMSKModulator`	Modulate using GMSK method
`comm.MSKDemodulator`	Demodulate using MSK method and the Viterbi algorithm
`comm.MSKModulator`	Modulate using MSK method
RF Impairments	
`comm.MemorylessNonlinearity`	Apply memoryless nonlinearity to input signal
`comm.PhaseFrequencyOffset`	Apply phase and frequency offsets to input signal. The `PhaseOffset` property of this object is not tunable in code generation.
`comm.PhaseNoise`	Apply phase noise to complex baseband signal
`comm.ThermalNoise`	Add receiver thermal noise
Synchronization – Timing Phase	

Supported Communications System Toolbox System Objects (Continued)

Object	Description
`comm.EarlyLateGateTimingSynchronizer`	Recover symbol timing phase using early-late gate method
`comm.GardnerTimingSynchronizer`	Recover symbol timing phase using Gardner's method
`comm.GMSKTimingSynchronizer`	Recover symbol timing phase using fourth-order nonlinearity method
`comm.MSKTimingSynchronizer`	Recover symbol timing phase using fourth-order nonlinearity method
`comm.MuellerMullerTimingSynchronizer`	Recover symbol timing phase using Mueller-Muller method
Synchronization Utilities	
`comm.CPMCarrierPhaseSynchronizer`	Recover carrier phase of baseband CPM signal
`comm.DiscreteTimeVCO`	Generate variable frequency sinusoid
Converters	
`comm.BitToInteger`	Convert vector of bits to vector of integers
`comm.IntegerToBit`	Convert vector of integers to vector of bits
Sequence Operators	
`comm.Descrambler`	Descramble input signal
`comm.GoldSequence`	Generate Gold sequence
`comm.Scrambler`	Scramble input signal

DSP System Toolbox System Objects

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

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
`dsp.BurgAREstimator`	Compute estimate of autoregressive model parameters using Burg method
`dsp.BurgSpectrumEstimator`	Compute parametric spectral estimate using Burg method
`dsp.CepstralToLPC`	Convert cepstral coefficients to linear prediction coefficients
`dsp.LevinsonSolver`	Solve linear system of equations using Levinson-Durbin recursion
`dsp.LPCToAutocorrelation`	Convert linear prediction coefficients to autocorrelation coefficients
`dsp.LPCToCepstral`	Convert linear prediction coefficients to cepstral coefficients
`dsp.LPCToLSF`	Convert linear prediction coefficients to line spectral frequencies
`dsp.LPCToLSP`	Convert linear prediction coefficients to line spectral pairs
`dsp.LPCToRC`	Convert linear prediction coefficients to reflection coefficients
`dsp.LSFToLPC`	Convert line spectral frequencies to linear prediction coefficients
`dsp.LSPToLPC`	Convert line spectral pairs to linear prediction coefficients
`dsp.RCToAutocorrelation`	Convert reflection coefficients to autocorrelation coefficients
`dsp.RCToLPC`	Convert reflection coefficients to linear prediction coefficients
Filters	
`dsp.AffineProjectionFilter`	Adaptive filter using the Affine Projection algorithm

Supported DSP System Toolbox System Objects (Continued)

Object	Description
`dsp.AllpoleFilter`	IIR Filter with no zeros. Only the `Denominator` property is tunable for code generation.
`dsp.BiquadFilter`	Model biquadratic IIR (SOS) filters
`dsp.CICDecimator`	Decimate input using Cascaded Integrator-Comb filter
`dsp.CICInterpolator`	Interpolate signal using Cascaded Integrator-Comb filter
`dsp.DigitalFilter`	Filter each channel of input over time using discrete-time filter implementations. The `SOSMatrix` and `ScaleValues` properties at not supported for code generation.
`dsp.FIRDecimator`	Filter and downsample input signals
`dsp.FIRFilter`	Static or time-varying FIR filter. Only the `Numerator` property is tunable for code generation.
`dsp.FIRInterpolator`	Upsample and filter input signals
`dsp.FIRRateConverter`	Upsample, filter and downsample input signals
`dsp.IIRFilter`	Infinite Impulse Response (IIR) filter. Only the `Numerator` and `Denominator` properties are tunable for code generation.
`dsp.LMSFilter`	Compute output, error, and weights using LMS adaptive algorithm
`dsp.RLSFilter`	Adaptive filter using the Recursive Least Squares (RLS) algorithm
Math Operations	
`dsp.ArrayVectorAdder`	Add vector to array along specified dimension
`dsp.ArrayVectorDivider`	Divide array by vector along specified dimension
`dsp.ArrayVectorMultiplier`	Multiply array by vector along specified dimension

Supported DSP System Toolbox System Objects (Continued)

Object	Description
`dsp.ArrayVectorSubtractor`	Subtract vector from array along specified dimension
`dsp.CumulativeProduct`	Compute cumulative product of channel, column, or row elements
`dsp.CumulativeSum`	Compute cumulative sum of channel, column, or row elements
`dsp.LDLFactor`	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
`dsp.LevinsonSolver`	Solve linear system of equations using Levinson-Durbin recursion
`dsp.LowerTriangularSolver`	Solve LX = B for X when L is lower triangular matrix
`dsp.LUFactor`	Factor square matrix into lower and upper triangular matrices
`dsp.Normalizer`	Normalize input
`dsp.UpperTriangularSolver`	Solve UX = B for X when U is upper triangular matrix
Quantizers	
`dsp.ScalarQuantizerDecoder`	Convert each index value into quantized output value
`dsp.ScalarQuantizerEncoder`	Perform scalar quantization encoding
`dsp.VectorQuantizerDecoder`	Find vector quantizer codeword for given index value
`dsp.VectorQuantizerEncoder`	Perform vector quantization encoding
Scopes	
`dsp.SpectrumAnalyzer`	Display frequency spectrum of time-domain signals. This System object does not generate code. It is automatically declared as an *extrinsic* variable using the `coder.extrinsic` function.
`dsp.TimeScope`	Display time-domain signals. This System object does not generate code. It is automatically declared as an *extrinsic* variable using the `coder.extrinsic` function.
Signal Management	

Supported DSP System Toolbox System Objects (Continued)

Object	Description
`dsp.Counter`	Count up or down through specified range of numbers
`dsp.DelayLine`	Rebuffer sequence of inputs with one-sample shift
Signal Operations	
`dsp.Convolver`	Compute convolution of two inputs
`dsp.Delay`	Delay input by specified number of samples or frames
`dsp.Interpolator`	Interpolate values of real input samples
`dsp.NCO`	Generate real or complex sinusoidal signals
`dsp.PeakFinder`	Determine extrema (maxima or minima) in input signal
`dsp.PhaseUnwrapper`	Unwrap signal phase
`dsp.VariableFractionalDelay`	Delay input by time-varying fractional number of sample periods
`dsp.VariableIntegerDelay`	Delay input by time-varying integer number of sample periods
`dsp.Window`	Generate or apply window function. This object has no tunable properties for code generation.
`dsp.ZeroCrossingDetector`	Calculate number of zero crossings of a signal
Sinks	
`dsp.AudioPlayer`	Write audio data to computer's audio device
`dsp.AudioFileWriter`	Write audio file
`dsp.UDPSender`	Send UDP packets to the network
Sources	
`dsp.AudioFileReader`	Read audio samples from an audio file
`dsp.AudioRecorder`	Read audio data from computer's audio device
`dsp.SignalSource`	Import variable from workspace

Supported DSP System Toolbox System Objects (Continued)

Object	Description
`dsp.SineWave`	Generate discrete sine wave. This object has no tunable properties for code generation.
`dsp.UDPReceiver`	Receive UDP packets from the network
Statistics	
`dsp.Autocorrelator`	Compute autocorrelation of vector inputs
`dsp.Crosscorrelator`	Compute cross-correlation of two inputs
`dsp.Histogram`	Output histogram of an input or sequence of inputs. This object has no tunable properties for code generation.
`dsp.Maximum`	Compute maximum value in input
`dsp.Mean`	Compute average or mean value in input
`dsp.Median`	Compute median value in input
`dsp.Minimum`	Compute minimum value in input
`dsp.RMS`	Compute root-mean-square of vector elements
`dsp.StandardDeviation`	Compute standard deviation of vector elements
`dsp.Variance`	Compute variance of input or sequence of inputs
Transforms	
`dsp.AnalyticSignal`	Compute analytic signals of discrete-time inputs
`dsp.DCT`	Compute discrete cosine transform (DCT) of input
`dsp.FFT`	Compute fast Fourier transform (FFT) of input
`dsp.IDCT`	Compute inverse discrete cosine transform (IDCT) of input
`dsp.IFFT`	Compute inverse fast Fourier transform (IFFT) of input

Phased Array System Toolbox System Objects

If you install Phased Array System Toolbox software, you can generate C/C++ code for the following Phased Array System Toolbox System objects. For more information on how to use these System objects, see "About Code Generation".

Supported Phased Array System Toolbox System Objects

Object	Description
Antenna and Microphone Elements	
`phased.CosineAntennaElement`	Cosine antenna element
`phased.CrossedDipoleAntennaElement`	Crossed-dipole antenna element
`phased.CustomAntennaElement`	Custom antenna element
`phased.CustomMicrophoneElement`	Custom microphone
`phased.IsotropicAntennaElement`	Isotropic antenna element
`phased.OmnidirectionalMicrophoneElement`	Omnidirectional microphone
`phased.ShortDipoleAntennaElement`	Short-dipole antenna element
Array Geometries and Analysis	
`phased.ULA`	Uniform linear array
`phased.URA`	Uniform rectangular array
`phased.ConformalArray`	Conformal array
`phased.PartitionedArray`	Phased array partitioned into subarrays
`phased.ReplicatedSubarray`	Phased array formed by replicated subarrays
`phased.SteeringVector`	Sensor array steering vector
`phased.ArrayGain`	Sensor array gain
`phased.ArrayResponse`	Sensor array response
`phased.ElementDelay`	Sensor array element delay estimator
Signal Radiation and Collection	
`phased.Collector`	Narrowband signal collector
`phased.Radiator`	Narrowband signal radiator

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
`phased.WidebandCollector`	Wideband signal collector **Note** Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation".
Waveforms, Transmitter, and Receiver	
`phased.LinearFMWaveform`	Linear FM pulse waveform
`phased.PhaseCodedWaveform`	Phase-coded pulse waveform
`phased.RectangularWaveform`	Rectangular pulse waveform
`phased.SteppedFMWaveform`	Stepped FM pulse waveform
`phased.FMCWWaveform`	FMCW Waveform
`phased.MatchedFilter`	Matched filter
`phased.Transmitter`	Transmitter
`phased.ReceiverPreamp`	Receiver preamp
Beamforming	
`phased.PhaseShiftBeamformer`	Narrowband phase shift beamformer
`phased.LCMVBeamformer`	Narrowband LCMV beamformer
`phased.MVDRBeamformer`	Narrowband MVDR (Capon) beamformer
`phased.SubbandPhaseShiftBeamformer`	Subband phase shift beamformer
`phased.FrostBeamformer`	Frost beamformer **Note** Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation".

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
`phased.TimeDelayBeamformer`	Time delay beamformer **Note** Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation".
`phased.TimeDelayLCMVBeamformer`	Time delay LCMV beamformer **Note** Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation".
`phased.SteeringVector`	Sensor array steering vector
Direction of Arrival (DOA) Estimation	
`phased.SumDifferenceMonopulseTracker`	Sum and difference monopulse for ULA
`phased.SumDifferenceMonopulseTracker2D`	Sum and difference monopulse for URA
`phased.BeamscanEstimator`	Beamscan spatial spectrum estimator for ULA
`phased.BeamscanEstimator2D`	2-D beamscan spatial spectrum estimator
`phased.MVDREstimator`	MVDR (Capon) spatial spectrum estimator for ULA
`phased.MVDREstimator2D`	2-D MVDR (Capon) spatial spectrum estimator
`phased.RootMUSICEstimator`	Root MUSIC direction of arrival (DOA) estimator
`phased.RootWSFEstimator`	Root WSF direction of arrival (DOA) estimator
`phased.ESPRITEstimator`	ESPRIT direction of arrival (DOA) estimator
`phased.BeamspaceESPRITEstimator`	Beamspace ESPRIT direction of arrival (DOA) estimator

Supported Phased Array System Toolbox System Objects (Continued)

Object	Description
Space-Time Adaptive Processing (STAP)	
`phased.STAPSMIBeamformer`	Sample matrix inversion (SMI) beamformer
`phased.DPCACanceller`	Displaced phase center array (DPCA) pulse canceller
`phased.ADPCACanceller`	Adaptive DPCA (ADPCA) pulse canceller
`phased.AngleDopplerResponse`	Angle-Doppler response
Detection	
`phased.CFARDetector`	Constant false alarm rate (CFAR) detector
`phased.MatchedFilter`	Matched filter
`phased.RangeDopplerResponse`	Range-Doppler response
`phased.StretchProcessor`	Stretch processor for linear FM waveform
`phased.TimeVaryingGain`	Time varying gain control
Environment and Target Models	
`phased.FreeSpace`	Free space environment **Note** Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.
`phased.RadarTarget`	Radar target
`phased.ConstantGammaClutter`	Constant gamma clutter simulation
`phased.BarrageJammer`	Barrage jammer
Coordinate Systems and Motion Modeling	
`phased.Platform`	Motion platform

Image Acquisition Toolbox System Objects

If you install Image Acquisition Toolbox™ software, you can generate C/C++ code for the VideoDevice System object. See `imaq.VideoDevice` and “Code Generation with VideoDevice System Object”.

System Objects

- "Create System Objects" on page 25-2
- "Set Up System Objects" on page 25-4
- "Process Data Using System Objects" on page 25-7
- "Tuning System object Properties in MATLAB" on page 25-10
- "Find Help and Examples for System Objects" on page 25-12
- "Use System Objects in MATLAB Code Generation" on page 25-13

Create System Objects

In this section...

"Create a System object" on page 25-2

"Change a System object Property" on page 25-3

"Check if a System object Property Has Changed" on page 25-3

"Run a System object" on page 25-3

"Display Available System Objects" on page 25-3

A System object™ is a MATLAB object-oriented implementation of an algorithm. System objects extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets.

Note System objects predefined in the software do not support sparse matrices. System objects you define support sparse matrices (see).

Create a System object

To use System objects, you must first create an object. For example,

```
Hram = hdlram        % Create default hdlram object, H
```

Change a System object Property

In general, you should set the object properties before you use the `step` method to run data through the object. To change the value of a property, use this format,

```
H.RAMType = 'Dual Port'   % Set the RAMType property
```

The property values of the `hdlram` object, `H`, are displayed.

Check if a System object Property Has Changed

To check if a tunable property has changed since `step` was last called, use this syntax:

```
flag = isChangedProperty(H,'Normalize')
```

`flag` is `true` if the `Normalize` property of object `H` has changed.

Run a System object

To execute a System object, use the `step` method.

```
Y = step(H,X);       % Process input data, X
```

The output data from the `step` method is stored in `Y`, which, in this case, is port input and output data.

Display Available System Objects

To see a list of all the System objects for a particular package, type . To display help for specific objects, properties, or methods, see "Find Help and Examples for System Objects" on page 25-12 .

Set Up System Objects

In this section...

"Create a New System object" on page 25-4

"Retrieve System object Property Values" on page 25-4

"Set System object Property Values" on page 25-4

Create a New System object

You must create a System object before using it. You can create the object at the MATLAB command line or within a program file. Your command-line code and programs can pass MATLAB variables into and out of System objects.

For general information about working with MATLAB objects, see "Object-Oriented Programming" in the MATLAB documentation.

Retrieve System object Property Values

System objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *Name-Value pair*. You can display the list of relevant property names and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handle>.<Name>`.

Example

This example retrieves and displays the `RAMType` property value for the previously created `hdlram` object:

```
H.RAMType
```

Set System object Property Values

You set the property values of a System object to model the desired algorithm.

Note When you use Name-Value pair syntax, the object sets property values in the order you list them. If you specify a dependent property value before its parent property, an error or warning may occur.

Set Properties for a New System object

To set a property when you first create the object, use Name-Value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
H1 = hdlram('RAMType','Single port')
```

where

- `H1` is the handle to the object
-
- `hdlram` is the object name.
- `CoefficientsSource` is the property name.

  `RAMType` is the property name.
- `` `Single port' `` is the property value.

Set Properties for an Existing System object

To set a property after you have created an object, use either of the following syntaxes:

```
H1.RAMType = 'Dual port'
```

or

```
set(H1,'RAMType','Dual port')
```

Use Value-Only Inputs

Some object properties have no useful default values or must be specified every time you create an object. For these properties, you can specify only the value without specifying the corresponding property name. If you use value-only inputs, those inputs must be in a specific order, which is the same as the order in which the properties are displayed. Refer to the object reference page for details.

Process Data Using System Objects

In this section...

"What are System object Methods?" on page 25-7

"The Step Method" on page 25-7

"Common Methods" on page 25-7

"Advantages of Using Methods" on page 25-9

What are System object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the step method and other available methods, see the descriptions in "Common Methods" on page 25-7.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the `release` method instead of a destructor. See "Understand System object Modes" on page 25-10.
`clone`	Creates another object with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked. See "Understand System object Modes" on page 25-10.
`reset`	Resets the internal states of the object to the initial values for that object
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`isChangedProperty`	Returns `true` if the specified tunable property value has changed since the last call to `step`.Example: `flag = isChangedProperty(obj,'propertyName')`
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

Method	Description
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, `getDiscreteState` returns an empty structure.

Advantages of Using Methods

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

Tuning System object Properties in MATLAB

In this section...

“Understand System object Modes” on page 25-10

“Change Properties While Running System Objects” on page 25-11

“Change System object Input Complexity or Dimensions” on page 25-11

Understand System object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

The object initializes and locks when it begins processing data. The typical way in which an object becomes locked is when the step method is called on that object. To determine if an object is locked, use the `isLocked` method. To unlock an object, use the `release` method. When the object is locked, you cannot change any of the following:

- Number of inputs or outputs
- Data type
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data. Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.
- Value of any nontunable property

Several System objects do not allow changing the complexity of inputs from real to complex. You can, however, change the input complexity from complex to real without unlocking the object.

These restrictions allow the object to maintain states and allocate memory appropriately.

Change Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding reference page or use a command of this form:

```
help hdlram.RAMType
```

where

- `hdlram` is the object name.
- `RAMType` is the property name.

Note Unless otherwise specified, System object properties are not tunable.

For information on locked and unlocked modes, see "Understand System object Modes" on page 25-10.

Change System object Input Complexity or Dimensions

During simulations you can change an input's complexity from complex to real, but not from real to complex. You cannot change any input complexity during code generation.

For objects that do not support variable-size input, if you change the input dimensions while the object is in locked mode, the object produces a warning and unlocks. The object then reinitializes the next time you call the step method. See the object's reference page for more information. You can change the value of a tunable property and the input size without a warning or error being produced. For all other changes at runtime, an error occurs.

Find Help and Examples for System Objects

Refer to the following resources for more information about System objects.

- Object help – `help hdlram`
- Documentation pages for object – `doc hdlram`
- Property help – `help hdlram.RAMType`
- Fixed-point property help – `hdlram.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.
- Method help – `help hdlram.step`, where `step` is the method name.

To view examples, go to the Help contents for the associated product. Under `Examples`, select `MATLAB Examples`.

Use System Objects in MATLAB Code Generation

In this section...

"Considerations for Using System Objects in Generated Code" on page 25-13

"Use System Objects with codegen" on page 25-17

"Use System Objects with the MATLAB Function Block" on page 25-17

"Use System Objects with MATLAB® Compiler™" on page 25-17

Considerations for Using System Objects in Generated Code

You can generate C/C++ code from System objects using MATLAB Coder product. Using this product with System objects, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. System objects also support code generation using the MATLAB Function block in Simulink and the MATLAB Coder `codegen` function.

For general information on generating code, see

- MATLAB Coder product
- Simulink Coder product
- Embedded Coder® product

The following example, which uses System objects, shows the key factors to consider, such as passing property values and using extrinsic functions, when you make MATLAB code suitable for code generation.

```
function lmssystemidentification
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.

    persistent hlms hfilt;
    % Initialize persistent System objects only once
    % Do this with 'if isempty(persistent variable).'
    % This condition will be false after the first time.

    if isempty(hlms)

        % Create LMS adaptive filter used for system
        % identification. Pass property value arguments
        % as constructor arguments. Property values must
        % be constants during compile time.

        hlms = dsp.LMSFilter(11, 'StepSize', 0.01);

        % Create system (an FIR filter) to be identified.

        hfilt = dsp.DigitalFilter(...
                    'TransferFunction', 'FIR (all zeros)', ...
                    'Numerator', fir1(10, .25));
    end

    x = randn(1000,1);                          % Input signal
    d = step(hfilt, x) + 0.01*randn(1000,1);    % Desired signal
    [~,~,w] = step(hlms, x, d);                 % Filter weights

    % Declare functions called into MATLAB that do not generate
    % code as extrinsic.

    coder.extrinsic('stem');

    stem([get(hfilt, 'Numerator').', w]);
end

% To compile this function use codegen lmssystemidentification.
% This produces a mex file with the same name in the current
% directory.
```

For a detailed code generation example, see “Generate Code for MATLAB Handle Classes and System Objects” in the MATLAB Coder product documentation.

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

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

- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note Variable-size properties in MATLAB Function block in Simulink are not supported. 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 do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
  - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
  - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation.
- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- Do not use cell arrays.
- Global variables are not supported. 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 Supported for Code Generation

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`
  - `getNumOutputs`

- `isDone` (for sources only)
- `release`
- `reset`
- `set` (for tunable properties)
- `step`

Use System Objects with codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See "Getting Started with MATLAB Coder" and "MATLAB Classes" for more information.

Use System Objects with the MATLAB Function Block

Using the MATLAB Function block, you can include a MATLAB language function in a Simulink model. This model can then generate embeddable code. You can include any System object in the MATLAB Function block. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see "What Is a MATLAB Function Block?" in the Simulink documentation.

Use System Objects with MATLAB Compiler

Note MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

Fixed-Point Designer for Simulink Models

Getting Started

- “Product Description” on page 26-2
- “What You Need to Get Started” on page 26-3
- “Physical Quantities and Measurement Scales” on page 26-5
- “Why Use Fixed-Point Hardware?” on page 26-14
- “Why Use the Fixed-Point Designer Software?” on page 26-16
- “Developing and Testing Fixed-Point Systems in Simulink” on page 26-17
- “Supported Data Types” on page 26-20
- “Configure Blocks with Fixed-Point Output” on page 26-21
- “Configure Blocks with Fixed-Point Parameters” on page 26-31
- “Pass Fixed-Point Data Between Simulink Models and the MATLAB Software” on page 26-35
- “Cast from Doubles to Fixed Point” on page 26-40

Product Description

Design and simulate fixed-point systems

Fixed-Point Designer enables the fixed-point capabilities of the Simulink product family, letting you use those products to design, simulate, and implement fixed-point control and signal processing algorithms.

With Fixed-Point Designer, you specify fixed-point data attributes, including word length and scaling for signals and parameters, in your model. You can perform bit-true simulations to observe the effects of limited range and precision on designs built with Simulink, Stateflow, DSP System Toolbox, and other Simulink products. Automated fixed-point advisors guide you through the steps of converting floating-point models to fixed point. Additional tools analyze your model or use simulation results to recommend data types and scaling.

Fixed-Point Designer supports C, HDL, and PLC code generation with Simulink code-generation products.

Key Features

- Fixed-point modeling and simulation in Simulink, Stateflow, and other Simulink products
- Bit-true, fixed-point arithmetic for code generated by Simulink C, HDL, and PLC code generation products
- Automated advisors that convert models from floating- to fixed-point data types
- Analysis tools for deriving ranges for all signals based on design information
- Data type tools that use range data to recommend word length and scaling
- Control of fixed-point data type and of scaling from 1- to 128-bit word sizes
- Customizable fixed-point operators and math functions for embedded code generation

What You Need to Get Started

In this section...

"Installation" on page 26-3

"Sharing Fixed-Point Models" on page 26-3

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 In the **Model Hierarchy** pane, select the root model.

2 From the Simulink model **Analysis** menu, select **Fixed-Point Tool**.

   In the Fixed-Point Tool:

   - 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`.

3 If you use `fi` objects or embedded numeric data types in your model, set the `fipref DataTypeOverride` property to `TrueDoubles` and the `DataTypeOverride` 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 26-5

"Selecting a Measurement Scale" on page 26-6

"Select a Measurement Scale for Temperature" on page 26-8

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:

$$V = \begin{cases} \dfrac{100^{\mathrm{o}}\,\mathrm{C}}{100} Q_1 & \text{Conversion 1} \\ \dfrac{100^{\mathrm{o}}\,\mathrm{C}}{255} Q_2 & \text{Conversion 2} \\ \dfrac{100^{\mathrm{o}}\,\mathrm{C}}{200} Q_3 & \text{Conversion 3} \end{cases}$$

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

$$V = SQ + B \pm Error.$$

If the quantized value Q is rounded to the nearest representable number, then

$$-\frac{S}{2} \le Error \le \frac{S}{2}.$$

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

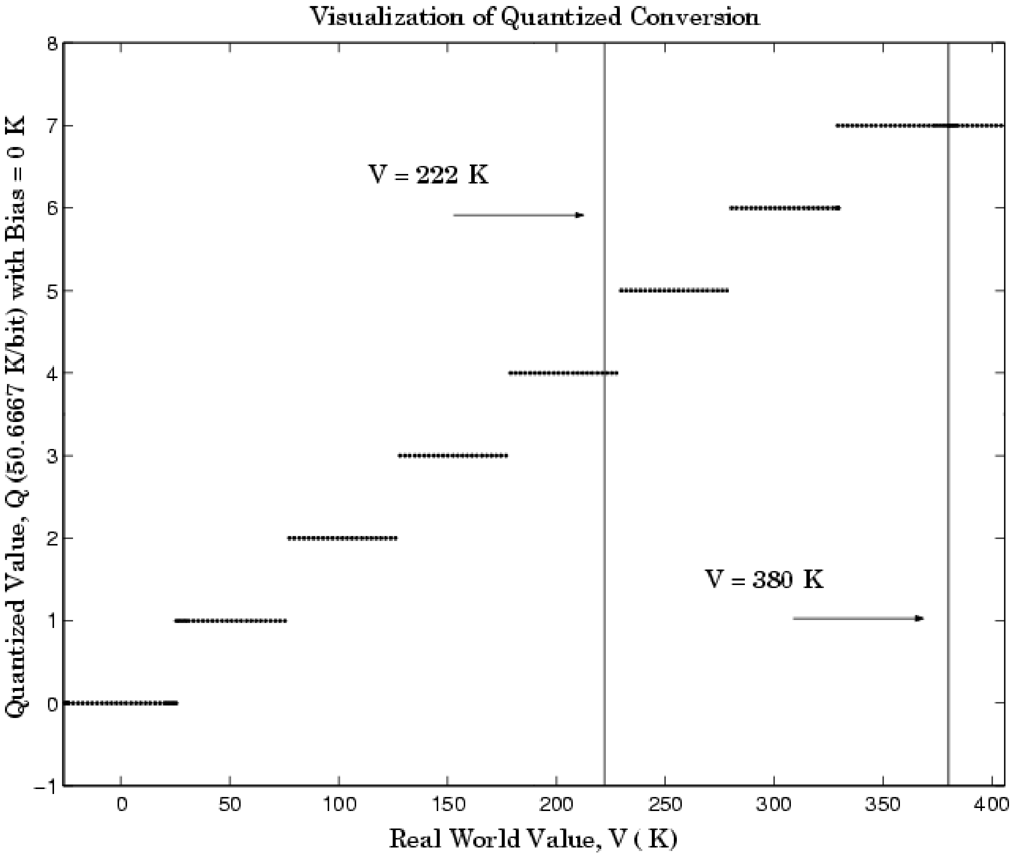
On typical electronically controlled internal combustion engines, the flow of fuel is regulated to obtain the desired ratio of air to fuel in the cylinders just prior to combustion. Therefore, knowledge of the current air flow rate is required. Some manufacturers use sensors that directly measure air 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

$$V = \frac{380\text{ K}}{7.5\text{ bit}}Q.$$

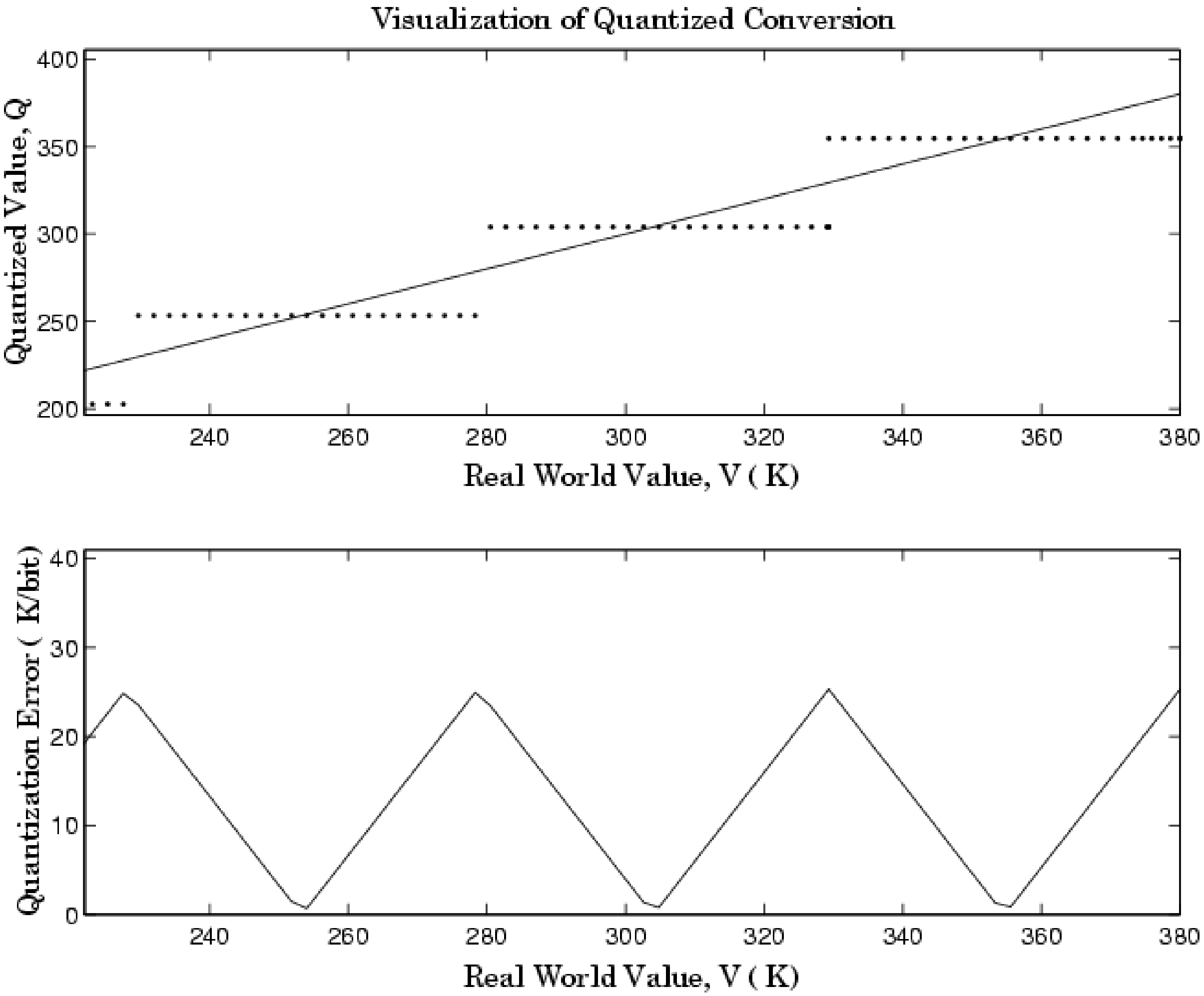
The quantized conversion and range of interest are shown in the following figure.



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 K/bit ≤ *Error* ≤ 25.33 K/bit.

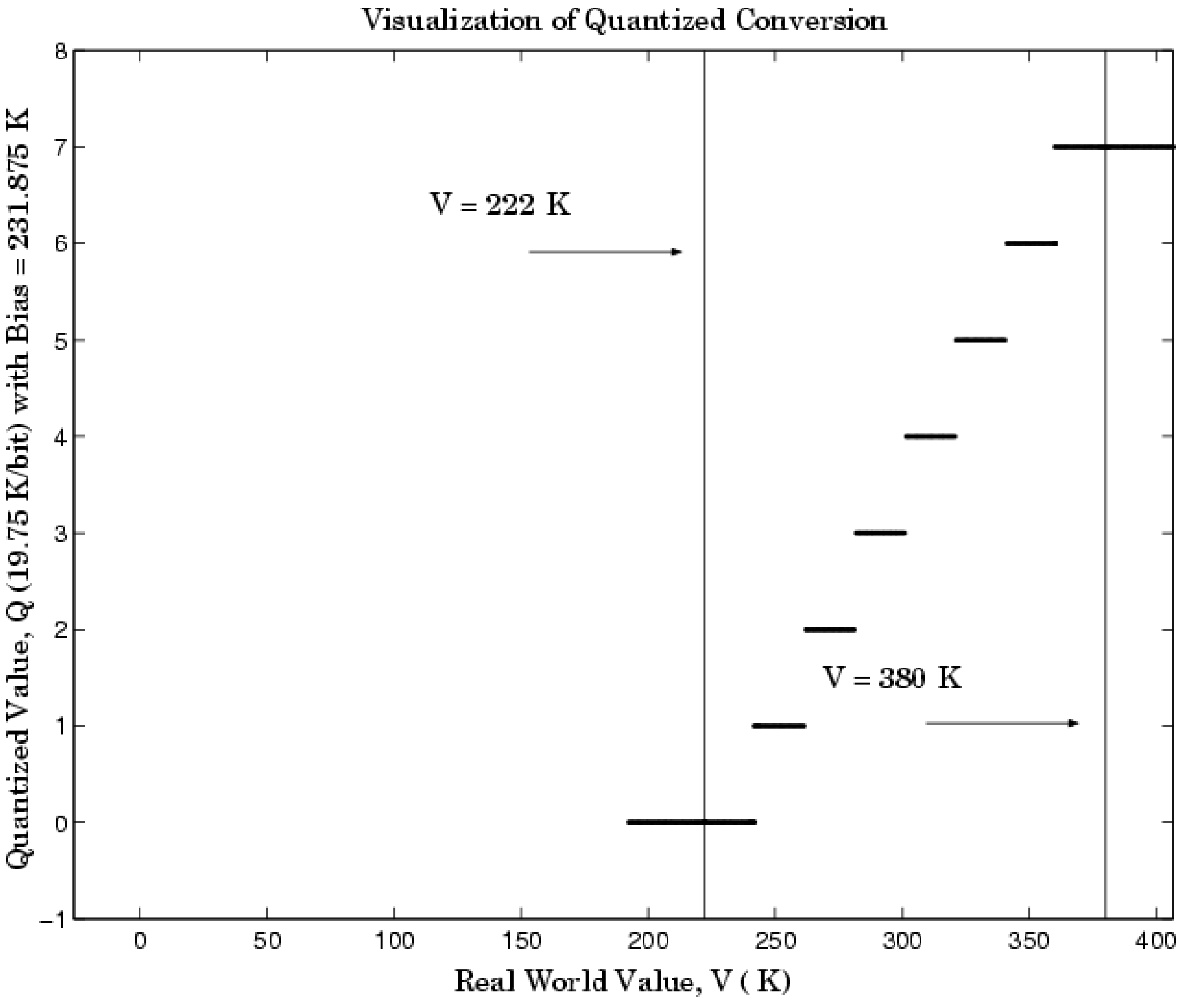
The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



As an alternative to the purely linear conversion, consider the general linear conversion with maximum precision:

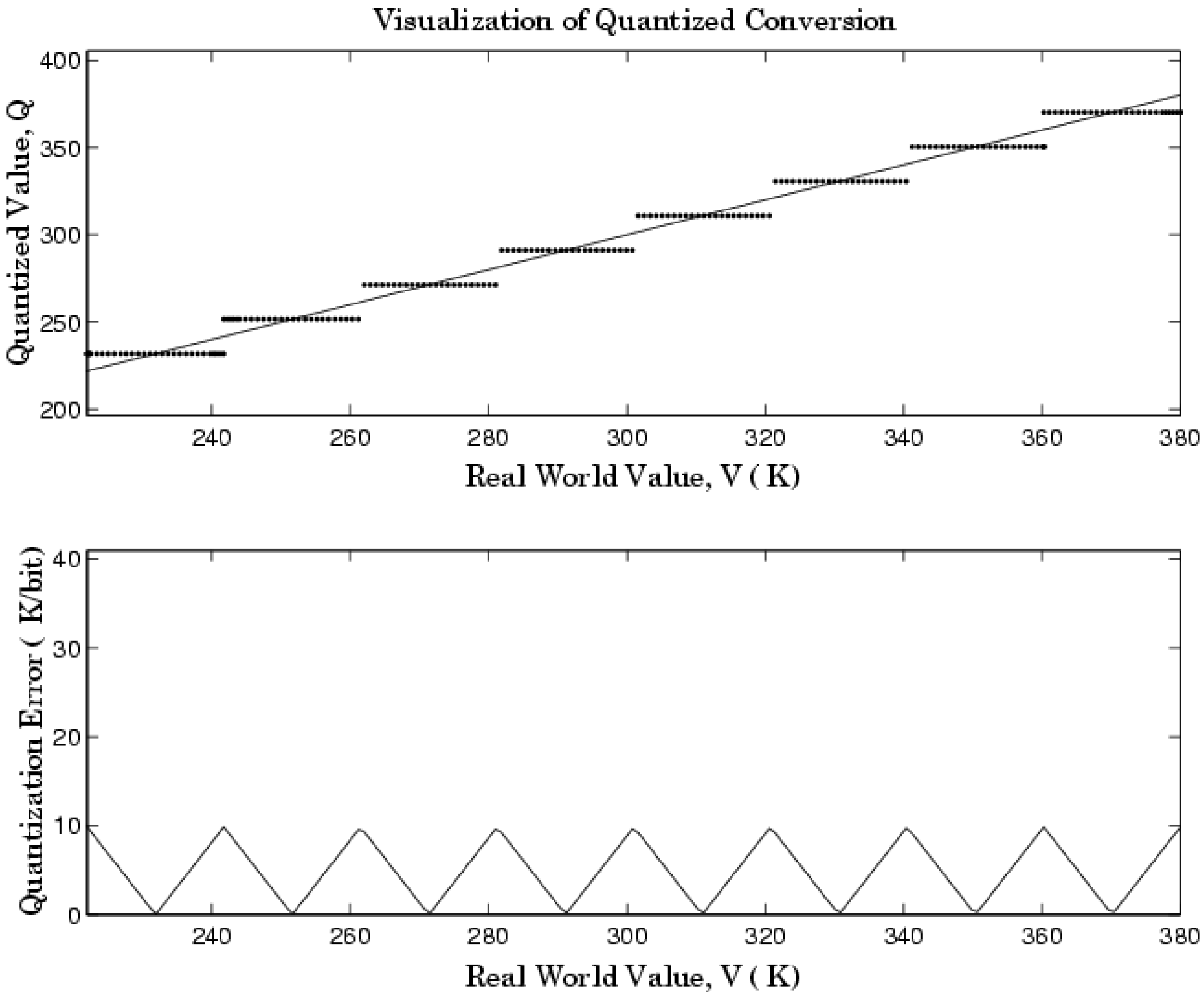
$$V = \left(\frac{380\ \text{K} - 222\ \text{K}}{8}\right)Q + 222\ \text{K} + 0.5\left(\frac{380\ \text{K} - 222\ \text{K}}{8}\right)$$

The quantized conversion and range of interest are shown in the following figure.



The quantization error is -9.875 K/bit ≤ *Error* ≤ 9.875 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.

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.

Why Use the Fixed-Point Designer Software?

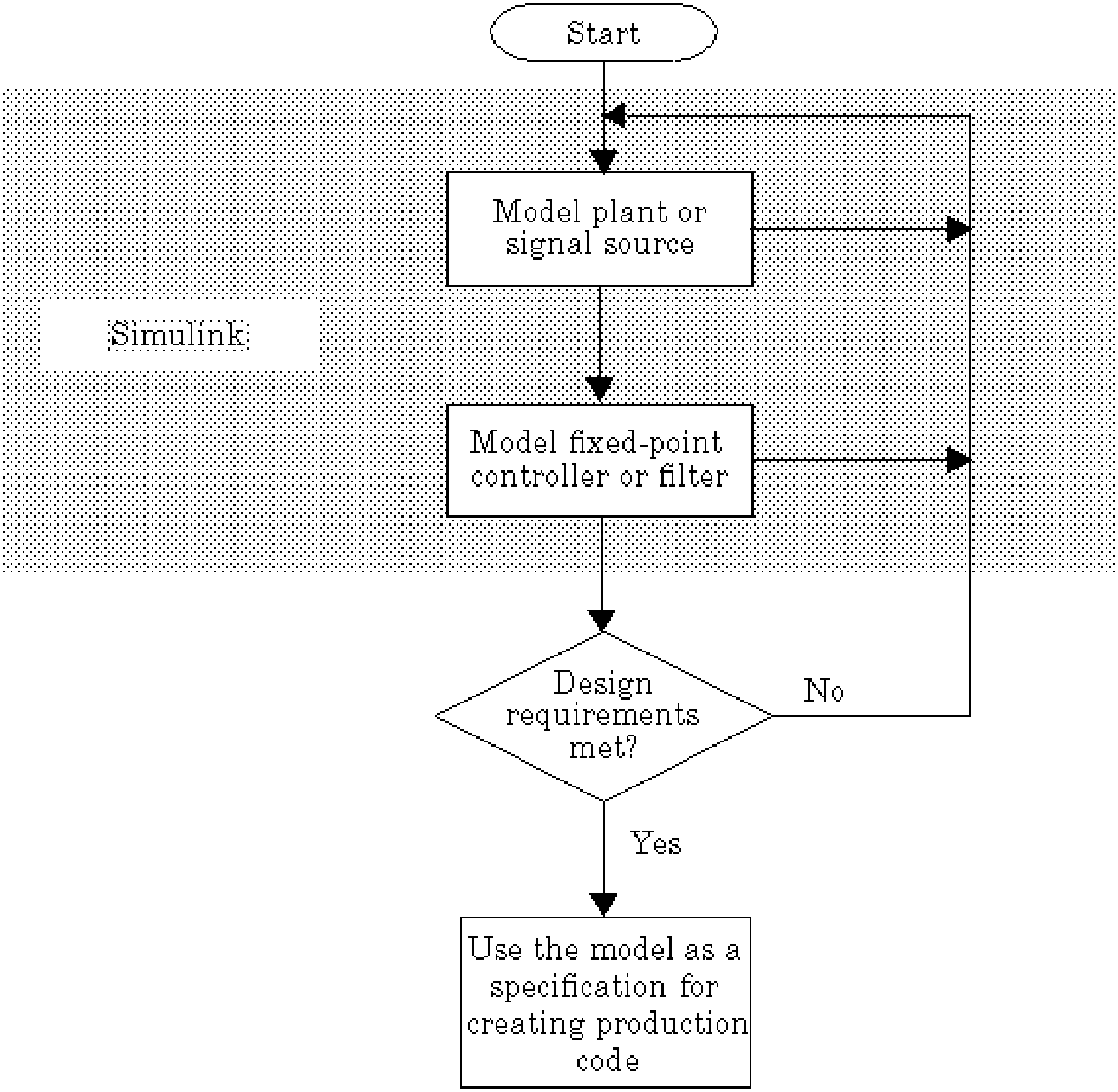
The Fixed-Point Designer software allows you to efficiently design control systems and digital filters that you will implement using fixed-point arithmetic. With the Fixed-Point Designer software, you can construct Simulink and Stateflow models that contain detailed fixed-point information about your 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.

Developing and Testing Fixed-Point Systems in Simulink

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.


Start
Model plant or signal source
Simulink
Model fixed-point controller or filter
Design requirements met?
No
Yes
Use the model as a specification for creating production code

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.

Supported Data Types

The Fixed-Point Designer software supports the following integer and fixed-point data types for simulation and code generation:

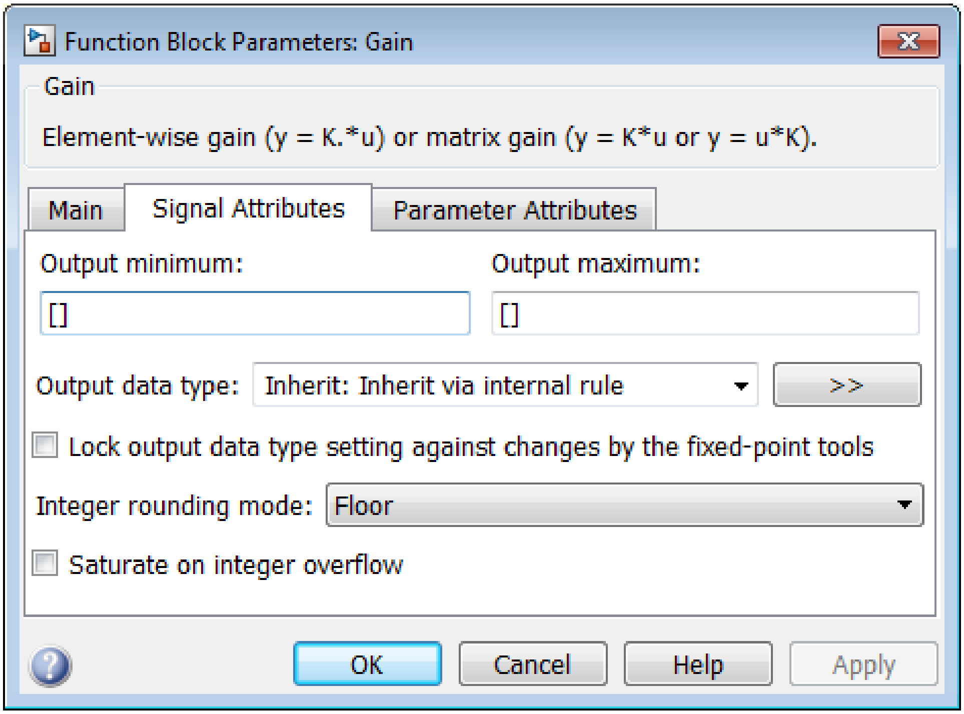
- Unsigned data types from 1 to 128 bits
- Signed data types from 2 to 128 bits
- Boolean, double, and single
- Scaled doubles

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.



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via internal rule

□ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

□ Saturate on integer overflow

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 26-22

“Specify Fixed-Point Data Types with the Data Type Assistant” on page 26-24

“Rounding” on page 26-27

“Overflow Handling” on page 26-28

“Lock the Output Data Type Setting” on page 26-28

“Real-World Values Versus Stored Integer Values” on page 26-29

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 “Specify Block Output Data Types”.

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 Numbers” on page 27-3 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 27-27 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 26-24). 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

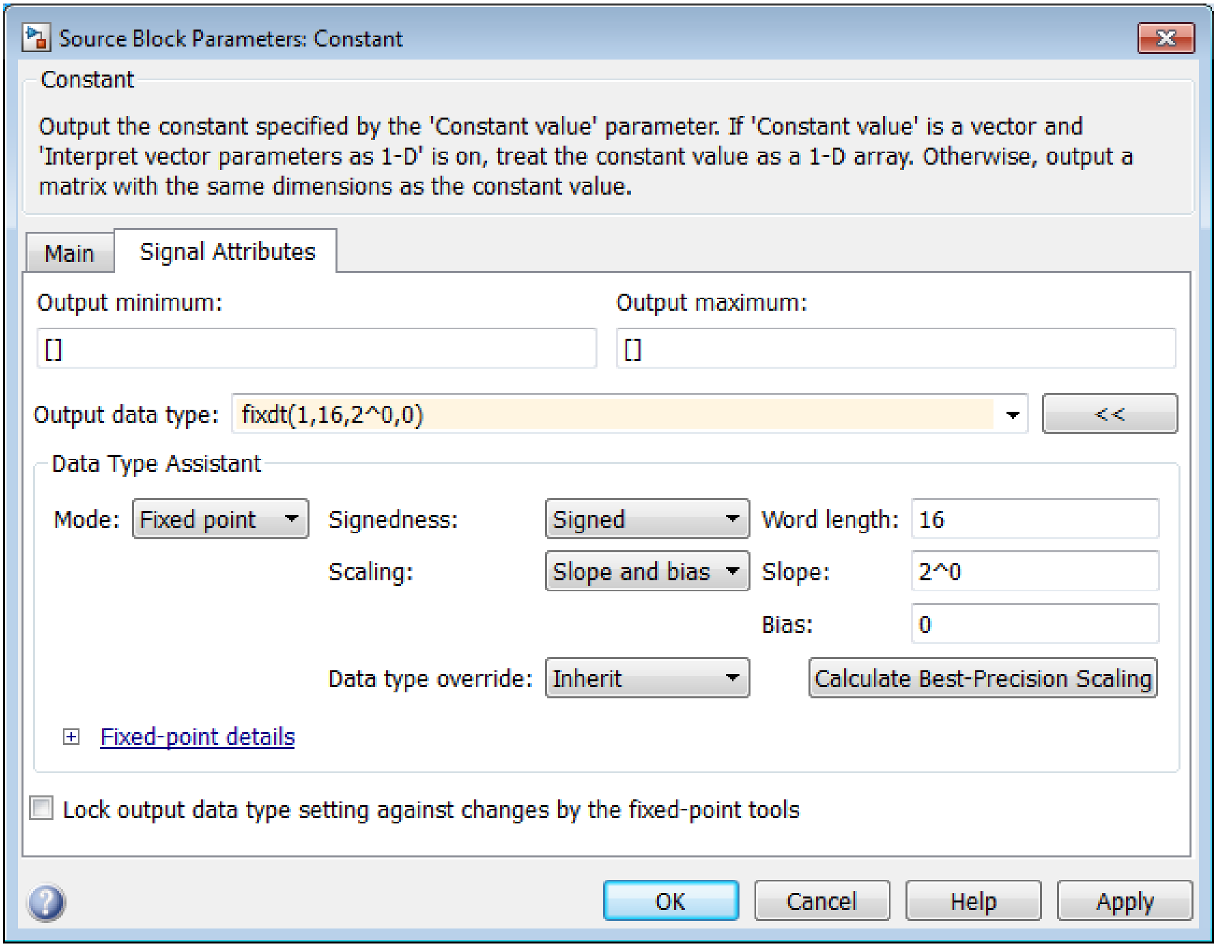
The Fixed-Point Designer software supports single-precision and double-precision floating-point numbers as defined by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. You can specify floating-point numbers with the Simulink `float` function.

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".

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:



Source Block Parameters: Constant

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Output minimum: []

Output maximum: []

Output data type: fixdt(1,16,2^0,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Slope and bias

Slope: 2^0

Bias: 0

Data type override: Inherit

□ Lock output data type setting against changes by the fixed-point tools

You can set the following fixed-point attributes:

Signedness

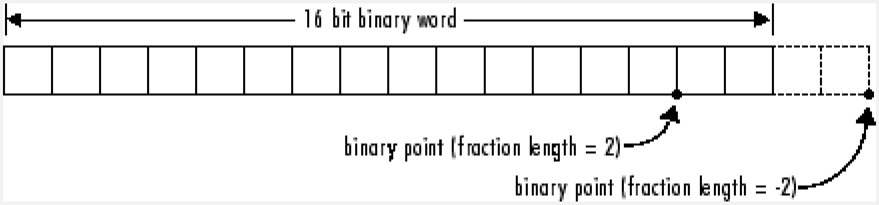
Select whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative quantities. Unsigned data represents positive values only.

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`	If you select this mode, the assistant displays the **Fraction length** field, specifying the binary point location. 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:  16 bit binary word binary point (fraction length = 2) binary point (fraction length = -2) See "Binary-Point-Only Scaling" on page 27-7 for more information.
`Slope and bias`	If you select this mode, the assistant displays fields for entering the **Slope** and **Bias**. • Slope can be any *positive* real number. • Bias can be any real number.

Scaling Mode	Description
	See "Slope and Bias Scaling" on page 27-8 for more information.
`Best precision`	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. See "Constant Scaling for Best Precision" on page 27-15 for more information.

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 28-4.

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.

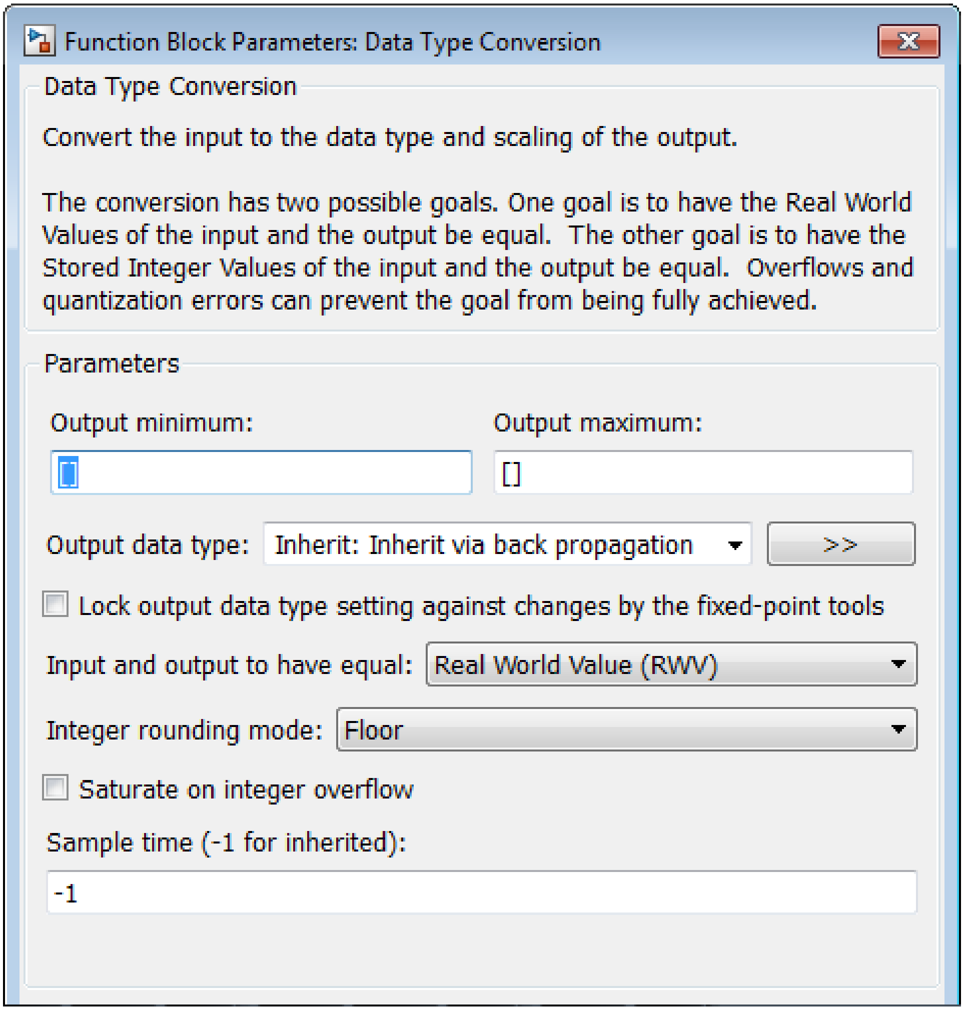
Lock the Output Data Type Setting

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. For more information, see . 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.



Function Block Parameters: Data Type Conversion

Data Type Conversion

Convert the input to the data type and scaling of the output.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

Parameters

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via back propagation

□ Lock output data type setting against changes by the fixed-point tools

Input and output to have equal: Real World Value (RWV)

Integer rounding mode: Floor

□ Saturate on integer overflow

Sample time (-1 for inherited): -1

The possible values are `Real World Value (RWV)` and `Stored Integer (SI)`.

In terms of the variables defined in “Scaling” on page 27-6, 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.

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 26-31

"Specify Fixed-Point Values Via Parameter Objects" on page 26-32

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 26-3 for more information.

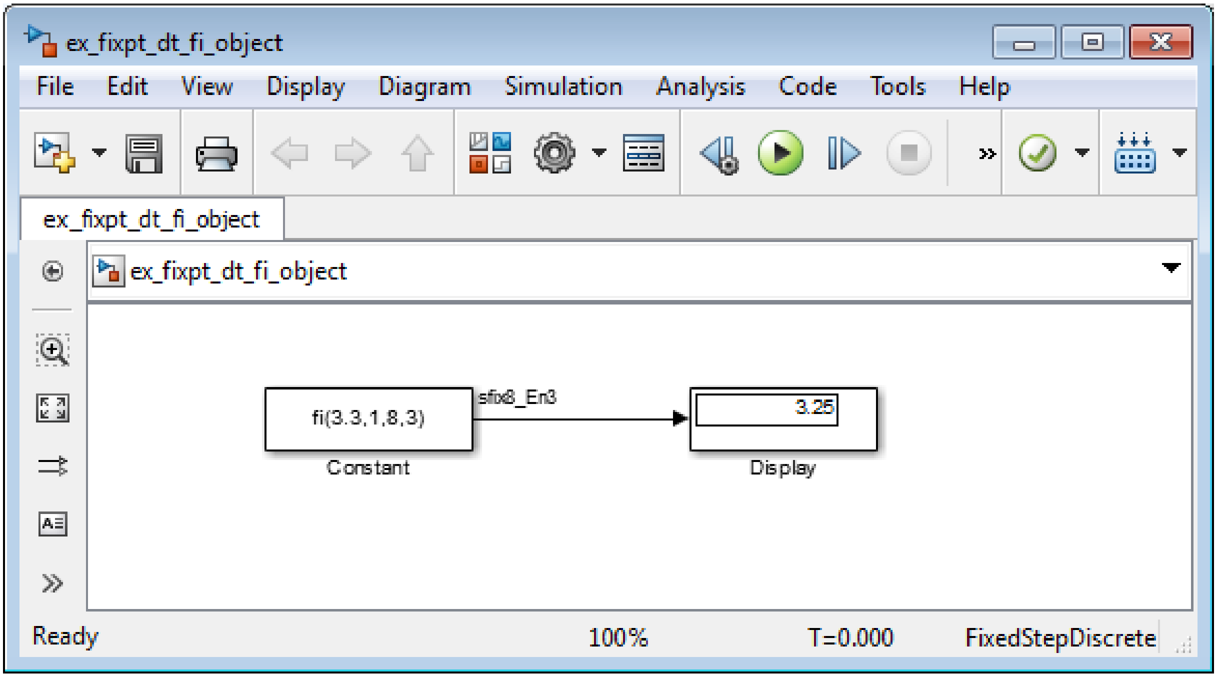
Specify Fixed-Point Values Directly

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
   ```

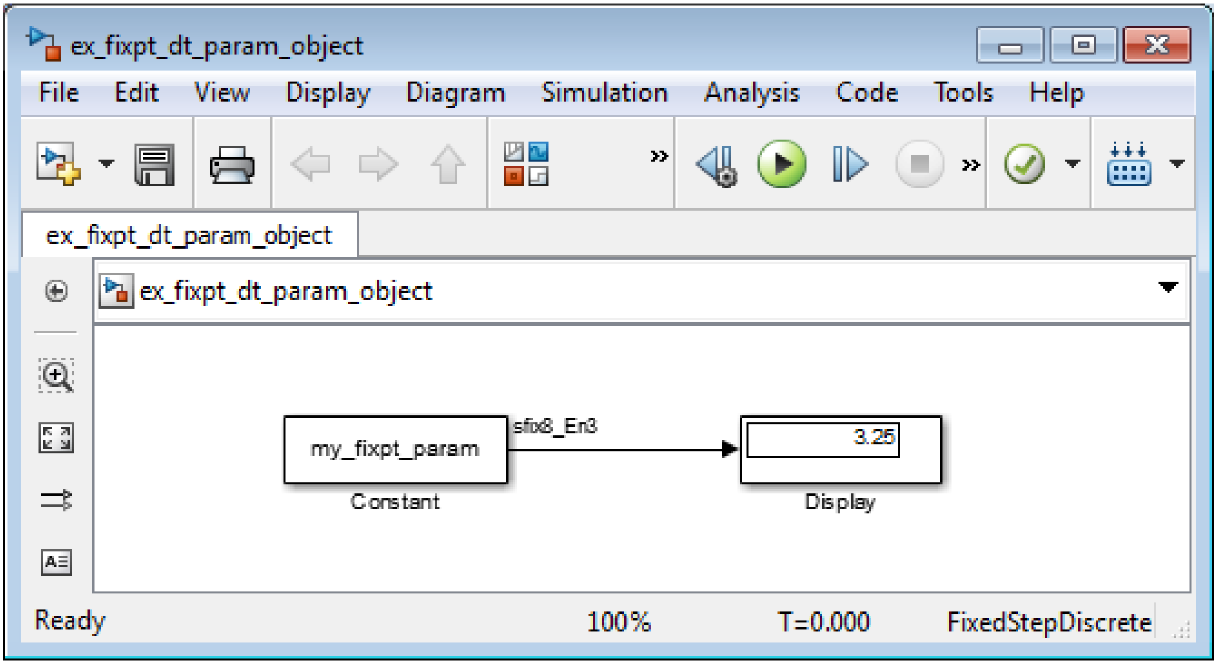
2 Specify either the name of a `fi` object or an expression that includes the `fi` constructor function as the parameter object's `Value` property:

```
my_fixpt_param.Value = fi(3.3,true,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` string:

```
my_fixpt_param.Value = 3.3;
my_fixpt_param.DataType = 'fixdt(true,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.

Pass Fixed-Point Data Between Simulink Models and the MATLAB Software

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`.

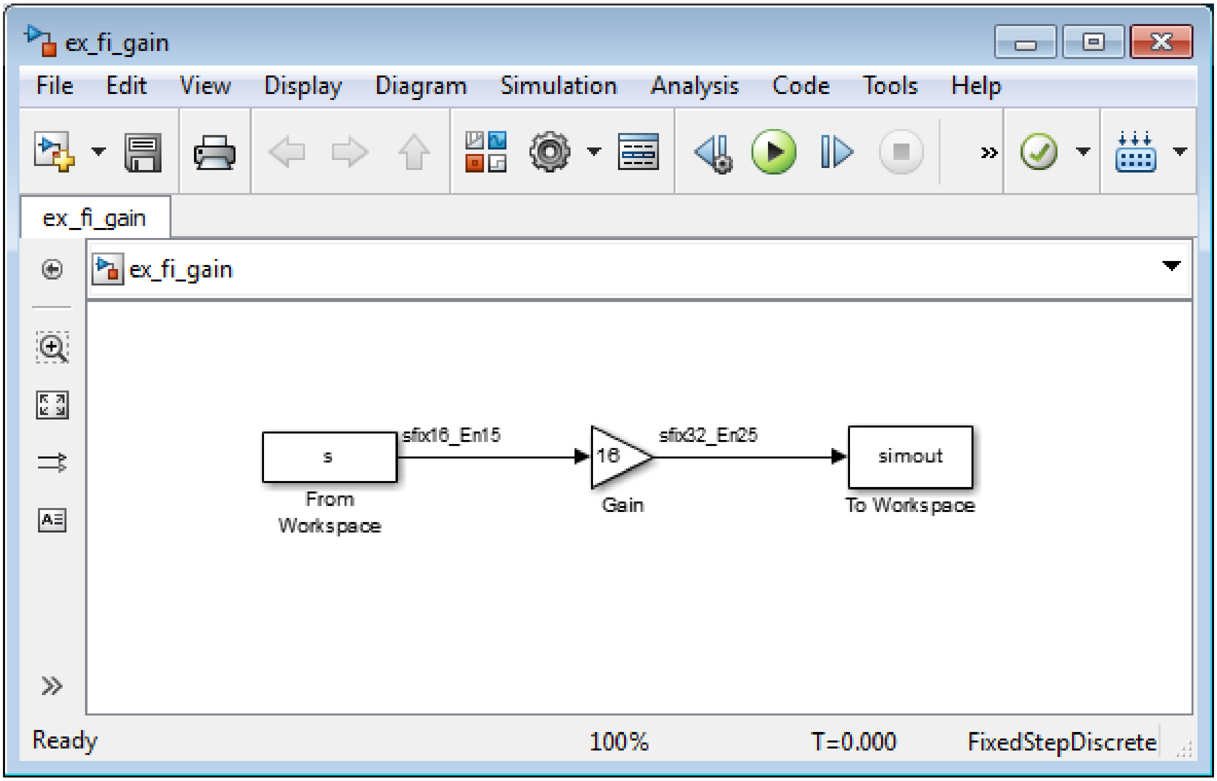
For example, you can use the following code to create a structure in the MATLAB workspace with a `fi` object in the `values` field. You can then use the From Workspace block to bring the data into a Simulink model.

```
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 Open the **Record** dropdown.

3 Select **Log Selected Signals**.

For more information, refer to “Signal Logging”.

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".

Cast from Doubles to Fixed Point

In this section...

"About This Example" on page 26-40

"Block Descriptions" on page 26-41

"Simulations" on page 26-42

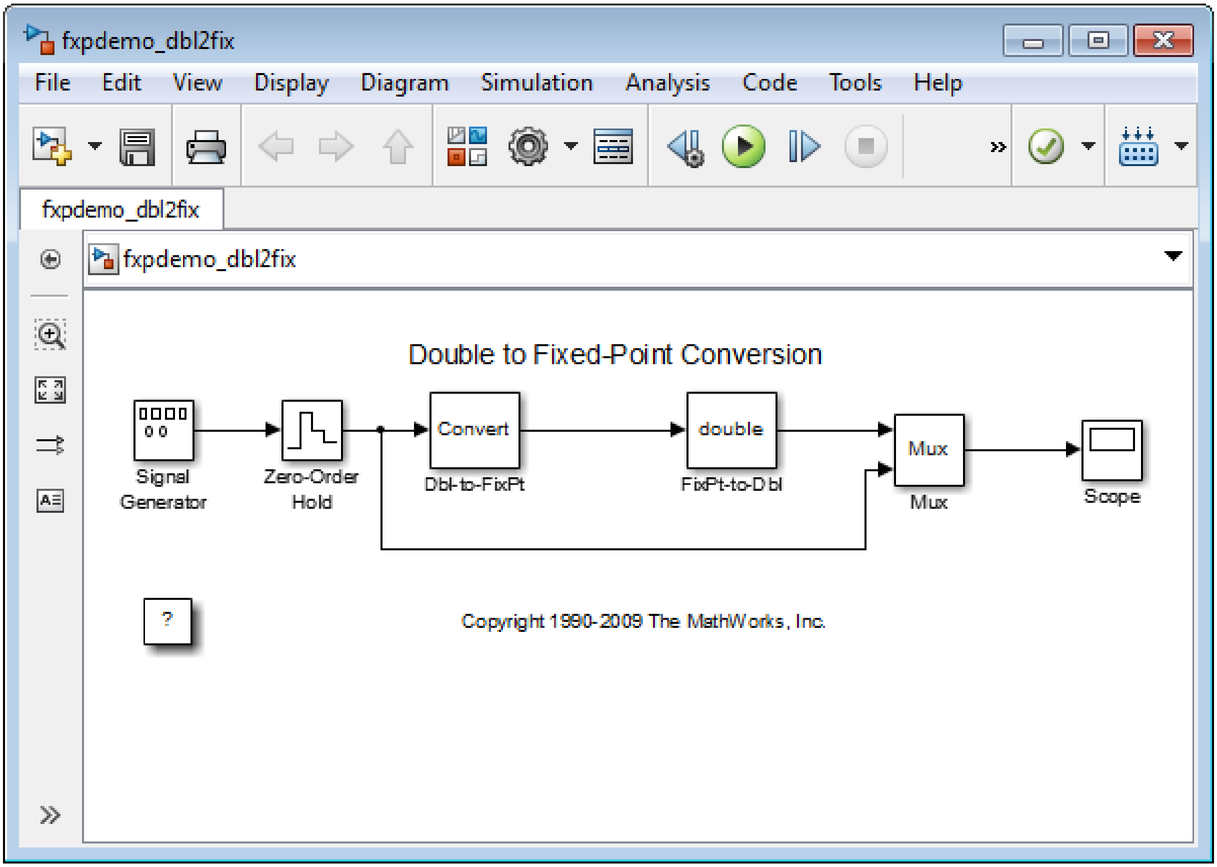
About This Example

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
```


fxpdemo_dbl2fix
File Edit View Display Diagram Simulation Analysis Code Tools Help
fxpdemo_dbl2fix
fxpdemo_dbl2fix
Double to Fixed-Point Conversion
Signal Generator
Zero-Order Hold
Convert
Dbl-to-FixPt
double
FixPt-to-Dbl
Mux
Mux
Scope
?
Copyright 1990-2009 The MathWorks, Inc.

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 27-6. For this scaling mode, the fractional slope *F* is 1 and there is no bias.

To run the simulation:

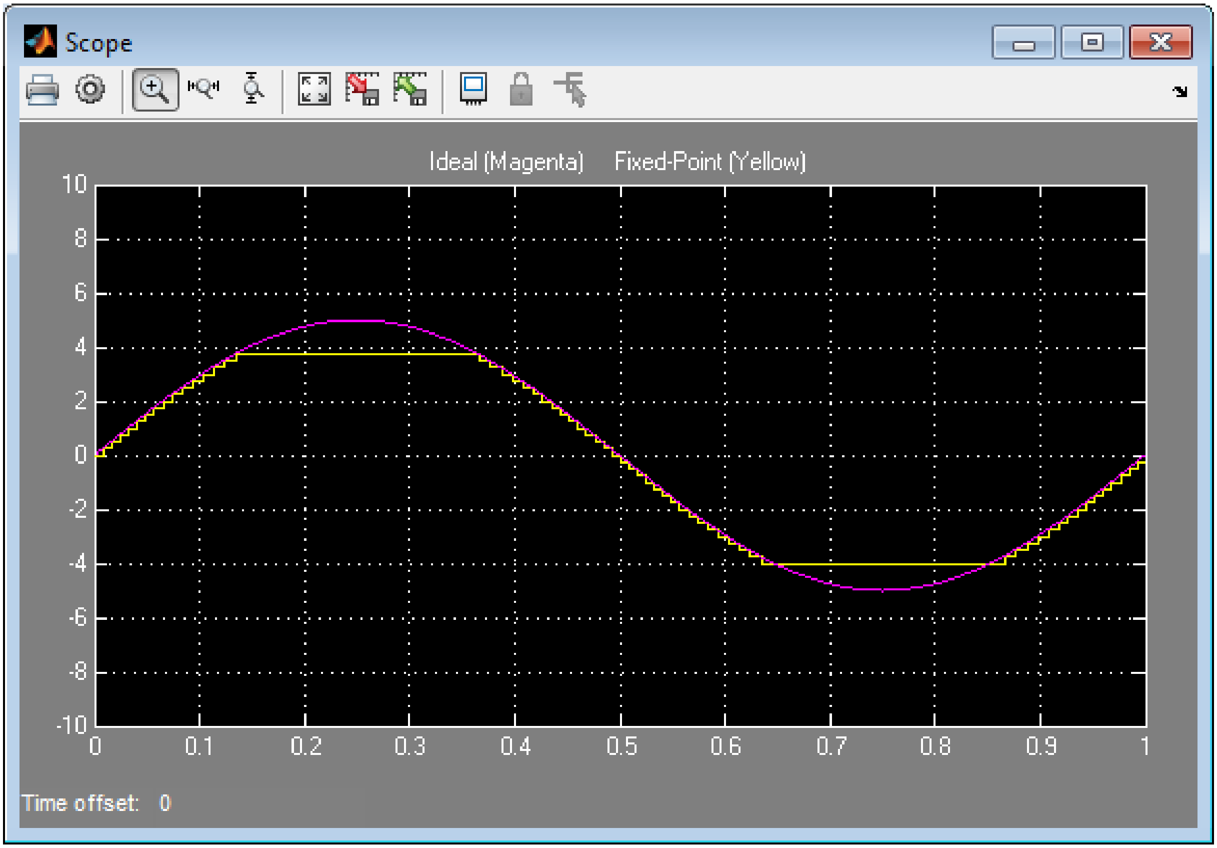
1. Configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval `[-5 5]`.
   - **a** Double-click the Signal Generator block to open the `Block Parameters` dialog.
   - **b** Set the **Wave form** parameter to `sine`.
   - **c** Set the **Amplitude** parameter 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 27-6. 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.(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 $1 \le F < 2$.

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].

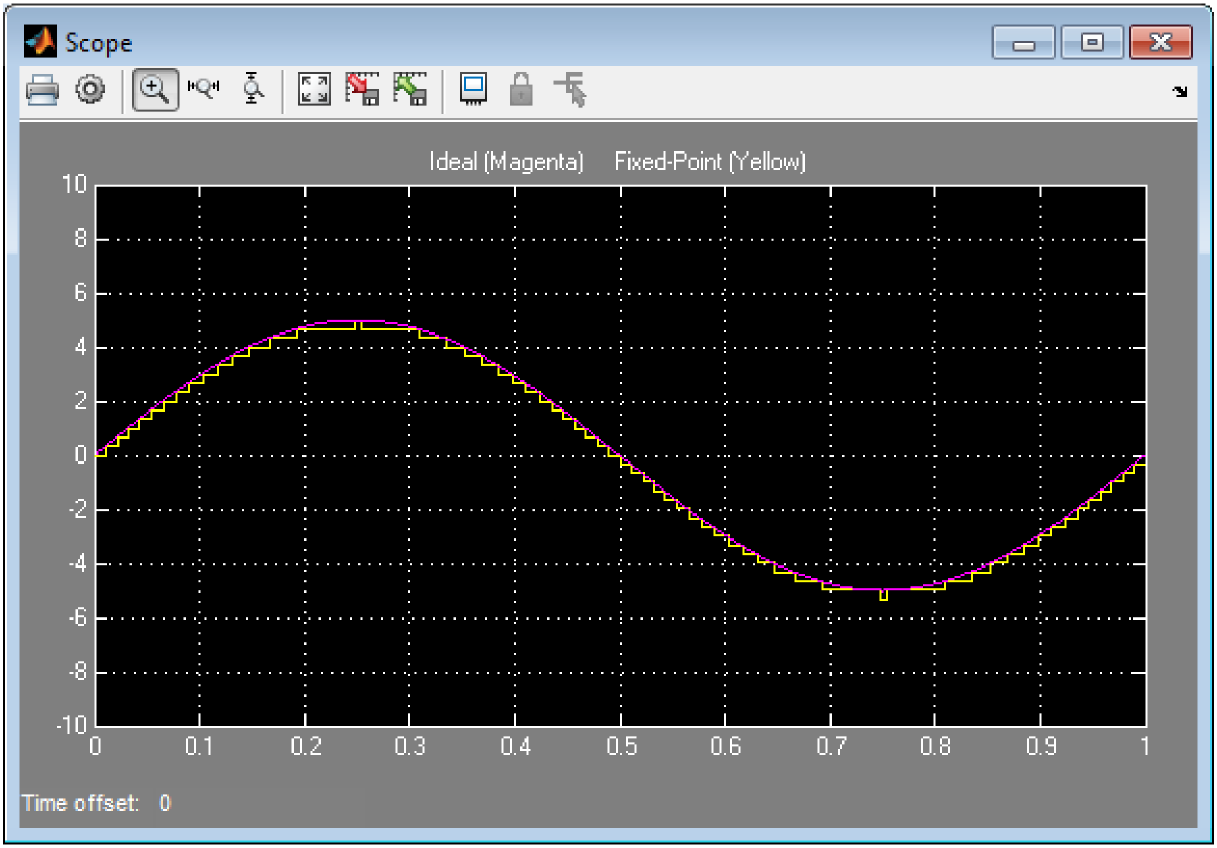
   c Verify that the **Integer rounding mode** parameter is `Floor`. `Floor` rounds the fixed-point result toward negative infinity.

   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.

Data Types and Scaling

- “Data Types and Scaling in Digital Hardware” on page 27-2
- “Fixed-Point Numbers” on page 27-3
- “Fixed-Point Numbers in Simulink” on page 27-15
- “Floating-Point Numbers” on page 27-27

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 binary point is the means by which fixed-point values are scaled. 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.

Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field. The blockset 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, doubles, and a nonstandard IEEE-style floating-point data type.

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 Numbers

In this section...

"Fixed-Point Numbers" on page 27-3

"Signed Fixed-Point Numbers" on page 27-4

"Binary Point Interpretation" on page 27-4

"Scaling" on page 27-6

"Quantization" on page 27-8

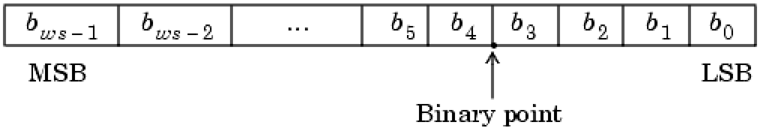
"Range and Precision" on page 27-10

Fixed-Point Numbers

Fixed-point numbers and their data types are characterized by their word size in bits, binary point, and whether they are signed or unsigned. The Fixed-Point Designer software supports integers and fixed-point numbers. The main difference among these data types is their binary point.

Note Fixed-point numbers can have a word size up to 128 bits.

A common representation of a binary fixed-point number , either signed or unsigned, is shown in the following figure.



where

- b_i are the binary digits (bits)
- ws is the word length in bits

- The most significant bit (MSB) is the leftmost bit, and is represented by location b_{ws-1}
- The least significant bit (LSB) is the rightmost bit, and is represented by location b_0
- The binary point is shown four places to the left of the LSB

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 supported by the 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.

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 $V = Q * 2 \hat{} E$. `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.

$$\begin{aligned} bbb. &= bbb.\times 2^{0} \\ bb.b &= bbb.\times 2^{-1} \\ b.bb &= bbb.\times 2^{-2} \\ .bbb &= bbb.\times 2^{-3} \end{aligned}$$

If the exponent were greater than 0 or less than -3, then the representation would involve lots of zeros.

$$\begin{aligned} bbb00000. &= bbb.\times 2^{5} \\ bbb00. &= bbb.\times 2^{2} \\ .00bbb &= bbb.\times 2^{-5} \\ .00000bbb &= bbb.\times 2^{-8} \end{aligned}$$

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-word 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".

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

$$V \approx \tilde{V} = SQ + B,$$

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.

The slope is partitioned into two components:

- 2^E specifies the binary point. E is the fixed power-of-two exponent.
- F is the slope adjustment factor. It is normalized such that $1 \le F < 2$.

Note S and B are constants and do not show up in the computer hardware directly. Only the quantization value Q is stored in computer memory.

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 28-33.

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:

- $F = 1$
- $S = F2^E = 2^E$
- $B = 0$

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

$$\tilde{V} = S.\left[\sum_{i=0}^{ws-1} b_i 2^i\right] + B,$$

while the value of a signed fixed-point quantity is given by

$$\tilde{V} = S.\left[-b_{ws-1}2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i\right] + B,$$

where

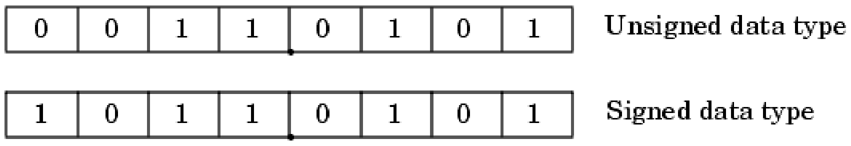
- b_i are binary digits, with $b_i = 1,0$, for $i = 0,1,...,ws-1$
- The word size in bits is given by ws, with ws = 1, 2, 3,..., 128.

- 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.



0	0	1	1	0	1	0	1	Unsigned data type
1	0	1	1	0	1	0	1	Signed data type

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

$$\begin{aligned}\tilde{V} &= \left(F2^E\right)Q = 2^E\left[\sum_{i=0}^{ws-1} b_i 2^i\right] \\ &= 2^{-4}\left(0\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0\right) \\ &= 3.3125.\end{aligned}$$

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

$$\begin{aligned}\tilde{V} &= \left(F2^{E}\right)Q = 2^{E}\left[\sum_{i=0}^{ws-1} b_i 2^i\right] \\ &= 2^{-4}\left(1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\right) \\ &= 11.3125,\end{aligned}$$

while the two's complement value is

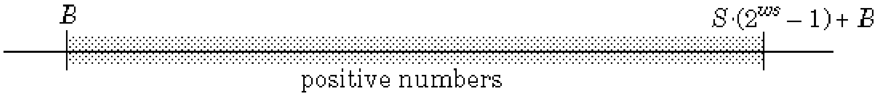
$$\begin{aligned}\tilde{V} &= \left(F2^{E}\right)Q = 2^{E}\left[-b_{ws-1}2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i\right] \\ &= 2^{-4}\left(-1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\right) \\ &= -4.6875.\end{aligned}$$

Range and Precision

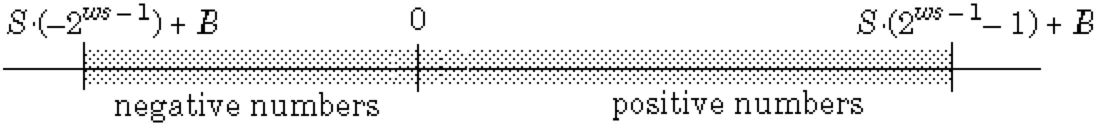
The *range* of a number gives the limits of the representation, while the *precision* gives the distance between successive numbers in the representation. The range and precision of a fixed-point number depend on the length of the word and the scaling.

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 Interpretation" on page 27-4 are given in the following table. See "Precision" on page 28-3 and "Range" on page 28-27 for more information.

Fixed-Point Data Type Range and Default Scaling

Name	Data Type	Low Limit	High Limit	Default Scaling (~Precision)
Unsigned Integer	`fixdt(0,ws,0)`	0	$2^{ws}-1$	1
Signed Integer	`fixdt(1,ws,0)`	-2^{ws-1}	$2^{ws-1}-1$	1
Unsigned Binary Point	`fixdt(0,ws,fl)`	0	$(2^{ws-1}-1)2^{-fl}$	2^{-fl}
Signed Binary Point	`fixdt(1,ws,fl)`	$-2^{ws-1-fl}$	$(2^{ws-1}-1)2^{-fl}$	2^{-fl}
Unsigned Slope Bias	`fixdt(0,ws,s,b)`	b	$s(2^{ws}-1)+b$	s
Signed Slope Bias	`fixdt(1,ws,s,b)`	$-s(2^{ws-1})+b$	$s(2^{ws-1}-1)+b$	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

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”) 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

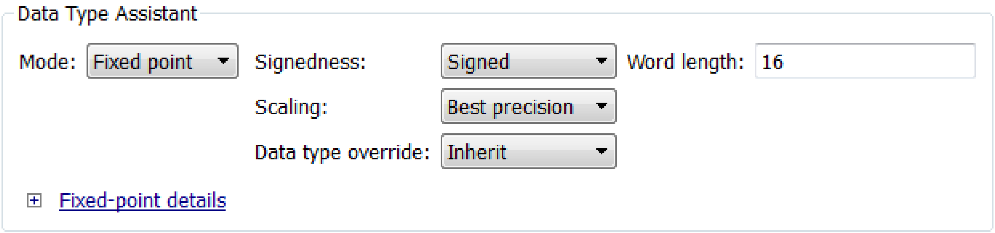
.

   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

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 27-6.

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		
`ufix`	Unsigned fixed-point data type	`ufix8` is an 8-bit unsigned fixed-point data type
`sfix`	Signed fixed-point data type	`sfix128` is a 128-bit signed fixed-point data type

Symbol	Description	Example
fltu	Scaled Doubles override of an unsigned fixed-point data type (ufix)	fltu32 is a scaled doubles override of ufix32
flts	Scaled Doubles override of a signed fixed-point data type (sfix)	flts64 is a scaled doubles override of sfix64
Number Encoding		
e	10^	125e8 equals 125*(10^(8))
n	Negative	n31 equals -31
p	Decimal point	1p5 equals 1.5 p2 equals 0.2
Scaling Encoding		
S	Slope	ufix16_S5_B7 is a 16-bit unsigned fixed-point data type with Slope of 5 and Bias of 7
B	Bias	ufix16_S5_B7 is a 16-bit unsigned fixed-point data type with Slope of 5 and Bias of 7
E	Fixed exponent (2^) A negative fixed exponent describes the fraction length	sfix32_En31 is a 32-bit signed fixed-point data type with a fraction length of 31
F	Slope adjustment factor	ufix16_F1p5_En50 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 **Note** If you pass this string to the slDataTypeAndScale function, it returns a valid fixdt data type.	No example available. For backwards compatibility only. To identify and replace calls to slDataTypeAndScale, use the "Check for calls to slDataTypeAndScale" Model Advisor check.
T or t	Compressed encoding for Slope **Note** If you pass this string to the slDataTypeAndScale, it returns a valid fixdt data type.	No example available. For .backwards compatibility only. To identify and replace calls to slDataTypeAndScale, use the "Check for calls to slDataTypeAndScale" Model Advisor check.

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 `sfix16_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 27-18. 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.

Consider an example where you are storing `0.75001` degrees Celsius in a data type `sfix16_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 `sfix16_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:

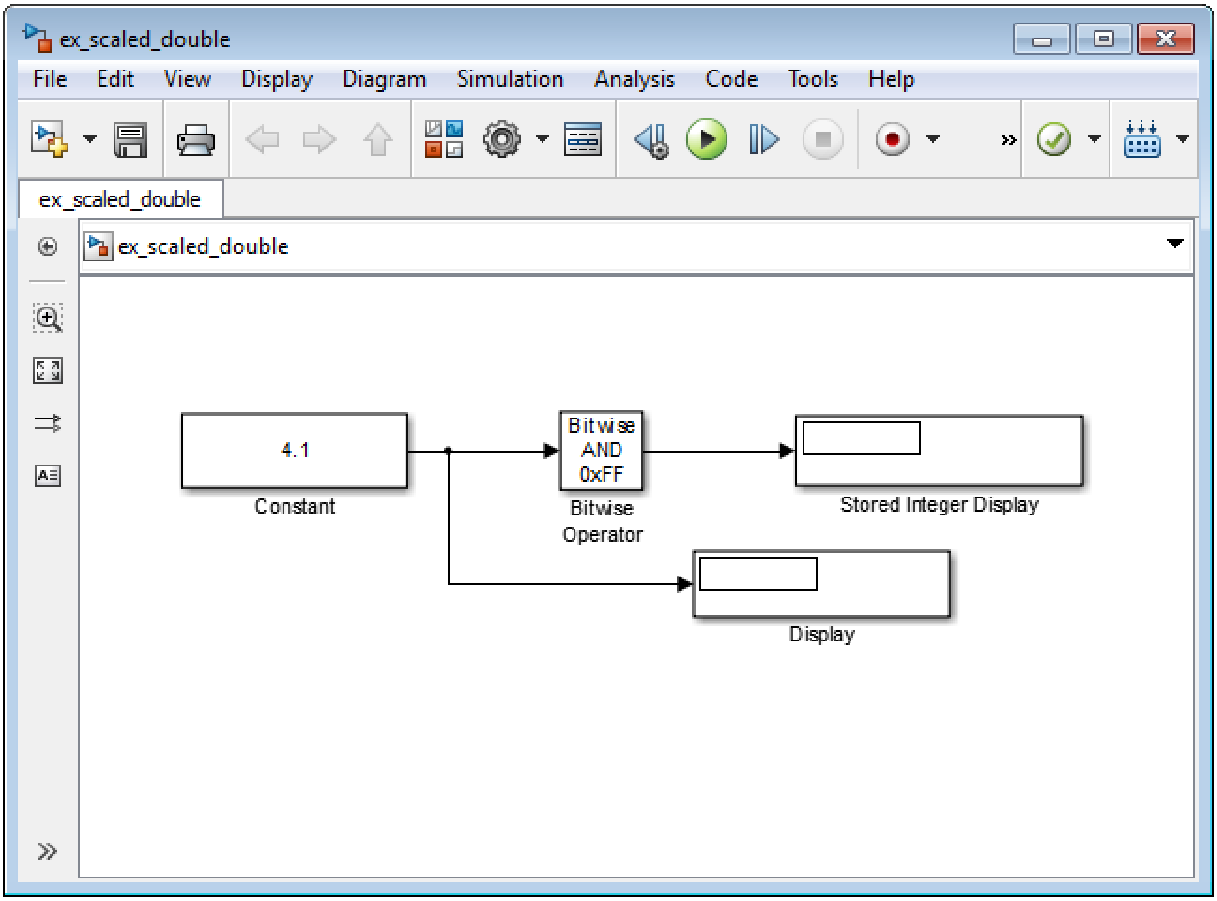
- Testing and debugging
- Applying data type overrides to individual subsystems

  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 27-20.

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 27-6.

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 > 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.

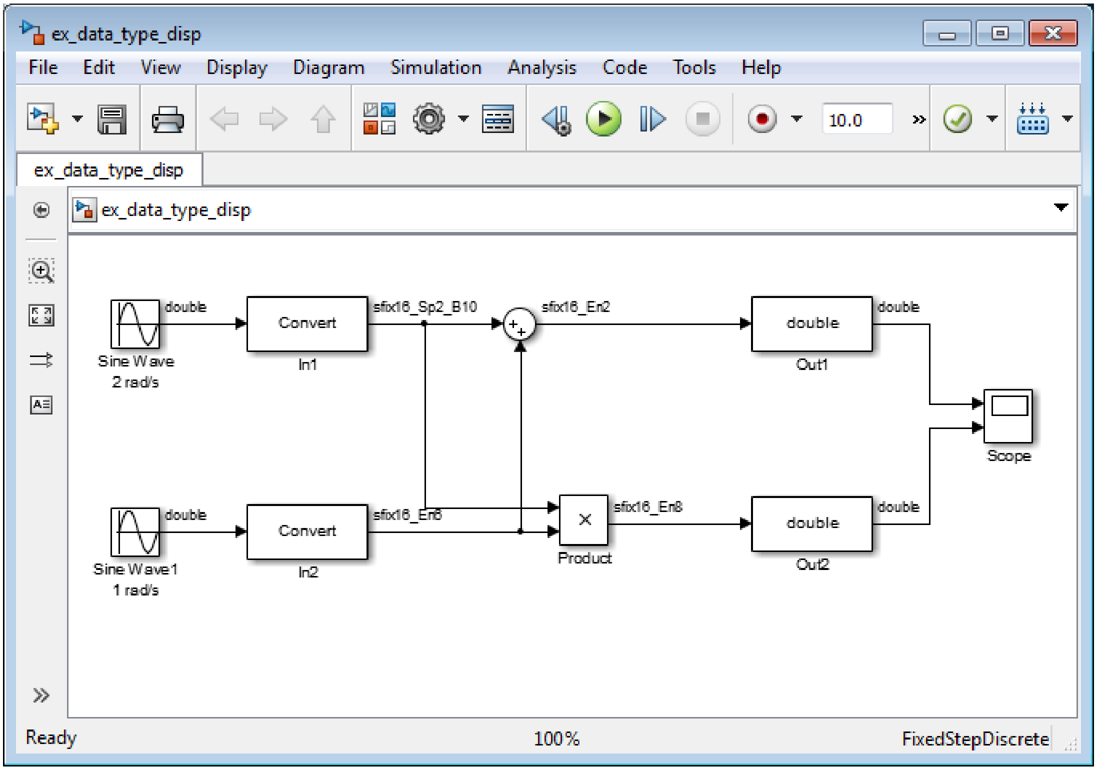
Display Port Data Types

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.

Floating-Point Numbers

In this section...

"Floating-Point Numbers" on page 27-27

"Scientific Notation" on page 27-28

"The IEEE Format" on page 27-28

"Range and Precision" on page 27-30

"Exceptional Arithmetic" on page 27-33

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. Additionally, a nonstandard IEEE-style number is supported.

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

$$\pm d.dddd \times 10^{p} = \pm ddddd.0 \times 10^{p-4} = \pm 0.ddddd \times 10^{p+1},$$

where $d = 0,\ldots,9$ 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

$$\pm b.bbbb \times 2^{q} = \pm bbbbb.0 \times 2^{q-4} = \pm 0.bbbbb \times 2^{q+1},$$

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 Interpretation” on page 27-4.

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.

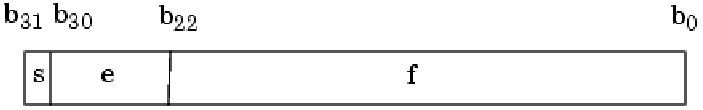
To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction of the form 1.*f* where *f* has a fixed size for a given data type. Since the leftmost fraction bit is always a 1, it is unnecessary to store this bit and is therefore implicit (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 27-33 and "NaN" on page 27-33.

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 27-28, "The Exponent Field" on page 27-29, and "The Fraction Field" on page 27-29. A representation of this format is given below.



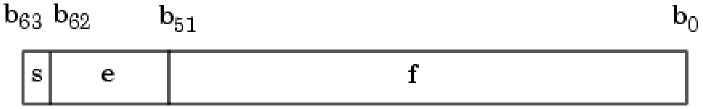
The relationship between this format and the representation of real numbers is given by

$$value = \begin{cases} (-1)^s(2^{e-127})(1.f) & \text{normalized, } 0 < e < 255, \\ (-1)^s(2^{e-126})(0.f) & \text{denormalized, } e = 0,\ f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

"Exceptional Arithmetic" on page 27-33 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 27-28, "The Exponent Field" on page 27-29, and "The Fraction Field" on page 27-29. 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

$$value = \begin{cases} (-1)^s(2^{e-1023})(1.f) & \text{normalized, } 0 < e < 2047, \\ (-1)^s(2^{e-1022})(0.f) & \text{denormalized, } e = 0,\ f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

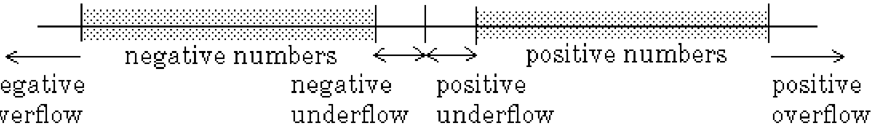
"Exceptional Arithmetic" on page 27-33 discusses denormalized values.

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 ±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	$2^{-52} \approx 10^{-16}$
Nonstandard	$2^{(1 - bias)}$	$(2 - 2^{-f}) \cdot 2^{bias}$	$2^{(e - 1)} - 1$	2^{-f}

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 $-\infty \leq$ (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: ∞–∞, –∞+∞, 0×∞, 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 28-2
- “Precision” on page 28-3
- “Range” on page 28-27
- “Recommendations for Arithmetic and Scaling” on page 28-33
- “Parameter and Signal Conversions” on page 28-44
- “Rules for Arithmetic Operations” on page 28-49
- “Conversions and Arithmetic Operations” on page 28-67

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.

Precision

In this section...

"Limitations on Precision" on page 28-3

"Rounding" on page 28-4

"Choose a Rounding Mode" on page 28-4

"Rounding Modes for Fixed-Point Simulink Blocks" on page 28-5

"Rounding Mode: Ceiling" on page 28-7

"Rounding Mode: Convergent" on page 28-8

"Rounding Mode: Floor" on page 28-10

"Rounding Mode: Nearest" on page 28-11

"Rounding Mode: Round" on page 28-12

"Rounding Mode: Simplest" on page 28-14

"Rounding Mode: Zero" on page 28-17

"Pad with Trailing Zeros" on page 28-19

"Limitations on Precision and Errors" on page 28-20

"Maximize Precision" on page 28-21

"Net Slope and Net Bias Precision" on page 28-21

"Detect Net Slope and Bias Precision Issues" on page 28-24

"Detect Fixed-Point Constant Precision Loss" on page 28-25

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 27-10.

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?

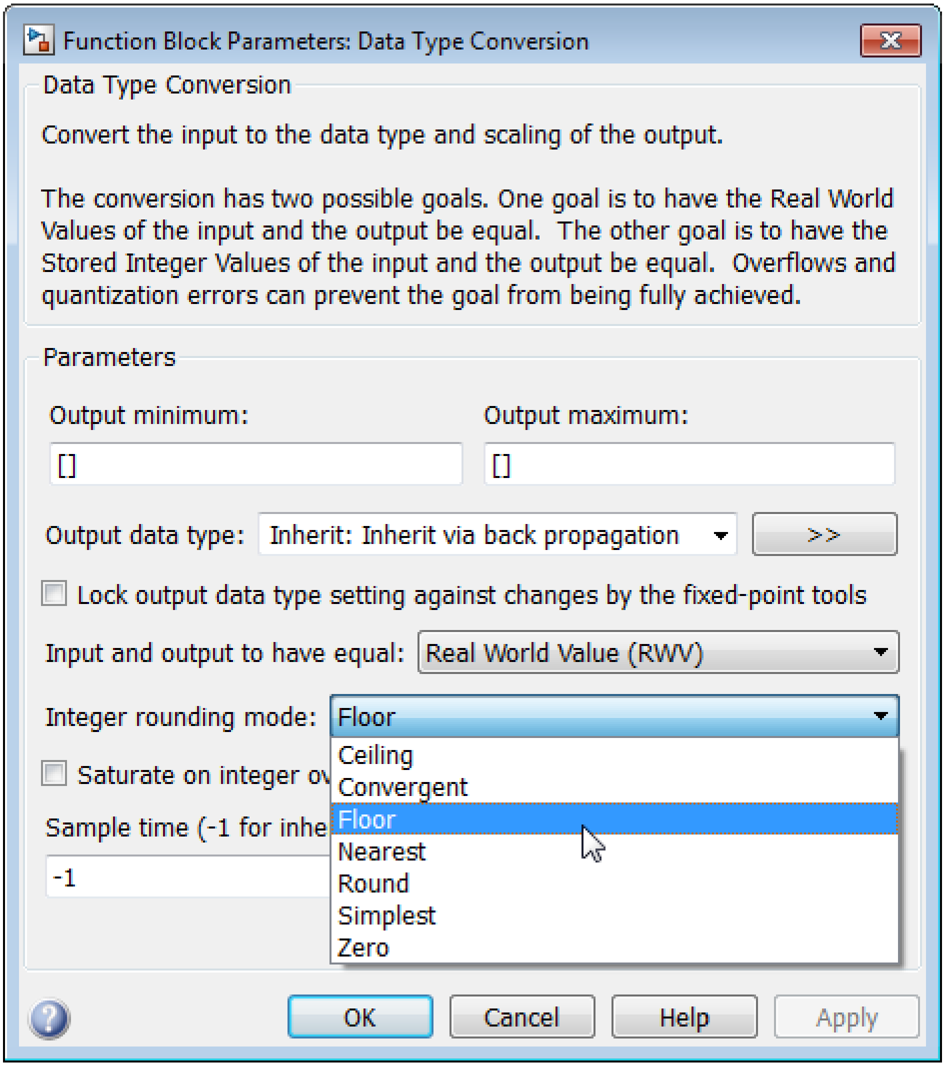
For more information on when to use each rounding mode, see “Rounding Methods” on page 1-6 in the *Fixed-Point Designer User’s Guide*.

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.

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.



Function Block Parameters: Data Type Conversion

Data Type Conversion

Convert the input to the data type and scaling of the output.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

Parameters

Output minimum: []

Output maximum: []

Output data type: Inherit: Inherit via back propagation

☐ Lock output data type setting against changes by the fixed-point tools

Input and output to have equal: Real World Value (RWV)

Integer rounding mode: Floor

☐ Saturate on integer ov

Sample time (-1 for inhe

-1

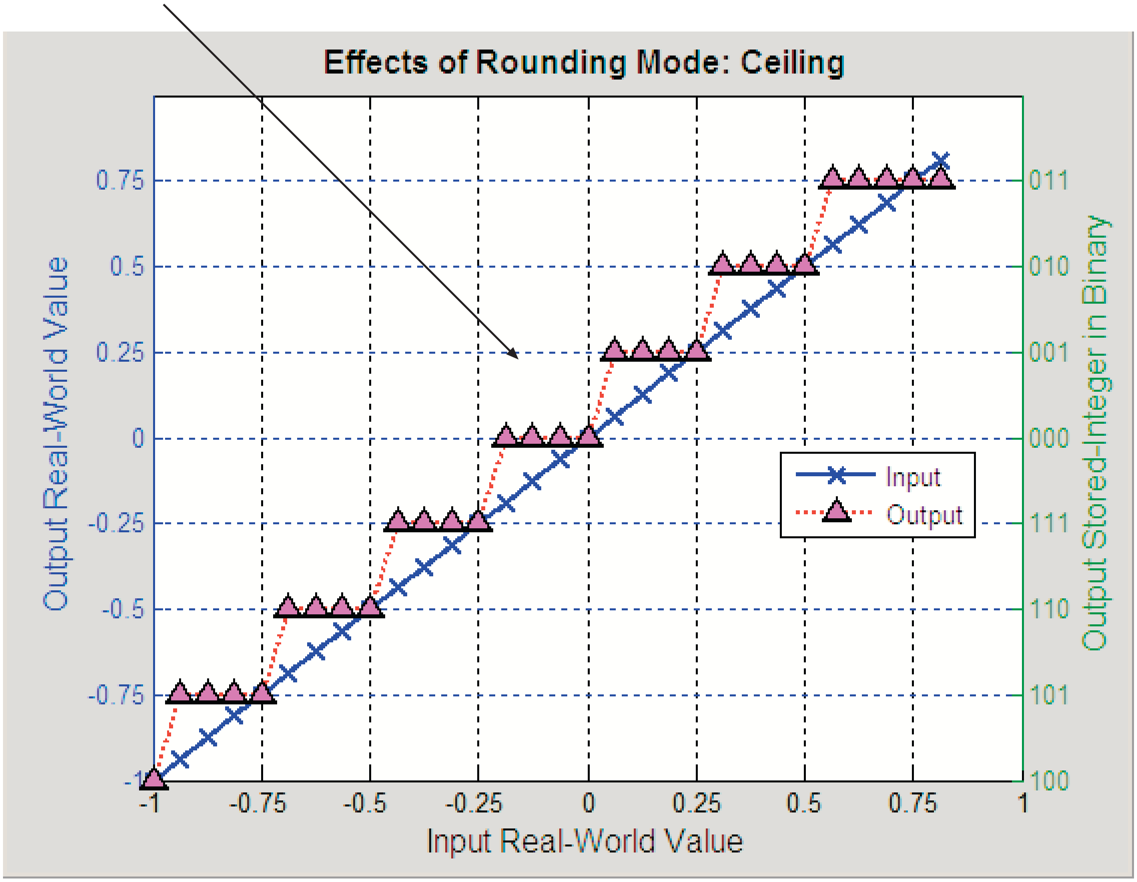
The following table illustrates the differences between these rounding modes:

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.	For positive numbers, ties are rounded toward the closest representable number in the direction of positive infinity. For negative numbers, ties are rounded toward the closest representable number in the direction of negative infinity.
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.



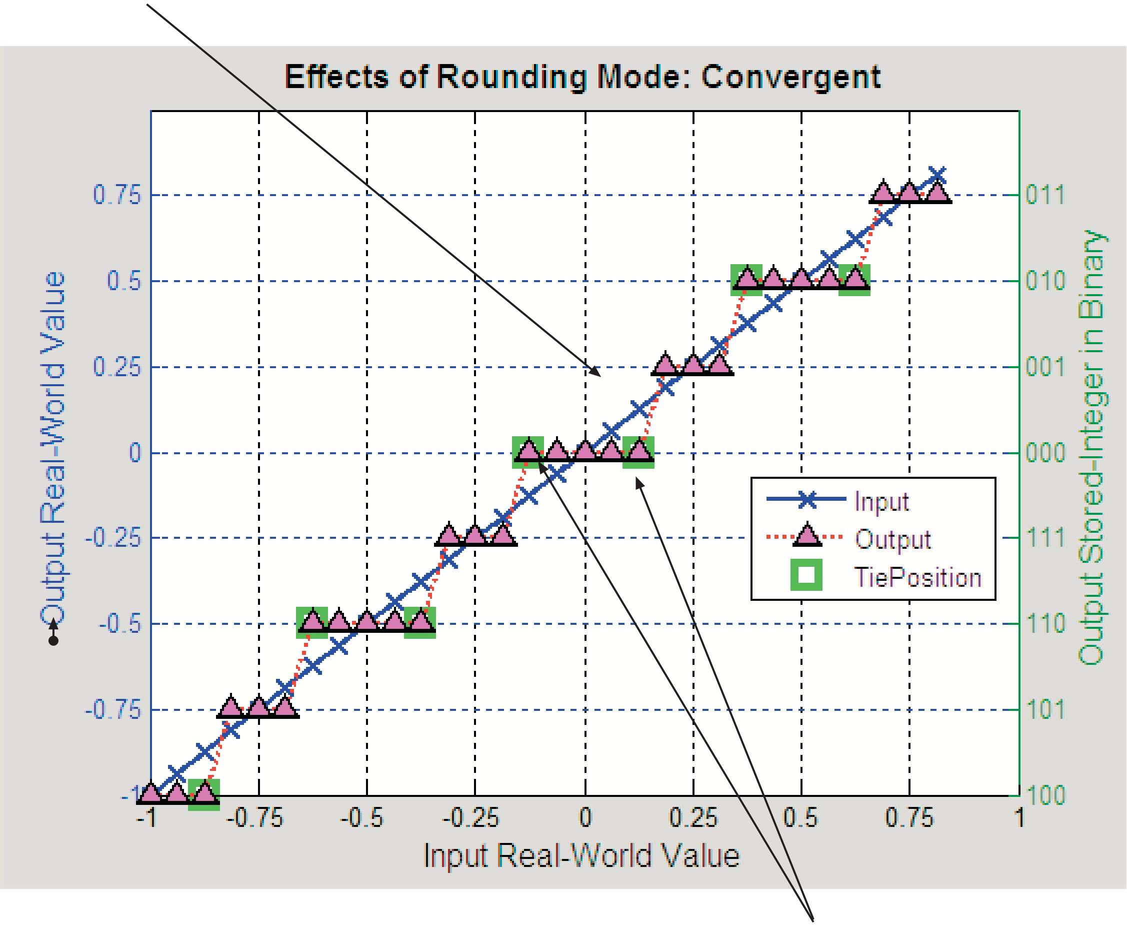
All numbers are rounded toward positive infinity

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

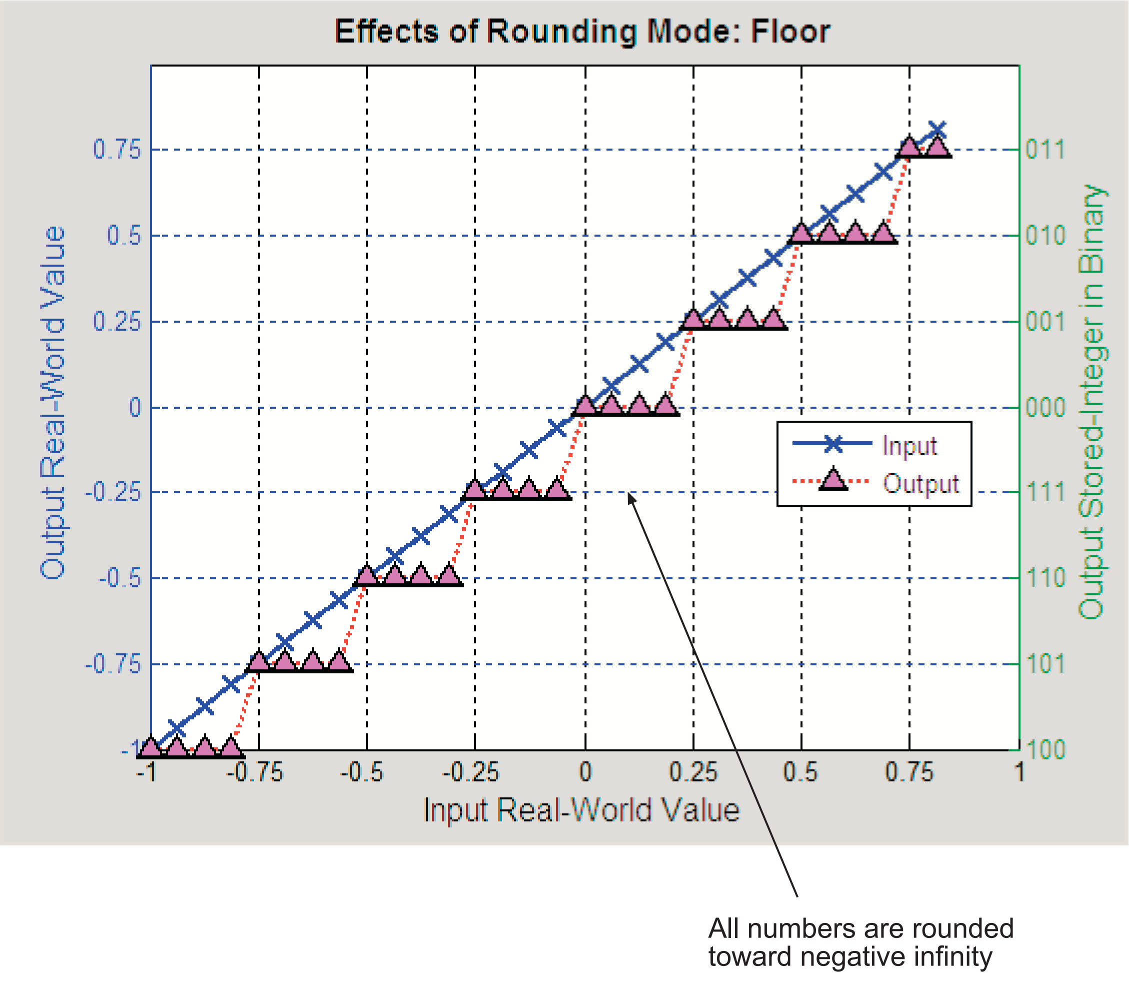


Ties are rounded to the nearest even number

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

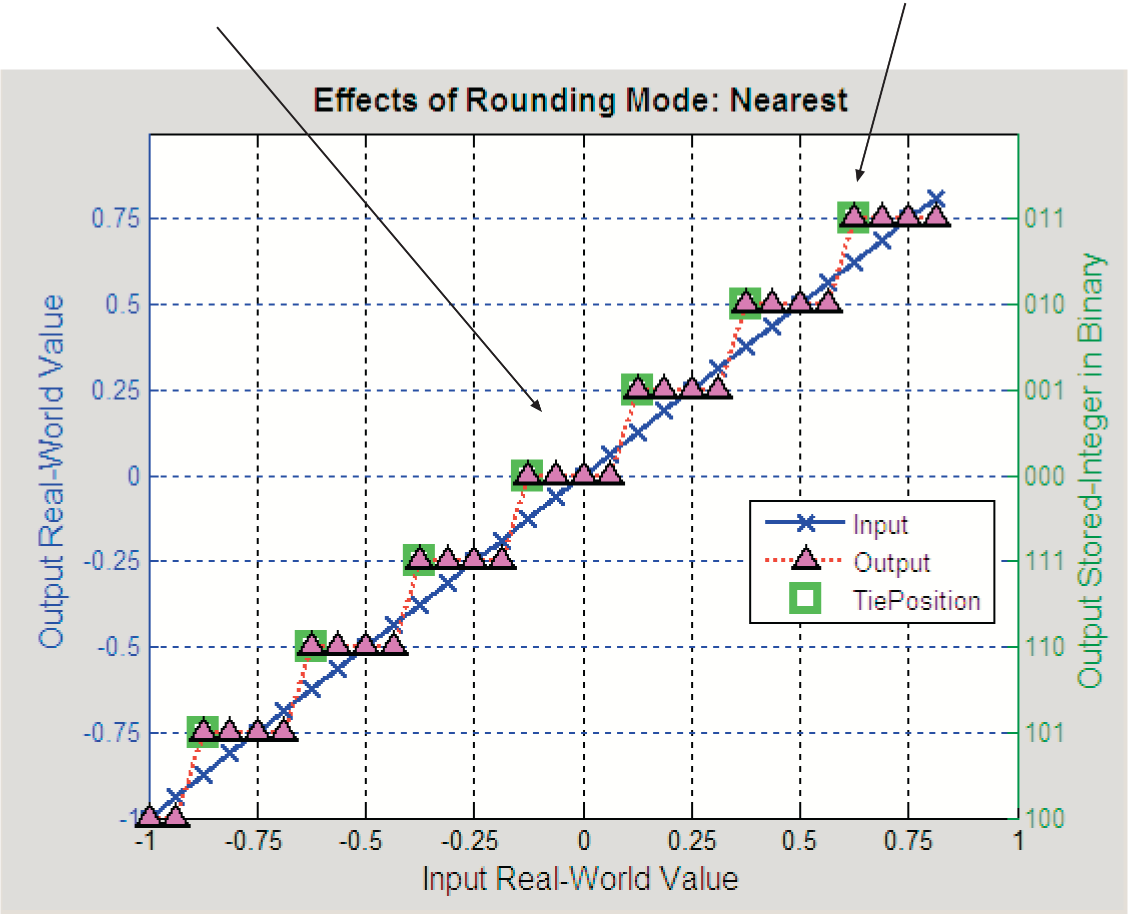
Rounding Mode: Nearest

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.

Ties are rounded to the closest representable number in the direction of positive infinity

All numbers are rounded to the nearest representable number



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.

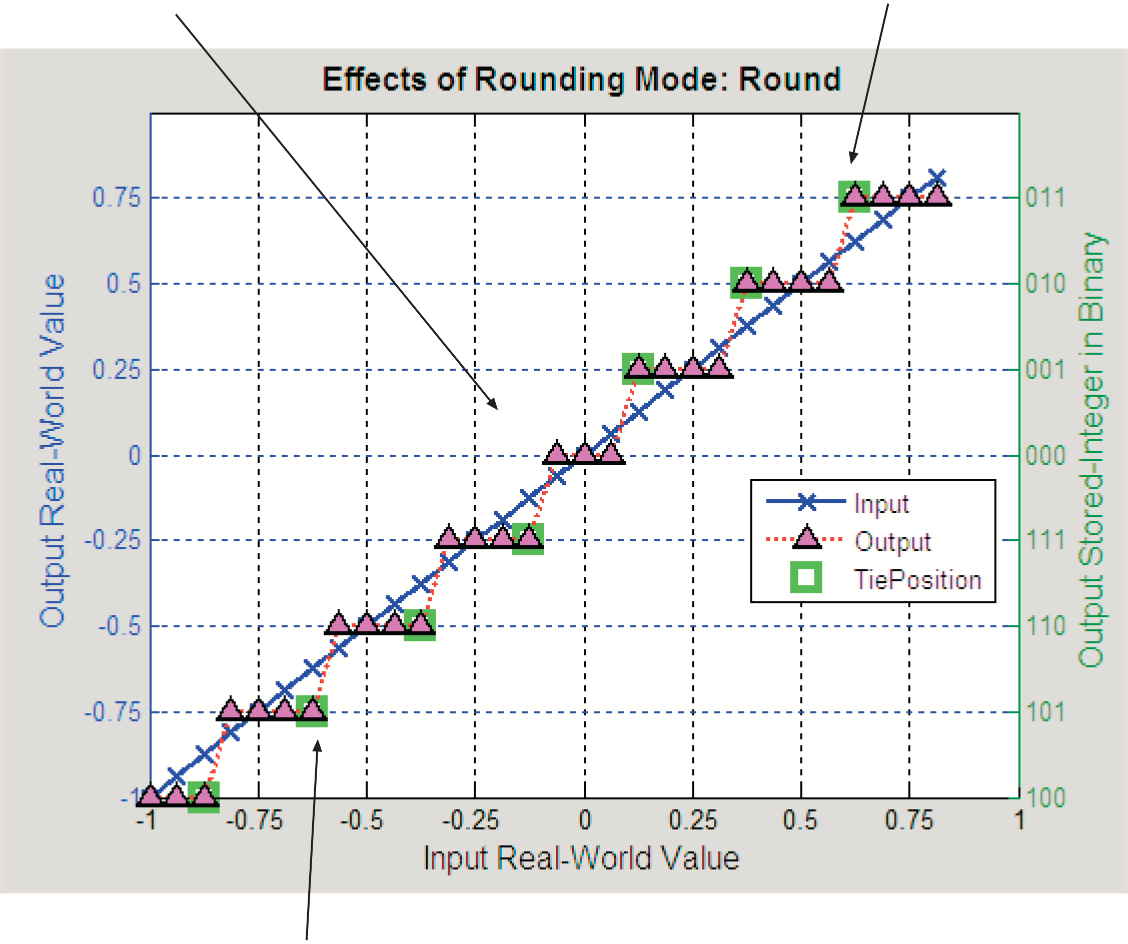
As a result:

- 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 28-14
- "Optimize Rounding for High-Level Arithmetic Operations" on page 28-15
- "Optimize Rounding for Intermediate Arithmetic Operations" on page 28-16

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.

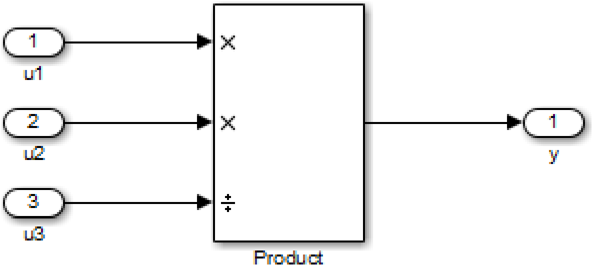
Optimize Rounding for Casts

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.

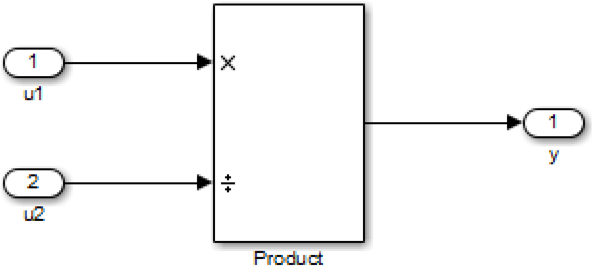
  To get the most efficient code, you must set the **Signed integer division rounds to** parameter to specify whether your production target rounds to zero or to floor for integer division. Most production targets round to zero for integer division operations. Note that `Simplest` rounding enables “mixed-mode” 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 “Fixed-Point Numbers” on page 27-3, 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:

$$y = \frac{u_1}{u_2} = \frac{S_1Q_1 + B_1}{S_2Q_2 + B_2}$$

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 28-15. 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.

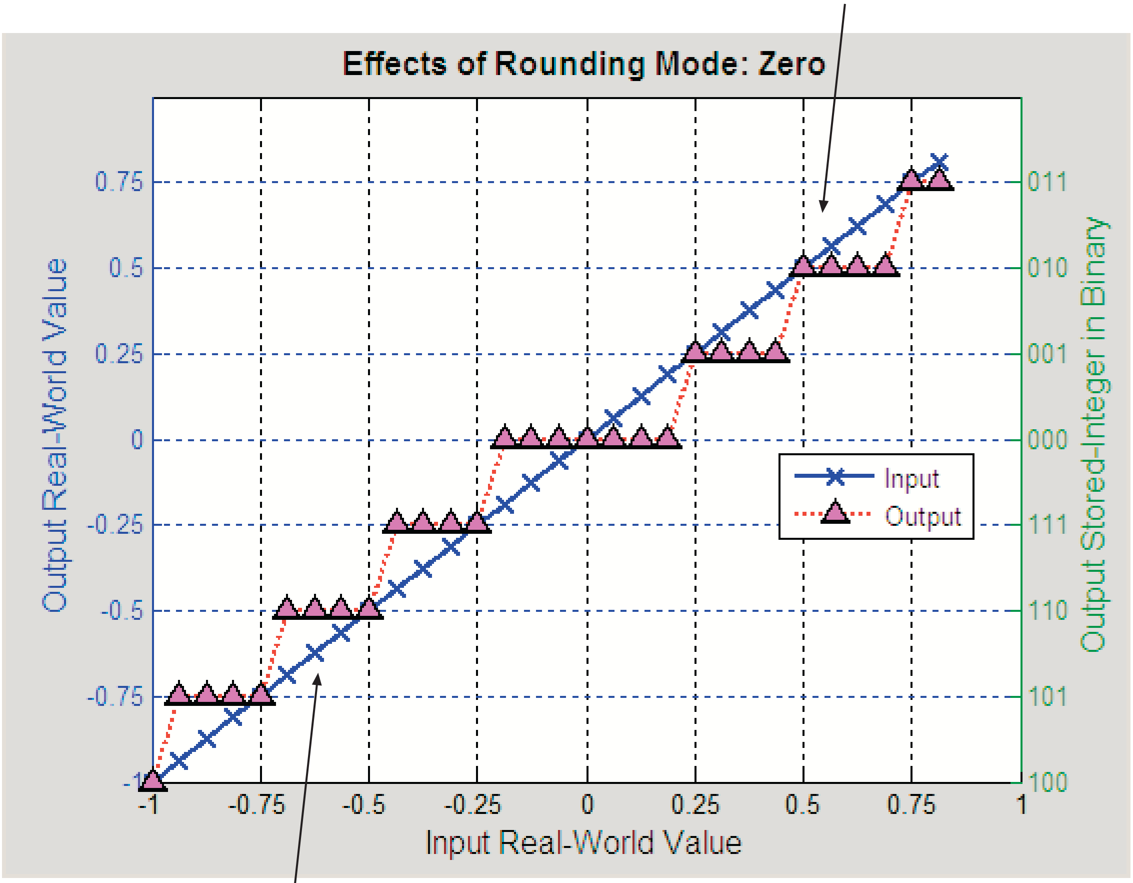
Note For signed fixed-point division where the denominator is an invariant constant power of 2, the simplest rounding mode does not generate the most efficient code. In this case, set the rounding mode to floor.

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.

Positive numbers are rounded
to smaller positive numbers



Negative numbers are rounded
to smaller negative numbers

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:

$$\begin{array}{r} 1.0000000 \times 2^{q} \\ \underline{-0.1111111 \times 2^{q}} \\ 0.0000001 \times 2^{q} \end{array}.$$

If the top number is padded by two zeros and the bottom number is padded with one zero, then the above equation becomes

$$\begin{array}{r} 1.000000000 \times 2^{q} \\ -0.111111110 \times 2^{q} \\ \hline 0.000000010 \times 2^{q} \end{array},$$

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 34-43.

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 27-6, 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:

$$\left|\tilde{V} - V\right| = 0.125 = \frac{S}{2} = \frac{F2^{E}}{2}.$$

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 27-6, these values are given by

$$\max(V) = F2^E\left(\max(Q)\right) + B$$
$$\min(V) = F2^E\left(\min(Q)\right) + B.$$

Solving for the slope, you get

$$F2^E = \frac{\max(V) - \min(V)}{\max(Q) - \min(Q)} = \frac{\max(V) - \min(V)}{2^{ws} - 1}.$$

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 28-24. 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 27-11.) 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 28-25.

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.

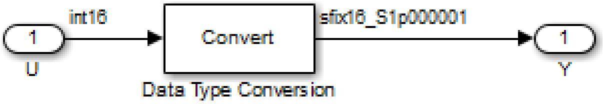
Detect Net Slope and Bias Precision Issues

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"	Diagnostic action when a fixed-point constant underflow occurs during simulation	Does not generate a warning or error.
"Detect overflow"	Diagnostic action when a fixed-point constant overflow occurs during simulation	Does not generate a warning or error.
"Detect precision loss"	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.

Range

In this section...

"Limitations on Range" on page 28-27

"What Are Saturation and Wrapping?" on page 28-28

"Saturation and Wrapping" on page 28-28

"Guard Bits" on page 28-31

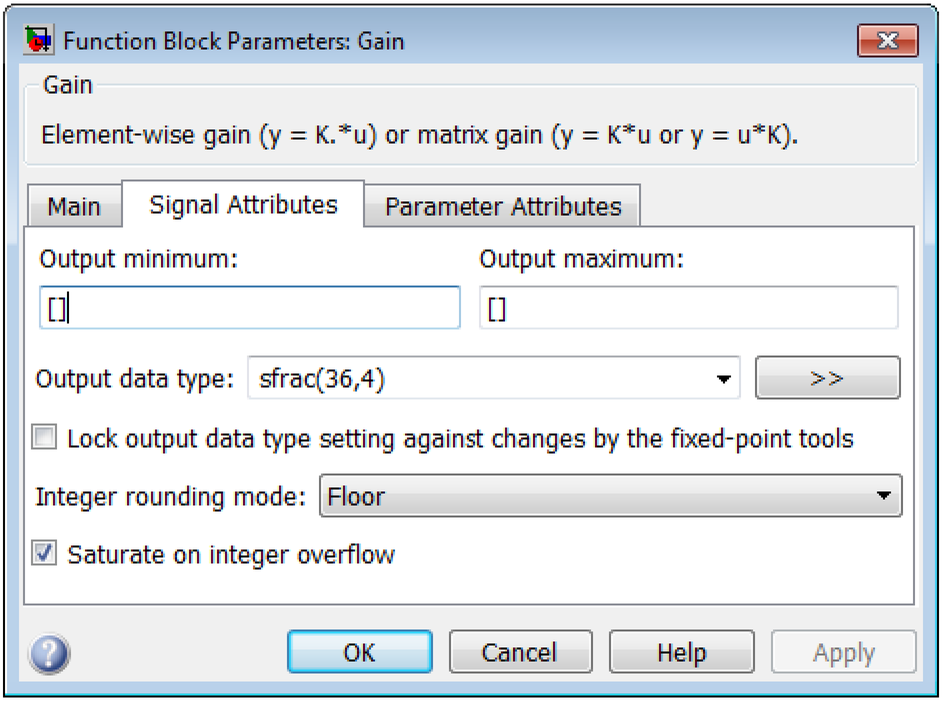
"Determine the Range of Fixed-Point Numbers" on page 28-31

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 27-10.

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).



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Output minimum: []

Output maximum: []

Output data type: sfrac(36,4)

□ Lock output data type setting against changes by the fixed-point tools

Integer rounding mode: Floor

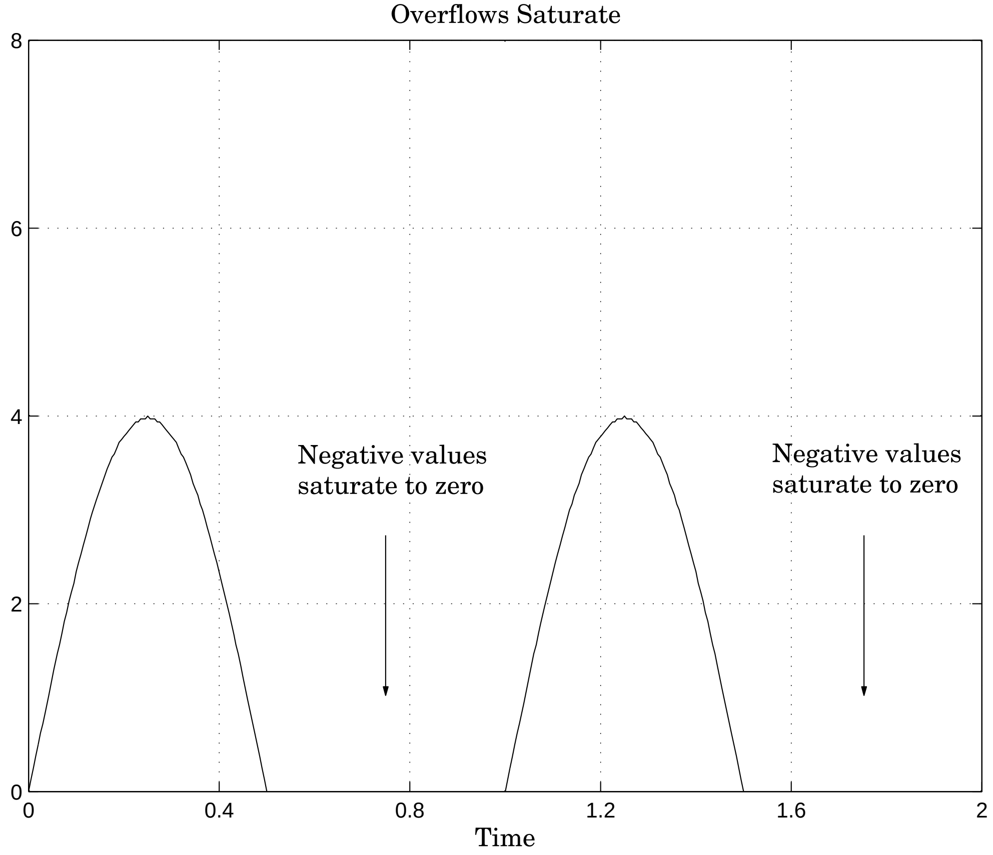
☑ Saturate on integer overflow

What Are Saturation and Wrapping?

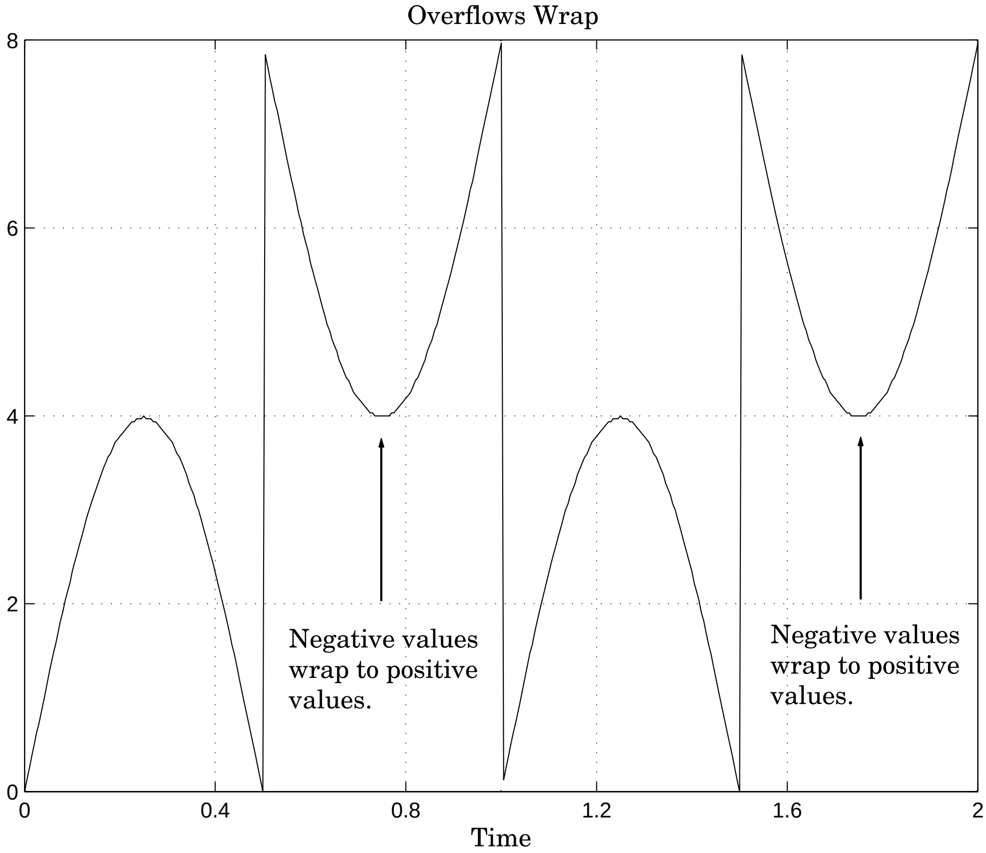
Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, the ADSP-2100 family of processors from Analog Devices™ supports either of these modes. If a register has a saturation mode of operation, then an overflow condition is set to the maximum 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 27-6, the approximate real-world value has the range

$$\max(\tilde{V}) - \min(\tilde{V}),$$

where

$$\min(\tilde{V}) = \begin{cases} \mathrm{B} & \text{unsigned,} \\ -F2^{E}\left(2^{ws-1}\right) + B & \text{signed,} \end{cases}$$

$$\max(\tilde{V}) = \begin{cases} F2^{E}\left(2^{ws} - 1\right) + \mathrm{B} & \text{unsigned,} \\ F2^{E}\left(2^{ws-1} - 1\right) + \mathrm{B} & \text{signed.} \end{cases}$$

If the real-world value exceeds the limited range of the approximate value, then the accuracy of the representation can become significantly worse.

Recommendations for Arithmetic and Scaling

In this section...

"Arithmetic Operations and Fixed-Point Scaling" on page 28-33

"Addition" on page 28-34

"Accumulation" on page 28-37

"Multiplication" on page 28-37

"Gain" on page 28-39

"Division" on page 28-41

"Summary" on page 28-43

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 27-6 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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

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 28-21 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

$$\begin{aligned}\min\left(\tilde{V}_a\right) &= \min\left(\tilde{V}_b\right) + \min\left(\tilde{V}_c\right),\\ \max\left(\tilde{V}_a\right) &= \max\left(\tilde{V}_b\right) + \max\left(\tilde{V}_c\right).\end{aligned}$$

You can now derive the maximum-precision slope:

$$\begin{aligned}F_a 2^{E_a} &= \frac{\max\left(\tilde{V}_a\right) - \min\left(\tilde{V}_a\right)}{2^{ws_a} - 1}\\ &= \frac{F_a 2^{E_b}\left(2^{ws_b} - 1\right) + F_c 2^{E_c}\left(2^{ws_c} - 1\right)}{2^{ws_a} - 1}.\end{aligned}$$

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

$$B_a = \min\left(\tilde{V}_a\right) - F_a 2^{E_a} \min\left(Q_a\right).$$

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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

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 28-21 shows that you can determine the range of a fixed-point operation from

$$\max\left(\tilde{V}_a\right)$$

and

$$\min\left(\tilde{V}_a\right).$$

For multiplication, you can determine the range from

$$\min\left(\tilde{V}_a\right) = \min\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$
$$\max\left(\tilde{V}_a\right) = \max\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$

where

$$V_{LL} = \min\left(\tilde{V}_b\right) \cdot \min\left(\tilde{V}_c\right),$$
$$V_{LH} = \min\left(\tilde{V}_b\right) \cdot \max\left(\tilde{V}_c\right),$$
$$V_{HL} = \max\left(\tilde{V}_b\right) \cdot \min\left(\tilde{V}_c\right),$$
$$V_{HH} = \max\left(\tilde{V}_b\right) \cdot \max\left(\tilde{V}_c\right).$$

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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i .$$

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

$$Q_a = \frac{Q_b}{Q_c} 2^{E_b - E_c - E_a} + \frac{\left(B_b / F_b\right)}{Q_c} 2^{-E_c - E_a}.$$

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 28-21 shows that you can determine the range of a fixed-point operation from

$$\max\left(\tilde{V}_a\right)$$

and

$$\min\left(\tilde{V}_a\right).$$

For division, you can determine the range from

$$\min\left(\tilde{V}_a\right) = \min\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$
$$\max\left(\tilde{V}_a\right) = \max\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$

where for nonzero denominators

$$V_{LL} = \min\left(\tilde{V}_b\right) / \min\left(\tilde{V}_c\right),$$
$$V_{LH} = \min\left(\tilde{V}_b\right) / \max\left(\tilde{V}_c\right),$$
$$V_{HL} = \max\left(\tilde{V}_b\right) / \min\left(\tilde{V}_c\right),$$
$$V_{HH} = \max\left(\tilde{V}_b\right) / \max\left(\tilde{V}_c\right).$$

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 28-44

"Parameter Conversions" on page 28-45

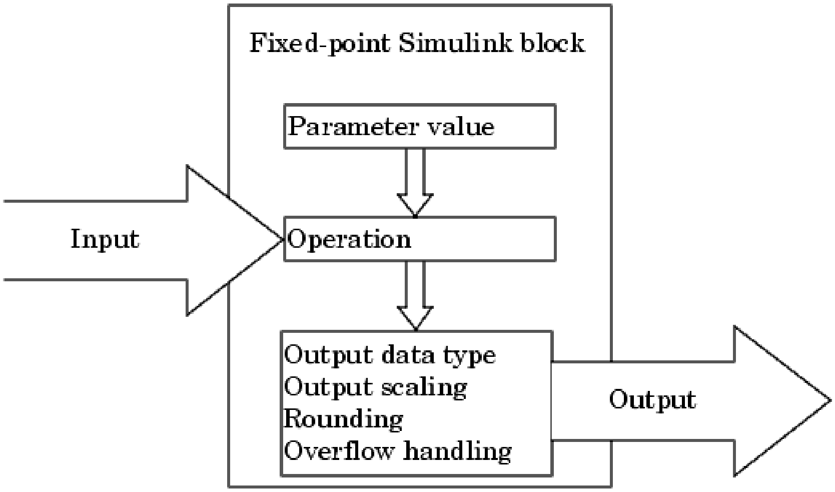
"Signal Conversions" on page 28-46

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 Designer data types
- When input signals are converted from one Fixed-Point Designer 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 28-49 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

$$y = \left(\frac{ab}{c}\right)u = Cu$$

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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

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:

   $$Q_{Temp} = convert\left(Q_{RawProduct}\right).$$

   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:

$$Q_a = convert\left(Q_b\right).$$

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

In this section...

"Introduction" on page 28-49

"Computational Units" on page 28-49

"Addition and Subtraction" on page 28-50

"The Summation Process" on page 28-52

"Multiplication" on page 28-55

"The Multiplication Process" on page 28-60

"Division" on page 28-62

"The Division Process" on page 28-63

"Shifts" on page 28-64

Introduction

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_b 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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

The equation in "Addition" on page 28-34 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:

   $$Q_{Temp} = convert\left(Q_{RawProduct}\right).$$

   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.

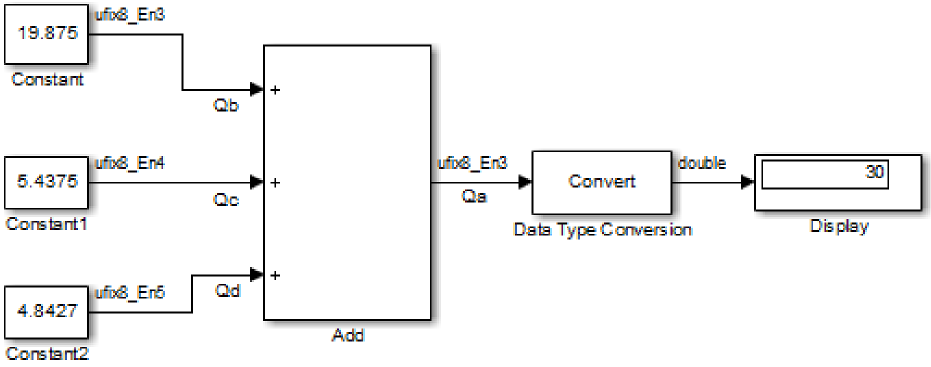
Streamlining Simulations and Generated Code

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.

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:

$$Q_c = 0101.0111,$$
$$Q_{Temp} = convert(Q_c)$$
$$Q_{Temp} = 00101.011.$$

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:

$$\begin{aligned} Q_a &= Q_a + Q_{Temp} \\ &= \frac{\begin{array}{r}10011.111\\ +00101.011\end{array}}{11001.010} = 25.250. \end{aligned}$$

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:

$$\begin{aligned} Q_d &= 100.11011, \\ Q_{Temp} &= convert\left(Q_d\right) \\ Q_{Temp} &= 00100.110. \end{aligned}$$

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:

$$\begin{aligned} Q_a &= Q_a + Q_{Temp} \\ &= \frac{\begin{array}{r} 11001.010 \\ +00100.110 \end{array}}{11110.000} = 30.000. \end{aligned}$$

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:

$$= \frac{\begin{array}{r} 10011.111 \\ 0101.0111 \\ +100.11011 \end{array}}{11110.001} = 30.125.$$

Blocks that perform addition and subtraction include the Sum, Gain, and Discrete FIR Filter blocks.

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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

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

$$\begin{aligned}
V_{1Temp} &= 2^{E_{1Temp}} Q_{1Temp}, \\
V_{2Temp} &= 2^{E_{2Temp}} Q_{2Temp}, \\
V_{3Temp} &= 2^{E_{3Temp}} Q_{3Temp}.
\end{aligned}$$

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:

  $$V_{Max} = F 2^{E} Q_{MaxInteger} + B,$$

$$V_{Min} = F2^{E} Q_{MinInteger} + B.$$

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:

$$V_{Original} = Q_{Original} + \text{-}43.25,$$

where $Q_{Original}$ is an 8-bit, unsigned data type. For this data type,

$$\begin{aligned} Q_{MaxInteger} &= 225, \\ Q_{MinInteger} &= 0, \end{aligned}$$

so

$$\begin{aligned} V_{Max} &= 211.75, \\ V_{Min} &= -43.25. \end{aligned}$$

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

$$\text{uint8\_T} \quad Q_{Original},$$

$$\text{uint16\_T} \quad Q_{Temp},$$

$$Q_{Temp} = \left((\text{uint16\_T}) Q_{Original} \text{ □ } 2 \right) - 173.$$

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:

$$Q_{Temp} = convert\left(Q_{RawProduct}\right).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. "Signal Conversions" on page 28-46 discusses conversions.

3 The multiplication

$$Q_{2RawProduct} = Q_{Temp} Q_{Net}$$

is performed.

4 The previous product is converted to the output data type:

$$Q_a = convert\left(Q_{2RawProduct}\right).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. "Signal Conversions" on page 28-46 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:

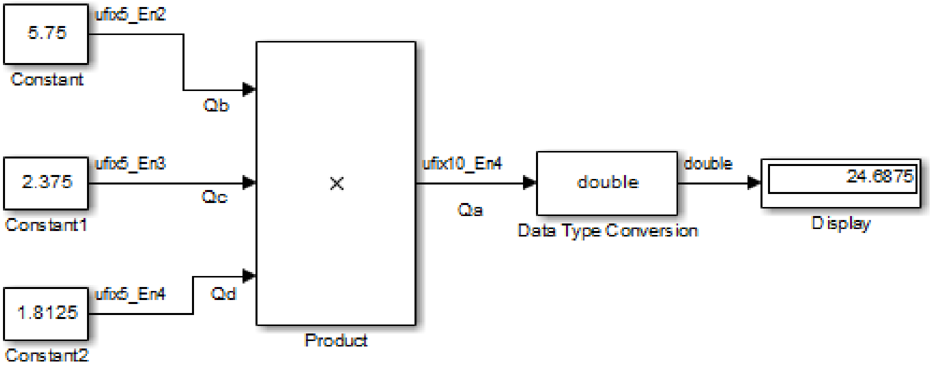
$$Q_a = convert(Q_{RawProduct}).$$

   This conversion includes any necessary bit shifting, rounding, or overflow handling. “Signal Conversions” on page 28-46 discusses conversions.

3. Steps 1 and 2 are repeated for each additional number to be multiplied.

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:

$$\begin{array}{rr} Q_{RawProduct} = & 101.11 \\ & \times 10.011 \\ \hline & 101.11 \cdot 2^{-3} \\ & 101.11 \cdot 2^{-2} \\ & +101.11 \cdot 2^{1} \\ \hline & 01101.10101 = 13.65625. \end{array}$$

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:

$$\begin{aligned} Q_{Temp} &= convert\left(Q_{RawProduct}\right) \\ &= 001101.1010 = 13.6250. \end{aligned}$$

"Signal Conversions" on page 28-46 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:

$$\begin{array}{rr} Q_{RawProduct} = & 01101.1010 \\ & \times 1.1101 \\ \hline & 1101.1010 \cdot 2^{-4} \\ & 1101.1010 \cdot 2^{-2} \\ & 1101.1010 \cdot 2^{-1} \\ & +1101.1010 \cdot 2^{0} \\ \hline & 0011000.10110010 = 24.6953125. \end{array}$$

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:

$$\begin{aligned} Q_a &= convert\left(Q_{RawProduct}\right) \\ &= 011000.1011 = 24.6875. \end{aligned}$$

"Signal Conversions" on page 28-46 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.

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 28-55.

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 27-6:

$$V_i = F_i 2^{E_i} Q_i + B_i .$$

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:

$$Q_a = 2^{E_b - E_c - E_a} \left(Q_b / Q_c \right).$$

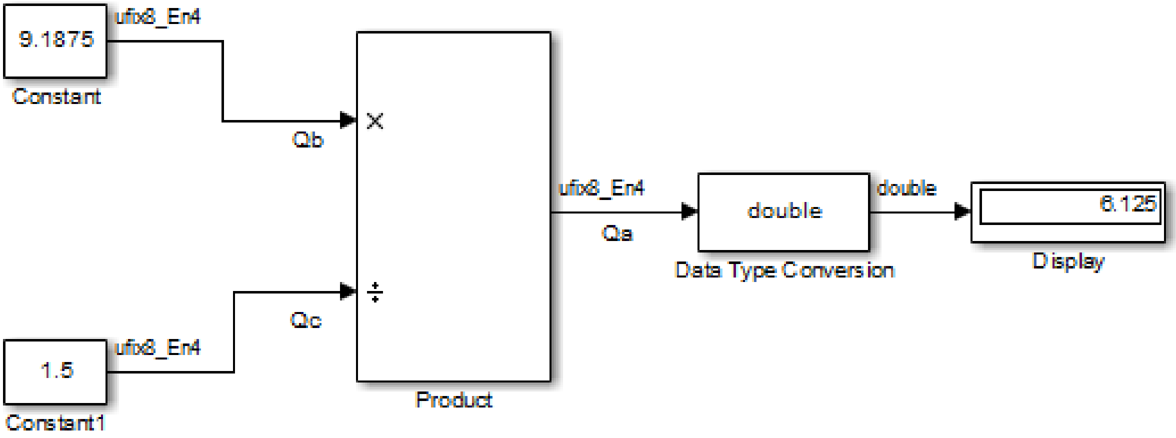
This equation involves an integer division and some bit shifts. If $E_a > E_b – E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b – E_c$, then the bit shifts are to the left and the implementation can be more complicated. The essential issue is that the 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.

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,

$$
\begin{aligned}
Q_a &= 2^{-4-(-4)-(-4)}\left(Q_b / Q_c\right) \\
&= 2^{4}\left(Q_b / Q_c\right).
\end{aligned}
$$

Assuming a large data type was available, this could be implemented as

$$
Q_a = \frac{\left(2^4 Q_b\right)}{Q_c},
$$

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.

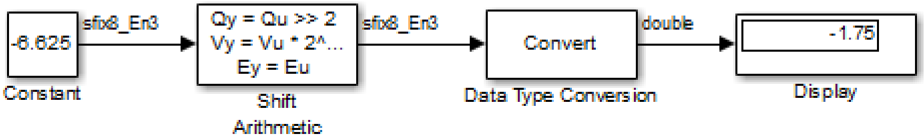
Shift Operation	Binary Value	Decimal Value
No shift (original number)	00110101	53
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

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

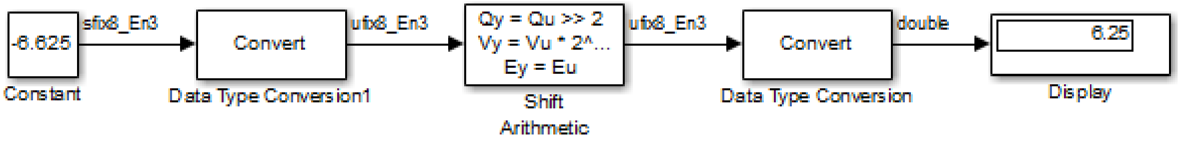
Shifting Bits to the Right

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).


-6.625
Constant
sfix8_En3
Convert
Data Type Conversion1
ufix8_En3
Qy = Qu >> 2
Vy = Vu * 2^...
Ey = Eu
Shift
Arithmetic
ufix8_En3
Convert
Data Type Conversion
double
6.25
Display

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 28-44 and “Rules for Arithmetic Operations” on page 28-49.

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

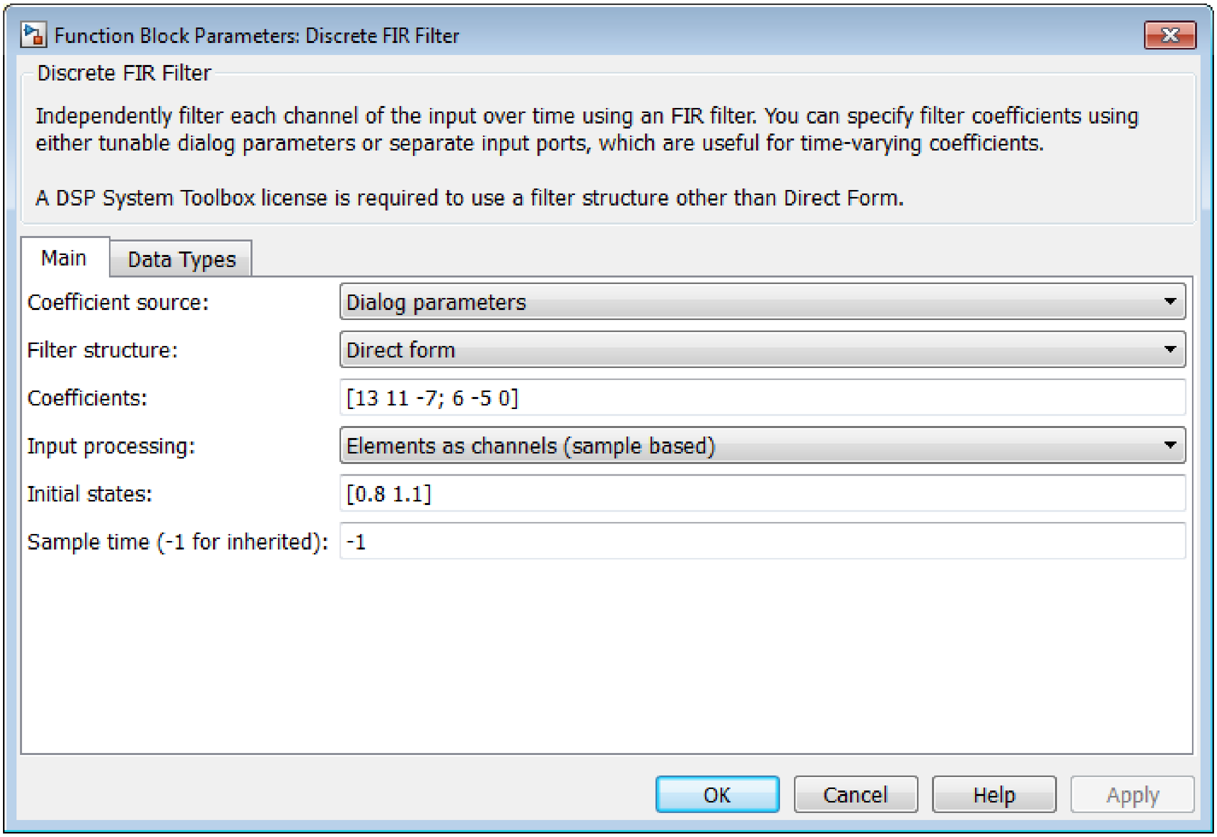
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2),$$

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.



Function Block Parameters: Discrete FIR Filter

Discrete FIR Filter

Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.

A DSP System Toolbox license is required to use a filter structure other than Direct Form.

Main | Data Types

Coefficient source: Dialog parameters

Filter structure: Direct form

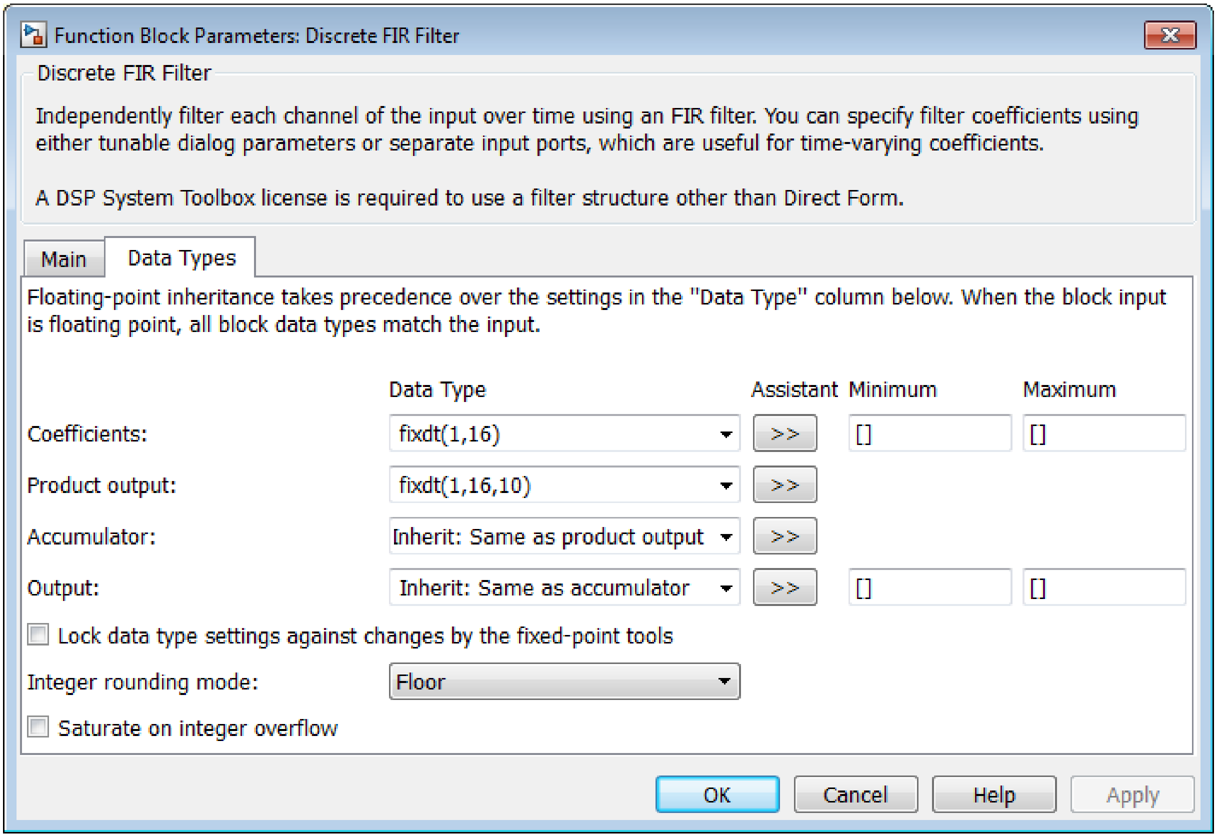
Coefficients: [13 11 -7; 6 -5 0]

Input processing: Elements as channels (sample based)

Initial states: [0.8 1.1]

Sample time (-1 for inherited): -1

Similarly, configure the options on the **Data Types** pane of the block dialog box to appear as follows:



Discrete FIR Filter

Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.

A DSP System Toolbox license is required to use a filter structure other than Direct Form.

Floating-point inheritance takes precedence over the settings in the "Data Type" column below. When the block input is floating point, all block data types match the input.

	Data Type	Minimum	Maximum
Coefficients:	fixdt(1,16)	[]	[]
Product output:	fixdt(1,16,10)		
Accumulator:	Inherit: Same as product output		
Output:	Inherit: Same as accumulator	[]	[]

□ Lock data type settings against changes by the fixed-point tools

Integer rounding mode: Floor

□ Saturate on integer overflow

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 29-2
- "Targeting an Embedded Processor" on page 29-4
- "Canonical Forms" on page 29-7

Realizing Fixed-Point Digital Filters

In this section...

"Introduction" on page 29-2

"Realizations and Data Types" on page 29-2

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 29-4.

Targeting an Embedded Processor

In this section...

"Introduction" on page 29-4

"Size Assumptions" on page 29-4

"Operation Assumptions" on page 29-4

"Design Rules" on page 29-5

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

In this section...

"Canonical Forms" on page 29-7

"Direct Form II" on page 29-8

"Series Cascade Form" on page 29-13

"Parallel Form" on page 29-15

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 in Simulink” on page 26-17, 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:

$$\begin{aligned} H_{ex}(z) &= \frac{1+2.2z^{-1}+1.85z^{-2}+0.5z^{-3}}{1-0.5z^{-1}+0.84z^{-2}+0.09z^{-3}} \\ &= \frac{\left(1+0.5z^{-1}\right)\left(1+1.7z^{-1}+z^{-2}\right)}{\left(1+0.1z^{-1}\right)\left(1-0.6z^{-1}+0.9z^{-2}\right)} \\ &= 5.5556-\frac{3.4639}{1+0.1z^{-1}}+\frac{-1.0916+3.0086z^{-1}}{1-0.6z^{-1}+0.9z^{-2}}. \end{aligned}$$

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.

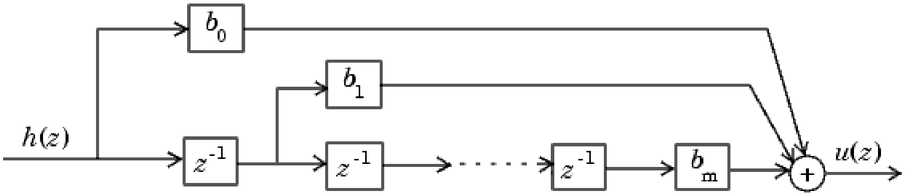
The canonical direct form II is presented as “Standard Programming” in *Discrete-Time Control Systems* by Ogata. It is known as the “Control

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

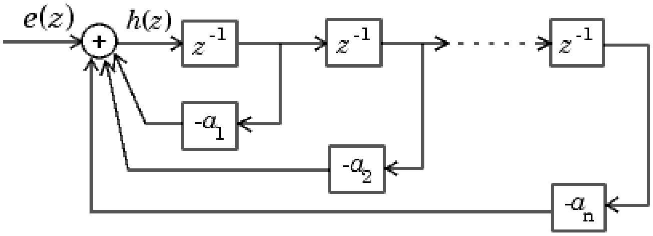
$$\begin{aligned}\frac{u(z)}{e(z)} &= \frac{u(z)}{h(z)} \cdot \frac{h(z)}{e(z)} \\ &= \underbrace{\left(b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}\right)}_{\frac{u(z)}{h(z)}} \underbrace{\frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} \ldots + a_n z^{-n}}}_{\frac{h(z)}{e(z)}}.\end{aligned}$$

The block diagram for *u*(*z*)/*h*(*z*) follows.



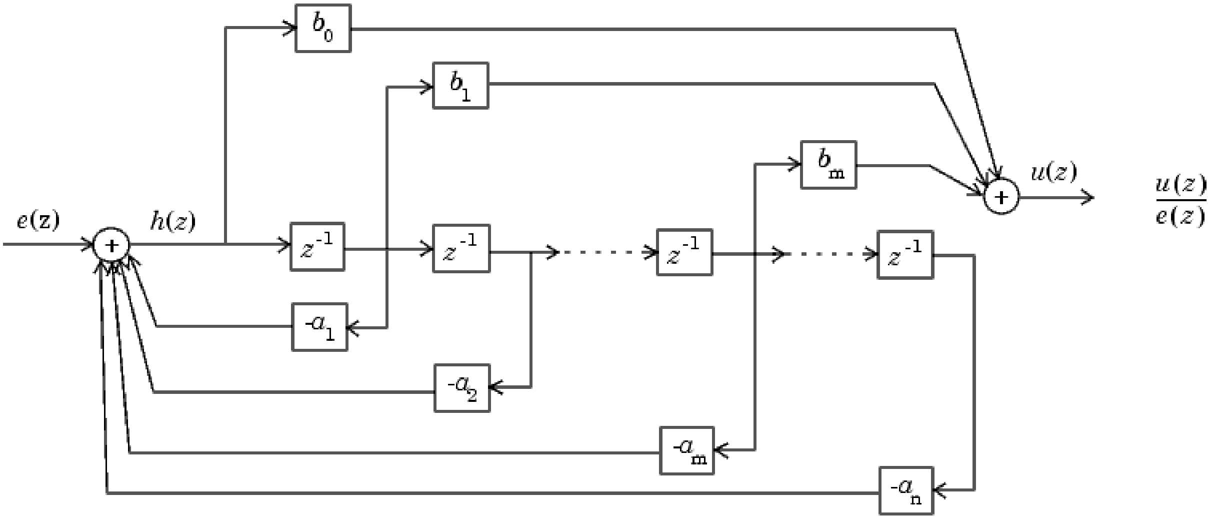
$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}$$

The block diagrams for *h*(*z*)/*e*(*z*) follow.



$$\frac{h(z)}{e(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}$$

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.



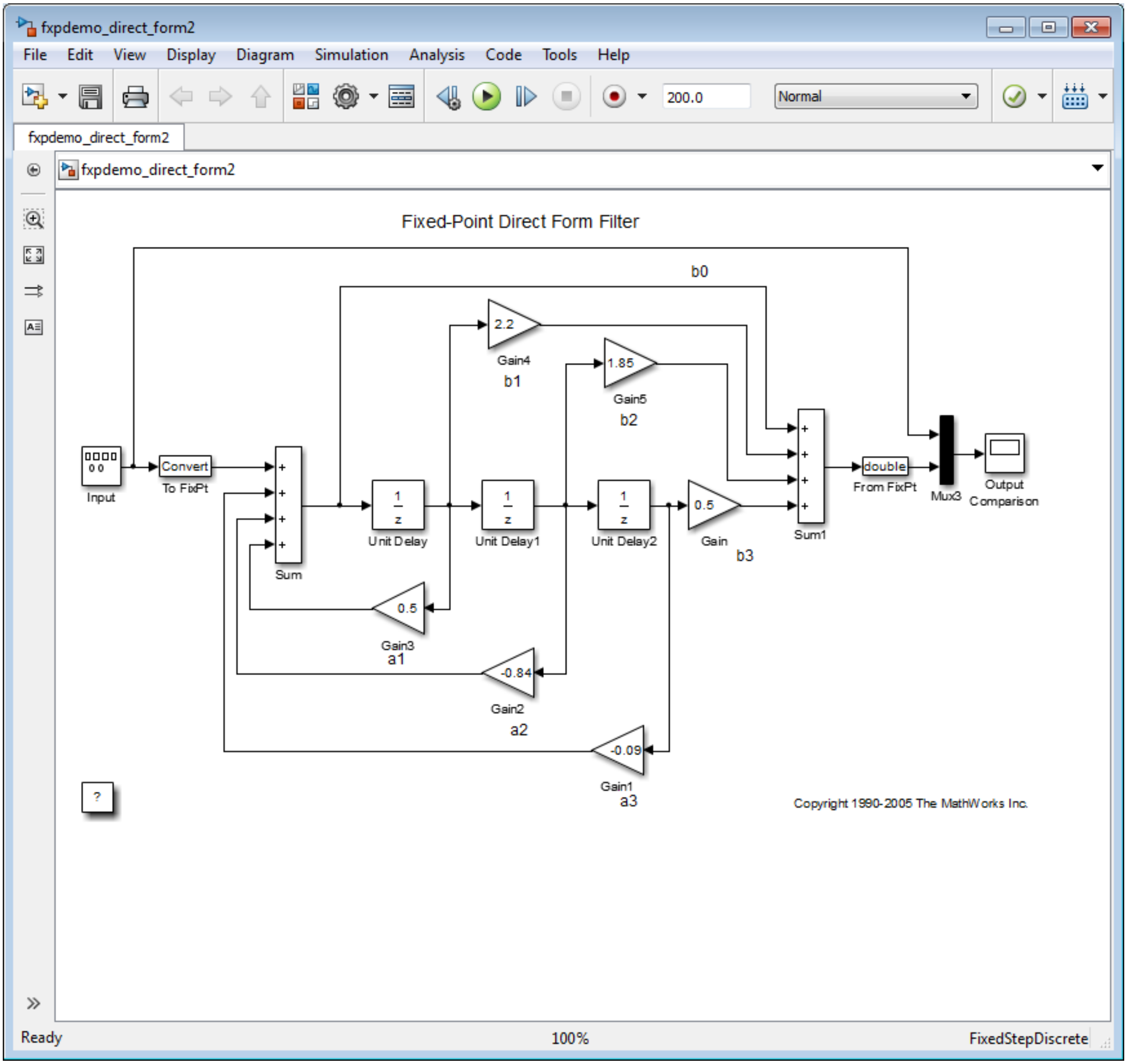
The direct form II example transfer function is given by

$$H_{ex}(z) = \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}}.$$

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.

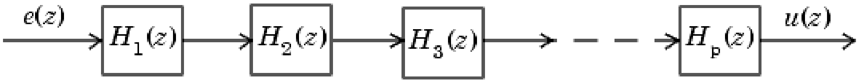

fxpdemo_direct_form2
File Edit View Display Diagram Simulation Analysis Code Tools Help
200.0
Normal
fxpdemo_direct_form2
fxpdemo_direct_form2
Fixed-Point Direct Form Filter
b0
2.2
Gain4
b1
1.85
Gain5
b2
Input
Convert
To FixPt
Sum
1
z
Unit Delay
1
z
Unit Delay1
1
z
Unit Delay2
0.5
Gain
b3
Sum1
double
From FixPt
Mux3
Output
Comparison
0.5
Gain3
a1
-0.84
Gain2
a2
-0.09
Gain1
a3
?
Copyright 1990-2005 The MathWorks Inc.
Ready
100%
FixedStepDiscrete

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:

$$H_i(z) = \frac{u(z)}{e(z)} = H_1(z) \cdot H_2(z) \cdot H_3(z) \ldots H_p(z).$$

This equation yields the canonical series cascade form.



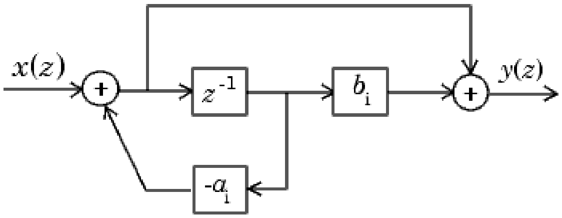
Factoring *H*(*z*) into $H_i(z)$ where *i* = 1,2,3,...,p can be done in a number of ways. Using the poles and zeros of *H*(*z*), you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles 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

$$\begin{aligned} H(z) &= H_1(z) H_2(z) \ldots H_p(z) \\ &= \prod_{i=1}^{j} \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^{p} \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}. \end{aligned}$$

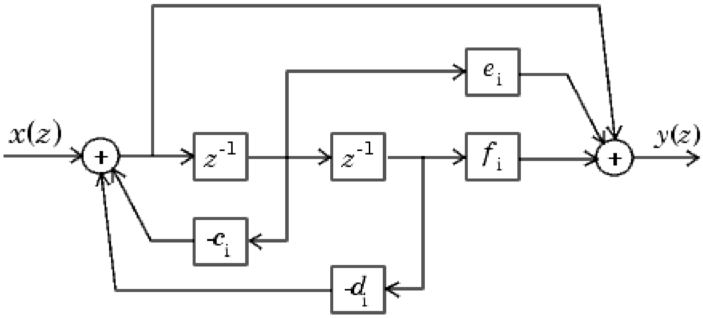
You must also take into consideration that the ordering of the individual $H_i(z)$'s will lead to systems with different numerical characteristics. You might want to try various orderings for a given set of $H_i(z)$'s to determine which gives the best numerical characteristics.

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.



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

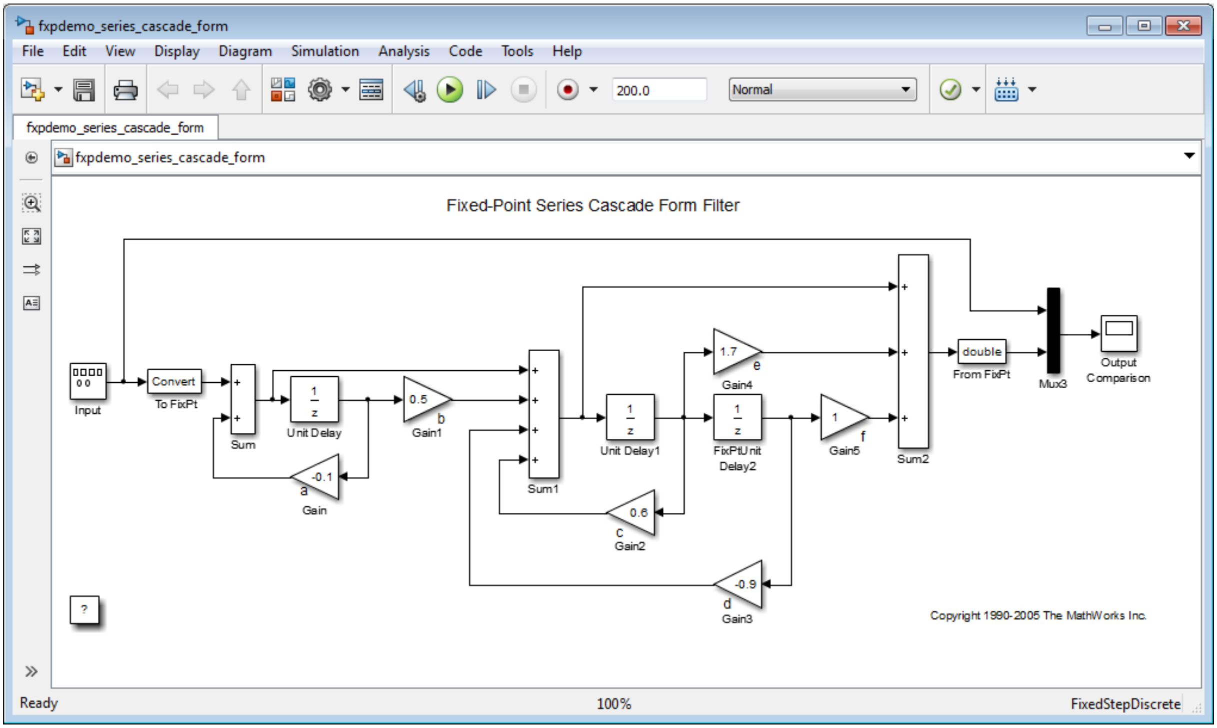
The series cascade form example transfer function is given by

$$H_{ex}(z) = \frac{\left(1+0.5z^{-1}\right)\left(1+1.7z^{-1}+z^{-2}\right)}{\left(1+0.1z^{-1}\right)\left(1-0.6z^{-1}+0.9z^{-2}\right)}.$$

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.

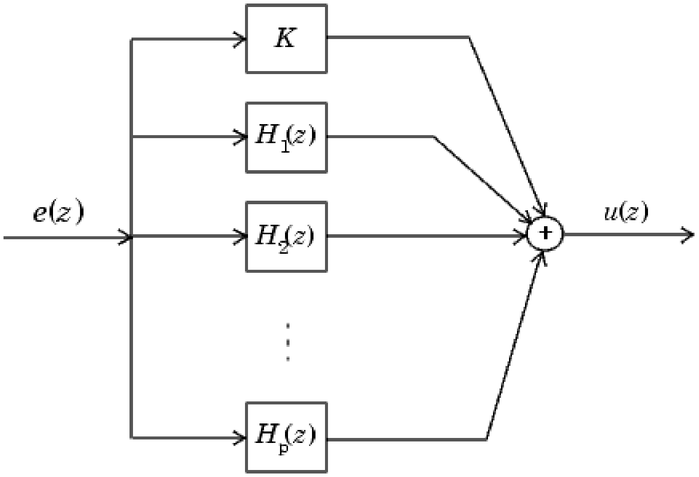

fxpdemo_series_cascade_form
File Edit View Display Diagram Simulation Analysis Code Tools Help
200.0
Normal
fxpdemo_series_cascade_form
fxpdemo_series_cascade_form
Fixed-Point Series Cascade Form Filter
Input
Convert
To FixPt
Sum
1/z
Unit Delay
-0.1
a
Gain
0.5
b
Gain1
Sum1
1/z
Unit Delay1
0.6
c
Gain2
1.7
e
Gain4
1/z
FixPtUnit Delay2
-0.9
d
Gain3
1
f
Gain5
Sum2
double
From FixPt
Mux3
Output Comparison
?
Copyright 1990-2005 The MathWorks Inc.
Ready
100%
FixedStepDiscrete

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:

$$H_i(z) = \frac{u(z)}{e(z)} = K + H_1(z) + H_2(z) + \ldots + H_p(z).$$

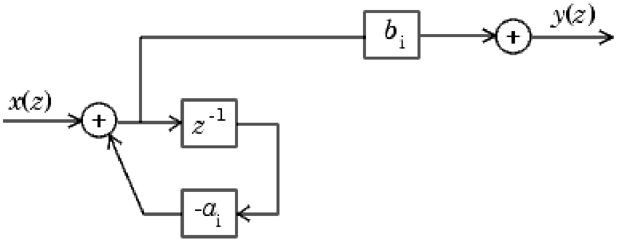
This expansion, where K is a constant and the $H_i(z)$ are the first- and second-order transfer functions, follows.



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

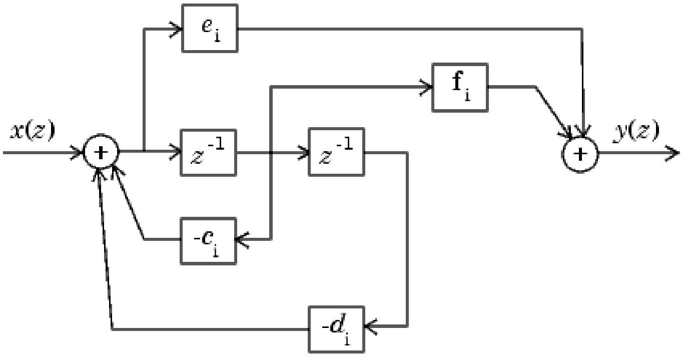
$$\begin{aligned} H(z) &= K + \sum_{i=1}^{j} H_i(z) + \sum_{i=j+1}^{p} H_i(z) \\ &= K + \sum_{i=1}^{j} \frac{b_i}{1 + a_i z^{-1}} + \sum_{i=j+1}^{p} \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}. \end{aligned}$$

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.



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

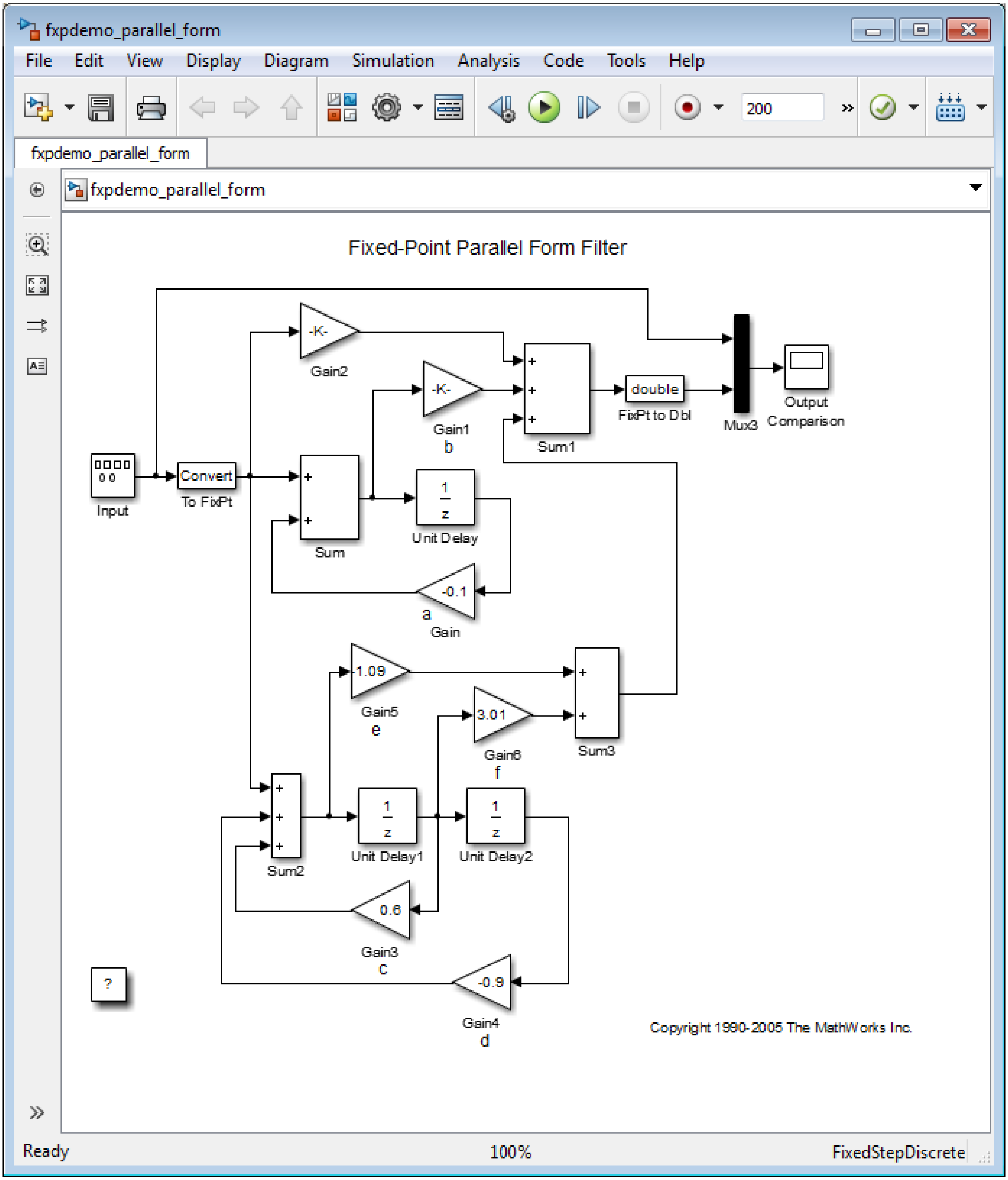
The parallel form example transfer function is given by

$$H_{ex}(z) = 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}.$$

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.


fxpdemo_parallel_form
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
200
fxpdemo_parallel_form
fxpdemo_parallel_form
Fixed-Point Parallel Form Filter
-K-
Gain2
-K-
Gain1
b
Sum1
double
FixPt to Dbl
Mux3
Output Comparison
Input
Convert
To FixPt
Sum
1
z
Unit Delay
-0.1
a
Gain
1.09
Gain5
e
3.01
Gain6
f
Sum3
Sum2
1
z
Unit Delay1
1
z
Unit Delay2
0.6
Gain3
c
-0.9
Gain4
d
?
Copyright 1990-2005 The MathWorks Inc.
Ready
100%
FixedStepDiscrete

Fixed-Point Advisor

- "Preparation for Fixed-Point Conversion" on page 30-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 30-11

Preparation for Fixed-Point Conversion

In this section...

"Introduction" on page 30-2

"Best Practices" on page 30-2

"Run the Fixed-Point Advisor" on page 30-4

"Fix a Task Failure" on page 30-5

"Manually Fixing Failures" on page 30-6

"Automatically Fixing Failures" on page 30-6

"Batch Fixing Failures" on page 30-7

"Restore Points" on page 30-7

"Save a Restore Point" on page 30-8

"Load a Restore Point" on page 30-9

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” and “Stop time” 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 30-8.

Run the Fixed-Point Advisor

1. Open a model.

2. Start the Fixed-Point Advisor by:

- Typing `fpcadvisor('*model_name/subsystem_name*')` at the MATLAB command line
- Selecting a subsystem and, from the menu, selecting **Analysis > Fixed-Point Tool** to open the Fixed-Point Tool. On the Fixed-Point Tool **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.
- Right-clicking a subsystem block and, from the subsystem context menu, selecting **Fixed-Point Tool** to open the Fixed-Point Tool. On the Fixed-Point Tool **Fixed-point preparation for selected system** pane, 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 to Failure** from the Run menu.
   - Right-clicking the Fixed-Point Advisor folder and selecting **Run to Failure** 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 30-5.

   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 30-6.
- Use the Action box. Use this method to automatically fix all failures. See "Automatically Fixing Failures" on page 30-6.
- Use the Model Advisor Results Explorer. Use this method to batch fix failures. See "Batch Fixing Failures" on page 30-7

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 an **Analysis Result** box 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 30-8.
2. In the **Analysis Result** box, review the recommended actions. Use the information to make changes to your model.
3. To verify that the task now passes, in the **Analysis** box, 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 **Analysis 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 30-8.

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 30-8.

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 30-8.

To learn how to load a restore point, see “Load a Restore Point” on page 30-9.

Save a Restore Point

When to Save a Restore Point

Consider saving a restore point:

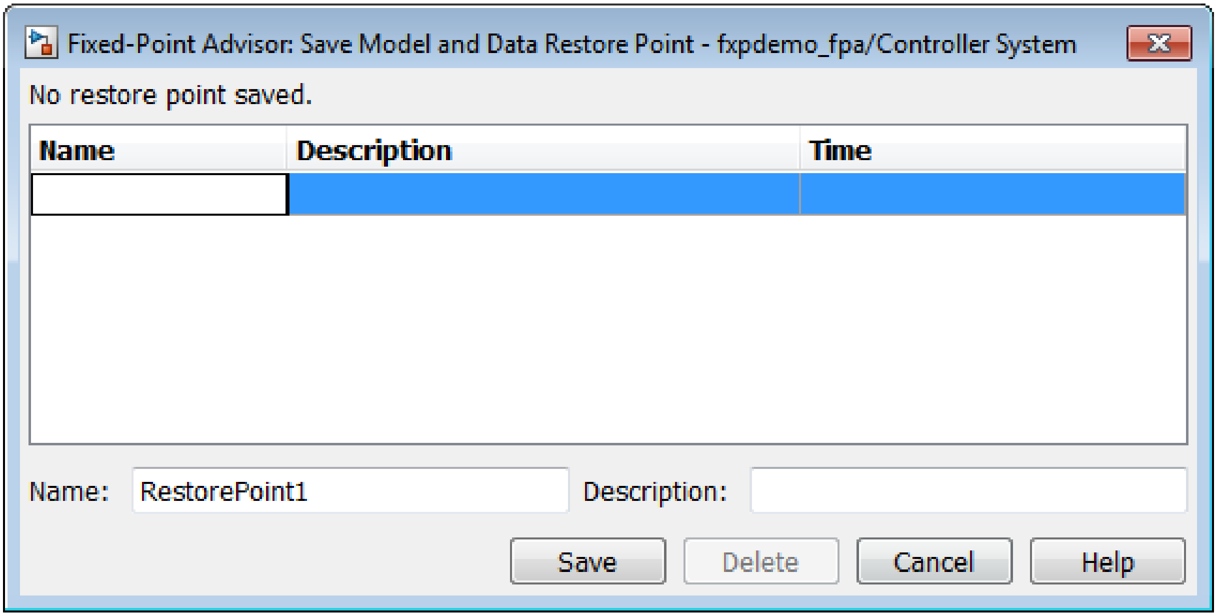
- Before applying changes to your model that might cause update diagram failure. For example, before applying proposed data types in task 3.1.
- Before attempting to fix failures.

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**.



Fixed-Point Advisor: Save Model and Data Restore Point - fxpdemo_fpa/Controller System

No restore point saved.

Name	Description	Time

Name: RestorePoint1

Description:

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` 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

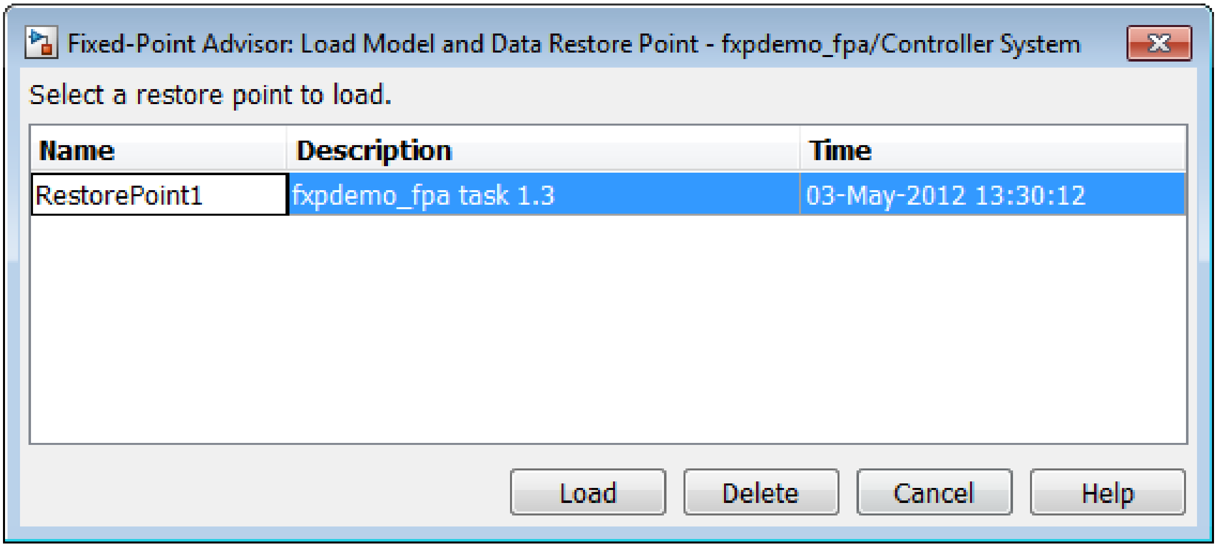
Load a restore point when:

- 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**.



Fixed-Point Advisor: Load Model and Data Restore Point - fxpdemo_fpa/Controller System

Select a restore point to load.

Name	Description	Time
RestorePoint1	fxpdemo_fpa task 1.3	03-May-2012 13:30:12

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.

Converting a Model from Floating- to Fixed-Point Using Simulation Data

In this section...

"About This Example" on page 30-11

"Starting the Preparation" on page 30-11

"Preparing Model for Conversion" on page 30-12

"Prepare for Data Typing and Scaling" on page 30-17

"Propose Data Types Based on the Simulation Reference Run" on page 30-20

"Apply the New Fixed-Point Data Types" on page 30-20

"Simulate the Model Using New Fixed-Point Settings" on page 30-21

About This Example

This example steps you through using the Fixed-Point Advisor to prepare the `fxpdemo_fpa` model for conversion from using floating-point data types to using fixed-point data types. This example shows you how to:

- Set model-wide configuration options.
- Set block-specific parameters.
- Obtain an initial fixed-point data types for the model.
- Validate the fixed-point data types against the floating-point model.

Starting the Preparation

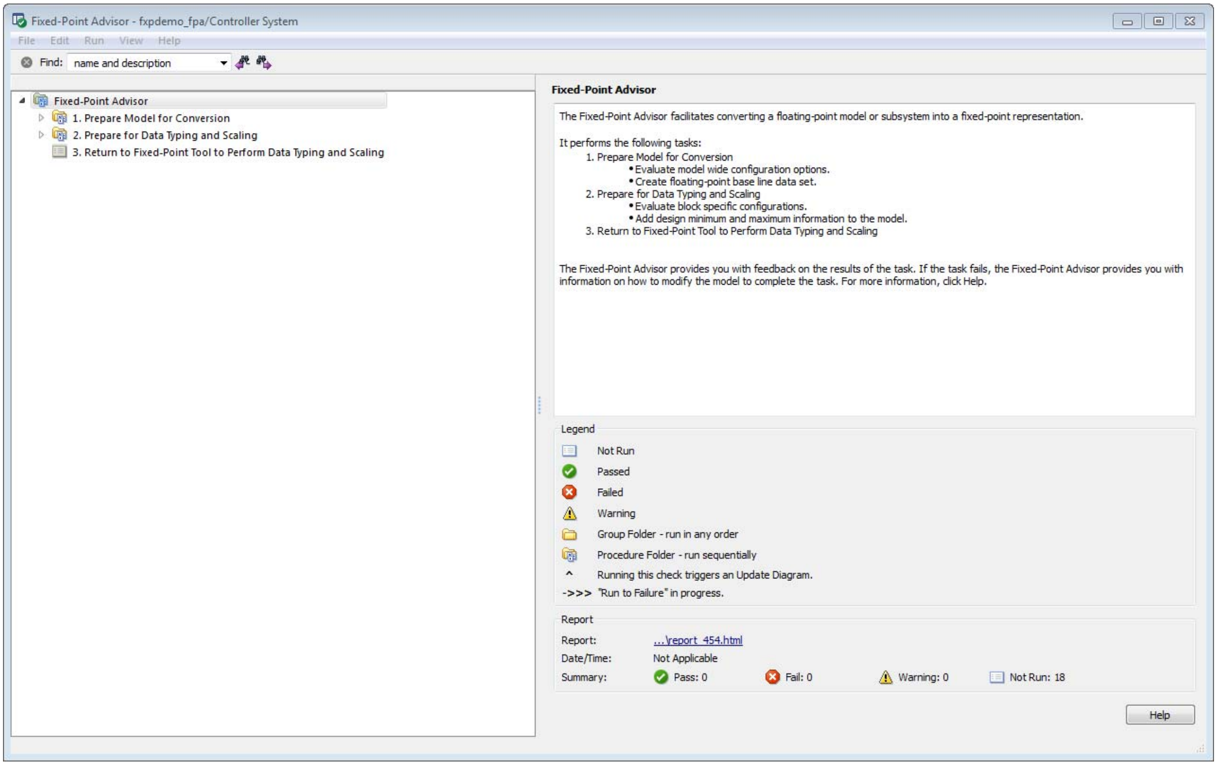
1 Open the model. At the command line, enter: `fxpdemo_fpa`.

2 To start the conversion:

a Right-click **Controller System** and, from the subsystem context menu, select **Fixed-Point Tool**.

b On the Fixed-Point Tool **Fixed-point preparation for selected system** pane, click the **Fixed-Point Advisor** button.

The Fixed-Point Advisor opens for the subsystem `Controller System`.


Fixed-Point Advisor - fxpdemo_fpa/Controller System
File Edit Run View Help
Find: name and description
Fixed-Point Advisor
1. Prepare Model for Conversion
2. Prepare for Data Typing and Scaling
3. Return to Fixed-Point Tool to Perform Data Typing and Scaling
Help

Fixed-Point Advisor

The Fixed-Point Advisor facilitates converting a floating-point model or subsystem into a fixed-point representation.

It performs the following tasks:
1. Prepare Model for Conversion
   - Evaluate model wide configuration options.
   - Create floating-point base line data set.
2. Prepare for Data Typing and Scaling
   - Evaluate block specific configurations.
   - Add design minimum and maximum information to the model.
3. Return to Fixed-Point Tool to Perform Data Typing and Scaling

The Fixed-Point Advisor provides you with feedback on the results of the task. If the task fails, the Fixed-Point Advisor provides you with information on how to modify the model to complete the task. For more information, click Help.

Legend

☐	Not Run
✓	Passed
✗	Failed
⚠	Warning
🗀	Group Folder - run in any order
🗀	Procedure Folder - run sequentially
^	Running this check triggers an Update Diagram.
->>>	"Run to Failure" in progress.

Report

Report: ...\report_454.html

Date/Time: Not Applicable

Summary: ✓ Pass: 0 ✗ Fail: 0 ⚠ Warning: 0 ☐ Not Run: 18

Preparing Model for Conversion

First, validate model-wide settings and create reference simulation data.

1 For the purpose of this tutorial, run the tasks in the Fixed-Point Advisor **Prepare Model for Conversion** folder one at a time. In the left pane, select **Verify model simulation settings** and, in the right pane, click **Run This Task**.

This task validates that model simulation settings allow signal logging and disable data type override to facilitate conversion to fixed point. These settings ensure that fixed-point data can be logged in downstream tasks.

The task passes.

2 Select and run **Verify update diagram status**.

Your model must be able to successfully complete an update diagram action 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 Fixed-Point Advisor cannot convert these blocks. To complete the conversion of your model, replace these blocks with Simulink built-in blocks that do support fixed-point data types. If a replacement block is not available, you can temporarily isolate the unsupported block with Data Type Conversion blocks.

The task fails because the model contains a block that does not support fixed-point data types.

4 Fix the failure by replacing the TrigFcn block with the provided replacement:

a Click the **Preview** link to view the replacement block.

b Click the link to the original block and view its settings.

c Double-click the replacement block and verify its settings match the settings of the original block.

Note 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.

- **d** In the Controller System subsystem, right-click the original TrigFcn block. From the context menu, select `Replace with Lookup Table`.

  The Fixed-Point Advisor replaces the original block.

- **e** In the Fixed-Point Advisor, rerun the task. The task passes.

5 Select and run **Set up signal logging**. Because you are using simulation minimum and maximum data, you must specify at least one signal to use in analysis and comparison in downstream checks. At a minimum, you should log the unique input and output signals.

The task runs and the Fixed-Point Advisor warns that signal logging is not specified for any signals.

6 Because you want to propose data types based on simulation data, fix the warning:

- **a** Click the **Explore Result** button.
- **b** In the Model Advisor Result Explorer, select the signals that you want to log and select the **EnableLogging** check box.

  For this tutorial, log the signals connected to the Inport and Outport blocks:

  - `Ctr_in`
  - `Ctr_out`

- **c** Close the Model Advisor Result Explorer.
- **d** In the Fixed-Point Advisor, rerun the task.

  The task passes because signal logging is enabled for at least one signal.

7 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 to use for analysis and comparison in later conversion tasks.

The task runs and the Fixed-Point Advisor warns that logging is not enabled.

If the simulation is set up to have a long simulation time, after starting 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 waiting for the long simulation.

8 To fix the failure, in the **Action** box click **Modify All**.

The **Modify All** action configures the model to the settings recommended in the Analysis Result. The **Action Result** box displays a table of changes.

Note Prior to automatically fixing failures, you should review the **Analysis Result** box to ensure that you want to apply all the recommended actions.

9 Click the **Run This Task** button.

The task passes and the tool stores the results in a run named `FPA_Reference`. You can view these results in the Fixed-Point Tool **Contents** pane.

10 Open 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 later tasks.

The task passes.

11 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.

12 Select and run **Check bus usage**. This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Note This is a Simulink check. For more information, see "Check bus usage" in the Simulink documentation.

The task passes.

13 Select and run **Simulation range checking**. This tasks verifies that the **Model Configuration Parameters** > **Diagnostics** > **Simulation range checking** option is not set to `none`. A warning is displayed because **Simulation range checking** is currently set to `none`. The recommended setting is `warning` so that warnings are generated when signals exceed the specified minimum or maximum values.

14 Fix the warning by applying the recommended setting using the **Modify All** button. Rerun the task.

The task passes.

15 Select and run **Check for implicit signal resolution**. This task checks for models that use implicit signal resolution. To use the Fixed-Point Advisor for Simulink signal object scaling, turn off implicit signal resolution by setting the **Diagnostics** > **Data Validity** > **Signal resolution** property in the Configuration Parameters dialog box to `Explicit only`. Enforce resolution for each of the signals and states that currently resolve successfully. For more information, see "Signal resolution" in the Simulink documentation.

The task passes because the model contains no Simulink signal objects.

The run to failure action has completed 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

The tasks in this folder prepare the model for automatic data typing by the Fixed-Point Tool. This folder contains tasks that set the block configuration options and output minimum and maximum values for blocks. The block settings from this task simplify the initial data typing and scaling. The optimal block configuration is achieved in later stages.

1. Right-click **Prepare for Data Typing and Scaling** and select **Run to Failure**.

   The Fixed-Point Advisor runs the **Review locked data type settings** task. This task identifies blocks that have their data type settings locked down, which excludes them from automatic data typing.

   The task passes because it finds no blocks with locked data types.

2. The Fixed-Point Advisor runs the **Remove output data type inheritance** task. This task identifies blocks with the **Output data type** property set to `Inherit`. Inherited data types might lead to data type propagation errors.

   The task fails because some blocks in the model have inherited output data types.

3. Fix the failure using the **Modify All** button to explicitly configure the output data types to the recommended values. Rerun the task.

   The task passes.

4. Continue running to failure. **Relax input data type settings** runs. This task identifies blocks with input data type constraints that might lead to data type propagation errors.

   The task passes because all blocks have flexible input data types.

5. **Verify Stateflow charts that have strong data typing with Simulink** runs. This task verifies that all Stateflow charts are configured to have strong data typing with Simulink I/O.

   The task passes because the model does not have any Stateflow charts.

6 **Remove redundant specification between signal objects and blocks** runs. This task identifies and removes redundant data type specification originating from blocks and Simulink signal objects.

   The task passes because the model contains no resolved Simulink signal objects.

7 **Verify hardware selection** runs. This task identifies the hardware device information on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

   The task fails because the **Model Configuration Parameters** > **Hardware Implementation** option does not provide values for the **Device vendor** and **Device type** parameters.

8 Fix the failure:

   a Click the Hardware Implementation Device settings link.

   b In the Configuration Parameters dialog box **Hardware Implementation** pane, change:

      - **Device vendor** to `Generic`
      - **Device type** to `32-bit Embedded Processor`

   c Click **OK** to apply the settings.

9 In the Fixed-Point Advisor window, rerun the task.

   The task fails because you must specify a default data type for floating-point data types that is suitable for the chosen hardware.

10 Fix the failure by setting **Default data type for all floating-point signals** to `int16`.

   The software uses this default data type for all output signals. The Fixed-Point Advisor proposes the `Same as embedded hardware integer` setting, which is `int32`. However, use `int16` because the model performs many multiplications and you want the product to fit into `int32`.

11 Rerun the task.

   The task passes.

12 Select and run **Specify block minimum and maximum values**.

The Fixed-Point Advisor warns you that you have not specified any minimum and maximum values. Optimally, 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, these values are determined during the design process based on the system that you are creating.

13 Fix the warning by specifying minimum and maximum values for Inport blocks:

a Click the **Explore Result** button.

The Model Advisor Result Explorer opens, displaying the Inport blocks that do not have an output minimum and maximum specified.

b On the Model Advisor Result Explorer center pane, select `Ctr_in`. For the purpose of this tutorial, specify the output minimum and maximum values for this block. Set **OutMin** to `-5` and set **OutMax** to `5`.

c Close the Model Advisor Result Explorer.

d In the Fixed-Point Advisor, rerun the task.

The task passes because minimum and maximum values are specified for all Inport blocks.

e For the purpose of this tutorial, do not specify other minimum and maximum values for other blocks.

f Review the results report found at the folder level.

14 Select and run **Return to the Fixed-Point Tool to perform data typing and scaling**.

15 On the Fixed-Point Tool **Contents** pane, examine the results for the simulation reference run. One of the TrigFcn block outputs overflowed multiple times, indicating that the fixed-point settings on this block are not suitable for the input range. To refine the fixed-point data types, first run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This action provides a floating-point benchmark that represents the ideal output. Then, propose new data types based on these “ideal” results.

Propose Data Types Based on the Simulation Reference Run

Use the Fixed-Point Tool to propose fixed-point data types based on the simulation reference (`FPA_Reference`) run.

1 In the Fixed-Point Tool:

   a Click the **Propose fraction lengths** button DT.

   b Because you are proposing data types based on fixed-point results, the tool issues a warning. In the warning dialog box, click **Yes**.

   The Fixed-Point Tool proposes new data types for objects in the model and updates the results on the **Contents** pane.

2 In the Fixed-Point Tool, set the **Column View** to `Automatic Data Typing with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed scaling in the **ProposedDT** column in the **Contents** pane.

   To accommodate the full simulation range, the Fixed-Point Tool proposes new data types for some blocks in the model. Because the `TrigFcn` block is a linked library, the tool does not propose new data types for this block.

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 detects no conflicts, so you are ready to apply the new data types as described in “Apply the New Fixed-Point Data Types” on page 32-25.

Apply the New Fixed-Point Data Types

1 Click **Apply accepted fraction lengths** to write the proposed data types to the model.
   DT✓

2 In the Fixed-Point Tool toolbar, select **Show > All results**.

The tool has set all the specified data types to the proposed types.

Simulate the Model Using New Fixed-Point Settings

1 On the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings.

2 On the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

3 Click **Simulate** to run the simulation.

The Simulink software simulates the model using the new fixed-point settings that you applied in the previous step and stores the results in the `NoOverride` run.

4 Examine the results. Because the tool did not propose new data types for the `TrigFcn` block, this block still overflows.

31

Fixed-Point Tool

- “Fixed-Point Tool” on page 31-2
- “Run Management” on page 31-5
- “Debug a Fixed-Point Model” on page 31-12
- “Logging Simulation Ranges for Referenced Models” on page 31-20
- “Log Simulation Ranges for Referenced Models” on page 31-24
- “Propose Data Types for a Referenced Model” on page 31-30
- “Logging Simulation Ranges for MATLAB Function Block” on page 31-34
- “Log Simulation Ranges for MATLAB Function Block” on page 31-35
- “View Signal Names in Fixed-Point Tool” on page 31-39

Fixed-Point Tool

In this section...
"Introduction to the Fixed-Point Tool" on page 31-2
"Using the Fixed-Point Tool" on page 31-2

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 too, 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"
Proposing data types based on simulation data	"Conversion Using Simulation Data"
Proposing data types based on derived ranges	"Conversion Using Range Analysis"
Proposing data types based on simulation data from multiple runs	"Propose Data Types Using Multiple Simulations" on page 34-67
Debugging fixed-point models	"Debug a Fixed-Point Model" on page 31-12

Using the Fixed-Point Tool

To open the Fixed-Point Tool, use any of the following methods:

- From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

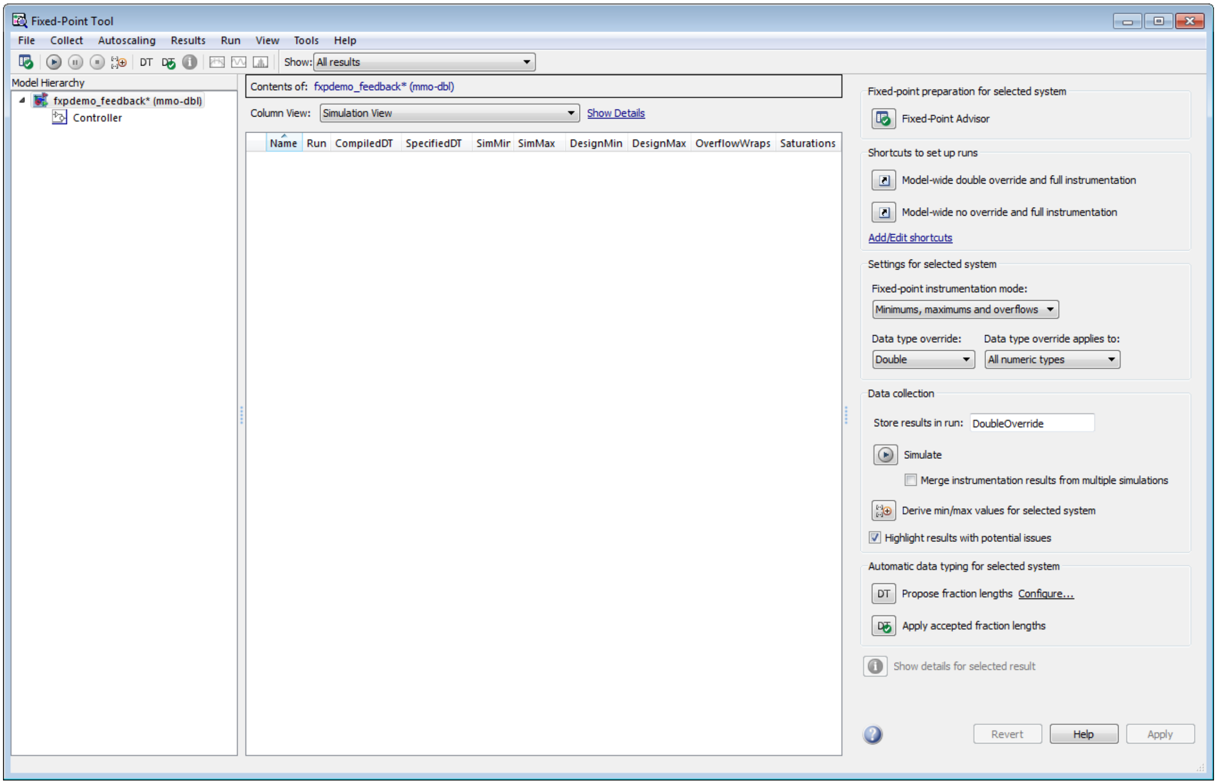
- From the model context menu, select **Fixed-Point Tool**.
- From a subsystem context menu, select **Fixed-Point Tool**.

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.
- Dialog pane — 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.
- **Shortcut Editor** — To open the **Shortcut Editor**, on the far right-hand pane, click the **Add/Edit shortcuts** 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 31-5.

For more information about each of these components, see `fxptdlg`.


Fixed-Point Tool
File
Collect
Autoscaling
Results
Run
View
Tools
Help
DT
Model Hierarchy
fxpdemo_feedback* (mmo-dbl)
Controller
Show Details
Name
Run
CompiledDT
SpecifiedDT
SimMir
SimMax
DesignMin
DesignMax
OverflowWraps
Saturations
Fixed-point preparation for selected system
Fixed-Point Advisor
Shortcuts to set up runs
Model-wide double override and full instrumentation
Model-wide no override and full instrumentation
Add/Edit shortcuts
Settings for selected system
Data collection
Simulate
Derive min/max values for selected system
Automatic data typing for selected system
DT
Propose fraction lengths
Configure...
Apply accepted fraction lengths
Show details for selected result
Revert
Help
Apply

Show: All results

Contents of: fxpdemo_feedback* (mmo-dbl)

Column View: Simulation View

Fixed-point instrumentation mode: Minimums, maximums and overflows

Data type override: Double

Data type override applies to: All numeric types

Store results in run: DoubleOverride

□ Merge instrumentation results from multiple simulations

☑ Highlight results with potential issues

Run Management

In this section…

"Run Management" on page 31-5

"Why Use Shortcuts to Manage Runs" on page 31-7

"When to Use Shortcuts to Manage Runs" on page 31-7

"Add Shortcuts" on page 31-8

"Edit Shortcuts" on page 31-9

"Delete Shortcuts" on page 31-10

"Capture Model Settings Using the Shortcut Editor" on page 31-10

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 shortcuts. 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 shortcuts, such as **Model-wide double override and full instrumentation**, 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 to create or edit shortcuts and to add and organize shortcut buttons in the Fixed-Point Tool **Shortcuts to set up runs** 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 **Data collection** pane, **Store results in run**.

  Provide a new run name before a simulation or collecting derived minimum and maximum values so that you do not overwrite existing runs.

- In the **Settings for selected system** pane:
  - **Fixed-point instrumentation mode**
  - **Data type override**
  - **Data type override applies to**

Why Use Shortcuts to Manage Runs

Shortcuts provide a quick and easy way to set up data type override and fixed-point instrumentation settings run 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 shortcuts. These defaults provide an efficient way to override the model with floating-point data types or remove existing data type overrides. For more information, see “Propose

To ...	Use...
	Fraction Lengths Using Simulation Range Data” on page 34-46.
Debug a model	Shortcuts to switch between different data type override and fixed-point instrumentation modes. For more information, see “Debug a Fixed-Point Model” on page 31-12.
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 Shortcut Editor 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 Shortcut Editor 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 31-10.

Add Shortcuts

1 On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2 For each subsystem that you want to specify a shortcut for, on the Shortcut Editor **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 31-9.

Edit Shortcuts

1. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2. In the Shortcut Editor, 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 shortcuts.

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, 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 **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2 On the Shortcut Editor **Manage shortcuts** pane, 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 **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2 In the Shortcut Editor, 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 31-12

"Debugging the Model" on page 31-14

"Simulating the Model Using a Different Input Stimulus" on page 31-16

"Debugging the Model with the New Input" on page 31-17

"Proposing Fraction Lengths for Math2 Based on Simulation Results" on page 31-18

"Verifying the New Settings" on page 31-18

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.

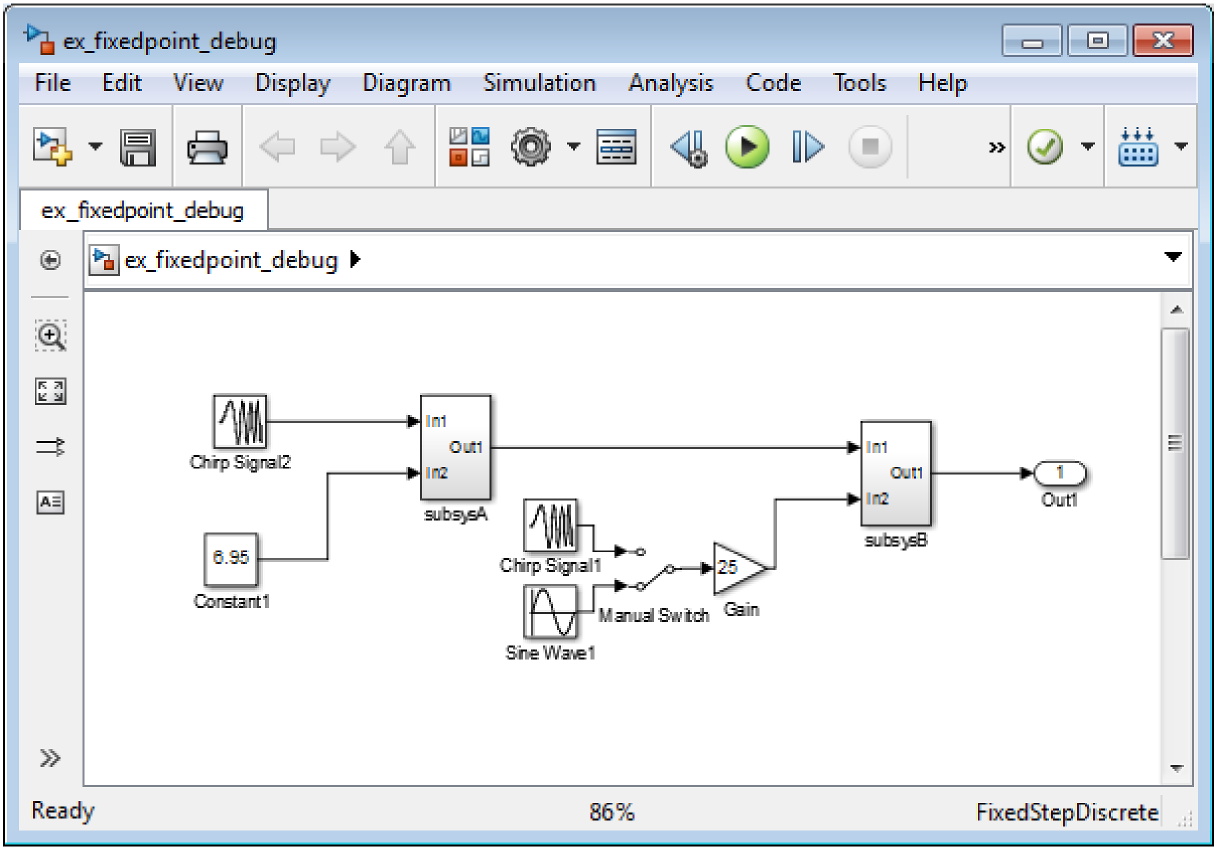
- Autoscale the model over the complete simulation range for both inputs.

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 **Fixed-Point Tool**.

3 In the Fixed-Point Tool, set up a shortcut for the initial system settings:

On the **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

4 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.

5 Use this shortcut to set up a run. Use the settings to simulate the model.

a On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

b On the **Shortcuts to set up runs** pane, click **Setting A**.

c Click the **Simulate** button, ▶

The Simulink software 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 **OverflowWraps** 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 On the **Manage shortcuts** pane, 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 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

2 On the **Shortcuts to set up runs** pane, click **Setting B** to override fixed-point settings on `Math2`.

3 Click the **Simulate** button.

The Simulink software 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. On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

2. On the **Shortcuts to set up runs** pane, click **Setting C** to override the fixed-point settings on `Math1`.

3. Click the **Simulate** button.

   The Simulink software 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. On the **Data collection** pane, select **Merge instrumentation 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 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug` and simulate using the original fixed-point settings for `Math1` and `Math2`.

   a On the **Shortcuts to set up runs** pane, click **Setting A**.

   b Click the **Simulate** button.

   The Simulink software 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 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

   b On the **Shortcuts to set up runs** pane, click **Setting B**.

   c Click the **Start** 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 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

   b On the **Shortcuts to set up runs** pane, click **Setting C**.

   c Click the **Start** 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 On the Fixed-Point Tool **Model Hierarchy** pane, select `Math2`.

2 On the **Automatic data typing for selected system** pane, click the **Propose fraction lengths** button.

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.

   In the **Automatic data typing for selected system** pane, click the **Apply accepted fraction lengths** button.

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 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

2 On the **Shortcuts to set up runs** pane, click **Setting A**.

3 On the **Data collection** pane, set **Store results in run** to `Setting A2` and click **Apply** so that the Fixed-Point Tool does not overwrite the previous results for this shortcut.

4 Click the **Simulate** button.

The Simulink software 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.

Logging Simulation Ranges for Referenced Models

In this section...

"Viewing Simulation Ranges for Referenced Models" on page 31-20

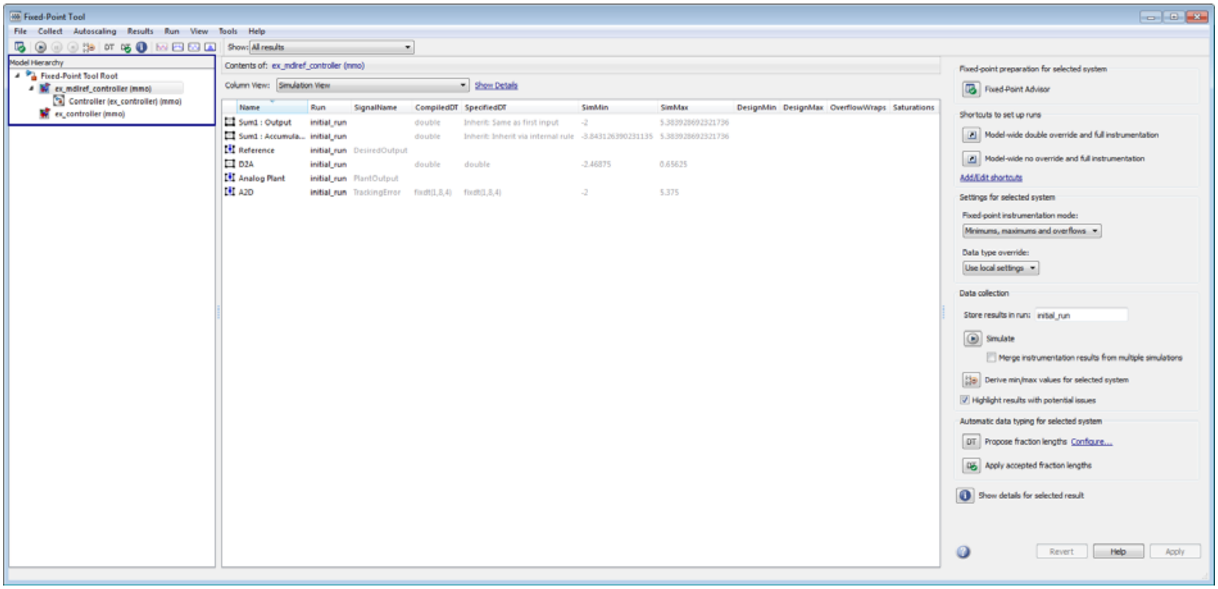
"Fixed-Point Instrumentation and Data Type Override Settings" on page 31-22

"See Also" on page 31-23

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_mdlref_controller` model contains a Model block that references the `ex_controller` model. The Fixed-Point Tool shows both models in the model hierarchy.



Show: All results

Contents of: ex_mdlref_controller (mmo)

Column View: Simulation View

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax	OverflowWraps	Saturations
Sum1 : Output	initial_run		double	Inherit: Same as first input	-2	5.383928692321736				
Sum1 : Accumula...	initial_run		double	Inherit: Inherit via internal rule	-3.843126390231135	5.383928692321736				
Reference	initial_run	DesiredOutput								
D2A	initial_run		double	double	-2.46875	0.65625				
Analog Plant	initial_run	PlantOutput								
A2D	initial_run	TrackingError	fixdt(1,8,4)	fixdt(1,8,4)	-2	5.375				

Fixed-point preparation for selected system

Shortcuts to set up runs

Model-wide double override and full instrumentation

Model-wide no override and full instrumentation

Settings for selected system

Fixed-point instrumentation mode: Minimums, maximums and overflows

Data type override: Use local settings

Data collection

Store results in run: initial_run

□ Merge instrumentation results from multiple simulations

Derive min/max values for selected system

☑ Highlight results with potential issues

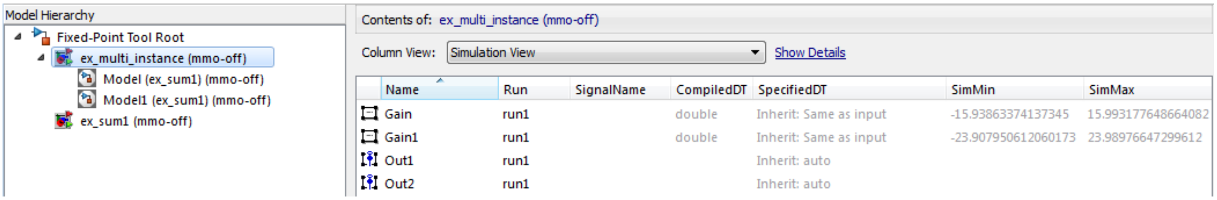
Automatic data typing for selected system

Propose fraction lengths

Apply accepted fraction lengths

Show details for selected result

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.

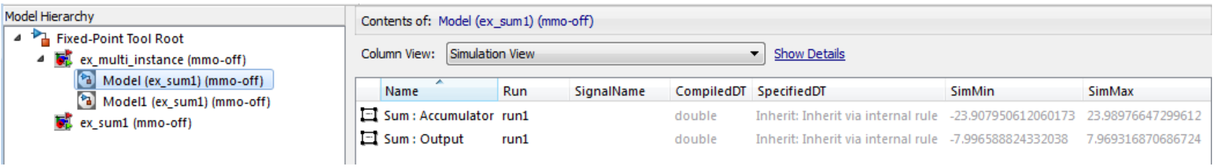

Model Hierarchy
Fixed-Point Tool Root
ex_multi_instance (mmo-off)
Model (ex_sum1) (mmo-off)
Model1 (ex_sum1) (mmo-off)
ex_sum1 (mmo-off)
Show Details

Contents of: ex_multi_instance (mmo-off)

Column View: Simulation View

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		

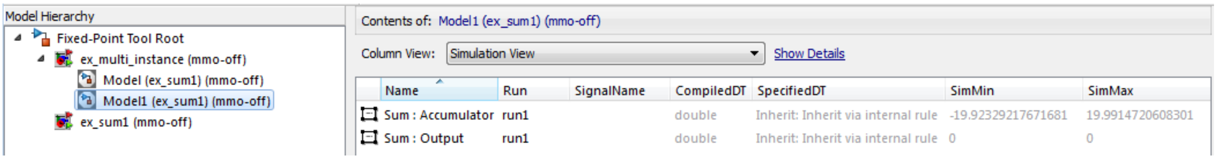
The tool logs and displays the results for each instance of the referenced model. For example, here are the results for the first instance of the referenced model `ex_sum1` in `ex_multi_instance`.



Contents of: Model (ex_sum1) (mmo-off)

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

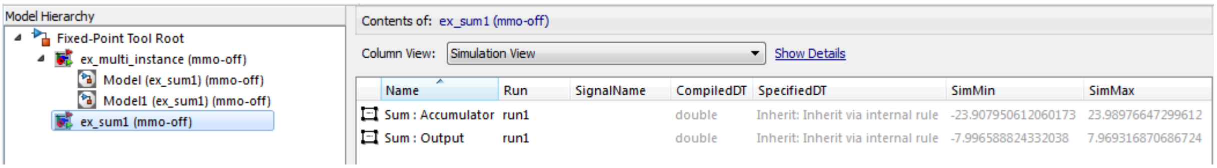
Here are the results for the second instance of `ex_sum1`.



Contents of: Model1 (ex_sum1) (mmo-off)

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.



Contents of: ex_sum1 (mmo-off)

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 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.

See Also

- "Log Simulation Ranges for Referenced Models" on page 31-24

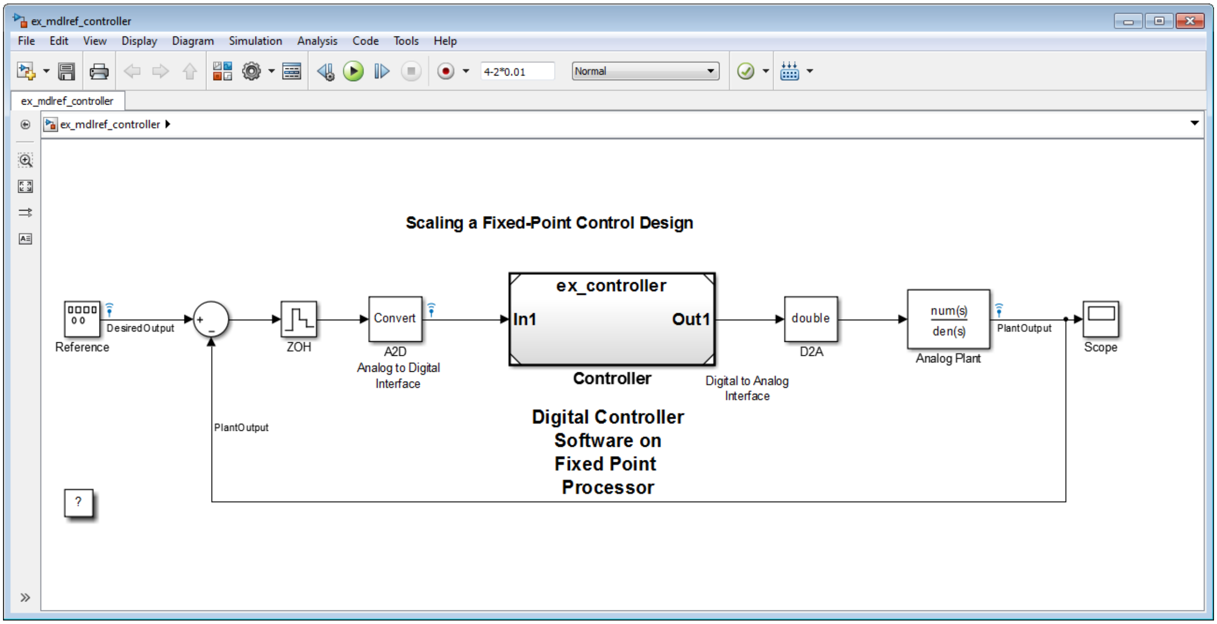
Log Simulation Ranges for Referenced Models

This example shows how to log simulation minimum and maximum values for a model that contains multiple instances of the same referenced model.

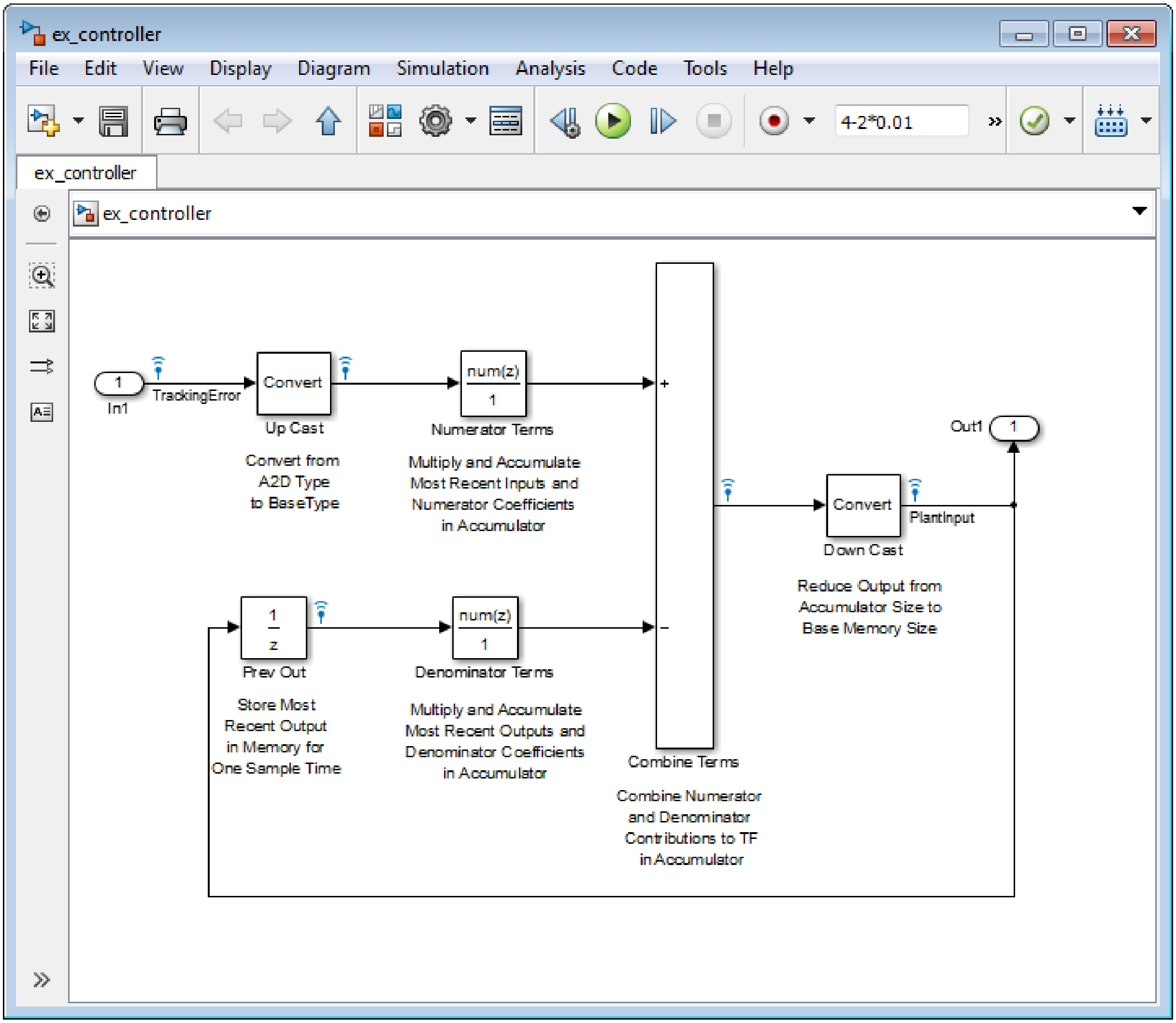
Simulate the Model Using Local Settings

1 Open the `ex_mdlref_controller` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_mdlref_controller
```

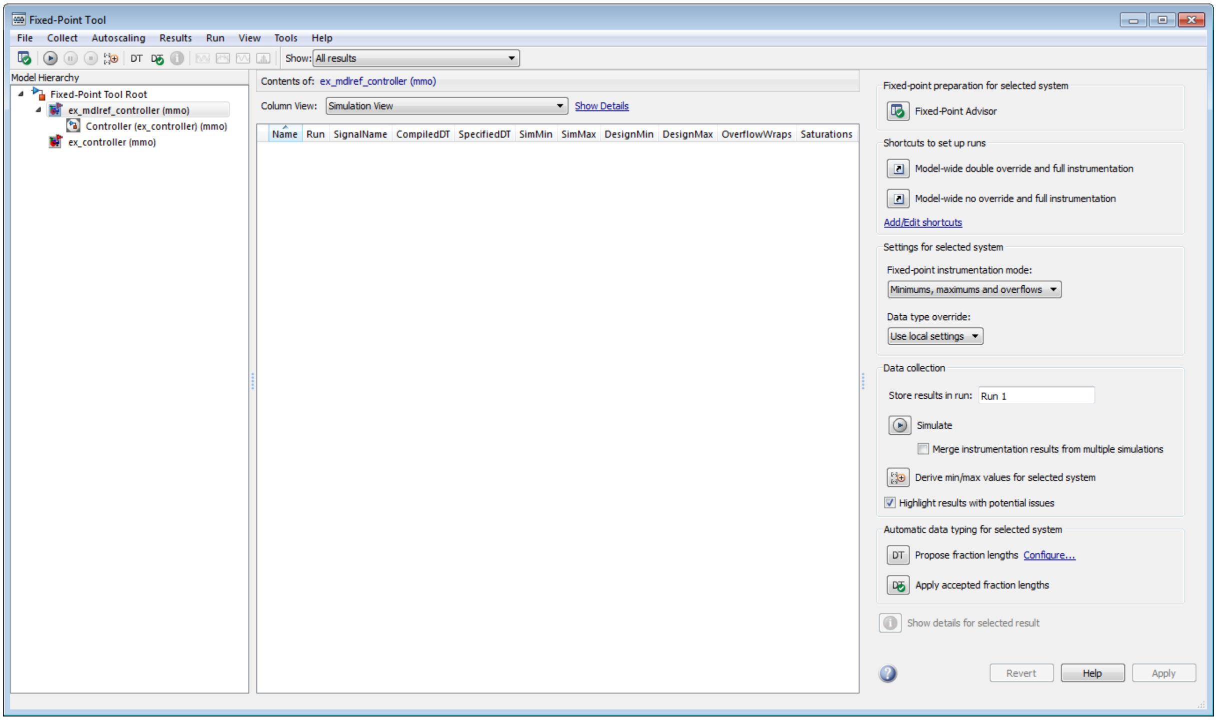

ex_mdlref_controller
File Edit View Display Diagram Simulation Analysis Code Tools Help
4-2*0.01
Normal
ex_mdlref_controller
Scaling a Fixed-Point Control Design
Reference
DesiredOutput
ZOH
Convert
A2D
Analog to Digital Interface
ex_controller
In1
Out1
Controller
double
D2A
Digital to Analog Interface
num(s)
den(s)
Analog Plant
PlantOutput
Scope
PlantOutput
Digital Controller Software on Fixed Point Processor
?

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_mdlref_controller` model menu, select **Analysis > Fixed-Point Tool**.

The Fixed-Point Tool opens. In its **Model Hierarchy** pane, the tool displays two model nodes, one for the `ex_mdlref_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.



Show: All results

Model Hierarchy
- Fixed-Point Tool Root
  - ex_mdlref_controller (mmo)
    - Controller (ex_controller) (mmo)
  - ex_controller (mmo)

Contents of: ex_mdlref_controller (mmo)

Column View: Simulation View

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax	OverflowWraps	Saturations

Fixed-point preparation for selected system

Shortcuts to set up runs
- Model-wide double override and full instrumentation
- Model-wide no override and full instrumentation

Settings for selected system

Fixed-point instrumentation mode: Minimums, maximums and overflows

Data type override: Use local settings

Data collection

Store results in run: Run 1

□ Merge instrumentation results from multiple simulations

Derive min/max values for selected system

☑ Highlight results with potential issues

Automatic data typing for selected system
- Propose fraction lengths
- Apply accepted fraction lengths

Show details for selected result

3 In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_mdlref_controller` model.

4 On the **Settings for selected system** pane, verify that:

- **Fixed-point instrumentation mode** is set to `Minimums, maximums and overflows`.
- **Data type override** is set to `Use local settings` so the model will log simulation data using the data types set up on the model.

5 In the Fixed-Point Tool Data collection pane, set **Store results in run** 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. The fixed-point instrumentation and data type override settings for the top-level model do not affect the settings in the referenced model. In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_controller` model and verify that:

   - **Fixed-point instrumentation mode** is set to `Minimums, maximums and overflows`
   - **Data type override** is set to `Use local settings`

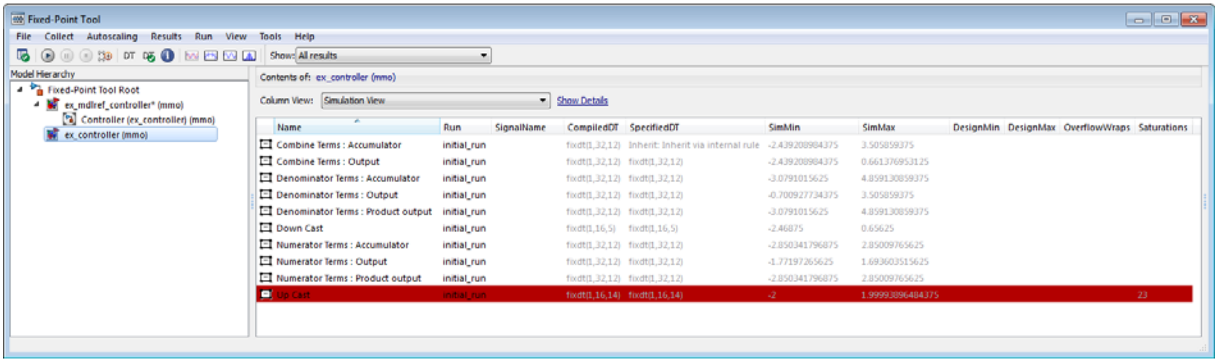
7. In the Fixed-Point Tool, click **Simulate**.

   The Simulink software 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.

   The Simulation Data Inspector tool opens. You can use this tool to inspect and compare signals in your model.

8. 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 **Saturations** column for this result shows that the block saturated 23 times, which indicates poor scaling.


Fixed-Point Tool
File Collect Autoscaling Results Run View Tools Help
DT
Model Hierarchy
Fixed-Point Tool Root
ex_mdlref_controller* (mmo)
Controller (ex_controller) (mmo)
ex_controller (mmo)
Show Details

Show: All results

Contents of: ex_controller (mmo)

Column View: Simulation View

Name	Run	SignalName	CompiledDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax	OverflowWraps	Saturations
Combine Terms : Accumulator	initial_run		fixdt(1,32,12)	Inherit: Inherit via internal rule	-2.439208984375	3.505859375				
Combine Terms : Output	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-2.439208984375	0.661376953125				
Denominator Terms : Accumulator	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-3.0791015625	4.859130859375				
Denominator Terms : Output	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-0.700927734375	3.505859375				
Denominator Terms : Product output	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-3.0791015625	4.859130859375				
Down Cast	initial_run		fixdt(1,16,5)	fixdt(1,16,5)	-2.46875	0.65625				
Numerator Terms : Accumulator	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-2.850341796875	2.85009765625				
Numerator Terms : Output	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-1.77197265625	1.693603515625				
Numerator Terms : Product output	initial_run		fixdt(1,32,12)	fixdt(1,32,12)	-2.850341796875	2.85009765625				
Up Cast	initial_run		fixdt(1,16,14)	fixdt(1,16,14)	-2	1.99993896484375				23

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. In the **Settings for selected system** pane, set **Data type override** to `Double`.

2. In the **Data collection** pane, set **Store results in run** to `double_run` and then click **Apply**.

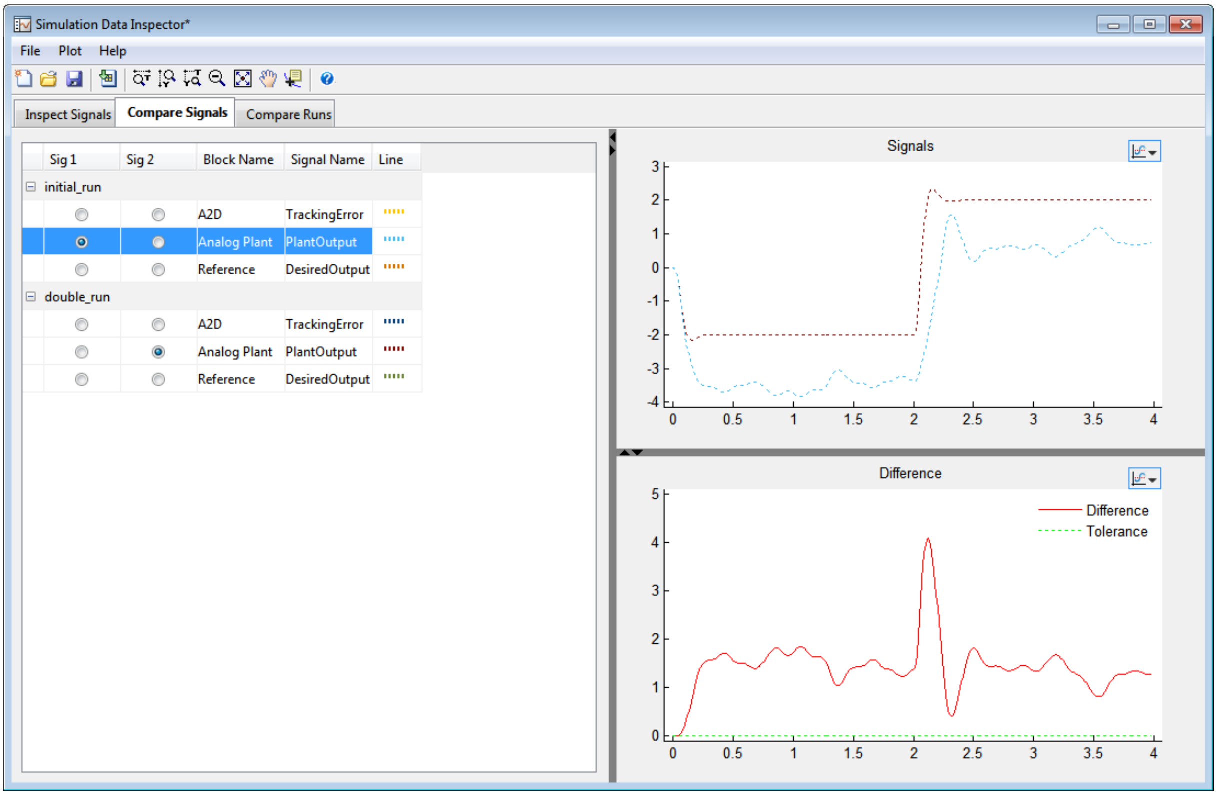
3. In the **Model Hierarchy** pane, select the `ex_mdlref_controller` model, set **Data type override** to `Double` and then click **Apply**.

   Setting data type override for the top-level model avoids data type propagation issues when you simulate the model.

4. In the Fixed-Point Tool, click **Simulate**.

   The Simulink software simulates the `ex_mdlref_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.

5. Use the Simulation Data Inspector to view the `initial_run` and `double_run` versions of the signal associated with the Analog Plant output (upper axes), and the difference between the signals (lower axes).



Sig 1	Sig 2	Block Name	Signal Name	Line
⊟ initial_run				
○	○	A2D	TrackingError	⋯
●	○	Analog Plant	PlantOutput	⋯
○	○	Reference	DesiredOutput	⋯
⊟ double_run				
○	○	A2D	TrackingError	⋯
○	●	Analog Plant	PlantOutput	⋯
○	○	Reference	DesiredOutput	⋯

Now you are ready to propose data types based on the simulation results from the doubles override run.

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.

1 In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `ex_controller` model.

2 In the **Automatic data typing for selected system** pane, click the **Configure** link and verify that **Propose fraction lengths for specified word lengths** is selected.

3 In the same pane, specify the **Safety margin for simulation min/max (%)** parameter as `20` and click **Apply**.

4 In the Fixed-Point Tool, click **Propose fraction lengths**, DT.

   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 **Percent 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.

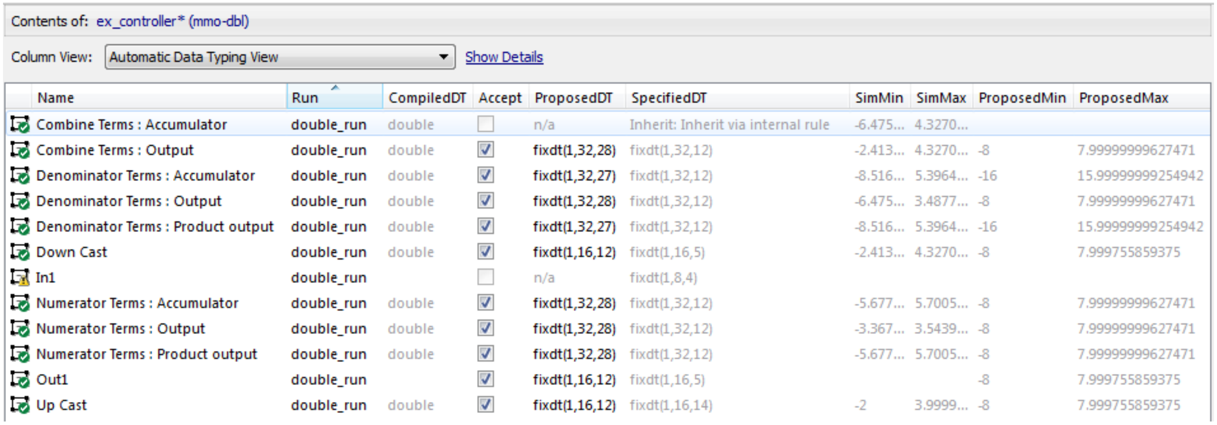
5 In the Propose Data Types dialog box, select `double_run` and click **OK**.

   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.

- **Output data type** parameter specifies a generalized fixed-point number.

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**, **ProposedMax**, which are relevant at this stage of the fixed-point conversion.



Contents of: ex_controller* (mmo-dbl)

Column View: Automatic Data Typing View

Name	Run	CompiledDT	Accept	ProposedDT	SpecifiedDT	SimMin	SimMax	ProposedMin	ProposedMax
Combine Terms : Accumulator	double_run	double	□	n/a	Inherit: Inherit via internal rule	-6.475...	4.3270...		
Combine Terms : Output	double_run	double	☑	fixdt(1,32,28)	fixdt(1,32,12)	-2.413...	4.3270...	-8	7.99999999627471
Denominator Terms : Accumulator	double_run	double	☑	fixdt(1,32,27)	fixdt(1,32,12)	-8.516...	5.3964...	-16	15.99999999254942
Denominator Terms : Output	double_run	double	☑	fixdt(1,32,28)	fixdt(1,32,12)	-6.475...	3.4877...	-8	7.99999999627471
Denominator Terms : Product output	double_run	double	☑	fixdt(1,32,27)	fixdt(1,32,12)	-8.516...	5.3964...	-16	15.99999999254942
Down Cast	double_run	double	☑	fixdt(1,16,12)	fixdt(1,16,5)	-2.413...	4.3270...	-8	7.999755859375
In1	double_run		□	n/a	fixdt(1,8,4)				
Numerator Terms : Accumulator	double_run	double	☑	fixdt(1,32,28)	fixdt(1,32,12)	-5.677...	5.7005...	-8	7.99999999627471
Numerator Terms : Output	double_run	double	☑	fixdt(1,32,28)	fixdt(1,32,12)	-3.367...	3.5439...	-8	7.99999999627471
Numerator Terms : Product output	double_run	double	☑	fixdt(1,32,28)	fixdt(1,32,12)	-5.677...	5.7005...	-8	7.99999999627471
Out1	double_run		☑	fixdt(1,16,12)	fixdt(1,16,5)			-8	7.999755859375
Up Cast	double_run	double	☑	fixdt(1,16,12)	fixdt(1,16,14)	-2	3.9999...	-8	7.999755859375

6 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 subsystem's blocks.

The Fixed-Point Tool does not propose a data type for `Combine Terms:Accumulator` and displays `n/a` in the **ProposedDT** column. The tool does not propose a data type because the **SpecifiedDT** is `Inherit: Inherit via internal rule`. To view more information about a proposal, click the **Show details for selected result** button ⓘ.

7. In the Fixed-Point Tool, click the **Apply accepted fraction lengths** button DT✓.

   The Fixed-Point Tool applies to the scaling proposals that you accepted in the previous step.

8. In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `ex_mdlref_controller` model.

   a. In the **Settings for selected system** pane, set **Data type override** to `Use local settings`. This option enables each of the model's subsystems to use its locally specified data type settings, however, it does not apply to the referenced model.

   b. In the **Data collection** pane, set **Store results in run** to `scaled_fixed_run` and then click **Apply**.

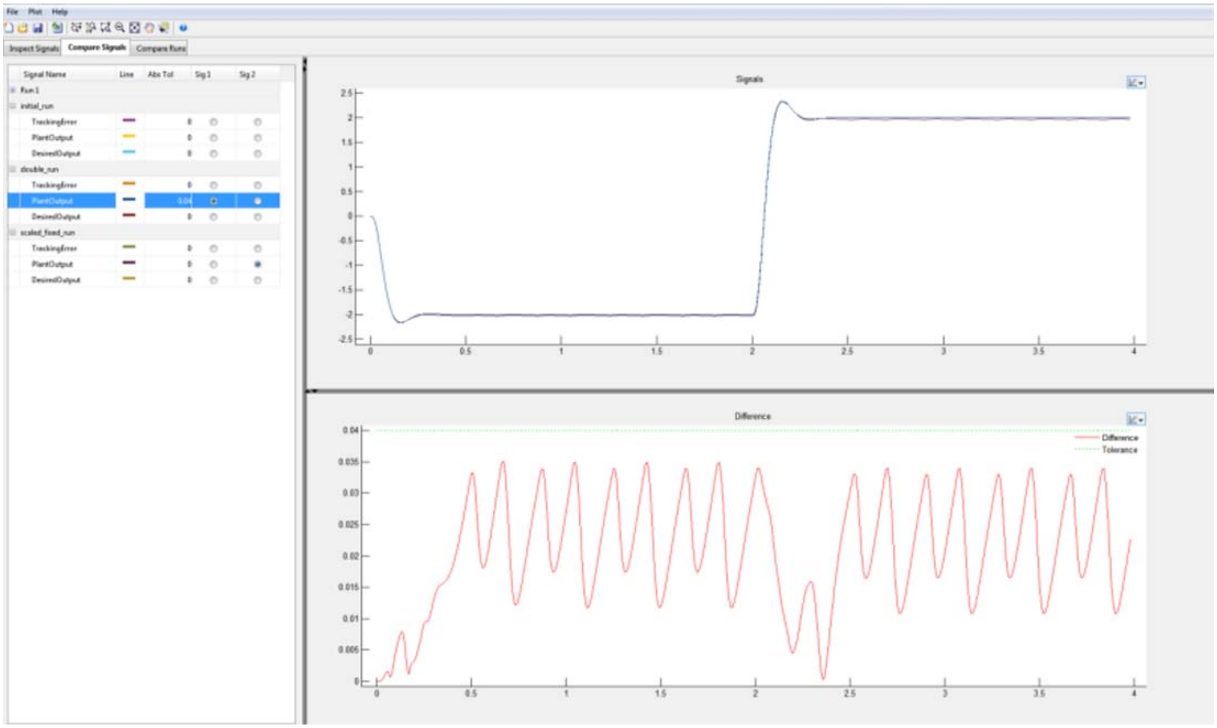
9. In the **Model Hierarchy** pane, select the `ex_controller` model and set its **Data type override** parameter as `Use local settings` and click **Apply**.

10. In the Fixed-Point Tool, click **Simulate**.

    The Simulink software simulates the `ex_mdlref_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.

11. 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`.



Signal Name	Line	Abs Tol	Sig 1	Sig 2
Run 1				
initial_run				
TrackingError	—	0	○	○
PlantOutput	—	0	○	○
DesiredOutput	—	0	○	○
double_run				
TrackingError	—	0	○	○
PlantOutput	—	0.04	●	○
DesiredOutput	—	0	○	○
scaled_fixed_run				
TrackingError	—	0	○	○
PlantOutput	—	0	○	●
DesiredOutput	—	0	○	○

Logging Simulation Ranges for MATLAB Function Block

You can log simulation minimum and maximum values for MATLAB Function blocks using the `Mininums, maximumx and overflows` logging control in the Fixed-Point Tool. The logged minimum and maximum values are displayed in the MATLAB Function Report. For fixed-point data types, the report also displays the percent of current range. You can use the simulation minimum/maximum data to help you determine the optimal word length and fraction length of fixed-point data types for signals in your model. After modifying your model to use fixed-point data types, simulate again to verify that the data types cover the full intended operating range.

Note The software does not log simulation minimum and maximum values for MATLAB Function blocks used as a reference (library) block or in a referenced model.

See Also

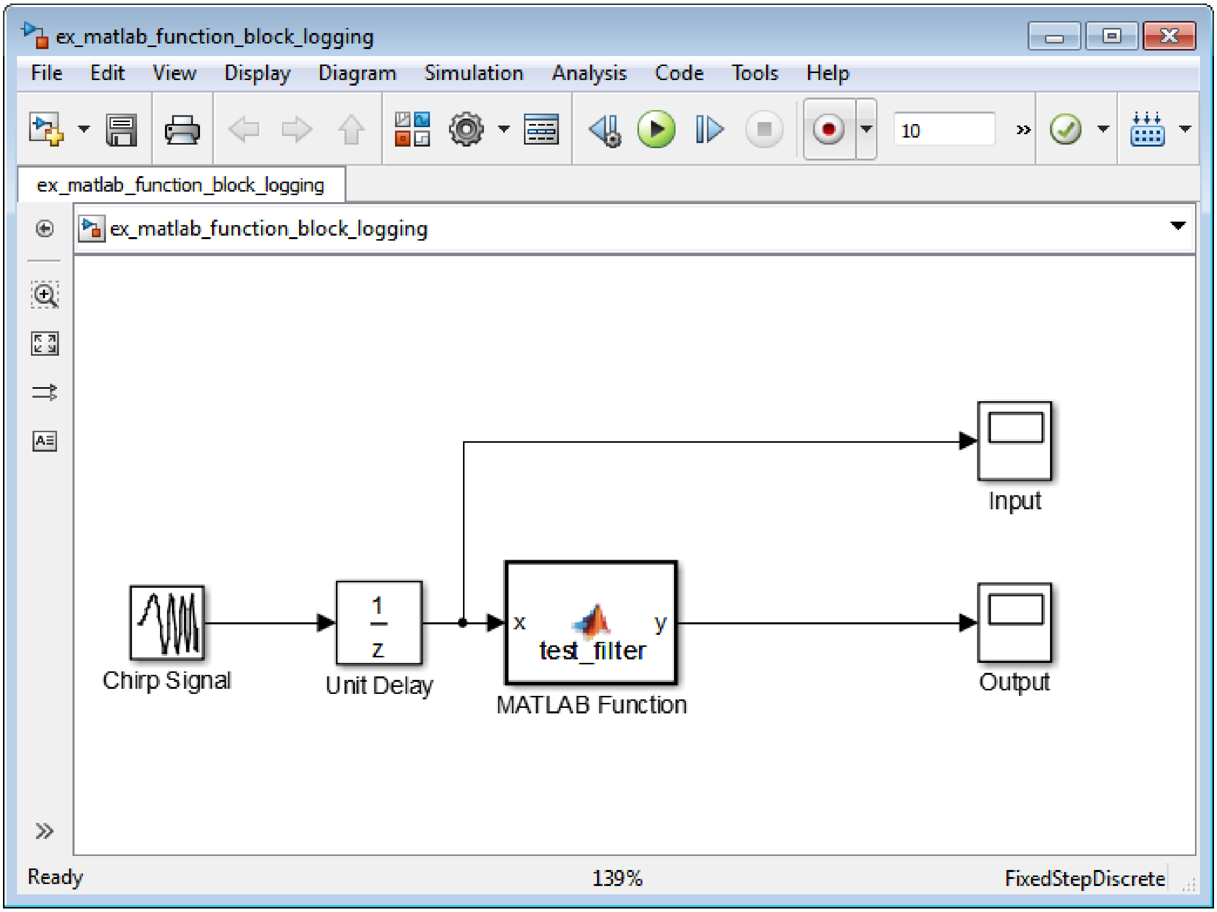
- "Log Simulation Ranges for MATLAB Function Block" on page 31-35

Log Simulation Ranges for MATLAB Function Block

This example shows how to log simulation minimum and maximum values for a MATLAB Function block and view these values in the MATLAB Function Report.

1 Open the `ex_matlab_function_block_logging` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_matlab_function_block_logging
```


ex_matlab_function_block_logging
File Edit View Display Diagram Simulation Analysis Code Tools Help
10
ex_matlab_function_block_logging
ex_matlab_function_block_logging
Input
Chirp Signal
1
z
Unit Delay
x
y
test_filter
MATLAB Function
Output
Ready
139%
FixedStepDiscrete

2 From the model **Analysis** menu, select **Fixed-Point Tool**.

3 In the Fixed-Point Tool, under **Settings for selected system**, **Fixed-point instrumentation mode** is set to `Minimums, maximums and overflows` so that the Fixed-Point Tool logs the simulation minimum and maximum values. **Data type override** is set to `Use local settings` so that the Fixed-Point Tool logs data using the data types specified in the model.

4 On the Fixed-Point Tool **Model Hierarchy** pane, select `ex_matlab_function_block_logging`.

5. Click the **Simulate** button, ▶.

   The Simulink software simulates the model using the specified fixed-point instrumentation and local data type settings.

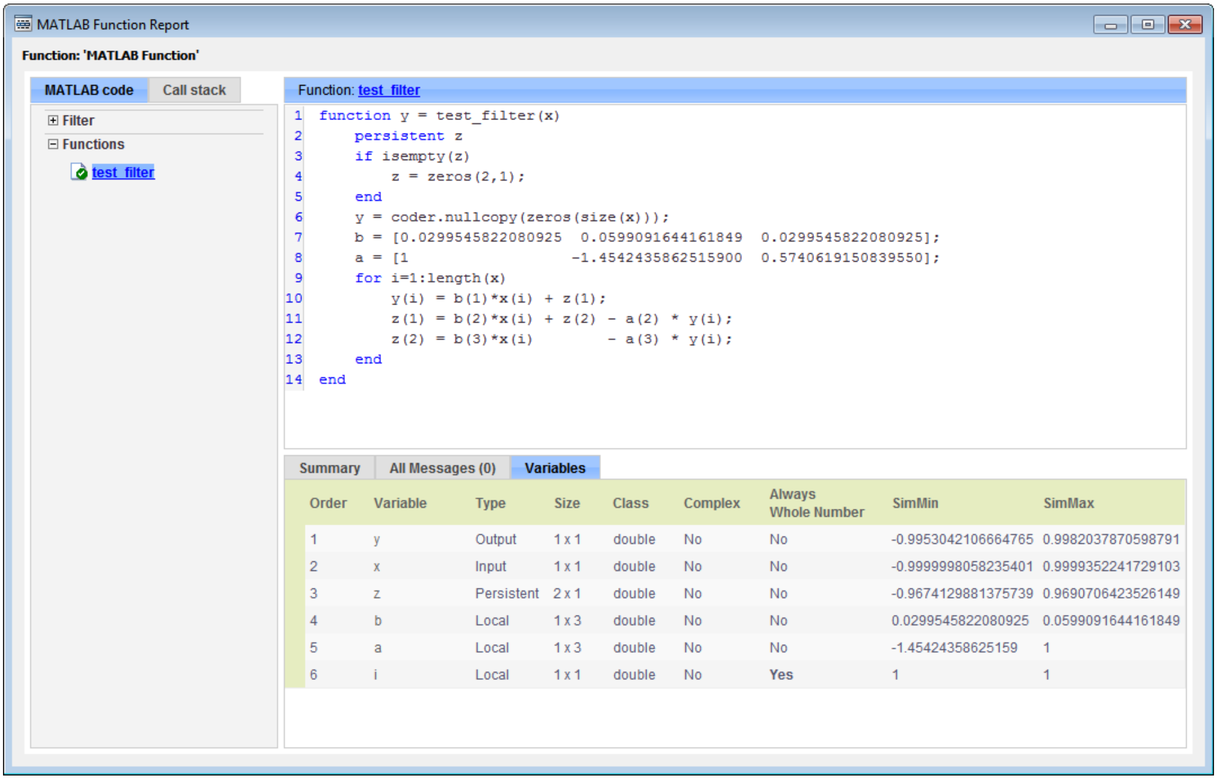
6. In the `ex_matlab_function_block_logging` model, double-click the MATLAB Function block.

   The MATLAB Function block code is displayed in the MATLAB editor window.

7. In MATLAB, on the **Editor** tab, click **View Report**.

8. In the **MATLAB Function Report**, click the **Variables** tab.

   The **Variables** tab displays the simulation minimum and maximum values for the MATLAB Function block input, output, and variables.



Function: test_filter

```
function y = test_filter(x)
    persistent z
    if isempty(z)
        z = zeros(2,1);
    end
    y = coder.nullcopy(zeros(size(x)));
    b = [0.0299545822080925  0.0599091644161849  0.0299545822080925];
    a = [1                  -1.4542435862515900  0.5740619150839550];
    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
```

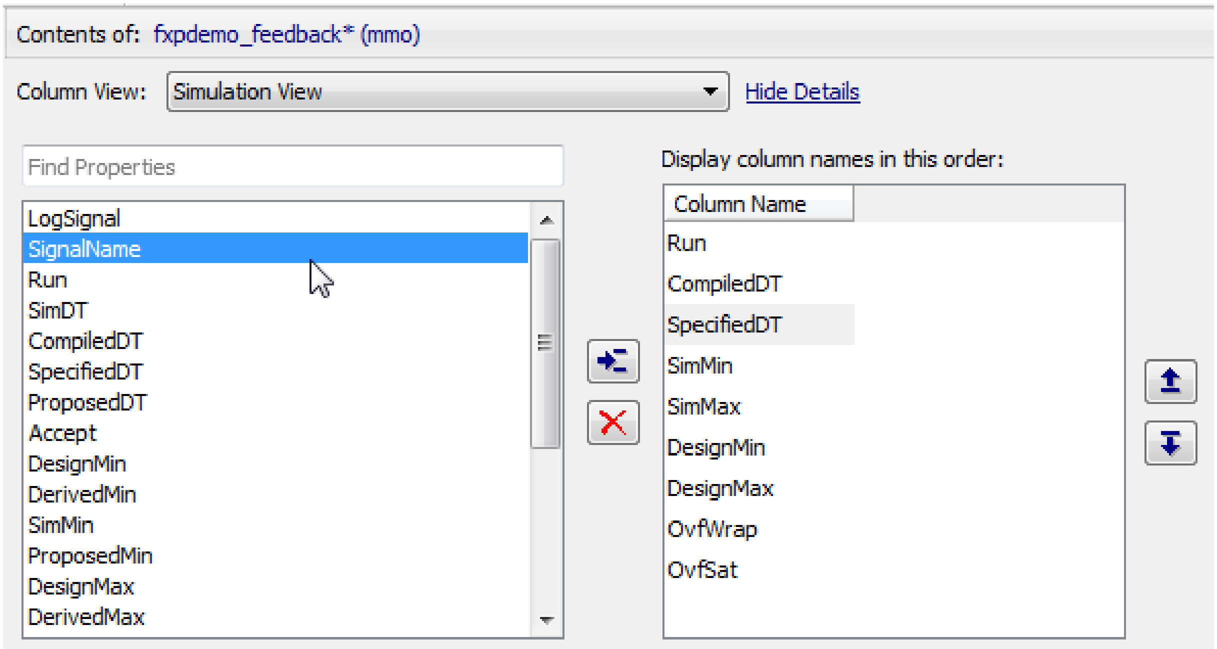
Order	Variable	Type	Size	Class	Complex	Always Whole Number	SimMin	SimMax
1	y	Output	1 x 1	double	No	No	-0.9953042106664765	0.9982037870598791
2	x	Input	1 x 1	double	No	No	-0.9999998058235401	0.9999352241729103
3	z	Persistent	2 x 1	double	No	No	-0.9674129881375739	0.9690706423526149
4	b	Local	1 x 3	double	No	No	0.0299545822080925	0.0599091644161849
5	a	Local	1 x 3	double	No	No	-1.45424358625159	1
6	i	Local	1 x 1	double	No	**Yes**	1	1

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`.



Contents of: fxpdemo_feedback* (mmo)

Column View: Simulation View

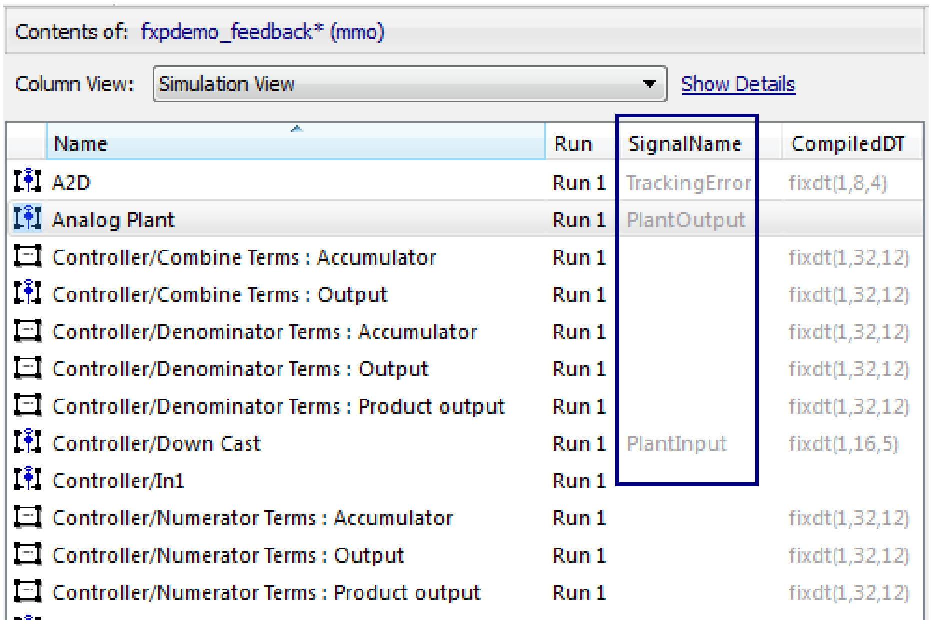
3 Click ⇥.

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.



Contents of: fxpdemo_feedback* (mmo)

Column View: Simulation View

Name	Run	SignalName	CompiledDT
A2D	Run 1	TrackingError	fixdt(1,8,4)
Analog Plant	Run 1	PlantOutput	
Controller/Combine Terms : Accumulator	Run 1		fixdt(1,32,12)
Controller/Combine Terms : Output	Run 1		fixdt(1,32,12)
Controller/Denominator Terms : Accumulator	Run 1		fixdt(1,32,12)
Controller/Denominator Terms : Output	Run 1		fixdt(1,32,12)
Controller/Denominator Terms : Product output	Run 1		fixdt(1,32,12)
Controller/Down Cast	Run 1	PlantInput	fixdt(1,16,5)
Controller/In1	Run 1		
Controller/Numerator Terms : Accumulator	Run 1		fixdt(1,32,12)
Controller/Numerator Terms : Output	Run 1		fixdt(1,32,12)
Controller/Numerator Terms : Product output	Run 1		fixdt(1,32,12)

Convert Floating-Point Model to Fixed Point

- “Learning Objectives” on page 32-2
- “Model Description” on page 32-4
- “Before You Begin” on page 32-7
- “Convert Floating-Point Model to Fixed Point” on page 32-9
- “Key Points to Remember” on page 32-28
- “Where to Learn More” on page 32-29

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 37-36.

- 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.

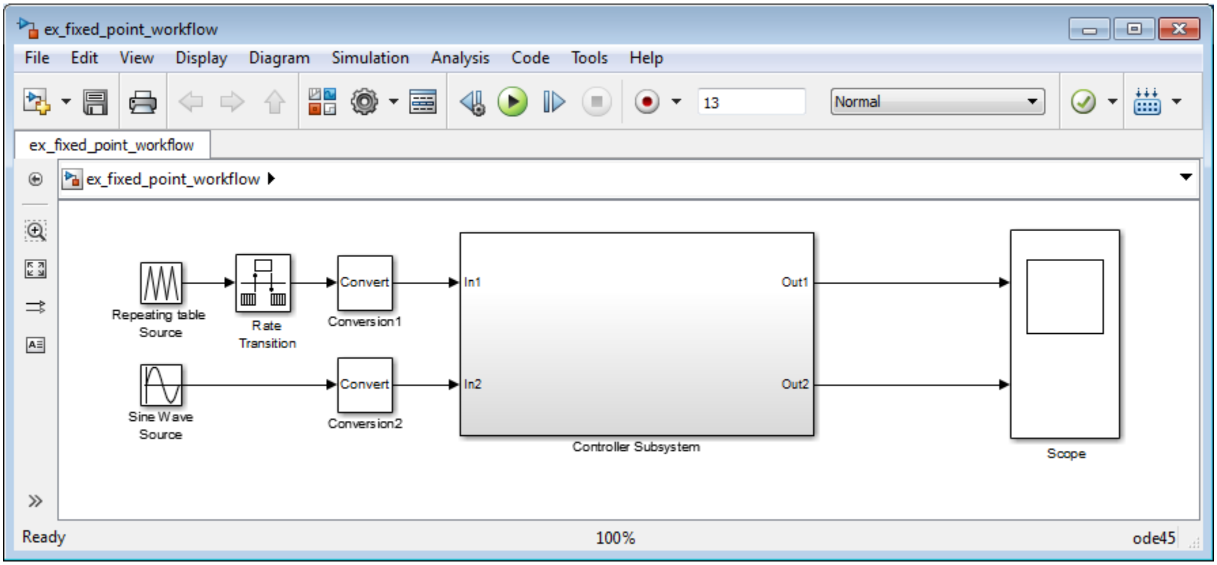
For floating-point inheritance blocks when inputs are floating point, all internal and output data types are floating point. The model in this example uses a Discrete Filter block, which is a floating-point inheritance block.

Model Description

In this section...
"Model Overview" on page 32-4
"Model Set Up" on page 32-5

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 32-9.

   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 32-18.

   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 32-19.

   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 32-19.

   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 32-9

"Prepare Floating-Point Model for Conversion to Fixed Point" on page 32-9

"Propose Data Types" on page 32-18

"Apply Fixed-Point Data Types" on page 32-19

"Verify Fixed-Point Settings" on page 32-19

"Test Fixed-Point Settings With New Input Data" on page 32-22

"Gather a Floating-Point Benchmark" on page 32-23

"Propose Data Types for the New Input" on page 32-24

"Apply the New Fixed-Point Data Types" on page 32-25

"Verify New Fixed-Point Settings" on page 32-25

"Prepare for Code Generation" on page 32-26

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 > Fixed-Point Tool**.

2 In the Fixed-Point Tool:

   a In the **Model Hierarchy** pane, select the Controller Subsystem.

   b In the **Fixed-point preparation for selected system** pane, 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 **Check bus usage**. This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Note This is a Simulink check. For more information, see "Check bus usage".

The task runs and generates a warning because this check works only from top-level models and you are running from the subsystem. Because this model uses no buses, ignore this warning. For models containing buses, you must run the Fixed-Point Advisor from the top-level model to perform this check.

12 Select and run **Simulation range checking**. This tasks verifies that the **Model Configuration Parameters** > **Diagnostics** > **Simulation range checking** option is not set to `none`.

The task generates a warning because the Simulation range checking option is `none`.

13 To fix the warning, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the Simulation range checking option to `warning`.

14 Rerun the task.

The task now passes because the **Simulation range checking** option is correct.

15 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.

16 To fix the failure, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the **Signal resolution** option to `Explicit only`.

17 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 **Remove output data type inheritance**. This task identifies blocks that have an inherited output signal data type that might lead to data type propagation errors.

This task fails because there are floating-point inheritance blocks in the model. For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point. Therefore, you must specify an input parameter data type for these blocks.

3 In the Fixed-Point Advisor **Input Parameters** pane, set **Data type for blocks with floating-point inheritance** to `int16`, and rerun the task.

The task fails and the Fixed-Point Advisor provides information about the failure in the Analysis Result box. The Fixed-Point Advisor recommends that you set:

- The input data type of the Discrete Filter block, which is a floating-point inheritance block, to a fixed-point data type to avoid floating-point inheritance.
- The output data type of all the other blocks that currently have their output data type set by inheritance rules to the compiled (current propagated) data type.

Tip Review the recommended data types prior to accepting them.

4 Fix the failure using the **Modify All** button to configure the output data types to the recommended values.

The Action Result box displays:

- A table showing the previous and current data types for all the floating-point inheritance blocks.
- A table showing the previous and current data types for blocks that use other types of inheritance.

5 Rerun the task.

The task passes.

6 Select and run **Relax input data type settings**. This task identifies blocks with input data type constraints that might cause data type propagation issues.

The task passes because the model contains no blocks that have inherited input data types.

7 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.

8 Select and run **Remove redundant specification between signal objects and blocks**. This task identifies and removes redundant data type specification originating from blocks and Simulink signal objects.

The task passes because the model contains no resolved Simulink signal objects.

9 Select and run **Verify hardware selection**. This task identifies the hardware device information in the **Hardware Implementation** pane of the Configuration Parameters dialog box. It also checks the default data type selected for floating-point signals in the model.

The task fails because the default data type for all floating-point signals is set to `Remain floating-point`. Because the target hardware is an embedded processor, the Fixed-Point Advisor recommends that you set this value to the hardware integer used by the embedded hardware.

10 To fix the failure, in the **Input Parameters** pane, set **Default data type of all floating-point signals** to `Same as embedded hardware integer`.

11 Rerun the task.

The task passes.

12 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.

The Fixed-Point Advisor warns you that you have not specified any minimum and maximum values.

13 Fix the warning by specifying minimum and maximum values for Inport blocks:

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.

The tool advises you to specify minimum and maximum values for all blocks if possible. For the purpose of this example, do not specify other minimum and maximum values for other 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, click the **Propose fraction lengths** button DT.

   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.
   - **Output data type** parameter specifies a generalized fixed-point number.
   - Data types are not inherited.

   The Fixed-Point Tool updates the results in the **Contents** pane.

2. In the Fixed-Point Tool, set the **Column View** to `Automatic Data Typing with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed data types in the **ProposedDT** column in the **Contents** pane. The tool does not propose data types for objects with inherited data types.

   To accommodate the full simulation range, the Fixed-Point Tool proposes data types for blocks that do not have inherited data types. 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 34-22.

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 34-17.

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 32-19.

Apply Fixed-Point Data Types

1 Click the **Apply accepted fraction lengths** button to write the proposed data types to the model. DT✓

   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 32-19.

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 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2 In the **Data collection** pane, set **Store results in run** 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.

3 Click the Fixed-Point Tool **Simulate** 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.

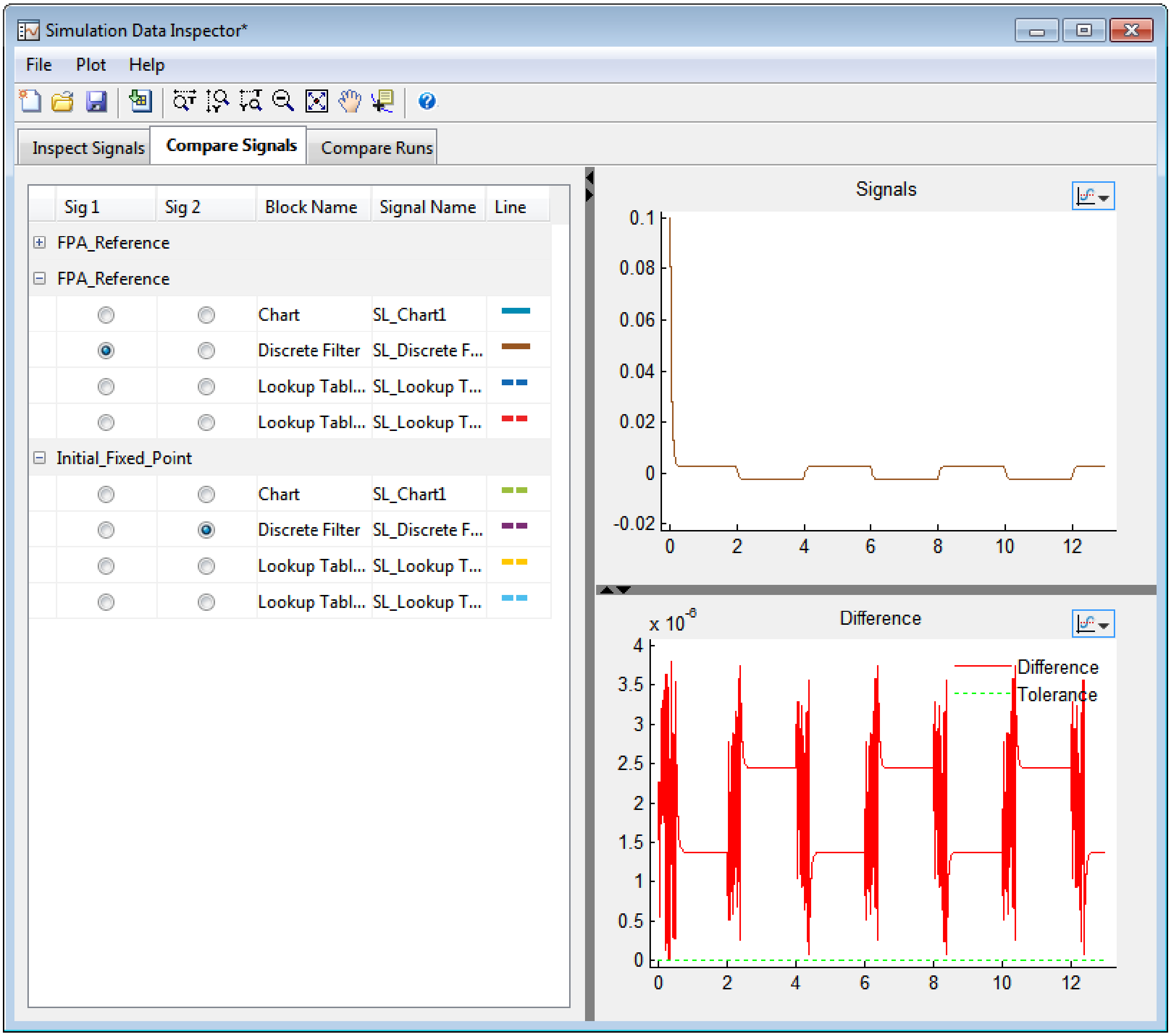
Tip In the **Contents** pane, click the **Run** column heading to sort the runs.

---

4 Examine the results to verify that there are no overflows or saturations.

5 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem. 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`.



Sig 1	Sig 2	Block Name	Signal Name	Line
⊞ FPA_Reference				
⊟ FPA_Reference				
○	○	Chart	SL_Chart1	━
●	○	Discrete Filter	SL_Discrete F...	━
○	○	Lookup Tabl...	SL_Lookup T...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍
⊟ Initial_Fixed_Point				
○	○	Chart	SL_Chart1	╍
○	●	Discrete Filter	SL_Discrete F...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍
○	○	Lookup Tabl...	SL_Lookup T...	╍

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 32-22.

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 **Model Hierarchy** pane , select the Controller Subsystem.

4. In the **Data collection** pane, set **Store results in run** to `Input2`.

5. Click the Fixed-Point Tool **Simulate** button ▶ to run the simulation.

   The Simulink software simulates the `ex_fixed_point_workflow` model. The Stateflow debugger reports a data overflow error in the Stateflow chart.

6. In the **Stateflow Debugging** window, under **Error checking options**, clear the **Data Range** option and close the debugger and the Chart.

   This action disables data range error detection and allows the simulation to run to completion.

7 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

   The Fixed-Point Tool **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.

8 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 32-23.

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 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2 In the **Settings for selected system** pane, set **Data type override** to `Double`.

   Using this 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.

3 In the **Data collection** pane, set **Store results in run** to `DTO_Input2`.

4. Click the Fixed-Point Tool **Simulate** button ▶ to run the simulation.

   The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.

5. 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 the **Propose fraction lengths** button DT.

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. In the Fixed-Point Tool, set the **Column View** to `Automatic Data Typing with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed data types in the **ProposedDT** column in the **Contents** pane. The tool does not propose data types for objects with inherited data types.

   To accommodate the full simulation range, the Fixed-Point Tool proposes new data types with reduced precision for the Chart/output and 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 32-25.

Apply the New Fixed-Point Data Types

1 Click **Apply accepted fraction lengths** to write the proposed data types to the model.

2 In the **Apply Data Types** dialog box, select `DTO_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 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2 In the **Settings for selected system** pane, set **Data type override** to `Use local settings`.

3 In the **Data collection** pane, set **Store results in run** to `Final_fixed_point`.

4 Click **Simulate** 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.

5 Examine the results to verify that there are no overflows or saturations.

6 In the Fixed-Point Tool **Model Hierarchy** pane , select the Controller Subsystem. In the **Contents** pane, select the `Discrete Filter:  Output` that corresponds to the `Initial_fixed_point` run, and then click the **Compare Signals** button.

7 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.

8 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.

Prepare for Code Generation

Optionally, use the Simulink Model Advisor to identify model settings that might lead to nonoptimal results in code generation.

1 From the Simulink **Analysis** menu, select **Model Advisor>Model Advisor**.

2 In the **System Selector** dialog box, select `Controller Subsystem`, and then click **OK**.

3 In the Model Advisor left pane, expand the **By Task** node.

4 Expand the **Code Generation Efficiency** node.

5 Select and run **Identify blocks that generate expensive rounding code**. This task optimizes the code to eliminate unnecessary rounding code.

   The Model Advisor warns that your model contains expensive rounding code.

6 Change the **Integer rounding mode** of the Look-Up Table1, Conversion1, and Conversion2 blocks from `Floor` to `Simplest`, and rerun the check.

   The task passes.

7 Select and run **Identify questionable fixed-point operations**. This task identifies fixed-point operations that can lead to nonoptimal results.

The Model Advisor warns that your model generates cumbersome multiplication code, and contains inefficient lookup blocks.

8 The Discrete Filter and Gain blocks contain expensive multiplication code.

   Under **Configuration Parameters>Hardware Implementation** change the **Device Type** to `64 bit Embedded Processor (LLP64)`, and select **Enable long long**.

9 The Model Advisor recommends that you change the breakpoints data of your lookup tables to have even power of 2 spacing, and use `Evenly-spaced points` index search method.

   Using the function `fixpt_look1_func_approx`, and following the steps outlined in "Fixed-Point Function Approximation", adjust the breakpoints data of the lookup table blocks.

10 Rerun the task.

   The task passes.

11 Select and run **Identify blocks that generate expensive fixed-point and saturation code**. This task identifies blocks that can lead to unnecessary saturation code.

   The Model Advisor warns that the model contains expensive saturation code.

12 Clear the Gain block's **Function Block Parameters** > **Signal Attributes** > **Saturate on integer overflow** parameter. Click apply, and close the window.

13 Rerun the task.

   The task passes.

Your model is now optimized for code generation.

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 30-2
Best practices for using the Fixed-Point Advisor	“Best Practices” on page 30-2
Using restore points in the Fixed-Point Advisor	“Restore Points” on page 30-7
Fixed-Point Tool capabilities	“Fixed-Point Tool” on page 31-2 `fxptdlg`
Best practices for using the Fixed-Point Tool	“Best Practices for Proposing Data Types” on page 34-5
Using the Fixed-Point Tool to merge multiple simulation results	“Propose Data Types Using Multiple Simulations” on page 34-67

Producing Lookup Table Data

- "Producing Lookup Table Data" on page 33-2
- "Worst-Case Error for a Lookup Table" on page 33-3
- "Create Lookup Tables for a Sine Function" on page 33-6
- "Use Lookup Table Approximation Functions" on page 33-24
- "Effects of Spacing on Speed, Error, and Memory Usage" on page 33-25

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 33-6 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 33-3 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

In this section...

"What Is the Worst-Case Error?" on page 33-3

"Approximate the Square Root Function" on page 33-3

What Is the Worst-Case Error?

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
```

Next, type

```
errworst = fixpt_look1_func_plot(xdata,ydata,funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

This returns the worst-case error of the lookup table as the variable `errworst`:

```
errworst =
    0.1250
```

It also generates the plots shown in the following figure. 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.

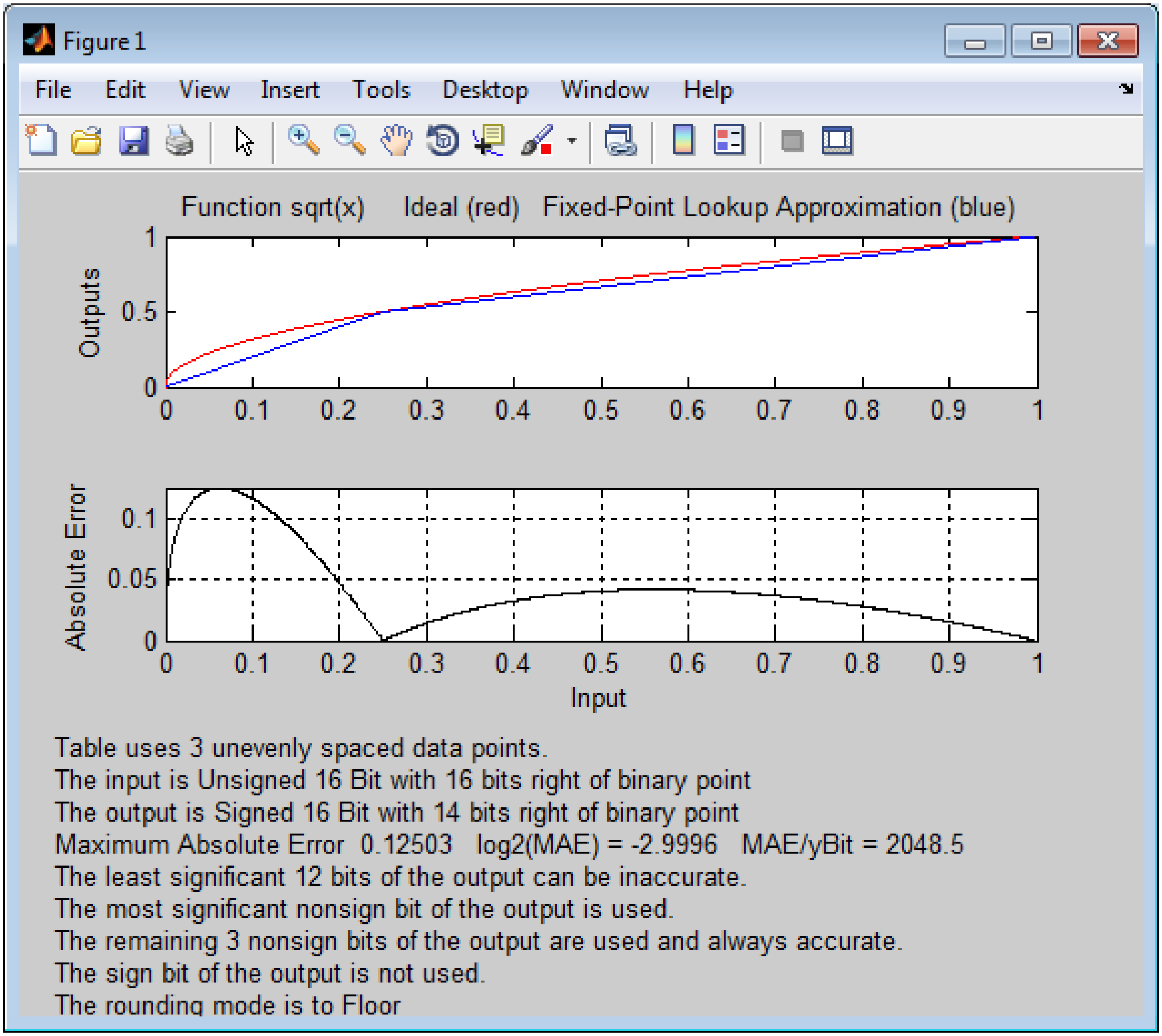


Table uses 3 unevenly spaced data points.
The input is Unsigned 16 Bit with 16 bits right of binary point
The output is Signed 16 Bit with 14 bits right of binary point
Maximum Absolute Error 0.12503   log2(MAE) = -2.9996   MAE/yBit = 2048.5
The least significant 12 bits of the output can be inaccurate.
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

Create Lookup Tables for a Sine Function

In this section...

"Introduction" on page 33-6

"Parameters for fixpt_look1_func_approx" on page 33-6

"Setting Function Parameters for the Lookup Table" on page 33-8

"Using errmax with Unrestricted Spacing" on page 33-8

"Using nptsmax with Unrestricted Spacing" on page 33-11

"Using errmax with Even Spacing" on page 33-14

"Using nptsmax with Even Spacing" on page 33-16

"Using errmax with Power of Two Spacing" on page 33-17

"Using nptsmax with Power of Two Spacing" on page 33-19

"Specifying Both errmax and nptsmax" on page 33-21

"Comparison of Example Results" on page 33-22

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(2πx) 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 33-8, 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 33-8.

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 worse case error. See “Using nptsmax with Unrestricted Spacing” on page 33-11.

The section “Specifying Both errmax and nptsmax” on page 33-21 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 33-13 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 33-8, 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(2π*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(2π*x*) to within the tolerance specified by errmax.

You can display the maximum error by typing errworst. This returns

```
errworst =
  9.7656e-004
```

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);
```

The resulting plots are shown.

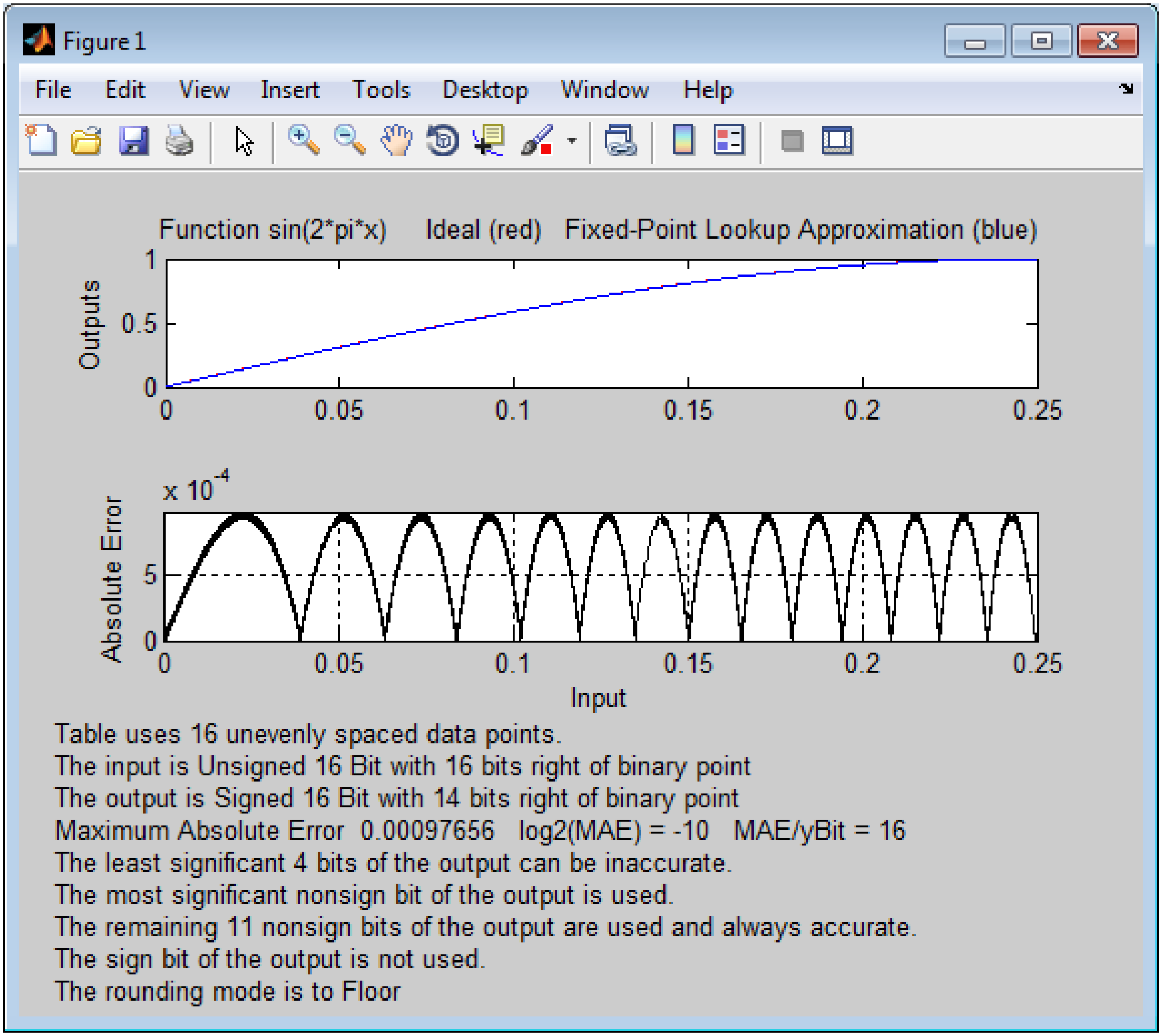


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10^-4
Absolute Error
5
0
0 0.05 0.1 0.15 0.2 0.25
Input

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 log2(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 33-8, 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. This returns

```
errworst =
  5.1139e-004
```

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,rndmeth);
```

The resulting plots are shown.

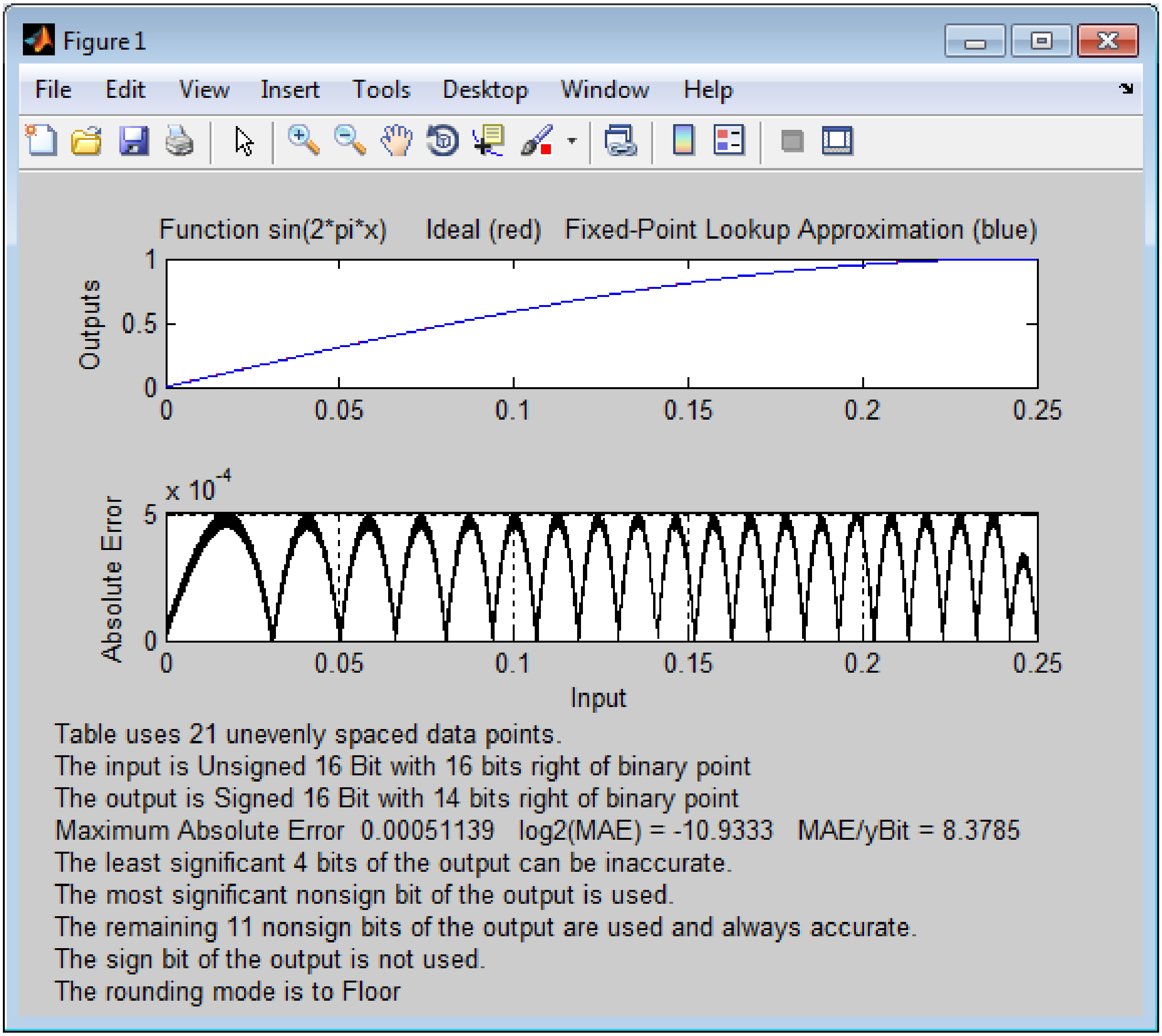


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10^-4
Absolute Error
5
0
0 0.05 0.1 0.15 0.2 0.25
Input

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 log2(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 33-25 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 33-8, 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);
```

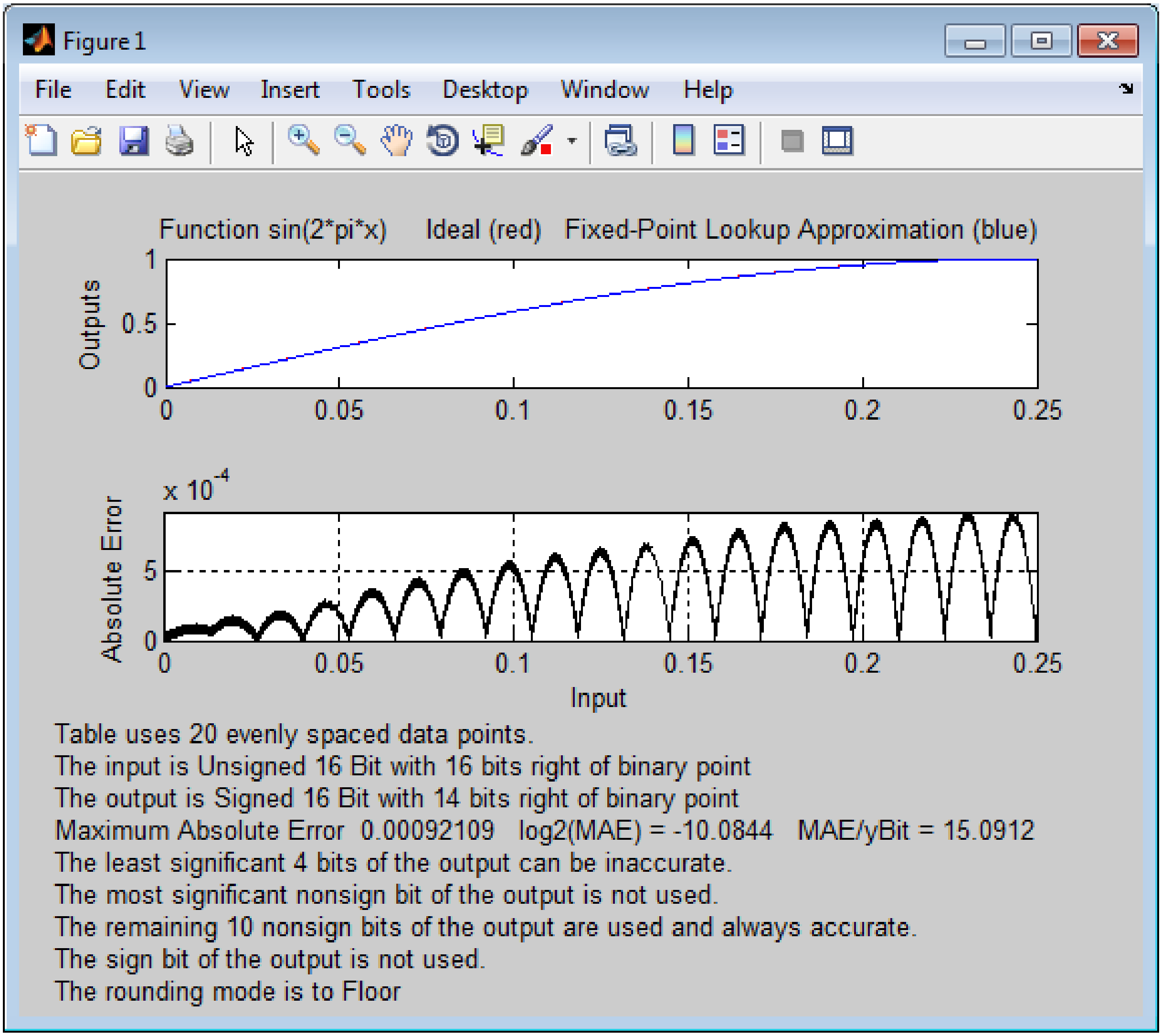
You can find the number of points in the lookup table by typing `length(xdata)`:

```
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);
```

This produces the following plots:



```
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   log2(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 33-8, 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);
```

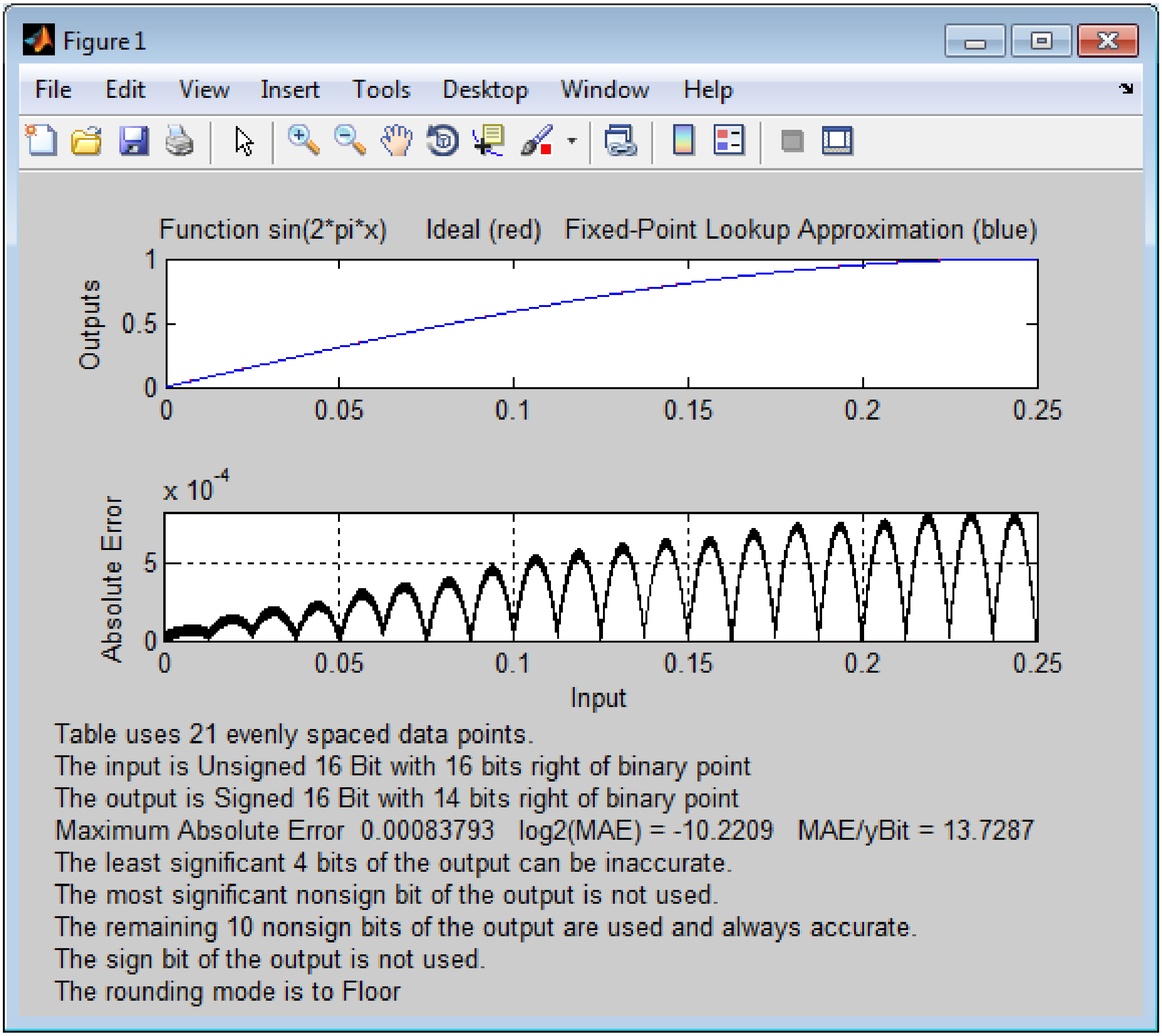


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10-4
Absolute Error
5
0
0 0.05 0.1 0.15 0.2 0.25
Input

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 log2(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 33-8, 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-004
```

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);
```

This displays the plots shown.

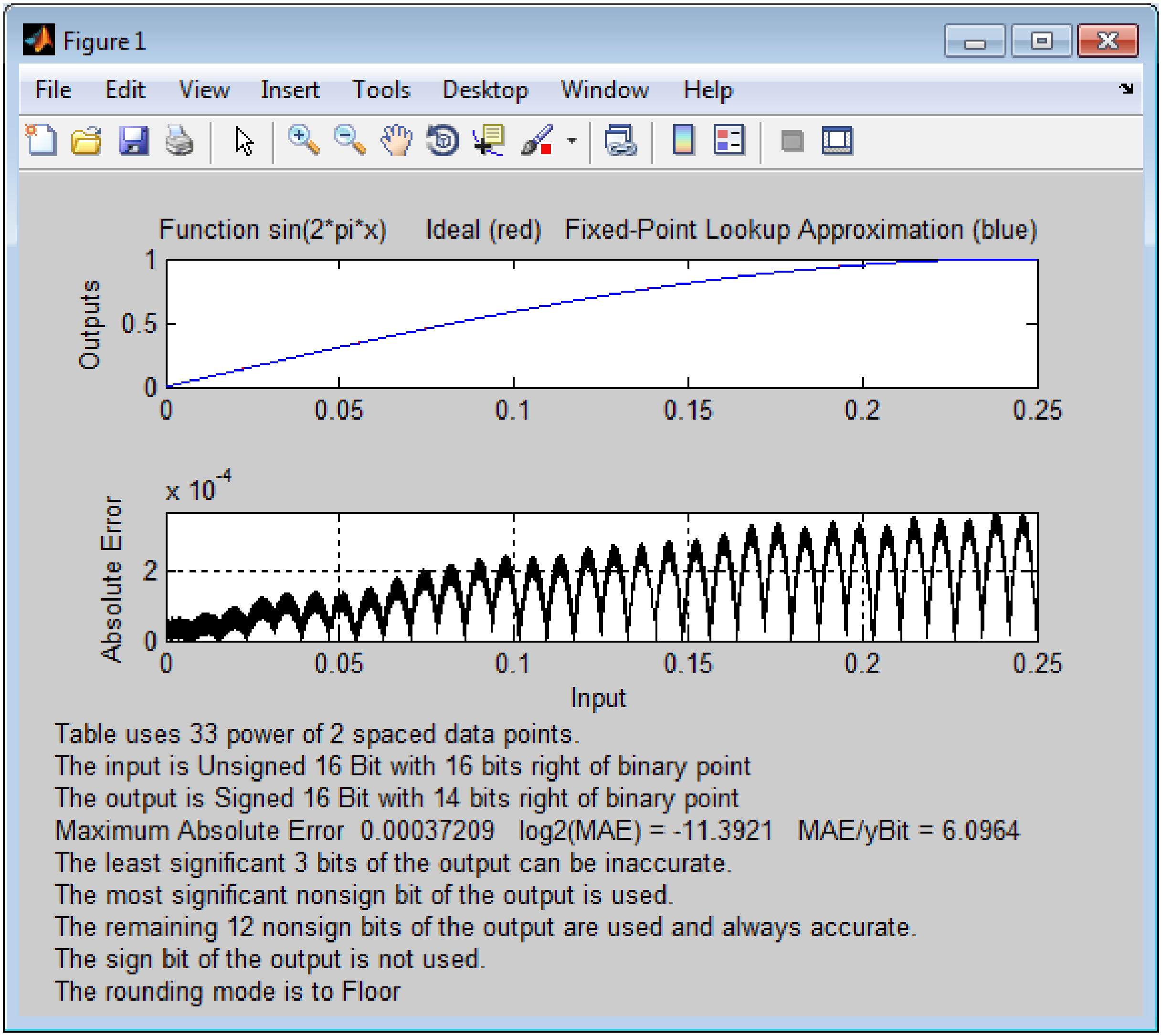


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10^-4
Absolute Error
2
0
0 0.05 0.1 0.15 0.2 0.25
Input

```
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   log2(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 nptsmax 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 33-8, 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,[],nptsmax,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);
```

This produces the plots shown below:

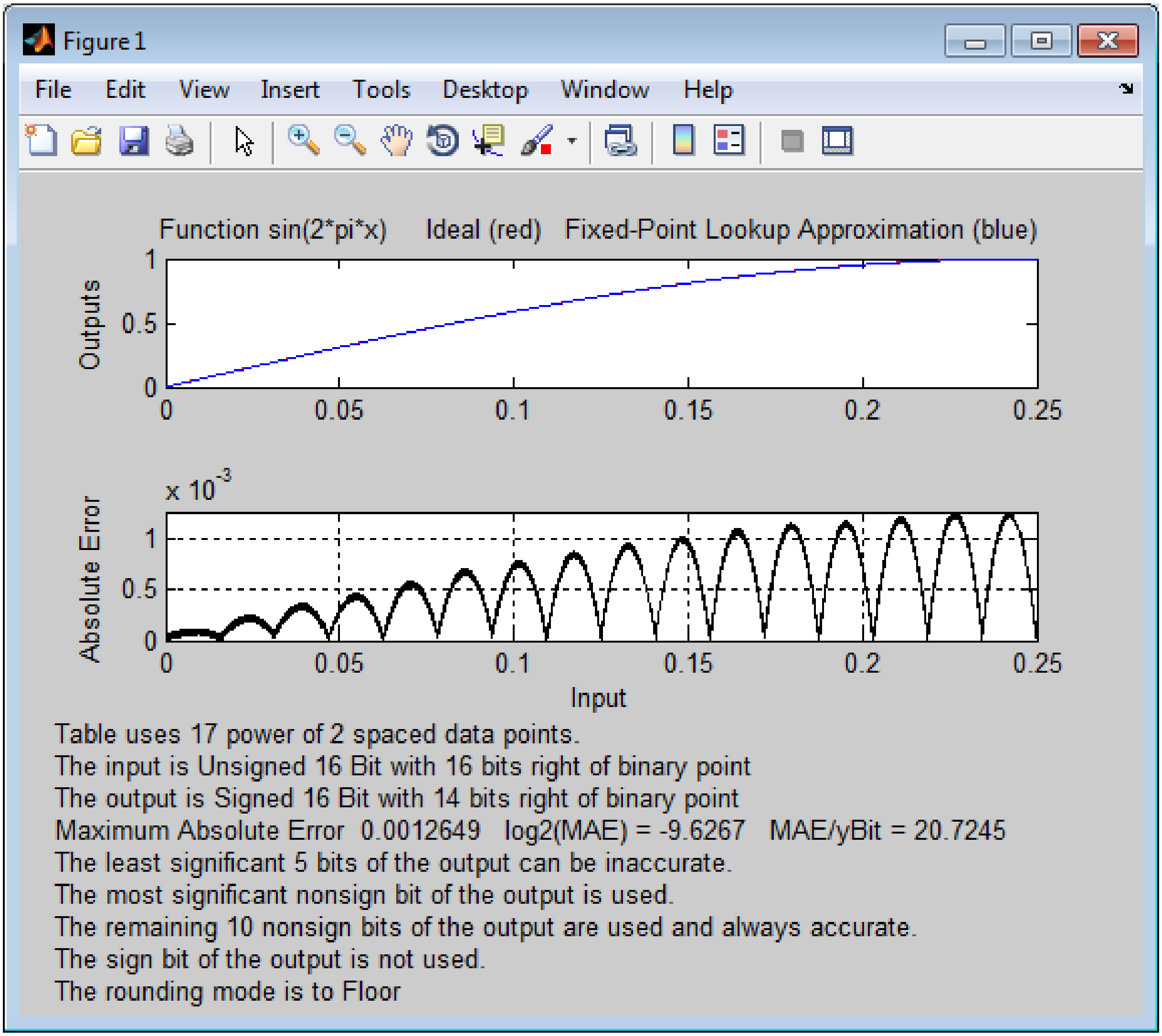


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sin(2*pi*x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
x 10^-3
Absolute Error
1
0.5
0
0 0.05 0.1 0.15 0.2 0.25
Input

```
Table uses 17 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  0.0012649   log2(MAE) = -9.6267   MAE/yBit = 20.7245
The least significant 5 bits of the output can be inaccurate.
The most significant nonsign bit of the output is 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
```

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 33-8, 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	`errmax=2^-10`	`'unrestricted'`	`2^-10`	16
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:
   1. The ideal function to approximate
   2. The range, `xmin` to `xmax`, over which to find X and Y data
   3. The fixed-point implementation: data type, scaling, and rounding method
   4. 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 33-25

"Model That Illustrates Effects of Breakpoint Spacing" on page 33-25

"Data ROM Required for Each Lookup Table" on page 33-26

"Determination of Out-of-Range Inputs" on page 33-27

"How the Lookup Tables Determine Input Location" on page 33-27

"Interpolation for Each Lookup Table" on page 33-29

"Summary of the Effects of Breakpoint Spacing" on page 33-32

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 33-32.

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;
iRght = 7; /* number of breakpoints minus 1 */

while ( ( iRght - iLeft ) > 1 )
{
  i = ( iLeft + iRght ) >> 1;

if ( uAngle < xuneven[i] )
  {
    iRght = i;
  }
  else
  {
    iLeft = i;
  }
}
```

The while loop executes up to log2(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[iRght] - yData[iLeft] ) * ( u - xData[iLeft] ) ...
    / ( xData[iRght] - xData[iLeft] ) + yData[iLeft]
```

The term `(xData[iRght] - 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[iRght] - xuneven[iLeft];
```

```
yDiff = yuneven[iRght] - yuneven[iLeft];

MUL_S32_S16_U16( bigProd, yDiff, xNum );

  DIV_NZP_S16_S32_U16_FLOOR( 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[iRght] - 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 log2(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.

34

Automatic Data Typing

- “About Automatic Data Typing” on page 34-2
- “Before Proposing Data Types for Your Model” on page 34-3
- “Best Practices for Proposing Data Types” on page 34-5
- “Models That Might Cause Data Type Propagation Errors” on page 34-7
- “Automatic Data Typing Using Simulation Data” on page 34-11
- “Automatic Data Typing Using Derived Ranges” on page 34-25
- “Propose Fraction Lengths” on page 34-39
- “Propose Word Lengths” on page 34-57
- “Propose Data Types Using Multiple Simulations” on page 34-67
- “View Simulation Results” on page 34-73
- “Viewing Results With the Simulation Data Inspector” on page 34-79

About Automatic Data Typing

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	• Useful if you know the inputs to use for the model. • You do not need to specify any design range information.	• 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.	• 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.	• Derivation might take a very long time.

Before Proposing Data Types for Your Model

Before you use the Fixed-Point Tool to propose data types your Simulink model, consider how automatic data typing affects your model:

- The Fixed-Point Tool proposes new data types for the fixed-point data types 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 Shortcut Editor.

  For more information, see "Best Practices for Proposing Data Types" on page 34-5.

- 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 a Block Diagram".

- 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 34-17.

  - 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 34-7.

Best Practices for Proposing Data Types

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 31-10.

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 31-12.

Isolate the System Under Conversion

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.

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.

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 34-17.

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 Proposing Data Types" on page 34-5.

The following models are likely to cause data type propagation issues.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
Buses	Does not detect the minimum, maximum, data type, and initial value information for bus objects and does not use them for automatic data typing. If you are using strict bus mode, the Fixed-Point Tool handles data type constraints introduced by connections using virtual buses that do not have any associated bus objects. The data type proposals take into account these constraints.	Fixed-Point Tool might propose data types that are inconsistent with the data types for the bus object or generate proposals that cause overflows.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
	How to enable strict bus mode To enable strict bus mode, set **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** to `error`.	
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.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
User-defined S-functions	Cannot detect the operation of user-defined S-functions.	• 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 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.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
Linked subsystems	Does not include linked subsystems when proposing data types.	Data type mismatch errors might occur at the linked subsystem boundaries.
MATLAB Function blocks	Does not propose data types for MATLAB Function blocks.	Fixed-Point Tool might propose data types that are inconsistent with the requirements of the MATLAB Function blocks. The proposed data types might cause data type mismatch errors or overflows.

Automatic Data Typing Using Simulation Data

In this section...

"Workflow for Automatic Data Typing Using Simulation Data" on page 34-11

"Set Up the Model" on page 34-12

"Prepare the Model for Conversion" on page 34-13

"Gather a Floating-Point Benchmark" on page 34-14

"Proposing Data Types" on page 34-15

"Propose Data Types" on page 34-17

"Examine Results to Resolve Conflicts" on page 34-17

"Apply Proposed Data Types" on page 34-22

"Verify New Settings" on page 34-22

"Automatic Data Typing of Simulink Signal Objects" on page 34-23

Workflow for Automatic Data Typing Using Simulation Data

1. Set up the model
2. Prepare the model for conversion
3. Run the model to gather floating-point benchmark
4. Propose data types
5. Examine results to resolve conflicts
6. Apply proposed data types
7. Verify new settings

Set Up the Model

To use the Fixed-Point Tool to generate data type proposals for your model based on simulation minimum and maximum values only, you must first set up your model in Simulink.

1 Back up your model in case of error and as a baseline for testing and validation.

2 Open your model in Simulink.

3 From the Simulink menu, 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 blocks.

   You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. For a list of blocks that permit you to specify these values, see "Blocks That Allow Signal Range Specification".

5 Specify fixed-point data types for blocks and signals in your model. For blocks with the **Data Type Assistant**, use the **Calculate Best-Precision Scaling** button to calculate best-precision scaling automatically. For more information, see "Specify Fixed-Point Data Types with the Data Type Assistant" on page 26-24. Use the Fixed-Point Advisor to prepare your model for conversion to an equivalent fixed-point representation For more information, see "Preparation for Fixed-Point Conversion" on page 30-2.

6 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 the **Lock output data type setting against changes by the fixed-point tools** parameter, the tool does not propose data types for that object.

7 From the Simulink **Simulation** menu, select **Update Diagram** to perform parameter range checking for all blocks in the model.

If **update diagram** fails, 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.

8. If the model changed, back up the model again in case of error and as a baseline for testing and validation.

9. Create a shortcut to capture the initial fixed-point instrumentation and data type override settings. For more information, see "Capture Model Settings Using the Shortcut Editor" on page 31-10.

Prepare the Model for Conversion

First use the Fixed-Point Advisor to prepare the model for conversion to fixed point. You do this preparation only once. The Fixed-Point Advisor gives advice about model and block configuration settings to prepare for automatic conversion to fixed point using the Fixed-Point Tool. The Fixed-Point Advisor:

- Checks the model against fixed-point guidelines.
- Identifies unsupported blocks.
- Removes output data type inheritance from blocks.

The Fixed-Point Advisor also makes recommendations for a model, such as model-level diagnostic settings and removal of inheritance rules. It configures the model for autoscaling by the Fixed-Point Tool. Therefore, even if your model uses only fixed-point data types, it is useful to run the Fixed-Point Advisor on the model.

To open the Fixed-Point Advisor:

1. From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

2. On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

3. On the **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.

   Use the Fixed-Point Advisor to prepare the model for conversion. For more information, see "Preparation for Fixed-Point Conversion" on page 30-2.

Gather a Floating-Point Benchmark

First, run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This action 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. From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

2. Enable signal logging for the system or subsystem of interest. Using the Fixed-Point Tool you can enable signal logging for multiple signals simultaneously. For more information, see "Signal Logging Options" in the `fxptdlg` Reference.

   To enable signal logging:

   a. On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem.

   b. Right-click the selected system to open the context menu.

   c. Use the **Enable Signal Logging** option to enable signal logging, as necessary.

   The **Contents** pane of the Fixed-Point Tool displays an antenna icon next to items that have signal logging enabled.

   Note You can plot results only for signals that have signal logging enabled.

3. On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem for which you want a proposal.

4. On the **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** 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 **Data collection** pane **Store results in run** field) to `DoubleOverride`

The Fixed-Point Tool performs a global override of the fixed-point data types with double-precision data types, thus avoiding quantization effects. During simulation, the tool logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `DoubleOverride`.

Note Data type override does not apply to `boolean` or enumerated data types.

---

5 Click the Fixed-Point Tool **Simulate** button ▶ to run the simulation.

The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.

Proposing Data Types

Unless you select an object's **Lock output data type setting against changes by the fixed-point tools** parameter or the data types are using inheritance rules, the Fixed-Point Tool proposes data types for model objects that specify fixed-point data types.

When proposing data types, the Fixed-Point Tool collects the following types of range data for model objects:

- Design minimum or maximum values — You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. For a list of blocks that permit you to specify these values, see "Blocks That Allow Signal Range Specification".
- Simulation minimum or maximum values — When simulating a system whose **Fixed-point instrumentation mode** parameter specifies `Minimums, maximums and overflows`, 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 `fxptdlg`.

- 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 for the model to derive range information for signals in your model. For more information, see “Derive Ranges”.

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 and derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the **Configure** link.

- The tool observes the derived range only when the **Derived min/max** option is selected. Otherwise, the tool ignores the derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the derived range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the **Configure** link.

- The tool observes the simulation range only when the **Simulation min/max** option is selected. Otherwise, the tool ignores the simulation range.

  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the **Configure** link.

Propose Data Types

1. In the **Automatic data typing for selected system Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**.

   If these options are not visible, use the **Configure** link to display them.

2. To use simulation min/max information only, clear **Derived min/max**.

3. If you have safety margins to apply:

   a. Enter **Safety margin for design and derived min/max (%)**, if applicable. For example, enter `10` for a 10% safety margin. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the **Configure** link.

   b. Enter **Safety margin for simulation min/max (%)**, if applicable.

4. Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

   Note When the Fixed-Point Tool proposes data types, it does not alter your model.

   If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box.

   If you do not see this warning, there are no conflicts in your model. Go to "Apply Proposed Data Types" on page 34-22.

Examine Results to Resolve Conflicts

You can examine each proposal using the **Result Details** dialog box, which displays the rationale underlying the proposed data types. Also, this dialog box describes potential issues or errors, and it suggests methods for resolving them. To open the dialog box:

1. On the **Contents** pane, select an object that has proposed data types.

2. Click the **Show details for selected result** button ℹ.

The **Result Details** dialog box provides the following information about the proposed data types, as appropriate.

Summary

Details which run the result is in and the current data type specified for the selected object.

Proposed Data Type Summary

Describes a proposal in terms of how it differs from the object's current data type. 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.

Needs Attention

Lists potential issues and errors associated with data type proposals. It describes the issues and suggests methods for resolving them. The dialog box uses the following icons to differentiate warnings from errors:

⚠	Indicates a warning message.
✖	Indicates an error message.

Shared Data Type Summary

This section of the dialog box informs you that the selected object must share 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 share the same data type. Similarly, blocks that are connected by the same element of a virtual bus must share the same data type.

The dialog box provides a hyperlink that you can click to highlight the objects that share data types in the model. To clear this highlighting, from the model **View** menu, select **Remove Highlighting**.

The Fixed-Point Tool allocates an identification tag to objects that must share the same data type. The tool displays this identification tag in the **DTGroup** column for the object. To display only the objects that must share data types, from the Fixed-Point Tool main toolbar, select the **Show** option.

Constrained Data Type Summary

Some Simulink blocks accept only certain data types on some ports. This section of the dialog box informs you when a block that connects to the selected object has data type constraints that impact the proposed data type of the selected object. The dialog box lists the blocks that have data type constraints, provides details of the constrained data types, and links to the blocks in the model.

Data Type Details

Provides a table with model object attributes that influence its data type proposal.

Item	Description
Currently Specified Data Type	Data type that an object specifies.
Proposed Data Type	Data type that the Fixed-Point Tool proposes for this object.
Proposed Representable Maximum	Maximum value that the proposed data type can represent.
Design Maximum	Design maximum value that an object specifies using, e.g., its **Output maximum** parameter.
Simulation Maximum	Maximum value that occurs during simulation.
Simulation Minimum	Minimum value that occurs during simulation.

Item	Description
Design Minimum	Design minimum value that an object specifies using, e.g., its **Output minimum** parameter.
Proposed Representable Minimum	Minimum value that the proposed data type can represent.

The dialog box table also includes a column titled **Percent Proposed Representable**. This column indicates the percentage of the proposed representable range that each value covers. Overflows occur when values lie outside this range.

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 provides an option that requires all of its inputs to have the same data type. Consequently, the dialog box table might also list attributes of other model objects that impact the proposal for the selected object. In such cases, the table displays the following types of shared values:

- **Initial Values**

  Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes 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. When data type dependencies exist, the tool considers how initial values impact the proposals for neighboring objects.

- **Model-Required Parameters**

  Some model objects require the specification of 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 “model-required” parameter value impacts the proposals for neighboring objects.

To Examine the Results and Resolve Conflicts

1 On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **Conflicts with proposed data types**.

The Fixed-Point Tool lists its data type proposals on the **Contents** pane under the **ProposedDT** column. The tool alerts you to potential issues for each object in the list by displaying a green, yellow, or red icon.

✓	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.

a Select the error, right-click and select **Highlight Block In Model** from the context menu to identify which block has a conflict.

b Click the **Show details for selected result** button i to open the **Result Details** dialog box.

c Use the information provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.

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 **Start** button ▶ to rerun the simulation.

The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.

5 Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

6 On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **All results**.

Apply Proposed Data Types

After reviewing the data type proposals, apply the proposed data types to your model. The Fixed-Point Tool allows you to apply its 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 check box indicates the status of a proposal:

☑ The Fixed-Point Tool will apply the proposed data type to this object. By default, the tool selects the **Accept** check box when a proposal differs from the object's current data type.

□ The Fixed-Point Tool will ignore the proposed data type and leave the current data type intact for this object.

▨ No proposal exists for this object, for example, when the object specifies a data type inheritance rule or is locked against automatic data typing.

1 Examine each result. For more information about a particular result, select the result and then click the **Show details for selected result** button ℹ to display the **Result Details** dialog box.

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 with the Fixed-Point Tool. 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 Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button DT✓ to write the proposed data types to the model.

   If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.

Verify New Settings

After applying proposed data types to your model, you simulate the model using the applied fixed-point data types.

1 On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem for which you want a proposal.

2 On the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings.

   This sets:

   - **Data type override** to `Use local settings`.
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.
   - The run name (in the **Data collection** pane **Store results in run** field) to `NoOverride`.

   Using these settings, the Fixed-Point Tool simulates the model using the new fixed-point settings and logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `NoOverride`.

3 Click the Fixed-Point Tool **Start** button ▶ to run the simulation.

4 Compare the ideal results stored in the **DoubleOverride** run 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 Signals** 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.

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 automatically 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.

Automatic Data Typing Using Derived Ranges

In this section...

"Prerequisites for Autoscaling Using Derived Ranges" on page 34-25

"Workflow for Autoscaling Using Derived Data" on page 34-25

"Set Up the Model" on page 34-26

"Prepare Model for Autoscaling Using Derived Data" on page 34-27

"Derive Minimum and Maximum Values" on page 34-28

"Resolve Range Analysis Issues" on page 34-30

"Proposing Data Types" on page 34-30

"Propose Data Types" on page 34-32

"Examine Results to Resolve Conflicts" on page 34-33

"Apply Proposed Data Types" on page 34-37

"Update Diagram" on page 34-38

Prerequisites for Autoscaling Using Derived Ranges

The Fixed-Point Tool uses range analysis to derive minimum and maximum values for objects in your model.

Range analysis:

- Requires a Fixed-Point Designer license.
- Works only for compatible models that use real signals. For more information, see "Model Compatibility with Range Analysis" on page 35-6.

Workflow for Autoscaling Using Derived Data

1. Verify that your model is compatible with range analysis. See "Model Compatibility with Range Analysis" on page 35-6.
2. Set up model.
3. Prepare model prior to automatic data typing using derived data.

4 Derive minimum and maximum values.

5 Resolve any issues.

6 Derive minimum and maximum values.

7 Propose data types.

8 Examine results to resolve conflicts.

9 Apply proposed data types.

10 Update diagram.

Set Up the Model

To use the Fixed-Point Tool to generate data type proposals for your model based on derived minimum and maximum values only, you must first set up your model in Simulink.

1 Back up your model in case of error and as a baseline for testing and validation.

2 Open your model in Simulink.

3 Select **Simulation > Normal** in the Simulink menu so that the model runs in **Normal** mode. The Fixed-Point Tool supports only **Normal** mode.

4 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 typically titled **Output minimum** and **Output maximum**. For a list of blocks that permit you to specify these values, see “Blocks That Allow Signal Range Specification”.

5 Specify fixed-point data types for blocks and signals in your model. For blocks with the **Data Type Assistant**, use the **Calculate Best-Precision Scaling** button to calculate best-precision scaling automatically. For more information, see “Specify Fixed-Point Data Types with the Data Type Assistant” on page 26-24.

6 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 the **Lock output data type setting against changes by the fixed-point tools** parameter, the tool does not propose data types for that object.

7 From the Simulink **Simulation** menu, select **Update Diagram** to perform parameter range checking for all blocks in the model.

   If **update diagram** fails, 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.

8 If the model changed, back up the model in case of error and as a baseline for testing and validation.

9 Create a shortcut to capture the initial fixed-point instrumentation and data type override settings. For more information, see “Capture Model Settings Using the Shortcut Editor” on page 31-10.

Prepare Model for Autoscaling Using Derived Data

First use the Fixed-Point Advisor to prepare the model for conversion to fixed point. You do this preparation only once. The Fixed-Point Advisor gives advice about model and block configuration settings to prepare for automatic conversion to fixed point using the Fixed-Point Tool. The Fixed-Point Advisor:

- Checks the model against fixed-point guidelines.
- Identifies unsupported blocks.
- Removes output data type inheritance from 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 37-36.
- Runs simulation range detection diagnostics. When preparing the model for automatic data typing using derived data, you can complete the preparation without setting up signal logging and creating a simulation reference run. However, creating at least one simulation run is useful for early error detection. Simulating the model helps to verify that the design minimum and maximum values specified on the model are correct and that the model conforms to modeling guidelines.

To open the Fixed-Point Advisor:

1. From the Simulink **Analysis** menu, select **Fixed-Point Tool**.
2. On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.
3. On the **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.

   Use the Fixed-Point Advisor to prepare the model for conversion. For more information, see “Preparation for Fixed-Point Conversion” on page 30-2.

Derive Minimum and Maximum Values

1. On the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.
2. On the **Settings for selected system** pane, set **Data type override** to `Double`.
3. Optionally, in the **Data collection** pane **Store results in run** field, specify a run name. Specifying a unique run name avoids overwriting results from previous runs.
4. In the Fixed-Point Tool, click **Derive min/max values for selected system**.

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. (See “View Derived Ranges in the Fixed-Point Tool” on page 35-12.) 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 34-30.

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 35-6
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 35-28
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.	"Providing More Design Range Information" on page 35-25

Proposing Data Types

Unless you select an object's **Lock output data type setting against changes by the fixed-point tools** parameter or the data types are using inheritance rules, the Fixed-Point Tool proposes data types for model objects that specify fixed-point data types. You set up the tool to either propose fraction lengths for specified word lengths or to propose word lengths for specified fraction lengths. For more information, see "Propose Fraction Lengths" on page 34-39 and "Propose Word Lengths" on page 34-57.

When generating data type proposals, the Fixed-Point Tool collects the following types of range data for model objects:

- Design minimum or maximum values — You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. For a list of blocks that permit you to specify these values, see “Blocks That Allow Signal Range Specification”.
- Simulation minimum or maximum values — When simulating a system whose **Fixed-point instrumentation mode** parameter specifies `Minimums, maximums and overflows`, 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 `fxptdlg`.
- 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 for the model to derive range information for signals in your model. For more information, see “Derive Ranges”.

  For models that contain floating-point operations, range analysis might report a range that is slightly larger than expected due to rounding errors in the analysis. Automatic data typing bases its proposal on this slightly larger derived range. To avoid this issue, use the safety margin for design and derived min/max.

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 and derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the **Configure** link.

  For more information, see “Safety margin for design and derived min/max (%)” in the `fxptdlg` reference.

- The tool observes the derived range only when the **Derived min/max** option is selected. Otherwise, the tool ignores the derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the derived range. For more information, see Percent safety margin for design and derived min/max in the `fxptdlg` reference.

- The tool observes the simulation range only when the **Simulation min/max** option is selected. Otherwise, the tool ignores the simulation range.

  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For more information, see “Safety margin for simulation min/max (%)” in the `fxptdlg` reference.

Propose Data Types

1. On the **Automatic data typing for selected system Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**, as applicable.

   If these options are not visible, use the **Configure** link to display them.

2. If you have a safety margin to apply, set **Safety margin for design and derived min/max (%)**. For example, enter `10` for a 10% safety margin.

3. Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

   Note When the Fixed-Point Tool proposes data types, it does not alter your model.

   If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box.

   If you do not see this warning, there are no conflicts in your model, go to “Apply Proposed Data Types” on page 34-22.

Examine Results to Resolve Conflicts

You can examine each data type proposal using the **Result Details** dialog box, which displays the rationale underlying the proposal. Also, this dialog box describes potential issues or errors, and provides methods for resolving them. To open the dialog box:

1. On the **Contents** pane, select an object that has proposed data types.

2. Click the **Show details for selected result** button ⓘ.

The **Result Details** dialog box provides the following information about the proposed data type, as appropriate.

Summary

Details about which run the result is in and the current data type specified for the selected object.

Proposed Data Type Summary

Describes a data type proposal in terms of how it differs from the object's current data type. For cases when the Fixed-Point Tool does not propose data types, provides a rationale. For example, the data type might be locked against changes by the fixed-point tool.

Needs Attention

Lists potential issues and errors associated with data type proposals. Describes the issues and suggests methods for resolving them. The dialog box uses the following icons to differentiate warnings from errors.

⚠	Indicates a warning message.
✖	Indicates an error message.

Shared Data Type Summary

This section of the dialog box informs you that the selected object must share 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 share the same data type.

**The Shared Data Type Summary also reports all source blocks that drive the same element of a bus object. **

The dialog box provides a hyperlink that you can click to highlight the objects that share data types in the model. To clear this highlighting, from the model **View** menu, select **Remove Highlighting**.

The Fixed-Point Tool allocates an identification tag to objects that must share the same data type. The tool displays this identification tag in the **DTGroup** column for the object. To display only the objects that must share data types, from the Fixed-Point Tool main toolbar, select the **Show** option.

Constrained Data Type Summary

Some Simulink blocks accept only certain data types on some ports. This section of the dialog box informs you when a block that connects to the selected object has data type constraints that impact the proposed data type of the selected object. The dialog box lists the blocks that have data type constraints, provides details of the constrained data types, and links to the blocks in the model.

Data Type Details

Provides a table that lists a model object attributes that influence its data type proposal.

Item	Description
Currently Specified Data Type	Data type that an object specifies.
Proposed Data Type	Data type that the Fixed-Point Tool proposes for this object.
Proposed Representable Maximum	Maximum value that the proposed data type can represent.

Item	Description
Design Maximum	Design maximum value that an object specifies using, e.g., its **Output maximum** parameter.
Simulation Maximum	Maximum value that occurs during simulation.
Simulation Minimum	Minimum value that occurs during simulation.
Design Minimum	Design minimum value that an object specifies using, e.g., its **Output minimum** parameter.
Proposed Representable Minimum	Minimum value that the proposed data type can represent.

The dialog box table also includes a column titled **Percent Proposed Representable**. This column indicates the percentage of the proposed representable range that each value covers. Overflows occur when values lie outside this range.

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 provides an option that requires all of its inputs to have the same data type. Consequently, the dialog box table might also list attributes of other model objects that impact the data type proposal for the selected object. In such cases, the table displays the following types of shared values:

- **Initial Values**

  Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes 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. When data type dependencies exist, the tool considers how initial values impact the proposals for neighboring objects.

- **Model-Required Parameters**

Some model objects require the specification of 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 "model-required" parameter value impacts the proposals for neighboring objects.

To Examine the Results and Resolve Conflicts

1 On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **Conflicts with proposed data types**.

   The Fixed-Point Tool lists its data type proposals on the **Contents** pane under the **ProposedDT** column. The tool alerts you to potential issues for each object in the list by displaying a green, yellow, or red icon.

   ✓ 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.

   a Select the error, right-click, and from the context menu, select **Highlight Block In Model** to identify which block has a conflict.

   b Click the **Show details for selected result** button ℹ to open the **Result Details** dialog box.

   c Use the information provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.

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 **Start** button ▶ to rerun the simulation.

The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.

5. Click the **Propose fraction lengths** or **Propose word lengths** button to generate a data type proposal, DT.

6. On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **All results**.

Apply Proposed Data Types

After reviewing the data type proposals, apply the proposed data types to your model. The Fixed-Point Tool allows you to apply its 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 check box indicates the status of a proposal:

☑ The Fixed-Point Tool will apply the proposed data type to this object. By default, the tool selects the **Accept** check box when a proposal differs from the object's current data type.

□ The Fixed-Point Tool will ignore the proposed data type and leave the current data type intact for this object.

▨ No proposal exists for this object, for example, when the object specifies a data type inheritance rule or is locked against automatic data typing.

1. Examine each result. For more information about a particular result, select the result and then click the **Show details for selected result** button ℹ to open the **Result Details** dialog box.

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, the Fixed-Point Tool enables you to 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. Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button DT✓ to write the proposed data types to the model.

   If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.

Update Diagram

From the model's **Simulation** menu, select **Update Diagram**.

After applying the data types to the model, **update diagram** to check for data type propagation issues.

If **update diagram** fails, 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 backed up model.

Propose Fraction Lengths

In this section...

"Propose Fraction Lengths" on page 34-39

"About the Feedback Controller Example Model" on page 34-40

"Propose Fraction Lengths Using Simulation Range Data" on page 34-46

Propose Fraction Lengths

1. On the Fixed-Point Tool **Automatic data typing for selected system** pane, select **Propose fraction lengths for specified word lengths**. If you cannot see this option, click **Configure** to display more options.

2. On the same pane:

   - For simulation min/max information only, clear **Derived min/max**.
   - For derived min/max information only, clear **Simulation min/max**.

3. If you have safety margins to apply, set **Safety margin for design and derived min/max (%)** and **Safety margin for design and derived min/max (%)**, as applicable.

4. Click the **Propose fraction lengths** button, DT.

Note When the Fixed-Point Tool proposes data types, it does not alter your model.

If there are conflicts in your model, the Fixed-Point Tool opens the **Result Details** dialog box.

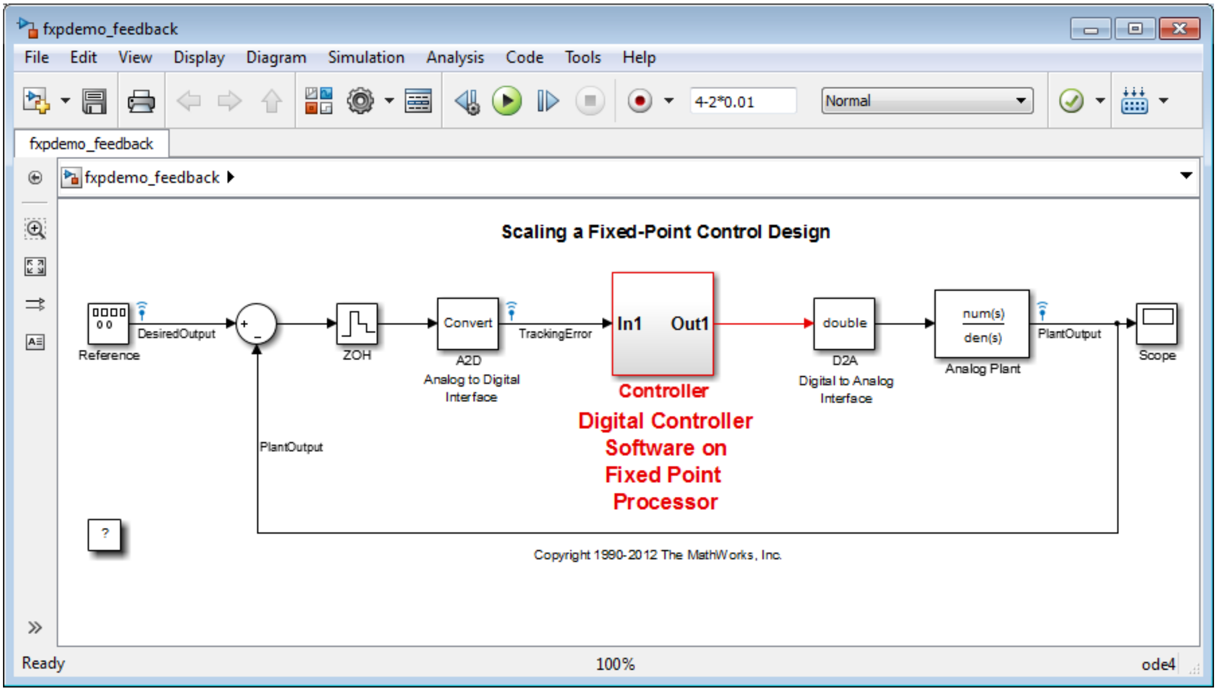
If you do not see this warning, there are no conflicts in your model. Review the proposed word lengths,

About the Feedback Controller Example Model

- "Opening the Feedback Controller Model" on page 34-40
- "Simulation Setup" on page 34-41
- "Idealized Feedback Design" on page 34-42
- "Digital Controller Realization" on page 34-43

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 34-43.

- **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.

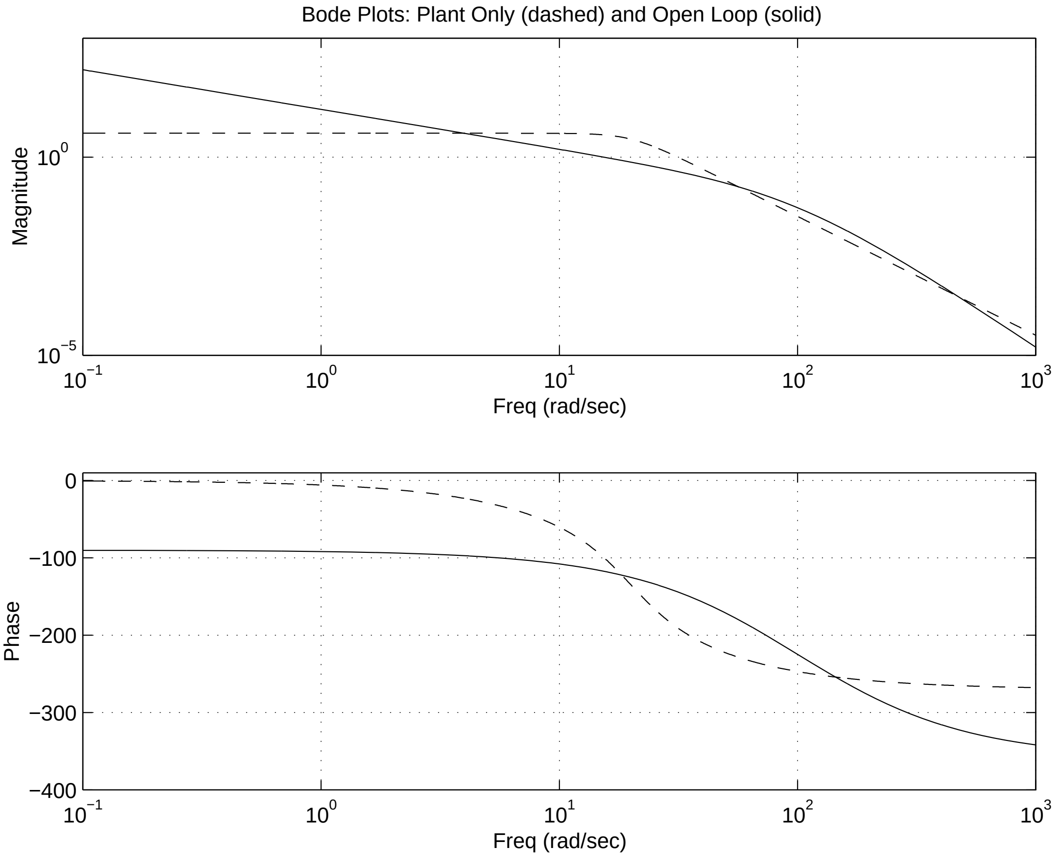
   You evaluate the behavior of the plant and the controller with a Bode plot. This evaluation is idealized, because all numbers, operations, and states are double-precision.

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

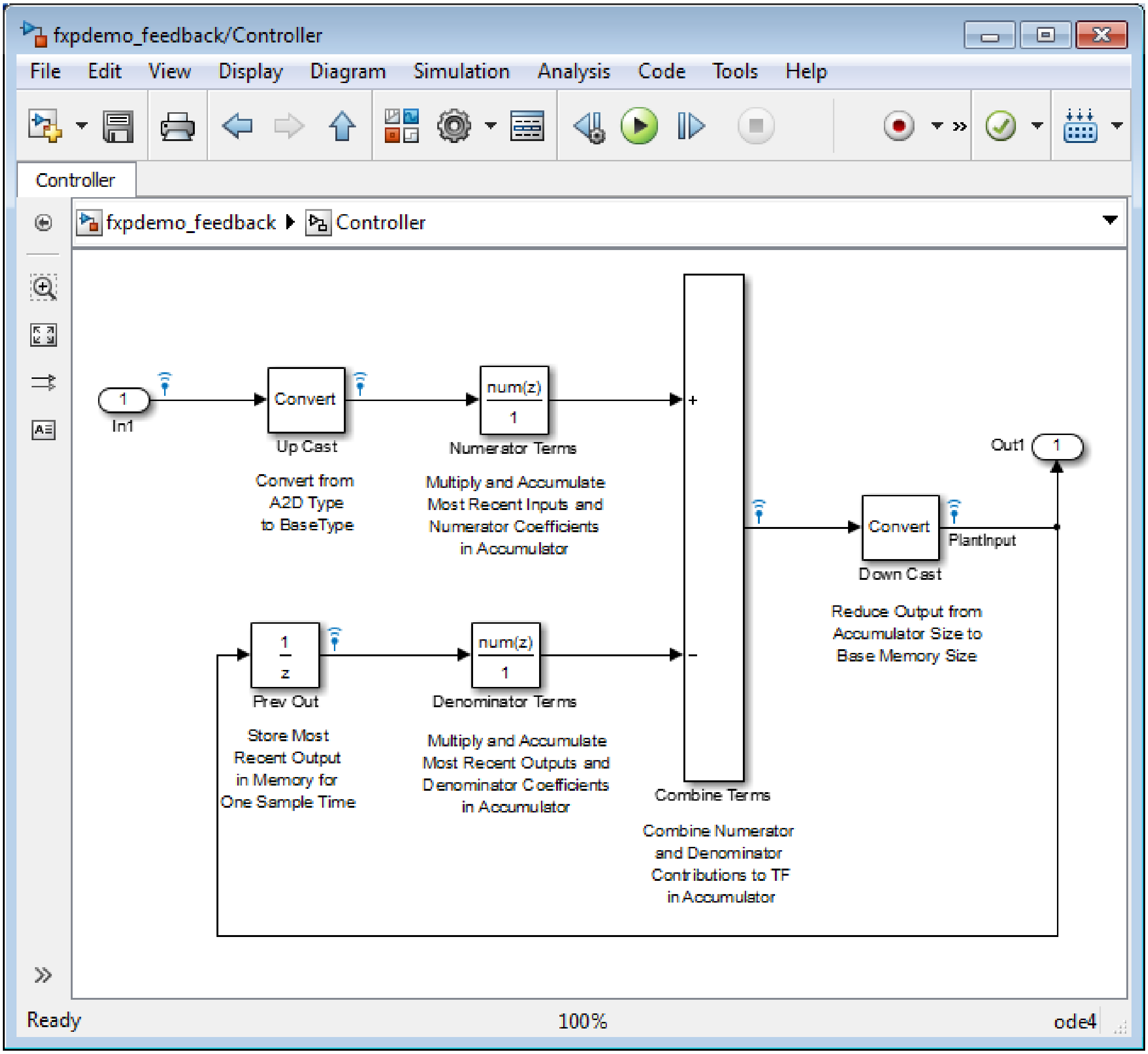
Open loop (controller and plant) and plant-only Bode plots for the “Scaling a Fixed-Point Control Design” model are shown in the following figure. The open loop Bode plot results from a digital controller described in the idealized world of continuous time, double-precision coefficients, storage of states, 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 34-47
- "Data Type Override" on page 34-49
- "Automatic Data Typing" on page 34-51

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 34-40). 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 34-47. Run an initial "proof of concept" 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 34-49. 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 34-51. 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 both the `fxpdemo_feedback` model and the Fixed-Point Tool.

2. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to set:

   - **Data type override** to `Use local settings`. This option enables each of the model's subsystems to use its locally specified data type settings.
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.
   - The run name to `NoOverride`.

3. In the Fixed-Point Tool, click the **Simulate** button ▶.

   The Simulink software simulates the `fxpdemo_feedback` model. Afterward, 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" in the `fxptdlg` function reference. For more information about customizing views, see "Control Model Explorer Contents Using Views".

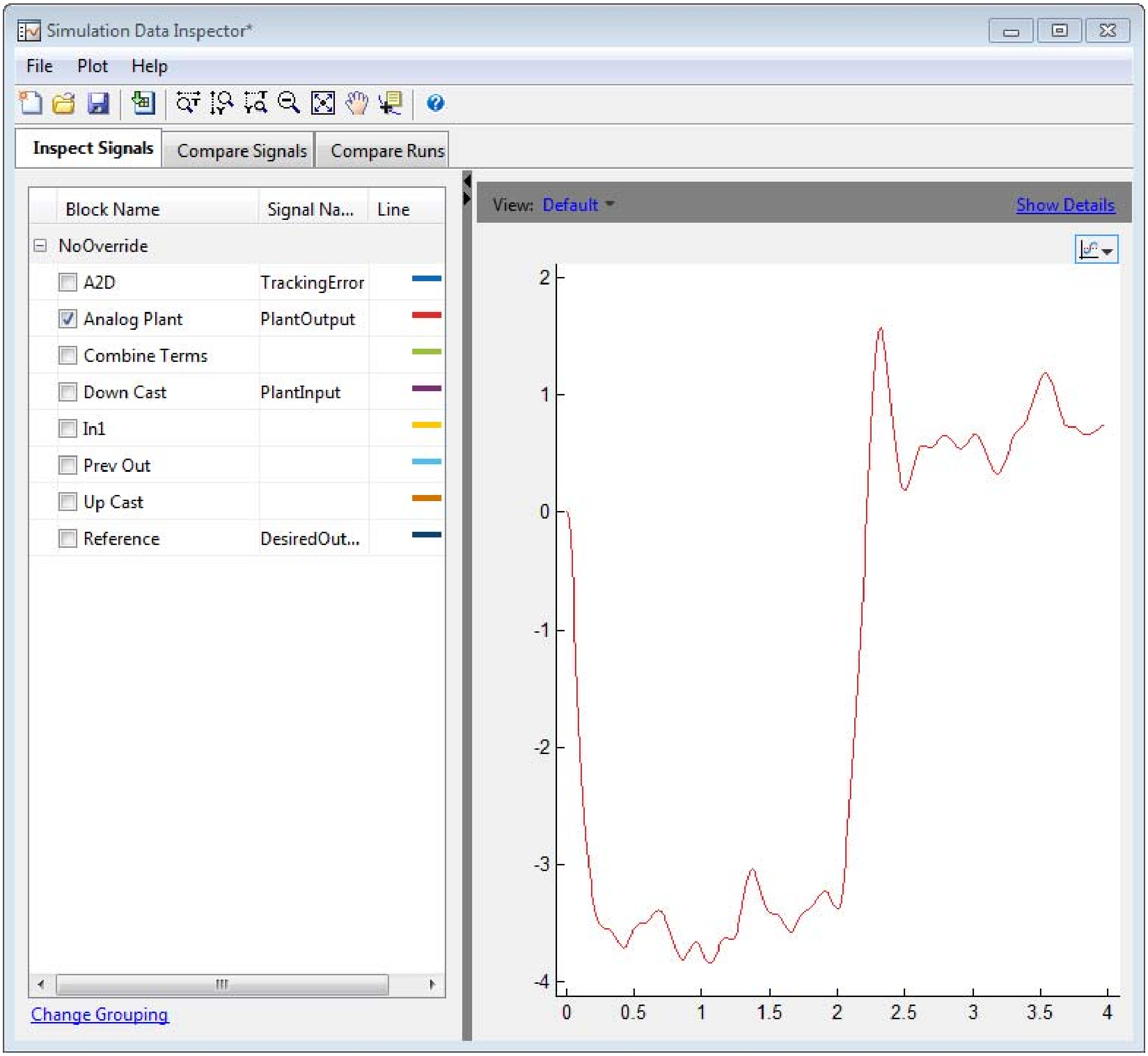
   The tool stores the results in the `NoOverride` run, denoted by the **NoOverride** label in the **Run** column. The Fixed-Point tool highlights the `Up Cast` block to indicate that there is an issue with this result. The **Saturations** column for this result shows that the block saturated 23 times, which indicates a poor guess for its scaling.

Tip In the main toolbar, use the **Show** option to view only blocks that have Overflows.

---

4 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.



Block Name	Signal Na...	Line
⊟ NoOverride		
□ A2D	TrackingError	🟦
☑ Analog Plant	PlantOutput	🟥
□ Combine Terms		🟩
□ Down Cast	PlantInput	🟪
□ In1		🟨
□ Prev Out		🟦
□ Up Cast		🟧
□ Reference	DesiredOut...	🟦

The preceding 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 main toolbar, use the `Show` option and select `Signal logging results`.

Data Type Override

Data type override mode 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. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** 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 (on the **Data collection** pane **Store results in run** field) to `DoubleOverride`

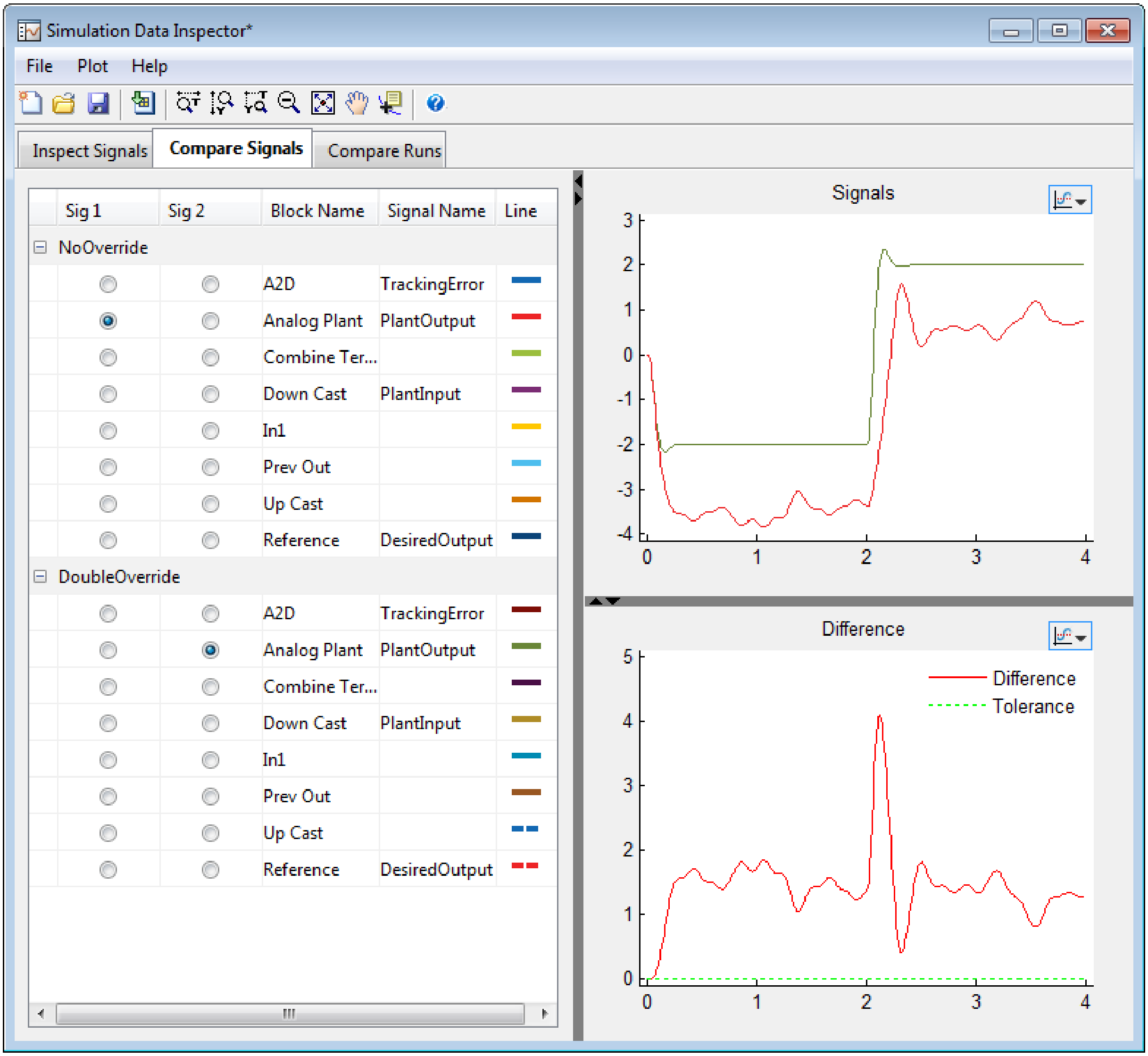
2. In the Fixed-Point Tool, click the **Simulate** button ▶.

   The Simulink software simulates the `fxpdemo_feedback` model in data type override mode and stores the results as 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 34-47). 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.



	Sig 1	Sig 2	Block Name	Signal Name	Line
⊟ NoOverride					
	○	○	A2D	TrackingError	—
	●	○	Analog Plant	PlantOutput	—
	○	○	Combine Ter...		—
	○	○	Down Cast	PlantInput	—
	○	○	In1		—
	○	○	Prev Out		—
	○	○	Up Cast		—
	○	○	Reference	DesiredOutput	—
⊟ DoubleOverride					
	○	○	A2D	TrackingError	—
	○	●	Analog Plant	PlantOutput	—
	○	○	Combine Ter...		—
	○	○	Down Cast	PlantInput	—
	○	○	In1		—
	○	○	Prev Out		—
	○	○	Up Cast		- -
	○	○	Reference	DesiredOutput	- -

Automatic Data Typing

Using the automatic data typing procedure, you can easily maximize the precision of the output data type while spanning the full simulation range.

Because no design min/max information is supplied, the simulation min/max data that was collected during the simulation run is used for proposing 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. For more information about how the Fixed-Point Tool calculates data type proposals, see “Proposing Data Types” on page 34-15.

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` block. This block is a subsystem that represents software running on the target, and it requires optimization.

1 On the **Model Hierarchy** pane of the Fixed-Point Tool, select the `Controller` subsystem. On the **Automatic data typing for selected system** pane, click the **Configure** link. Select **Simulation min/max** for **Propose using information from design min/max and**, then specify the **Safety margin for simulation min/max** parameter as `20`. Click **Apply**.

2 In the Fixed-Point Tool:

   a Click the **Propose fraction lengths** button DT.

   b 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 not selected.
- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited types.

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 with Simulation Min/Max View** to provide information, such as **ProposedDT**, **ProposedMin**, **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 `fxptdlg` in the Simulink Reference.

3. 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.

4. In the Fixed-Point Tool, click the **Apply accepted fraction lengths** button DT✓.

   The Fixed-Point Tool applies the scaling proposals that you accepted in the previous step to the `Controller` subsystem's blocks.

5. On the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system.

  a. On the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings.

  b. On the **Data collection** pane, set **Store results in run** 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.

6. In the Fixed-Point Tool, click **Simulate**.

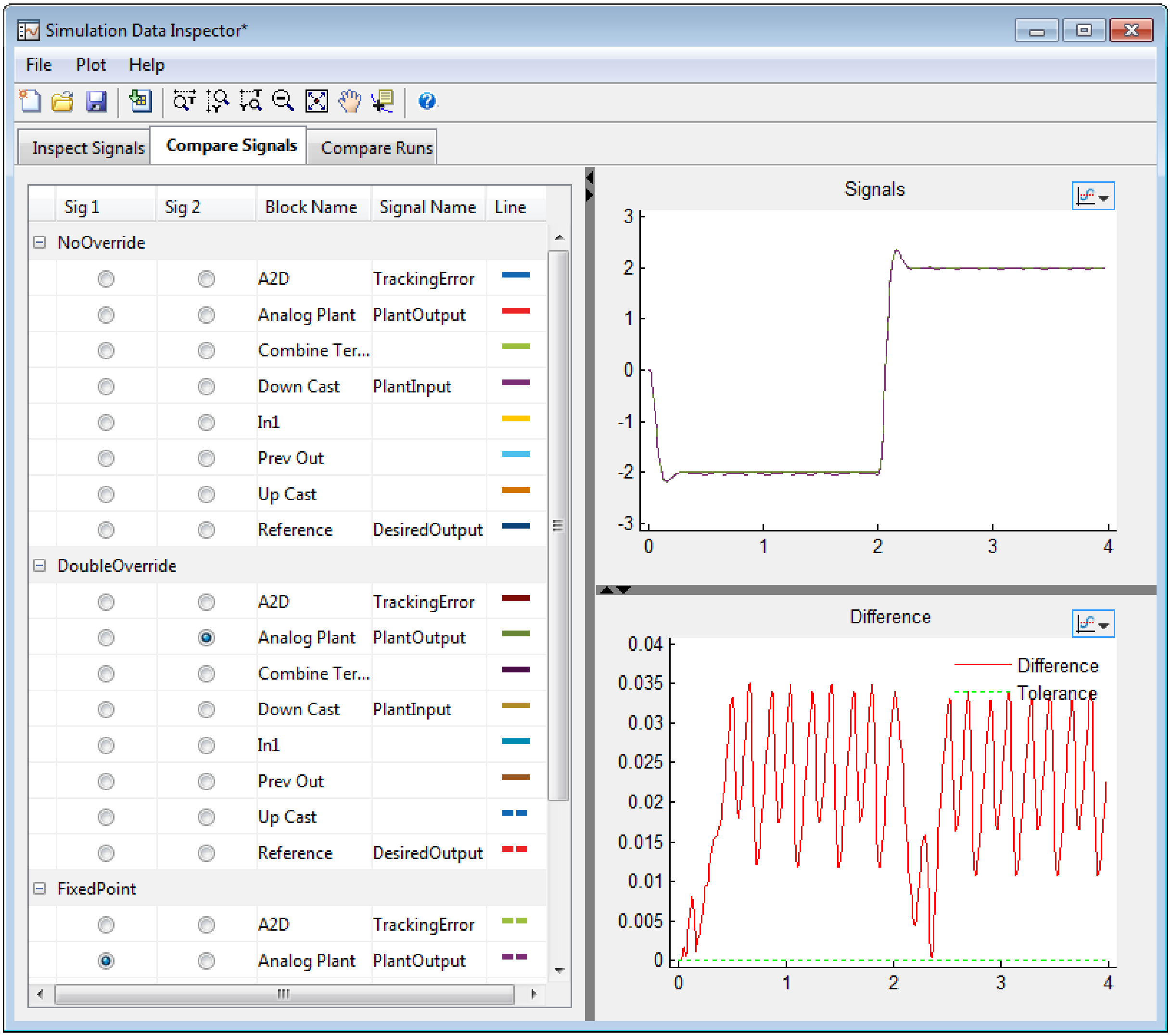
   The Simulink software simulates the `fxpdemo_feedback` model using the new scaling that you applied. Afterward, in its **Contents** pane, the Fixed-Point Tool displays information about blocks that logged fixed-point data. The compiled data type (**CompiledDT**) column for the `FixedPoint` run shows that the Controller subsystem's blocks used fixed-point data types with the new scaling.

7. 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.



Sig 1	Sig 2	Block Name	Signal Name	Line
⊟ NoOverride				
○	○	A2D	TrackingError	▬
○	○	Analog Plant	PlantOutput	▬
○	○	Combine Ter...		▬
○	○	Down Cast	PlantInput	▬
○	○	In1		▬
○	○	Prev Out		▬
○	○	Up Cast		▬
○	○	Reference	DesiredOutput	▬
⊟ DoubleOverride				
○	○	A2D	TrackingError	▬
○	●	Analog Plant	PlantOutput	▬
○	○	Combine Ter...		▬
○	○	Down Cast	PlantInput	▬
○	○	In1		▬
○	○	Prev Out		▬
○	○	Up Cast		╍
○	○	Reference	DesiredOutput	╍
⊟ FixedPoint				
○	○	A2D	TrackingError	╍
●	○	Analog Plant	PlantOutput	╍

Tip 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 that you want to view more closely.

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.

Propose Word Lengths

In this section...

"How the Fixed-Point Tool Proposes Word Lengths" on page 34-57

"Propose Word Lengths" on page 34-59

"Propose Word Lengths Based on Simulation Data" on page 34-60

How the Fixed-Point Tool Proposes Word Lengths

To use the Fixed-Point Tool to propose word lengths, you must specify the target hardware and 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 fraction length to zero. The Fixed-Point Tool uses these specified fraction lengths to recommend the minimum word length for fixed-point data types 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 for blocks, signal objects.
- The signedness and fraction lengths of the default data types that you specify in the Fixed-Point Tool or Fixed-Point Advisor.
- The production hardware implementation settings specified in the Configuration Parameters dialog box.

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 **Derived min/max** and **Simulation min/max** settings.

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 safety margins, the Fixed-Point Tool takes these margins 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 you specify a safety margin of 10% for design range, the Fixed-Point Tool uses a range of `[-11,11]` for the proposal.

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. The tool uses the following rules.

Target Hardware	Ideal Word Length	Proposed Word Length	Restrictions
FPGA/ASIC	Ideal word length=<128	Ideal word length	None
	Ideal word length>128	128	Maximum word length is 128

Target Hardware	Ideal Word Length	Proposed Word Length	Restrictions
Embedded Processor	Ideal word length=< character bit length for the embedded processor (`char`)	`char`	Rounds up word length
	char <Ideal word length=< short bit length for the embedded processor (`short`)	`short`	Rounds up word length
	short<Ideal word length=< integer bit length for the embedded processor (`int`)	`int`	Rounds up word length
	int<Ideal word length=<long bit length for the embedded processor (`long`)	`long`	Rounds up word length
	Ideal word length>long bit length for the embedded processor	`long`	Maximum word length is the target hardware `long`

Propose Word Lengths

1 Specify the target hardware.

   a In the model, select **Simulation > Model Configuration Parameters**.

   b In the Configuration Parameters dialog box, select **Hardware Implementation**.

   c. On the **Hardware Implementation** pane, specify the **Device vendor** and **Device type**, and then click **Apply**.

2. On the Fixed-Point Tool **Automatic data typing for selected system** pane, select **Propose word lengths for specified fraction lengths**. If you cannot see this option, click **Configure** to display more options.

3. On the same pane:

   - For simulation min/max information only, clear **Derived min/max**.
   - For derived min/max information only, clear **Simulation min/max**.

4. If you have safety margins to apply, set **Safety margin for design and derived min/max (%)** and **Safety margin for design and derived min/max (%)**, as applicable.

5. Click the **Propose word lengths** button, DT.

Note When the Fixed-Point Tool proposes data types, it does not alter your model.

If there are conflicts in your model, the Fixed-Point Tool opens the **Result Details** dialog box.

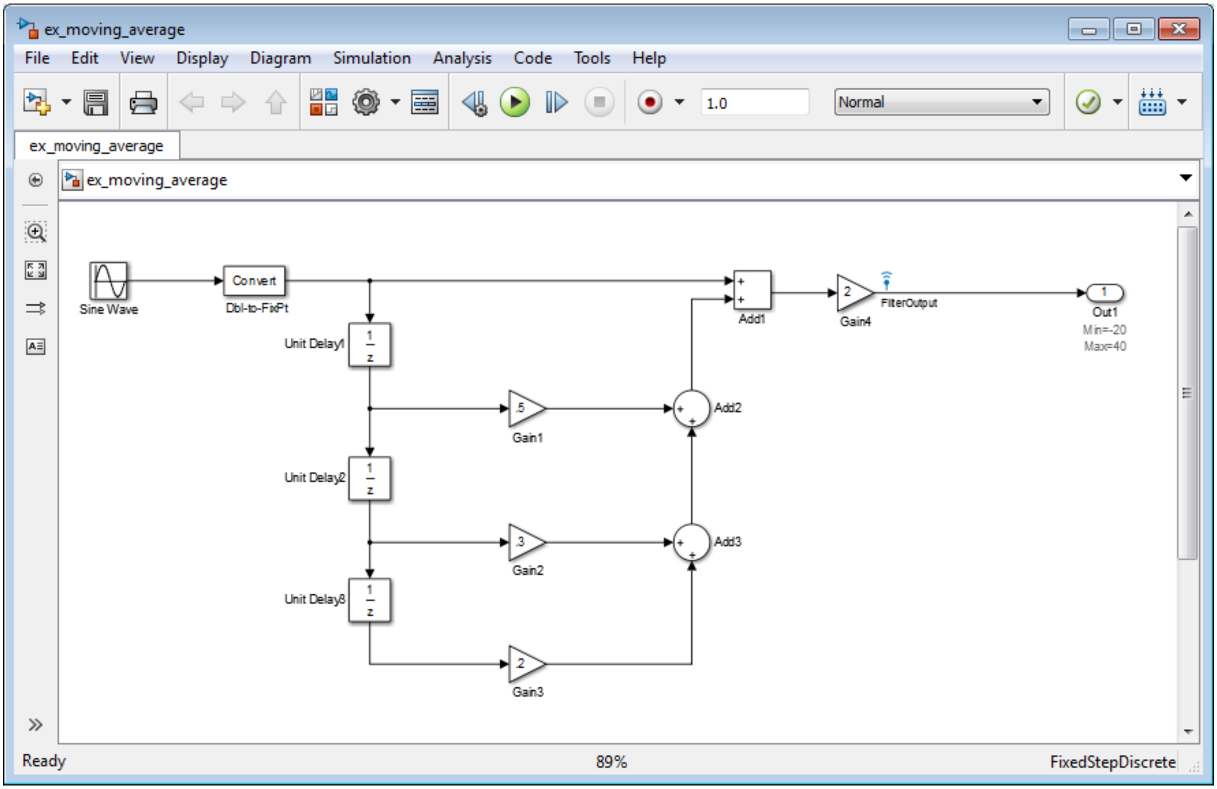
If you do not see this warning, there are no conflicts in your model. Review the proposed word lengths,

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
`Dbl2Fixpt`	`fixdt(1,16,10)`
`Gain1`	`fixdt(1,32,17)`
`Gain2`	`fixdt(1,32,17)`
`Gain3`	`fixdt(1,32,17)`

Block	Data Type Specified on Block
`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.

   a. In the model, select **Simulation > Model Configuration Parameters**.

   b. In the Configuration Parameters dialog box, select **Hardware Implementation**.

      On the **Hardware Implementation** pane, the **Device vendor** is `Generic` and the **Device type** is `16 bit embedded processor`.

   c. Close the Configuration Parameters dialog box.

3. From the model **Analysis** menu, select **Fixed-Point Tool**.

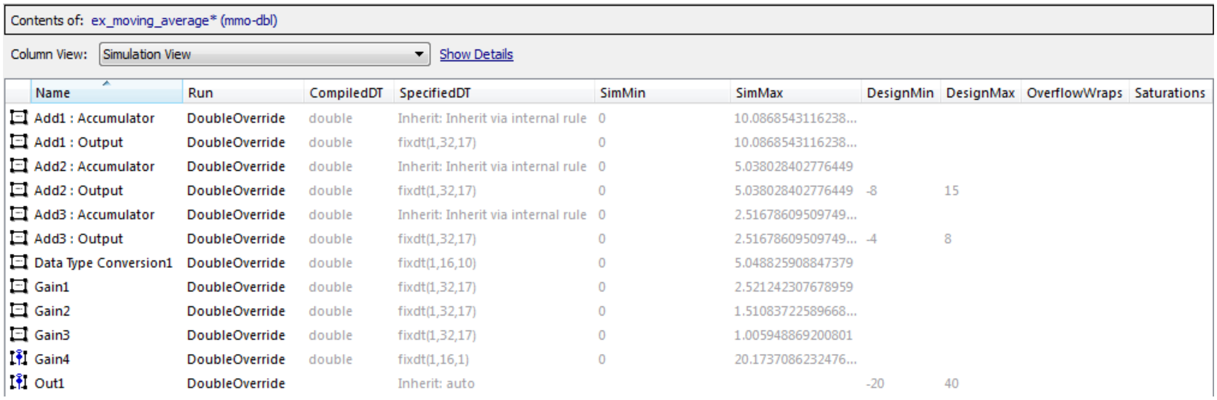
4. On the **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:

   - **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 **Data collection** pane **Store results in run** field) 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 current system or subsystem in the run `DoubleOverride`.

5. Click the Fixed-Point Tool **Simulate** 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

Name	Run	CompiledDT	SpecifiedDT	SimMin	SimMax	DesignMin	DesignMax	OverflowWraps	Saturations
Add1 : Accumulator	DoubleOverride	double	Inherit: Inherit via internal rule	0	10.0868543116238...				
Add1 : Output	DoubleOverride	double	fixdt(1,32,17)	0	10.0868543116238...				
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	2.51678609509749...				
Add3 : Output	DoubleOverride	double	fixdt(1,32,17)	0	2.51678609509749...	-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	1.51083722589668...				
Gain3	DoubleOverride	double	fixdt(1,32,17)	0	1.005948869200801				
Gain4	DoubleOverride	double	fixdt(1,16,1)	0	20.1737086232476...				
Out1	DoubleOverride		Inherit: auto			-20	40		

6 On the **Automatic data typing for selected system** pane:

a Click **Configure** to display more options.

b Select **Propose word lengths for specified fraction lengths**, then click **Apply**.

7 Click the **Propose word lengths** 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 **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. In this example, no safety margins are set.

- The tool observes the simulation range because you selected the **Simulation min/max** option.

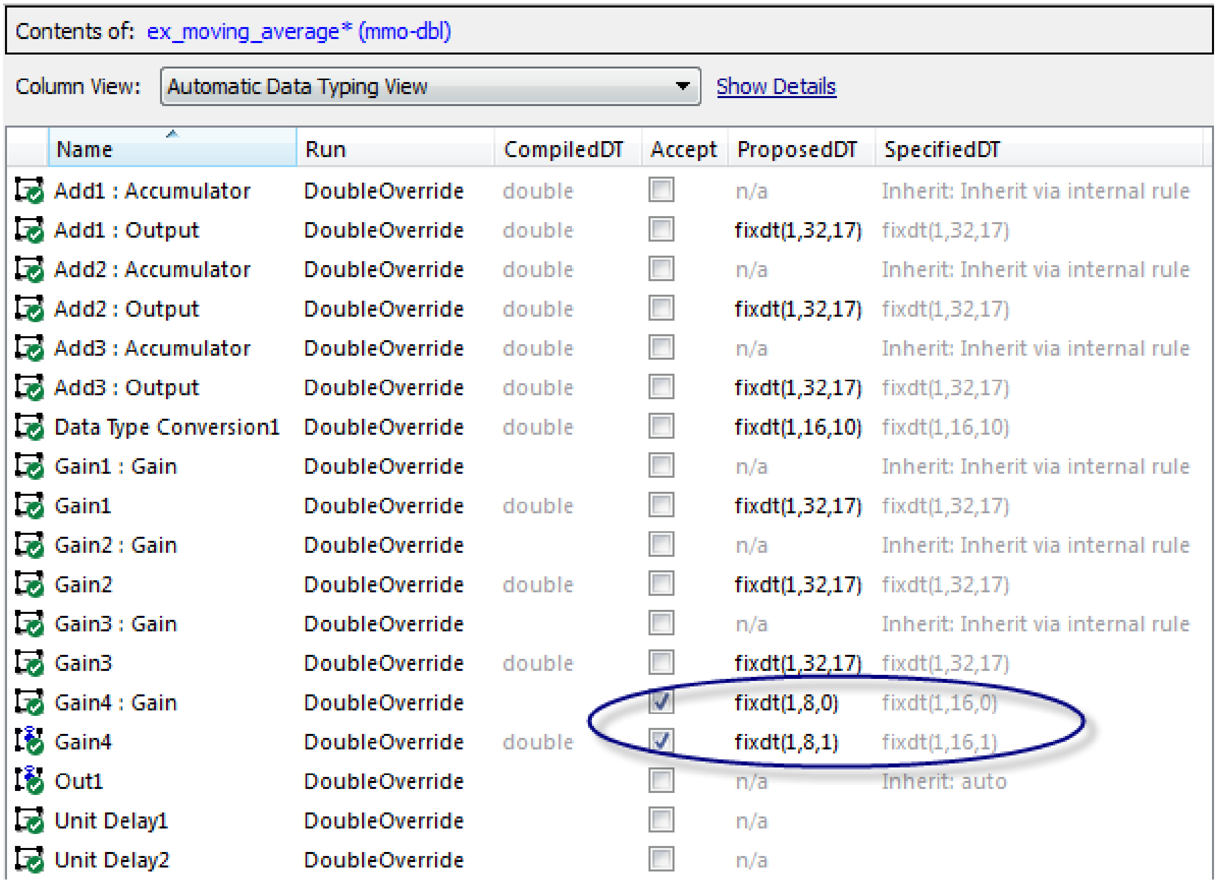
  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. In this example, no safety margins are set.

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 not selected.
- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited types.

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 34-58.

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

Name	Run	CompiledDT	Accept	ProposedDT	SpecifiedDT
Add1 : Accumulator	DoubleOverride	double	□	n/a	Inherit: Inherit via internal rule
Add1 : Output	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Add2 : Accumulator	DoubleOverride	double	□	n/a	Inherit: Inherit via internal rule
Add2 : Output	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Add3 : Accumulator	DoubleOverride	double	□	n/a	Inherit: Inherit via internal rule
Add3 : Output	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Data Type Conversion1	DoubleOverride	double	□	fixdt(1,16,10)	fixdt(1,16,10)
Gain1 : Gain	DoubleOverride		□	n/a	Inherit: Inherit via internal rule
Gain1	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Gain2 : Gain	DoubleOverride		□	n/a	Inherit: Inherit via internal rule
Gain2	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Gain3 : Gain	DoubleOverride		□	n/a	Inherit: Inherit via internal rule
Gain3	DoubleOverride	double	□	fixdt(1,32,17)	fixdt(1,32,17)
Gain4 : Gain	DoubleOverride		☑	fixdt(1,8,0)	fixdt(1,16,0)
Gain4	DoubleOverride	double	☑	fixdt(1,8,1)	fixdt(1,16,1)
Out1	DoubleOverride		□	n/a	Inherit: auto
Unit Delay1	DoubleOverride		□	n/a	
Unit Delay2	DoubleOverride		□	n/a	

8 To see how the target hardware affects the word length proposal, change the target hardware to FPGA/ASIC.

a In the model, select **Simulation > Model Configuration Parameters**.

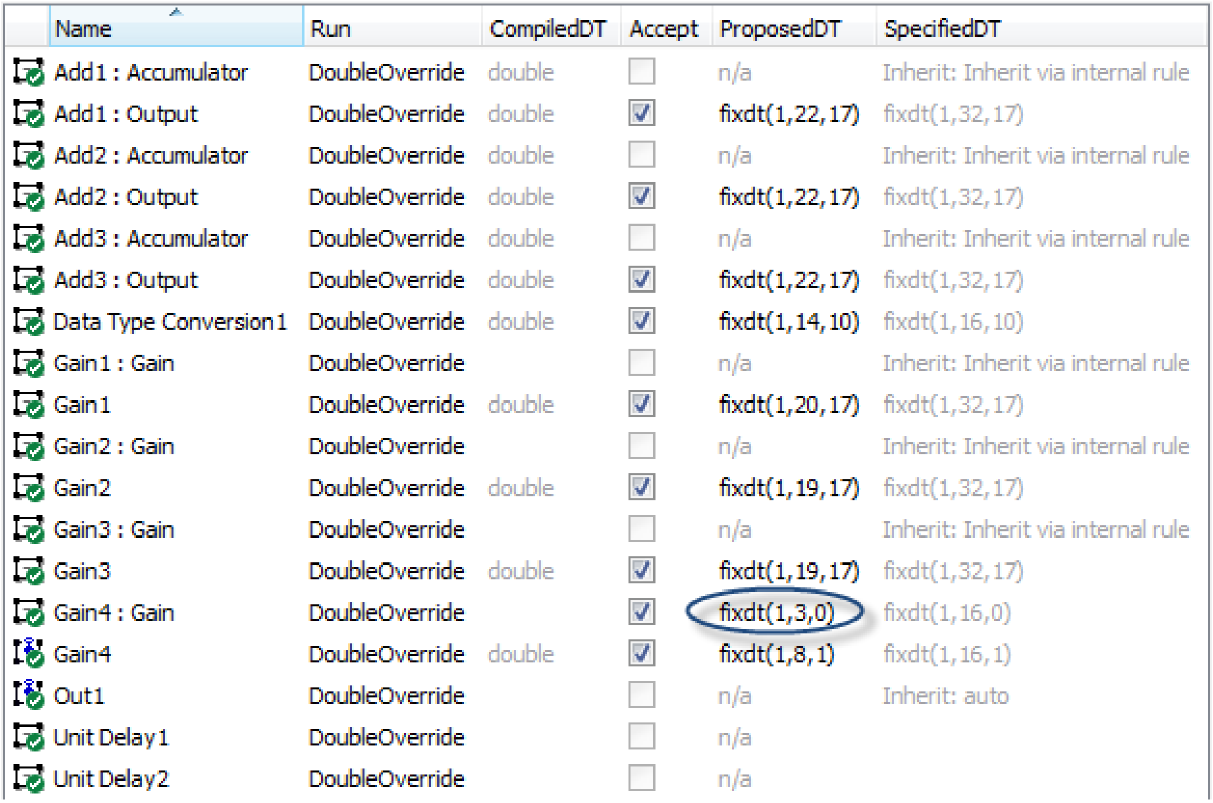
b In the Configuration Parameters dialog box, select **Hardware Implementation**.

c On the **Hardware Implementation** pane, set **Device vendor** to `ASIC/FPGA`. Simulink automatically sets the **Device type** to `ASIC/FPGA`.

d Click **Apply** and close the Configuration Parameters dialog box.

9 On the Fixed-Point Tool **Automatic data typing for selected system** pane, click the **Propose word lengths** button.

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 3.



Name	Run	CompiledDT	Accept	ProposedDT	SpecifiedDT
Add1 : Accumulator	DoubleOverride	double	☐	n/a	Inherit: Inherit via internal rule
Add1 : Output	DoubleOverride	double	☑	fixdt(1,22,17)	fixdt(1,32,17)
Add2 : Accumulator	DoubleOverride	double	☐	n/a	Inherit: Inherit via internal rule
Add2 : Output	DoubleOverride	double	☑	fixdt(1,22,17)	fixdt(1,32,17)
Add3 : Accumulator	DoubleOverride	double	☐	n/a	Inherit: Inherit via internal rule
Add3 : Output	DoubleOverride	double	☑	fixdt(1,22,17)	fixdt(1,32,17)
Data Type Conversion1	DoubleOverride	double	☑	fixdt(1,14,10)	fixdt(1,16,10)
Gain1 : Gain	DoubleOverride		☐	n/a	Inherit: Inherit via internal rule
Gain1	DoubleOverride	double	☑	fixdt(1,20,17)	fixdt(1,32,17)
Gain2 : Gain	DoubleOverride		☐	n/a	Inherit: Inherit via internal rule
Gain2	DoubleOverride	double	☑	fixdt(1,19,17)	fixdt(1,32,17)
Gain3 : Gain	DoubleOverride		☐	n/a	Inherit: Inherit via internal rule
Gain3	DoubleOverride	double	☑	fixdt(1,19,17)	fixdt(1,32,17)
Gain4 : Gain	DoubleOverride		☑	fixdt(1,3,0)	fixdt(1,16,0)
Gain4	DoubleOverride	double	☑	fixdt(1,8,1)	fixdt(1,16,1)
Out1	DoubleOverride		☐	n/a	Inherit: auto
Unit Delay1	DoubleOverride		☐	n/a	
Unit Delay2	DoubleOverride		☐	n/a	

Propose Data Types Using Multiple Simulations

In this section...

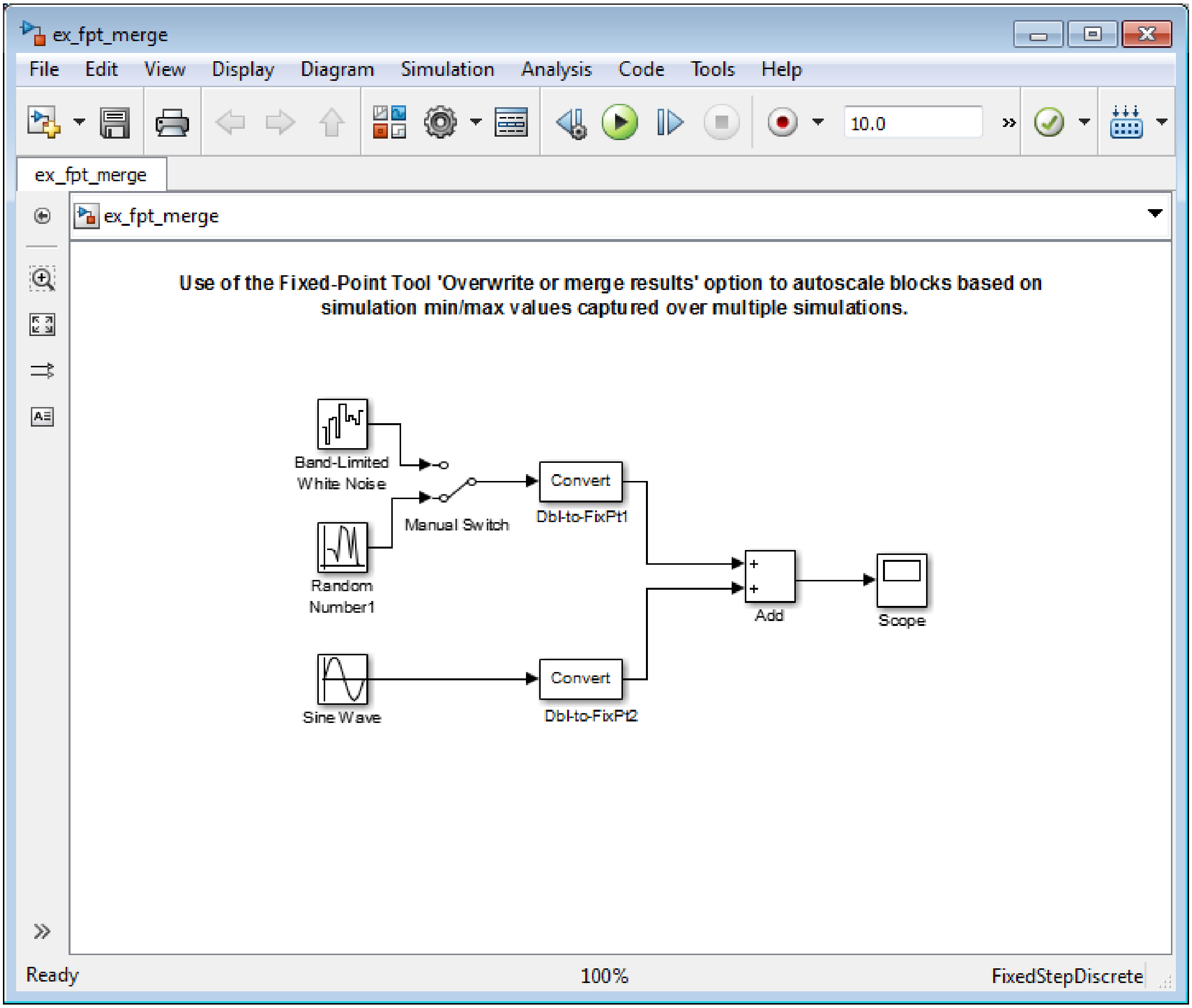
"About This Example" on page 34-67

"Running the Simulation" on page 34-70

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.



Use of the Fixed-Point Tool 'Overwrite or merge results' option to autoscale blocks based on simulation min/max values captured over multiple simulations.

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 34-70. 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. The Fixed-Point Tool uses these simulation settings:

   - **Fixed-point instrumentation mode**: `Minimums, maximums and overflows`
   - **Data type override**: `Double`
   - **Data type override applies to**: `All numeric types`
   - **Merge instrumentation results from multiple simulations** is not selected.

   This run provides the simulation results for the random uniform noise input only.

2. "Simulate Model Using Band-Limited White Noise" on page 34-71. 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 34-71. 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 34-72. 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. From the model main menu, select **Analysis > Fixed-Point Tool**.

3. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:

   - **Data type override** to `Double`. This option enables each of the model’s subsystems to use its locally specified data type settings.

   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.

   - The run name to `DoubleOverride`.

4. In the Fixed-Point Tool, click the **Simulate** button ▶.

   The Simulink software 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`, denoted by the **DoubleOverride** label in the **Run** column.

5 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.

   The Simulink software 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 On the Fixed-Point Tool **Data collection** pane, select **Merge instrumentation 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`

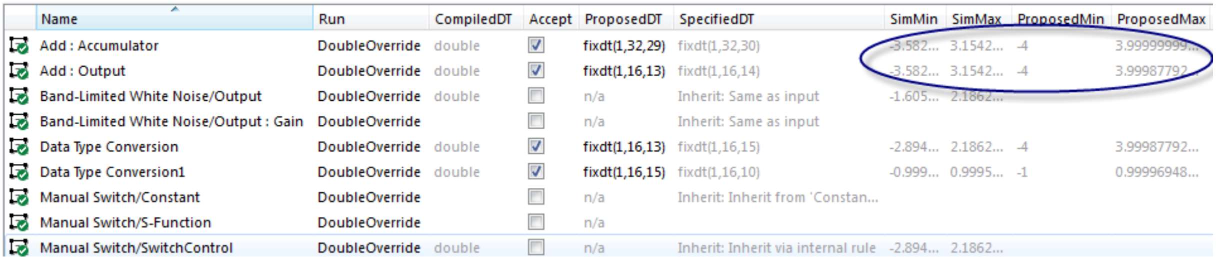
Propose Fraction Lengths Based on Merged Results

1 On the **Automatic data typing for selected system** pane, click the **Propose fraction lengths** 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 not selected.
- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited.

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	CompiledDT	Accept	ProposedDT	SpecifiedDT	SimMin	SimMax	ProposedMin	ProposedMax
Add : Accumulator	DoubleOverride	double	☑	fixdt(1,32,29)	fixdt(1,32,30)	-3.582...	3.1542...	-4	3.99999999...
Add : Output	DoubleOverride	double	☑	fixdt(1,16,13)	fixdt(1,16,14)	-3.582...	3.1542...	-4	3.99987792...
Band-Limited White Noise/Output	DoubleOverride	double	□	n/a	Inherit: Same as input	-1.605...	2.1862...		
Band-Limited White Noise/Output : Gain	DoubleOverride		□	n/a	Inherit: Same as input				
Data Type Conversion	DoubleOverride	double	☑	fixdt(1,16,13)	fixdt(1,16,15)	-2.894...	2.1862...	-4	3.99987792...
Data Type Conversion1	DoubleOverride	double	☑	fixdt(1,16,15)	fixdt(1,16,10)	-0.999...	0.9995...	-1	0.99996948...
Manual Switch/Constant	DoubleOverride		□	n/a	Inherit: Inherit from 'Constan...				
Manual Switch/S-Function	DoubleOverride		□	n/a					
Manual Switch/SwitchControl	DoubleOverride	double	□	n/a	Inherit: Inherit via internal rule	-2.894...	2.1862...		

View Simulation Results

In this section...

"Compare Runs" on page 34-73

"Compare Signals" on page 34-74

"Inspect Signals" on page 34-76

"Histogram Plot of Signal" on page 34-77

"See Also" on page 34-78

Compare Runs

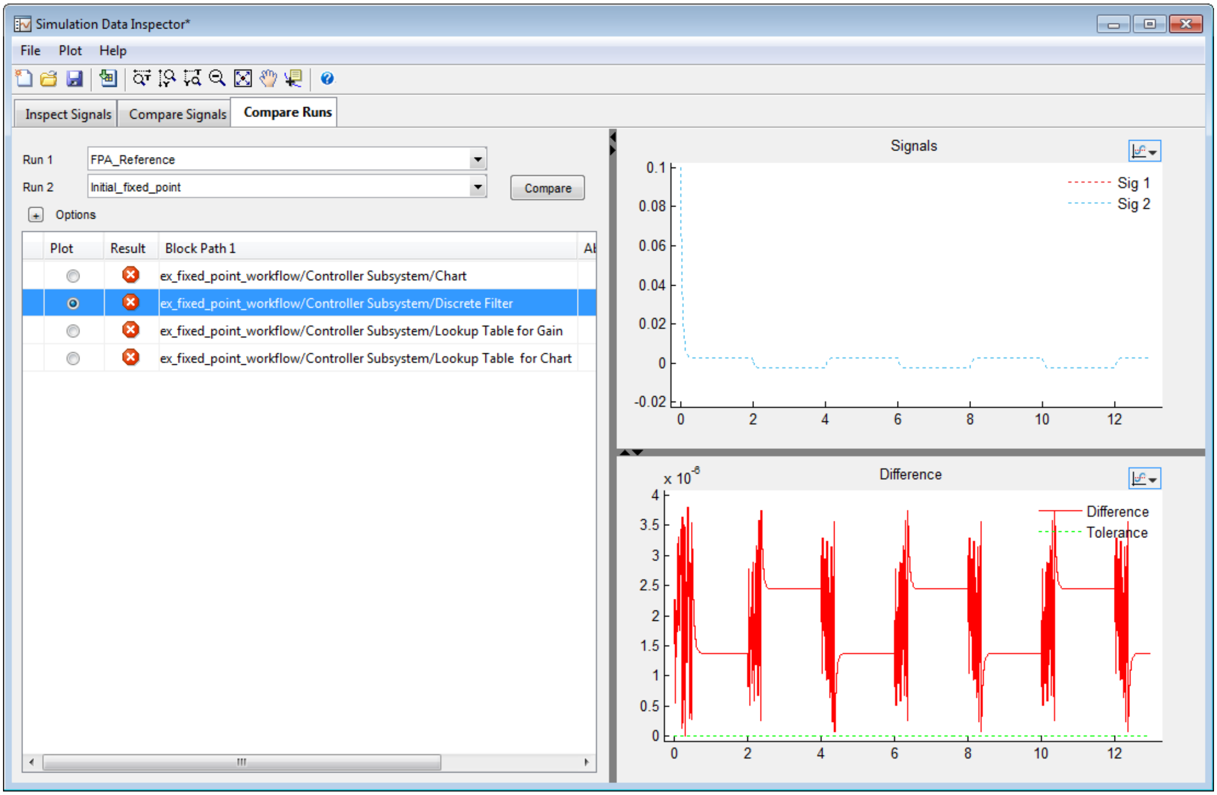
To compare runs:

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.



Run 1 FPA_Reference

Run 2 Initial_fixed_point

Plot	Result	Block Path 1	Al
○	☒	ex_fixed_point_workflow/Controller Subsystem/Chart	
●	☒	ex_fixed_point_workflow/Controller Subsystem/Discrete Filter	
○	☒	ex_fixed_point_workflow/Controller Subsystem/Lookup Table for Gain	
○	☒	ex_fixed_point_workflow/Controller Subsystem/Lookup Table for Chart	

Compare Signals

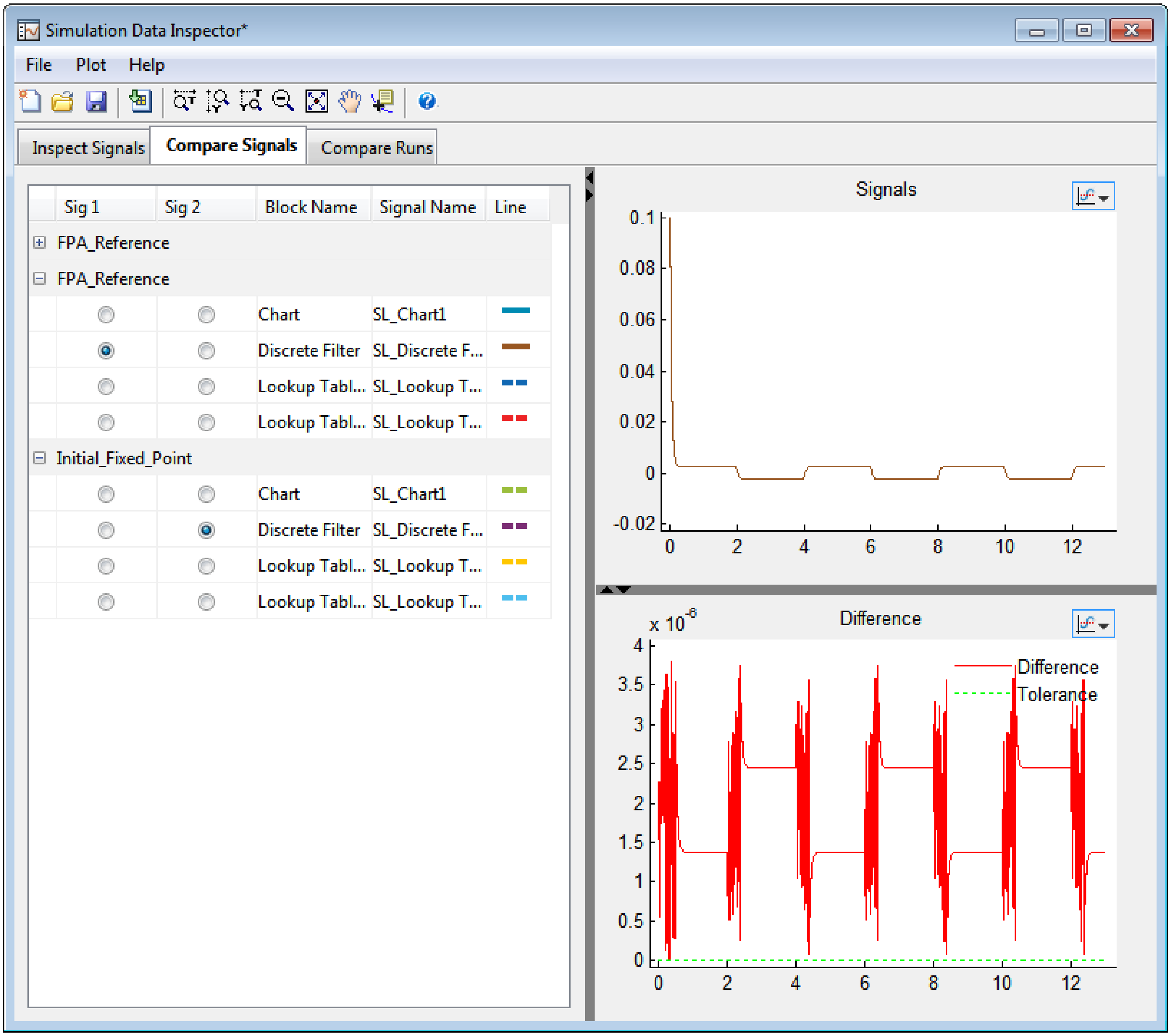
To compare signals:

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.



Sig 1	Sig 2	Block Name	Signal Name	Line
⊞ FPA_Reference				
⊟ FPA_Reference				
○	○	Chart	SL_Chart1	▬
●	○	Discrete Filter	SL_Discrete F...	▬
○	○	Lookup Tabl...	SL_Lookup T...	- -
○	○	Lookup Tabl...	SL_Lookup T...	- -
⊟ Initial_Fixed_Point				
○	○	Chart	SL_Chart1	- -
○	●	Discrete Filter	SL_Discrete F...	- -
○	○	Lookup Tabl...	SL_Lookup T...	- -
○	○	Lookup Tabl...	SL_Lookup T...	- -

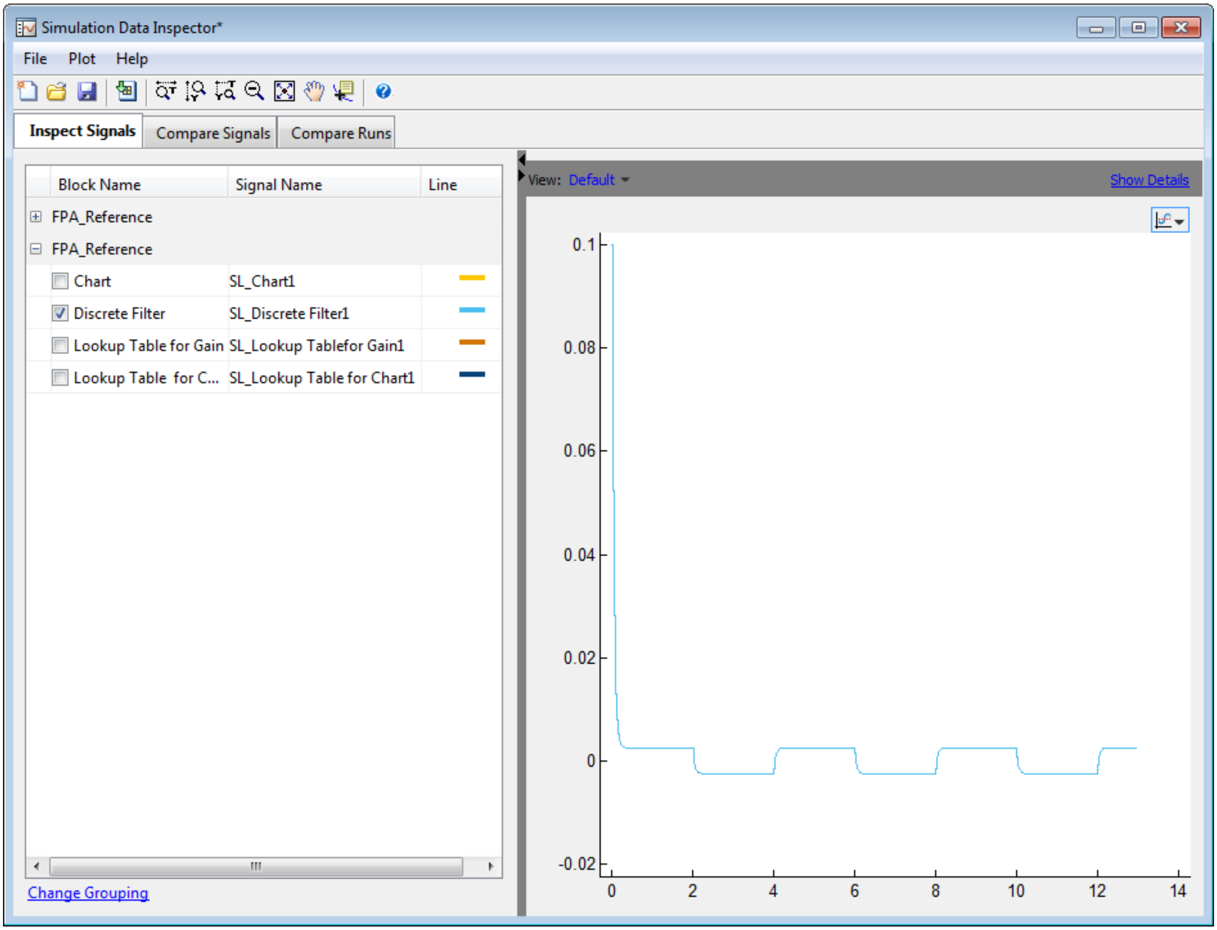
Inspect Signals

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.



	Block Name	Signal Name	Line
⊞	FPA_Reference		
⊟	FPA_Reference		
	□ Chart	SL_Chart1	▬
	☑ Discrete Filter	SL_Discrete Filter1	▬
	□ Lookup Table for Gain	SL_Lookup Tablefor Gain1	▬
	□ Lookup Table for C...	SL_Lookup Table for Chart1	▬

Histogram Plot of Signal

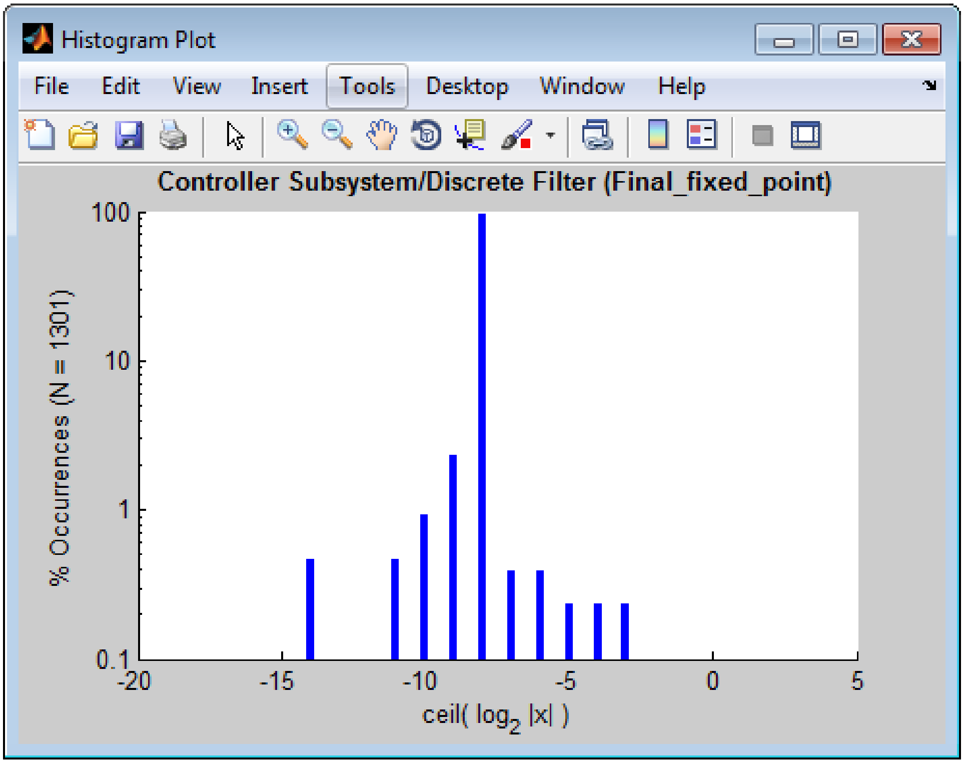
To view the histogram plot of a signal:

1. Select the logged signal that you want to plot.
2. From the Fixed-Point Tool menu, select **Results > Histogram Plot of Signal** or click 📊.

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.



See Also

- “Viewing Results With the Simulation Data Inspector” on page 34-79
- “Propose Fraction Lengths Using Simulation Range Data” on page 34-46
- “Converting a Model from Floating- to Fixed-Point Using Simulation Data” on page 30-11

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 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.

See Also

- "Validate System Behavior"
- `fxptdlg`

Range Analysis

- "How Range Analysis Works" on page 35-2
- "Derive Ranges" on page 35-7
- "Derive Ranges at the Subsystem Level" on page 35-10
- "View Derived Ranges in the Fixed-Point Tool" on page 35-12
- "Range Analysis Examples" on page 35-14
- "Derive Ranges for a Referenced Model" on page 35-31
- "Propose Data Types for a Referenced Model" on page 35-37
- "Deriving Ranges for a Referenced Model" on page 35-39
- "Unsupported Simulink Software Features" on page 35-41
- "Supported and Unsupported Simulink Blocks" on page 35-43

How Range Analysis Works

In this section...

"Analyzing a Model with Range Analysis" on page 35-2

"Automatic Stubbing" on page 35-5

"Model Compatibility with Range Analysis" on page 35-6

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 35-6.

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 Outport. 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 35-5.

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
You specify design minimum and maximum data for a block output.	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: • Specified minimum and maximum values • Derived minimum and maximum values	"Derive Ranges Using Design Minimum and Maximum Values" on page 35-14
A parameter on a block has initial conditions and a design range.	The analysis takes both factors into account by taking the union of the design range and the initial conditions.	"Derive Ranges Using Block Initial Conditions" on page 35-16
The model contains a global tunable parameter with a specified range. (See "Global Tunable Parameters")	The analysis takes into account the range specified for the parameter and ignores the value.	"Derive Ranges Using Design Range Information for Simulink.Parameter Objects" on page 35-19
The model contains a global nontunable parameter with a specified range.	The analysis does not take into account the range specified for the parameter. Instead, it uses the parameter value.	"Derive Ranges Using Design Range Information for Simulink.Parameter Objects" on page 35-19

When...	How the Analysis Works	Examples
The model contains insufficient design range information.	The analysis cannot determine derived ranges. You must specify more design range information and rerun the analysis.	"Providing More Design Range Information" on page 35-25 The analysis results might depend on block sorting order which determines the order in which the software analyzes the blocks. For more information, see "Control and Display the Sorted Order".
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 35-28

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 “Supported and Unsupported Simulink Blocks” on page 35-43.

Model Compatibility with Range Analysis

To verify that your model is compatible with range analysis, see:

- “Unsupported Simulink Software Features” on page 35-41
- “Supported and Unsupported Simulink Blocks” on page 35-43
- “Limitations of Support for Model Blocks” on page 35-53

Derive Ranges

1 Verify that your model is compatible with range analysis. See "Model Compatibility with Range Analysis" on page 35-6.

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 34-26.

3 From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

4 In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

5 Use the Fixed-Point Advisor to prepare the model for conversion.

   a In the Fixed-Point Tool **Fixed-point preparation for selected system** pane, 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 30-2.

   The Fixed-Point Advisor:

   - Checks the model against fixed-point guidelines.
   - Identifies unsupported blocks.
   - Removes output data type inheritance from 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 37-36.

6 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

   This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

7 Optionally, in the **Data collection** pane **Store results in run** field, specify a run name. Specifying a unique run name avoids overwriting results from previous runs.

8 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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 35-12
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 35-6
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 35-28
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.	"Providing More Design Range Information" on page 35-25

Derive Ranges at the Subsystem Level

In this section...

"Deriving Ranges at the Subsystem Level" on page 35-10

"Derive Ranges at the Subsystem Level" on page 35-11

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 35-7.

To derive ranges at the subsystem level, the key points to remember are:

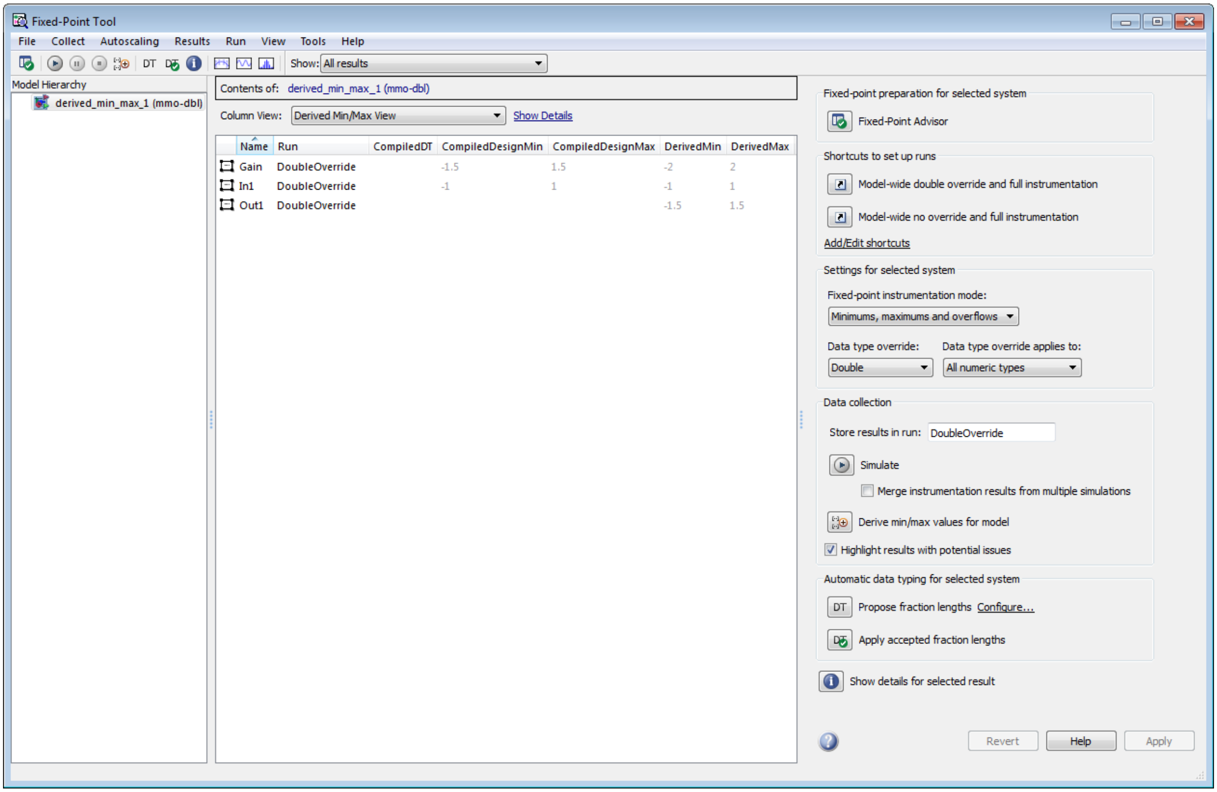
- The subsystem or subchart must be atomic.
- In the Fixed-Point Tool **Model Hierarchy** pane, select the subsystem of interest.
- In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

  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.
- In the **Data collection** pane **Store results in run** field, 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.


Fixed-Point Tool
File Collect Autoscaling Results Run View Tools Help
DT
Model Hierarchy
derived_min_max_1 (mmo-dbl)
Show Details
Fixed-point preparation for selected system
Fixed-Point Advisor
Shortcuts to set up runs
Model-wide double override and full instrumentation
Model-wide no override and full instrumentation
Add/Edit shortcuts
Settings for selected system
Data collection
Simulate
Derive min/max values for model
Automatic data typing for selected system
Propose fraction lengths
Configure...
Apply accepted fraction lengths
Show details for selected result
Revert
Help
Apply

Show: All results

Contents of: derived_min_max_1 (mmo-dbl)

Column View: Derived Min/Max View

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	DoubleOverride		-1.5	1.5	-2	2
In1	DoubleOverride		-1	1	-1	1
Out1	DoubleOverride				-1.5	1.5

Fixed-point instrumentation mode: Minimums, maximums and overflows

Data type override: Double

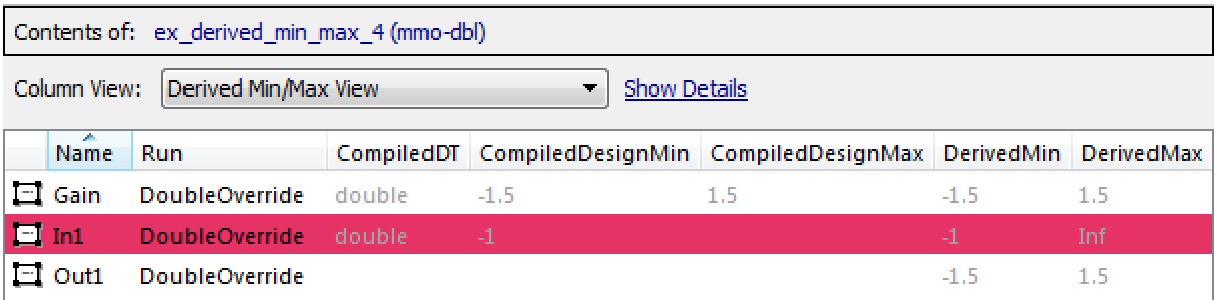
Data type override applies to: All numeric types

Store results in run: DoubleOverride

□ Merge instrumentation results from multiple simulations

☑ Highlight results with potential issues

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 35-22.



Contents of: ex_derived_min_max_4 (mmo-dbl)

Column View: Derived Min/Max View

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
In1	DoubleOverride	double	-1		-1	Inf
Out1	DoubleOverride				-1.5	1.5

Range Analysis Examples

In this section...

"Derive Ranges Using Design Minimum and Maximum Values" on page 35-14

"Derive Ranges Using Block Initial Conditions" on page 35-16

"Derive Ranges Using Design Range Information for Simulink.Parameter Objects" on page 35-19

"Insufficient Design Range Information" on page 35-22

"Providing More Design Range Information" on page 35-25

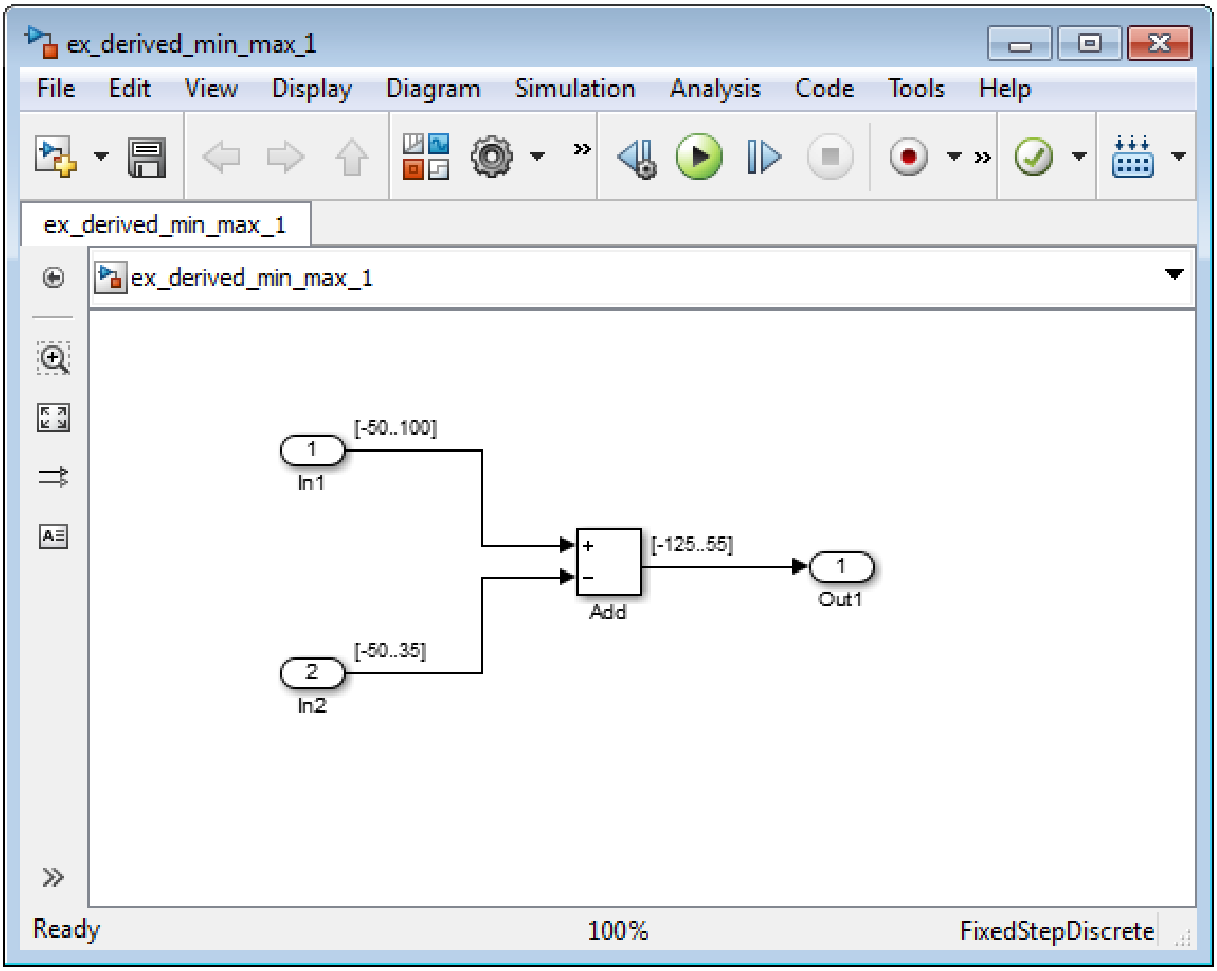
"Fixing Design Range Conflicts" on page 35-28

Derive Ranges Using Design Minimum and Maximum Values

This example shows how the range analysis narrows the derived range for the Outport 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 **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

   This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

4 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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.

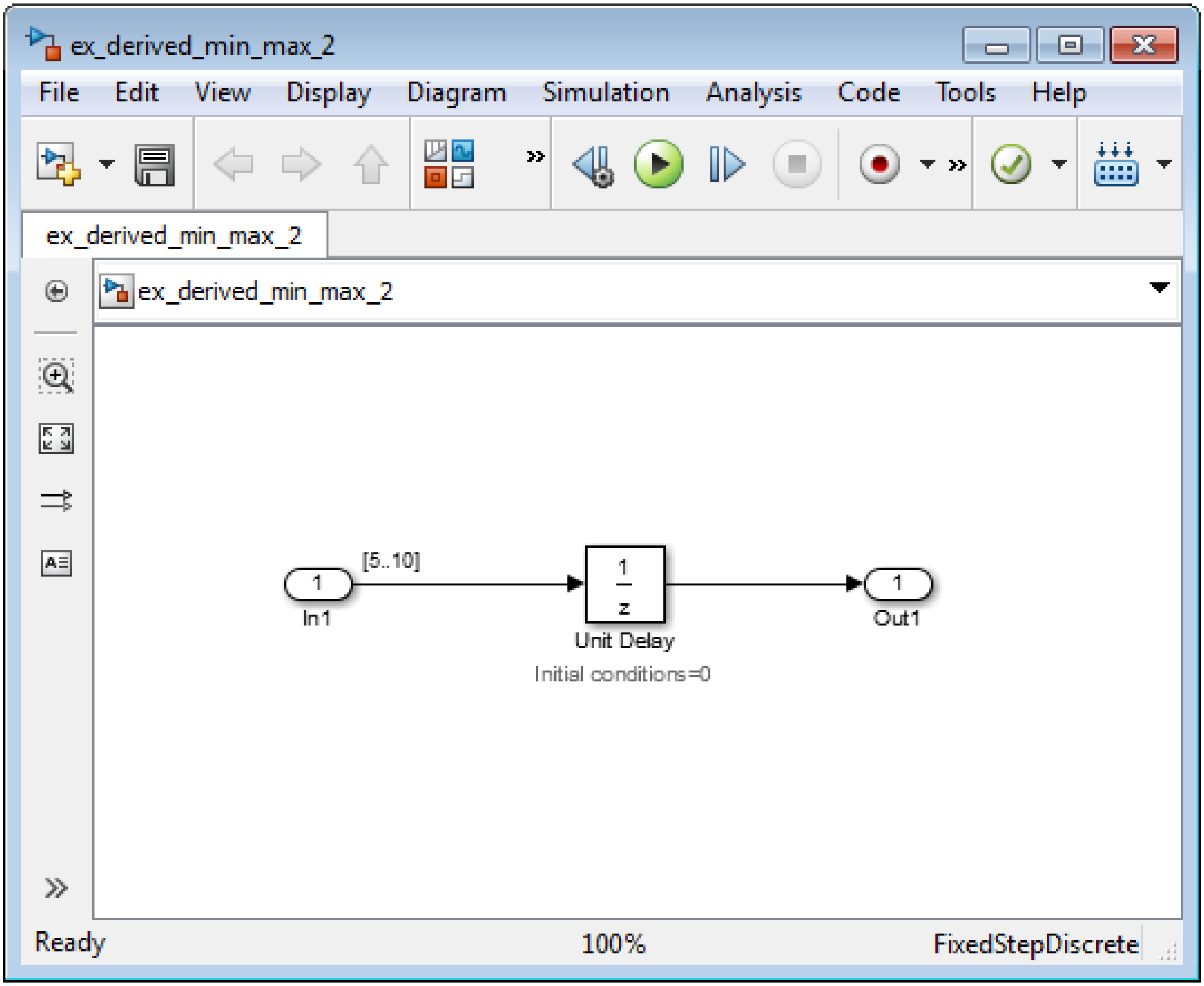
   - The derived range for the Add block output signal is narrowed to `[-85..55]`. This derived range is the intersection of the range derived from the block inputs, `[-85..150]` and the design minimum and maximum values specified for the block output, `[-125..55]`.
   - The derived range for the Outport block `Out1` is `[-85..55]`, the same as the Add block output.

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 From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

   This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

4 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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]`.

5 Change the initial value of the Unit Delay block to `7`.

   a Double-click the Unit Delay block.

   b In the **Block Parameters** dialog box, set **Initial conditions** to `7`, then click **OK**.

   c In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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 still `[5..10]`.

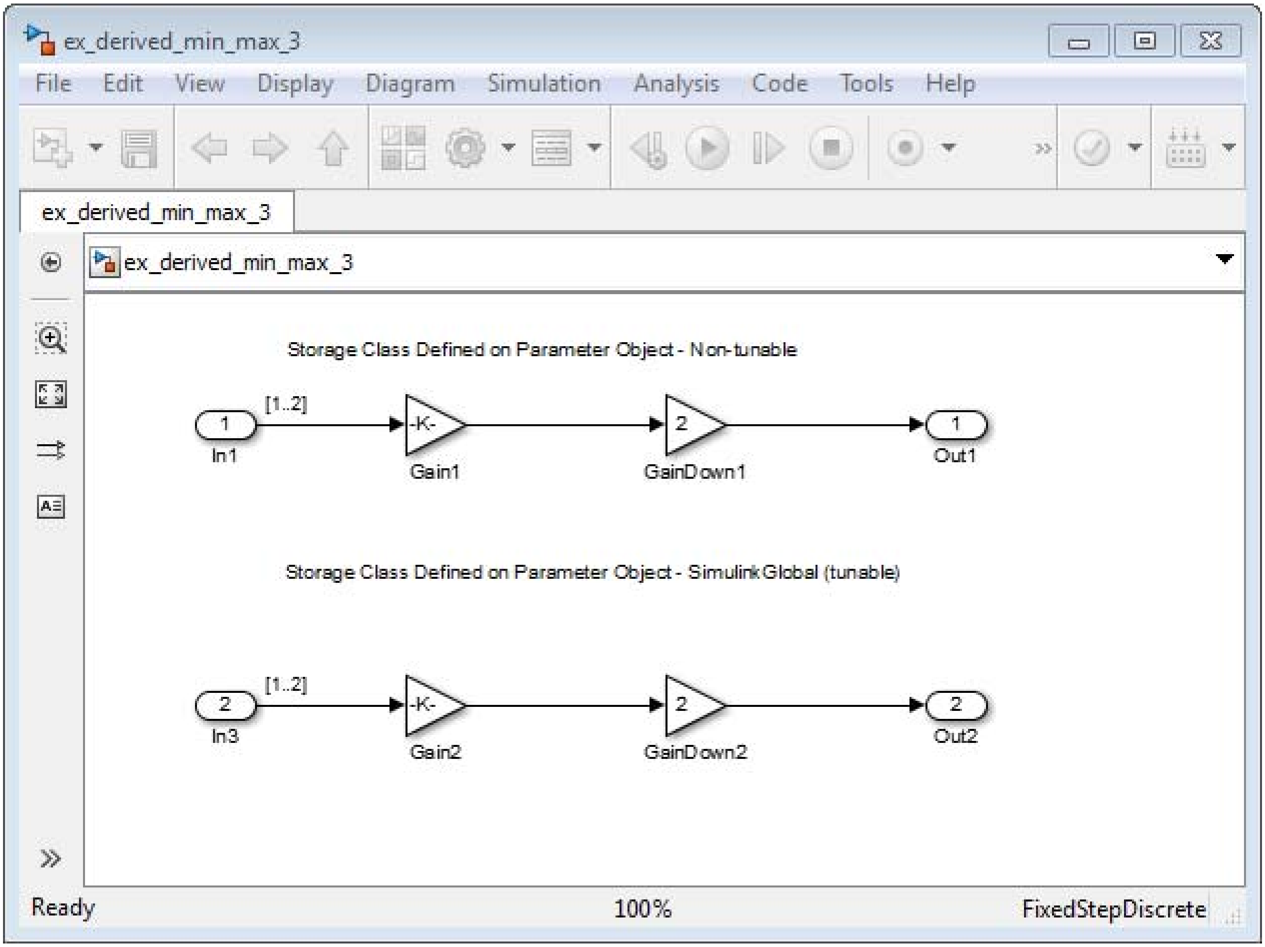
Derive Ranges Using Design Range Information for Simulink.Parameter Objects

This example shows how the range analysis takes into account design range information for `Simulink.Parameter` objects only if they are global tunable parameters. (See “Global Tunable Parameters”.) Otherwise, 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`


ex_derived_min_max_3
File Edit View Display Diagram Simulation Analysis Code Tools Help
ex_derived_min_max_3
ex_derived_min_max_3
Storage Class Defined on Parameter Object - Non-tunable
[1..2]
1
In1
-K-
Gain1
2
GainDown1
1
Out1
Storage Class Defined on Parameter Object - SimulinkGlobal (tunable)
[1..2]
2
In3
-K-
Gain2
2
GainDown2
2
Out2
Ready
100%
FixedStepDiscrete

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 **Fixed-Point Tool**.

4 In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

5 In the Fixed-Point Tool, click the **Derive min/max values for selected system** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model.

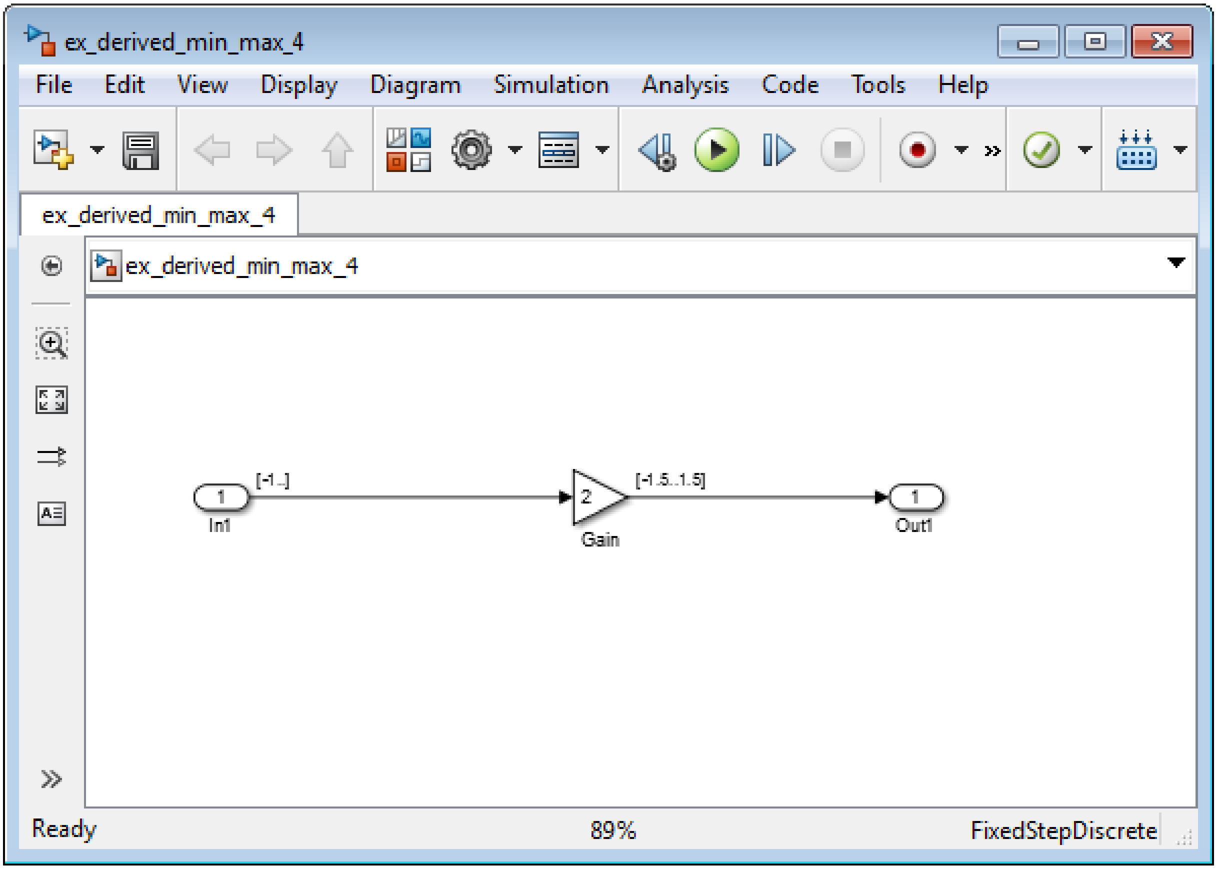
Block	Derived Range	Reason
Gain1	[2..4]	The gain parameter, paramObjOne, specified on Gain block Gain1 is a Simulink.Parameter object that has its storage class specified as Auto. This is a non-tunable parameter. The range analysis uses the **value** of the Simulink.Parameter object, which is 2, and ignores the design range specified for these parameters.
Gain2	[1..20]	The gain parameter, paramObjTwo, specified on Gain block Gain2 is a Simulink.Parameter object that has its storage class specified as SimulinkGlobal. This is a tunable parameter. The range analysis takes into account the design range, [1..10], specified for this parameter.

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**.

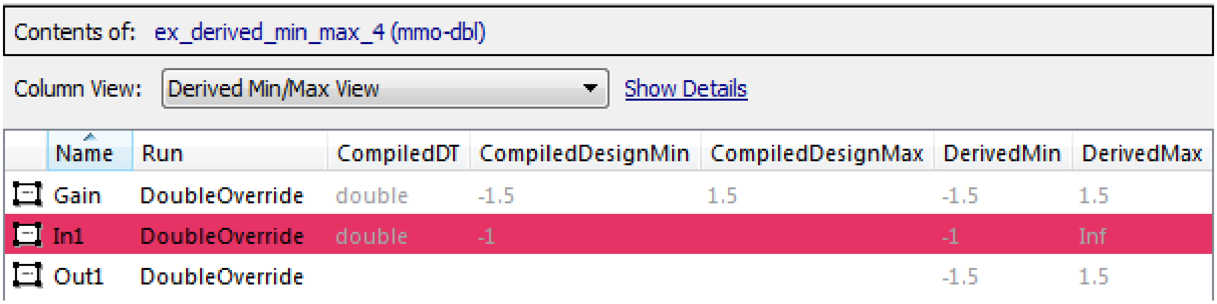
2 From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

4 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
In1	DoubleOverride	double	-1		-1	Inf
Out1	DoubleOverride				-1.5	1.5

5 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`.

6 In the Fixed-Point Tool, click the **Derive min/max values for selected system** button to rerun the range analysis.

The range analysis can now derive ranges for the Inport and Gain blocks.

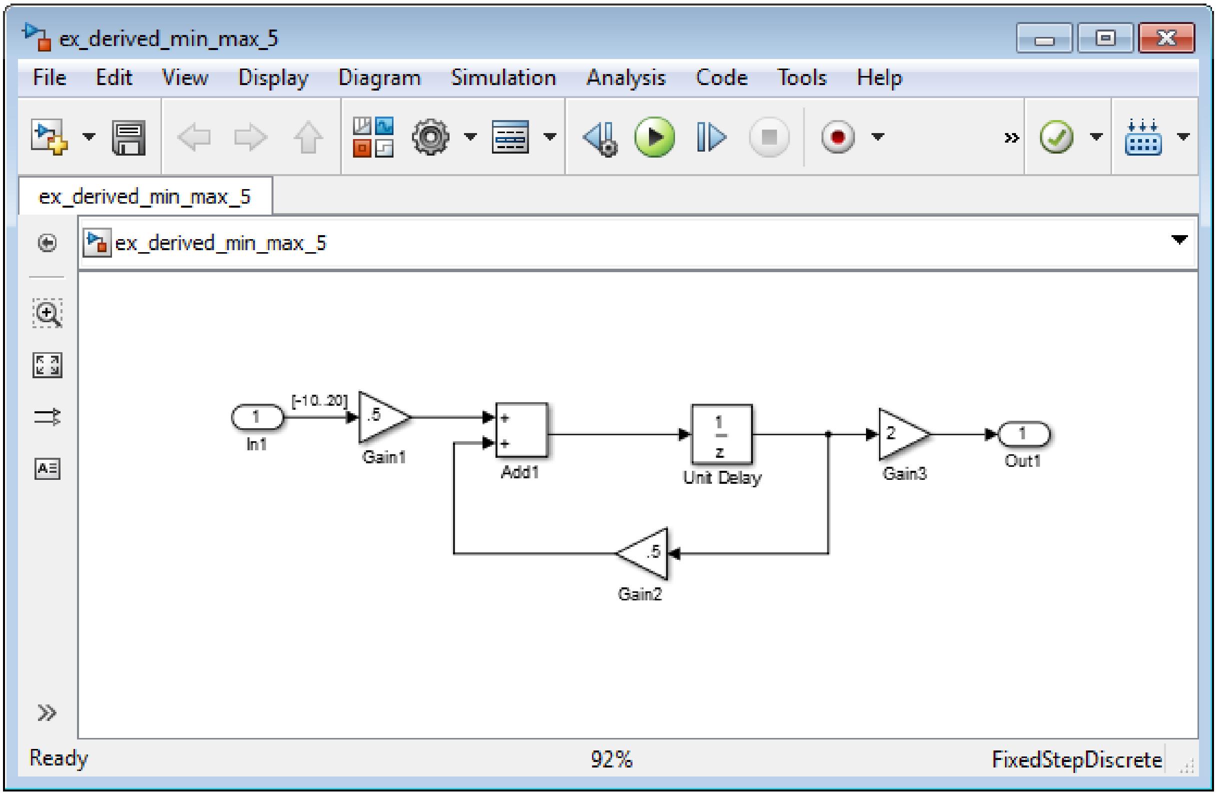
Block	Derived Range	Reason
Inport In1	[-1..1]	Uses specified design range on the block.
Gain	[-1.5..1.5]	The design range specified on the Gain block is[-1.5..1.5]. The derived range at the block input is [-1..1] (the derived range at the output of In1). Therefore, because the gain is 2, the derived 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.

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 output design minimum and maximum values.

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**.

---

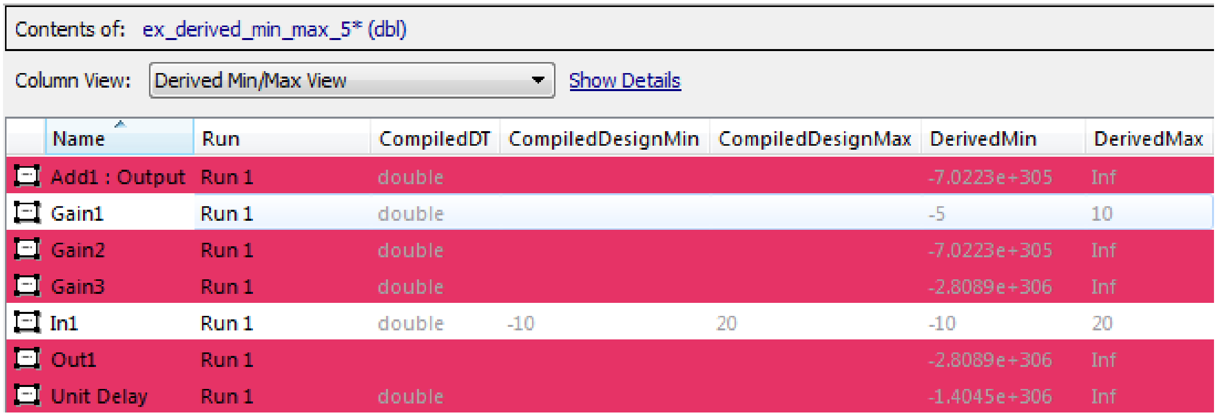
2 From the Simulink **Analysis** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

4 In the Fixed-Point Tool, click the **Derive min/max values for selected system** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model. Because one of the Add block inputs is fed back from its output, 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* (dbl)

Column View: Derived Min/Max View ▾ Show Details

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Add1 : Output	Run 1	double			-7.0223e+305	Inf
Gain1	Run 1	double			-5	10
Gain2	Run 1	double			-7.0223e+305	Inf
Gain3	Run 1	double			-2.8089e+306	Inf
In1	Run 1	double	-10	20	-10	20
Out1	Run 1				-2.8089e+306	Inf
Unit Delay	Run 1	double			-1.4045e+306	Inf

5 To fix the issue, specify design minimum and maximum values for `Out1`:

a In the model, double-click the Outport block, `Out1`.

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**.

6 In the Fixed-Point Tool, click the **Derive min/max values for selected system** button to rerun the range analysis.

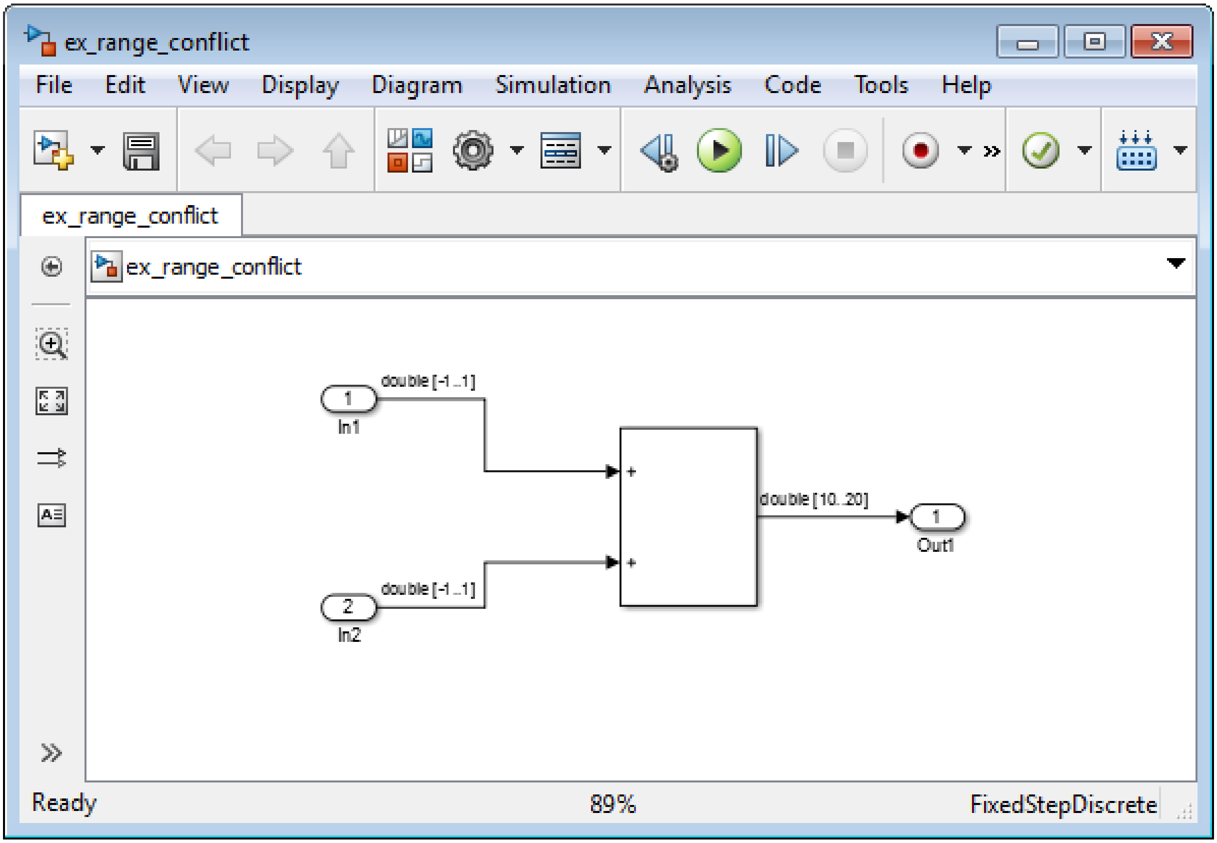
The range analysis uses the minimum and maximum values specified for `Out1`, `[-20..40]` and the gain value of `Gain3`, `2`, to derive an input range for `Gain3`, `[-10..20]`. Because the input of `Gain3` feeds back to the input of the Add block, the analysis now derives ranges for all objects in the model.

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 **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

   This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

4 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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]`.

5 Close the error dialog box.

6 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**.

7 In the Fixed-Point Tool, click the **Derive min/max values for selected system** 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.

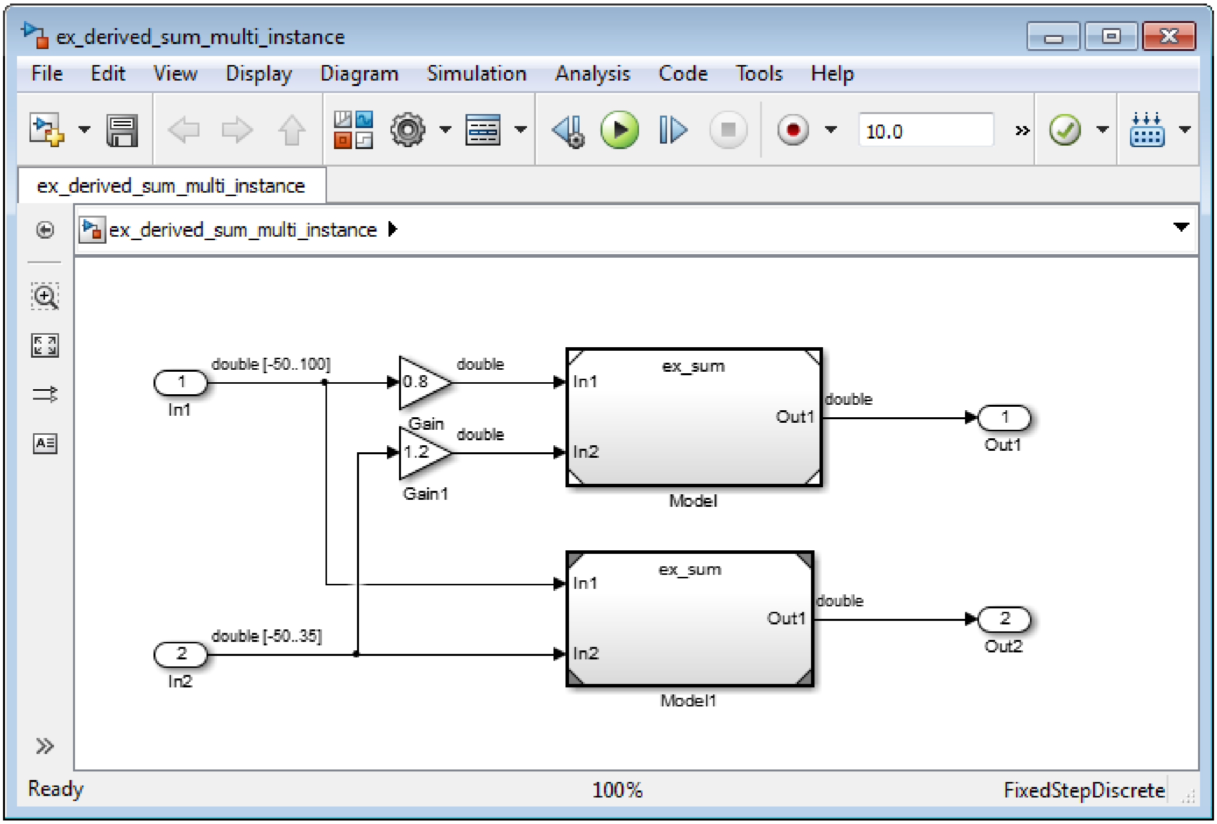
Derive Ranges for a Referenced Model

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
```

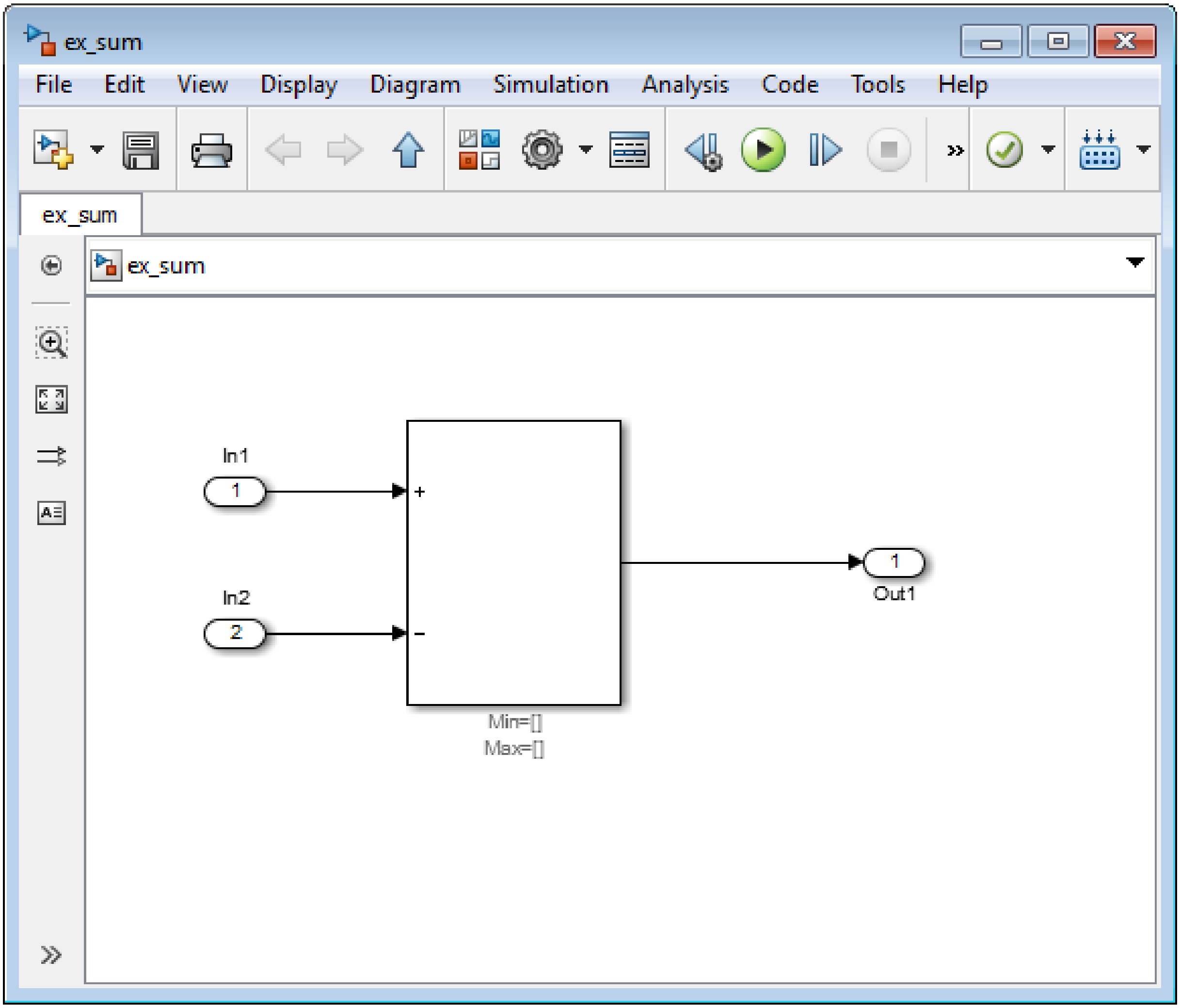


The model displays the specified design minimum and maximum values for the blocks in the model.

- 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.


ex_sum
File Edit View Display Diagram Simulation Analysis Code Tools Help
ex_sum
ex_sum
In1
1
+
In2
2
–
1
Out1
Min=[]
Max=[]

Initially, the Sum block has no design range information.

2 From the `ex_derived_sum_multi_instance` model **Analysis** menu, select **Fixed-Point Tool**.

3 In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_derived_sum_multi_instance` model.

4 In the **Settings for selected system** pane, set **Data type override** to `Double`.

   This global override of the fixed-point data types using double-precision numbers avoids quantization effects.

5 In the Fixed-Point Tool Data collection pane, set **Store results in run** 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 In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_sum` model.

   The **Data type override** setting is `Off`. The setting in the parent model does not affect the setting in the referenced model — you must change it manually in the referenced model.

7 For the `ex_sum` model, set **Data type override** to `Double` and then click **Apply**.

   Changing the setting for any instance of the referenced model changes the setting on all instances and on the referenced model itself.

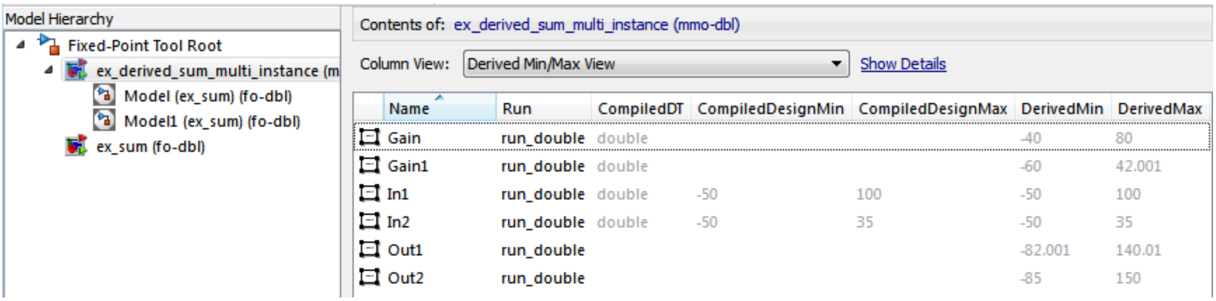
8 Save the models in a local writable folder.

   The Fixed-Point Tool cannot derive ranges if your model contains unsaved changes.

9 In the Fixed-Point Tool, select the `ex_derived_sum_multi_instance` model and then click **Derive min/max values for selected system**.

   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.



Model Hierarchy

- Fixed-Point Tool Root
  - ex_derived_sum_multi_instance (m
    - 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

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	run_double	double			-40	80
Gain1	run_double	double			-60	42.001
In1	run_double	double	-50	100	-50	100
In2	run_double	double	-50	35	-50	35
Out1	run_double				-82.001	140.01
Out2	run_double				-85	150

View Derived Ranges for Referenced Model

1 In the **Model Hierarchy** pane, select the first instance of the referenced model, `Model(ex_sum)`.

The tool displays the derived minimum and maximum values for this instance of the referenced model, `[-82.001..140.001]`. This range is derived from the outputs of the two Gain blocks, `[-40,80]` and `[-60..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.

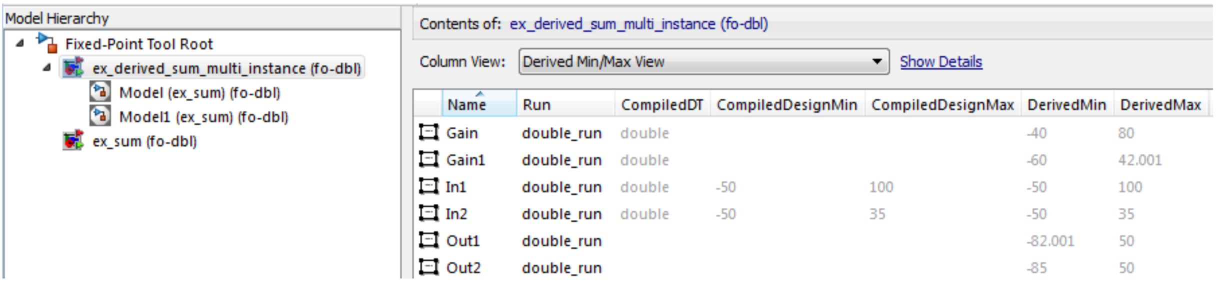
Add Design Range for Sum Block and Derive Ranges

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. In the Fixed-Point Tool, select the `ex_derived_sum_multi_instance` model and then click **Derive min/max values for selected system**.

   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.



Contents of: ex_derived_sum_multi_instance (fo-dbl)

Column View: Derived Min/Max View

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	double_run	double			-40	80
Gain1	double_run	double			-60	42.001
In1	double_run	double	-50	100	-50	100
In2	double_run	double	-50	35	-50	35
Out1	double_run				-82.001	50
Out2	double_run				-85	50

4. You can now propose data types for the model based on these derived minimum and maximum values.

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" example.

1 In the Fixed-Point Tool **Model Hierarchy** pane, select the `ex_derived_sum_multi_instance` model.

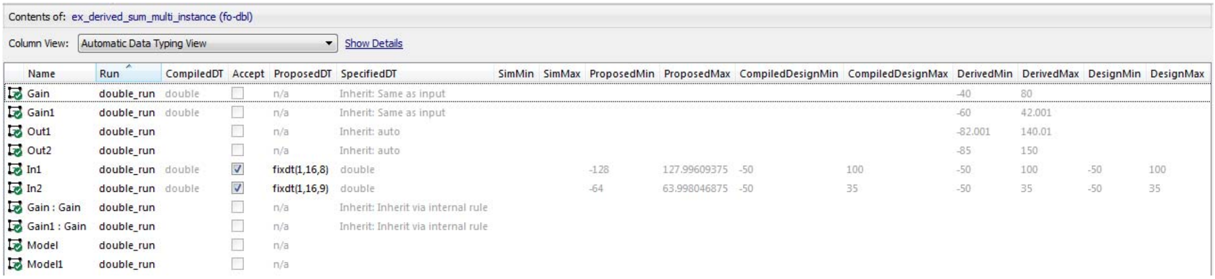
2 In the **Automatic data typing for selected system** pane, click the **Configure** link, set **Default data type of all floating-point signals** to `fixdt(1,16,4)` and then click **Apply**.

3 In the same pane, click **Propose fraction lengths**, DT.

   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.

   The tool does not propose data types for the other blocks because they use inherited data types. Instead, it displays `n/a` in the **ProposedDT** column. The Fixed-Point Tool might not be able to propose data types for other reasons, to view more information, click the **Show details for selected result** button ⓘ.

   The tool displays the proposed scaling in its **Contents** pane. It displays the Automatic Data Typing View to provide information, such as **ProposedDT**, **ProposedMin**, **ProposedMax**, which are relevant at this stage of the fixed-point conversion.


Contents of: ex_derived_sum_multi_instance (fo-dbl)

Column View: Automatic Data Typing View

Name	Run	CompiledDT	Accept	ProposedDT	SpecifiedDT	SimMin	SimMax	ProposedMin	ProposedMax	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax	DesignMin	DesignMax
Gain	double_run	double	□	n/a	Inherit: Same as input							-40	80		
Gain1	double_run	double	□	n/a	Inherit: Same as input							-60	42.001		
Out1	double_run		□	n/a	Inherit: auto							-82.001	140.01		
Out2	double_run		□	n/a	Inherit: auto							-85	150		
In1	double_run	double	☑	fixdt(1,16,8)	double			-128	127.99609375	-50	100	-50	100	-50	100
In2	double_run	double	☑	fixdt(1,16,9)	double			-64	63.998046875	-50	35	-50	35	-50	35
Gain : Gain	double_run		□	n/a	Inherit: Inherit via internal rule										
Gain1 : Gain	double_run		□	n/a	Inherit: Inherit via internal rule										
Model	double_run		□	n/a											
Model1	double_run		□	n/a											

4 After reviewing the data type proposals, click **Apply accepted fraction lengths** to apply the proposed data types to your model.

Deriving Ranges for a Referenced Model

In this section...

"Viewing Derived Minimum and Maximum Values for Referenced Models" on page 35-39

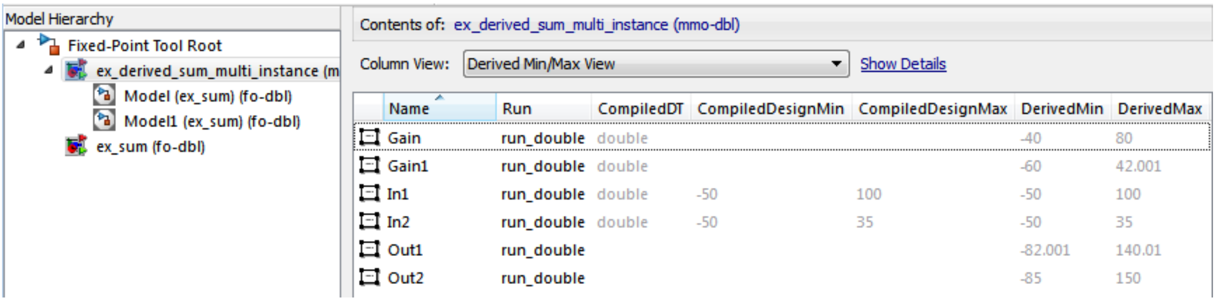
"Data Type Override Settings" on page 35-40

"See Also" on page 35-40

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.



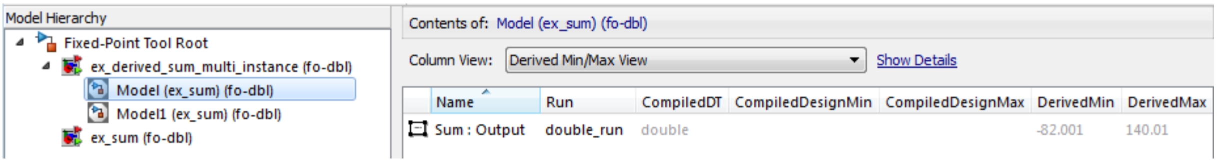
Model Hierarchy
- Fixed-Point Tool Root
  - ex_derived_sum_multi_instance (m
    - Model (ex_sum) (fo-dbl)
    - Model1 (ex_sum) (fo-dbl)
  - ex_sum (fo-dbl)

Contents of: ex_derived_sum_multi_instance (mmo-dbl)

Column View: Derived Min/Max View

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	run_double	double			-40	80
Gain1	run_double	double			-60	42.001
In1	run_double	double	-50	100	-50	100
In2	run_double	double	-50	35	-50	35
Out1	run_double				-82.001	140.01
Out2	run_double				-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_sum1` in `ex_multi_instance`.



Model Hierarchy

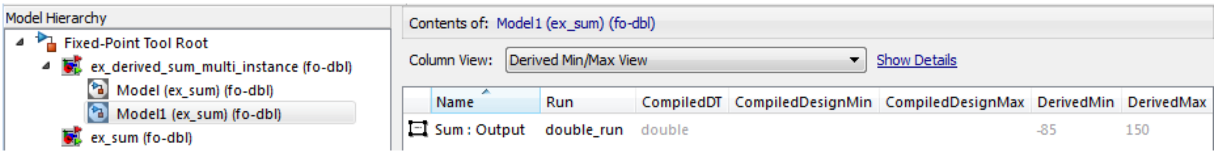
- Fixed-Point Tool Root
  - ex_derived_sum_multi_instance (fo-dbl)
    - Model (ex_sum) (fo-dbl)
    - Model1 (ex_sum) (fo-dbl)
  - ex_sum (fo-dbl)

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_sum1`.



Model Hierarchy

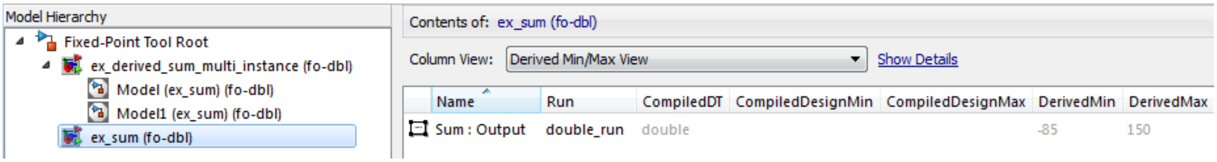
- Fixed-Point Tool Root
  - ex_derived_sum_multi_instance (fo-dbl)
    - Model (ex_sum) (fo-dbl)
    - Model1 (ex_sum) (fo-dbl)
  - ex_sum (fo-dbl)

Contents of: Model1 (ex_sum) (fo-dbl)

Column View: Derived Min/Max View ▾ Show Details

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.



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: ex_sum (fo-dbl)

Column View: Derived Min/Max View ▾ Show Details

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Sum : Output	double_run	double			-85	150

Data Type Override Settings

When you derive minimum and maximum values for a model that contains referenced models, the data type override setting for the top-level model does not control the setting for the referenced models. You must specify the data type override setting separately for the referenced model.

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.

See Also

- “Derive Ranges for a Referenced Model” on page 35-31

Unsupported Simulink Software Features

The software does not support the following Simulink software features. Avoid using these unsupported features.

Not Supported	Description
Variable-step solvers	The software supports only fixed-step solvers. For more information, see “Choosing a Fixed-Step Solver”.
Callback functions	The software does not execute model callback functions during the analysis. The results that the analysis generates may behave inconsistently with the expected behavior. • 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 `PreLoadFcn` or `PostLoadFcn` model callbacks, are fully supported.
Model callback functions	The software only supports model callback functions if the `InitFcn` callback of the model is empty.
Algebraic loops	The software does not support models that contain algebraic loops. For more information, see “Algebraic Loops”.
Masked subsystem initialization functions	The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.
Complex signals	The software supports only real signals. For more information, see “Complex Signals”.

Not Supported	Description
Variable-size signals	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.
Arrays of buses	The software does not support arrays of buses. For more information, see "Combine Buses into an Array of Buses".
Multiword fixed-point data types	The software does not support multiword fixed-point data types.
Nonfinite data	The software does not support nonfinite data (for example, `NaN` and `Inf`) and related operations.
Signals with nonzero sample time offset	The software does not support models with signals that have nonzero sample time offsets.
Models with no output ports	The software only supports models that have one or more output ports.

Supported and Unsupported Simulink Blocks

Overview of Simulink Block Support

The following tables summarize the 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 35-5.

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	Not supported and not stubbable
PID Controller	Not supported
PID Controller (2 DOF)	Not supported
Second Order Integrator	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	The software analyzes through the filter. It does not derive any range information for the filter.
Discrete PID Controller	Supported
Discrete PID Controller (2 DOF)	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

Block	Support Notes
Transfer Fcn First Order	Supported
Transfer Fcn Lead or Lag	Supported
Transfer Fcn Real Zero	Supported
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: • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `4`. or • 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`.

Block	Support Notes
n-D Lookup Table	Not supported when: • The **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`. or • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `5`.
Lookup Table Dynamic	Supported
Prelookup	Supported
Sine	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	Not supported
Divide	Supported
Dot Product	Supported
Find Nonzero Elements	Supported
Gain	Supported
Magnitude-Angle to Complex	Not supported

<table>
<tr><th>Block</th><th>Support Notes</th></tr>
<tr><td>Math Function</td><td>All signal types support the following <b>Function</b> parameter settings.
<table>
<tr><td><code>conj</code></td><td><code>hermitian</code></td><td><code>magnitude^2</code></td><td><code>mod</code></td></tr>
<tr><td><code>rem</code></td><td><code>reciprocal</code></td><td><code>square</code></td><td><code>transpose</code></td></tr>
</table>
The software does not support the following <b>Function</b> parameter settings.
<table>
<tr><td><code>10^u</code></td><td><code>exp</code></td><td><code>hypot</code></td></tr>
<tr><td><code>log</code></td><td><code>log10</code></td><td><code>pow</code></td></tr>
</table></td></tr>
<tr><td>Matrix Concatenate</td><td>Supported</td></tr>
<tr><td>MinMax</td><td>Supported</td></tr>
<tr><td>MinMax Running Resettable</td><td>Supported</td></tr>
<tr><td>Permute Dimensions</td><td>Supported</td></tr>
<tr><td>Polynomial</td><td>Supported</td></tr>
<tr><td>Product</td><td>Supported</td></tr>
<tr><td>Product of Elements</td><td>Supported</td></tr>
<tr><td>Real-Imag to Complex</td><td>Not supported</td></tr>
<tr><td>Reciprocal Sqrt</td><td>Not supported</td></tr>
<tr><td>Reshape</td><td>Supported</td></tr>
<tr><td>Rounding Function</td><td>Supported</td></tr>
<tr><td>Sign</td><td>Supported</td></tr>
<tr><td>Signed Sqrt</td><td>Not supported</td></tr>
<tr><td>Sine Wave Function</td><td>Not supported</td></tr>
<tr><td>Slider Gain</td><td>Supported</td></tr>
<tr><td>Sqrt</td><td>Not supported</td></tr>
</table>

Block	Support Notes
Squeeze	Supported
Subtract	Supported
Sum	Supported
Sum of Elements	Supported
Trigonometric Function	Supported when **Function** is `sin`, `cos`, `sincos` , or `atan2`, and **Approximation method** is `CORDIC`.
Unary Minus	Supported
Vector Concatenate	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

Block	Support Notes
Enable	Supported
Enabled Subsystem	Range analysis does not consider the design minimum and maximum values specified for blocks connected to the outport 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 outport of the subsystem.
For Each	Not supported
For Each Subsystem	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 outport 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 35-53.
Model Variants	Supported except for the limitations described in “Limitations of Support for Model Blocks” on page 35-53.
Outport	Supported
Subsystem	Supported
Switch Case	Supported
Switch Case Action Subsystem	Supported

Block	Support Notes
Trigger	Supported
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 outport 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	Range analysis does not consider the minimum and maximum values specified on the Data Store Memory block. When deriving ranges, the analysis considers only local model writes. It does not consider updates to global data store memory by other processes.
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
Clock	Supported
Constant	Supported unless **Constant value** is `inf`.
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 `timeseries` 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 `^`. Supports only the mathematical functions `abs`, `ceil`, `fabs`, `floor`, `rem`, and `sgn`.
Interpreted MATLAB Function	Not supported

MATLAB Function	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:
	`function y = fcn(u,v)`
	`%#codegen`
	`y = 2*u;`
	`y = y + v;`
	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 outport of the block.
	The analysis does not consider design ranges for the instance of `y` in `y = 2*u;` because it is an intermediate instance.

Level-2 MATLAB S-Function	Not supported
S-Function	Not supported
S-Function Builder	Not supported

Limitations of Support for Model Blocks

The software supports the Model block, but with the following limitations. The software cannot analyze a model that contains one or more Model blocks if:

- The referenced model is protected. Protected referenced models are encoded to obscure their contents. This feature allows third parties to use the referenced model without being able to view the intellectual property that makes up the model.

  Note For more information, see "Protected Model".

- The parent model or any of the referenced models gives 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".

- The parent model and the referenced model have mismatched data type override settings. The data type override setting of the parent model and all of 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.

Code Generation

- "Generating and Deploying Production Code" on page 36-2
- "Code Generation Support" on page 36-3
- "Accelerating Fixed-Point Models" on page 36-5
- "Using External Mode or Rapid Simulation Target" on page 36-7
- "Optimize Your Generated Code" on page 36-9
- "Optimizing Your Generated Code with the Model Advisor" on page 36-34

Generating and Deploying Production Code

You can generate C code with the Fixed-Point Designer software by using the Simulink Coder product. The code generated from fixed-point models uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations. You can use the generated code on embedded fixed-point processors or on rapid prototyping systems even if they contain a floating-point processor. For more information about code generation, refer to the Simulink Coder documentation.

You can generate code for testing on a rapid prototyping system using products such as xPC Target™, Real-Time Windows Target™, or dSPACE® software. The target compiler and processor may support floating-point operations in software or in hardware. In any case, the fixed-point portions of a model generate pure integer code and do not use floating-point operations. This allows valid bit-true testing even on a floating-point processor.

You can also generate code for non-real-time testing. For example, you can generate code to run in nonreal time on computers running any supported operating system. Even though the processors have floating-point hardware, the code generated by fixed-point blocks is pure integer code. The Generic Real-Time Target (GRT) in the Simulink Coder product and acceleration modes in the Simulink software are examples of where non-real-time code is generated and run.

When used with HDL Coder, Fixed-Point Designer lets you generate bit-true synthesizable Verilog® and VHDL® code from Simulink models, Stateflow charts, and MATLAB Function blocks.

Code Generation Support

In this section...

"Introduction" on page 36-3

"Languages" on page 36-3

"Data Types" on page 36-3

"Rounding Modes" on page 36-3

"Overflow Handling" on page 36-3

"Blocks" on page 36-4

"Scaling" on page 36-4

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.

Data Types

Fixed-point code generation supports all integer and fixed-point data types that are supported by simulation. See "Supported Data Types" on page 26-20.

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`.

Scaling

Any binary-point-only scaling and [Slope Bias] scaling that is supported in simulation is supported, bit-true, in code generation.

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 36-7

"External Mode" on page 36-7

"Rapid Simulation Target" on page 36-8

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 27-15 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.

Optimize Your Generated Code

In this section...

"Reducing ROM Consumption or Model Execution Time" on page 36-9

"Restrict Data Type Word Lengths" on page 36-10

"Avoid Fixed-Point Scalings with Bias" on page 36-11

"Wrap and Round to Floor or Simplest" on page 36-11

"Limit the Use of Custom Storage Classes" on page 36-13

"Limit the Use of Unevenly Spaced Lookup Tables" on page 36-13

"Minimize the Variety of Similar Fixed-Point Utility Functions" on page 36-13

"Handle Net Slope Correction" on page 36-14

"Use Integer Division to Handle Net Slope Correction" on page 36-15

"Improve Numerical Accuracy of Simulation Results with Integer Division to Handle Net Slope" on page 36-16

"Improve Efficiency of Generated Code with Integer Division to Handle Net Slope" on page 36-21

"Optimize Generated Code Using Specified Minimum and Maximum Values" on page 36-27

"Eliminate Unnecessary Utility Functions Using Specified Minimum and Maximum Values" on page 36-30

Reducing ROM Consumption or Model Execution Time

Tip	Reduces ROM	Reduces Model Execution Time
"Restrict Data Type Word Lengths" on page 36-10	Yes	Yes
"Avoid Fixed-Point Scalings with Bias" on page 36-11	Yes	Yes
"Wrap and Round to Floor or Simplest" on page 36-11	Yes	Yes

Tip	Reduces ROM	Reduces Model Execution Time
“Limit the Use of Custom Storage Classes” on page 36-13	Yes	No
“Limit the Use of Unevenly Spaced Lookup Tables” on page 36-13	Yes	Yes
“Minimize the Variety of Similar Fixed-Point Utility Functions” on page 36-13	Yes	No
“Handle Net Slope Correction” on page 36-14	Dependent on model configuration, compiler, and target hardware	Dependent on model configuration, compiler, and target hardware
“Optimize Generated Code Using Specified Minimum and Maximum Values” on page 36-27	Yes	Yes

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 28-33 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 28-14.

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 36-11, 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.

Limit the Use of Unevenly Spaced Lookup Tables

If possible, use lookup tables with nontunable, evenly spaced axes. A table with an unevenly spaced axis requires a search routine and memory for each input axis, which increases ROM and execution time. However, keep in mind that an unevenly spaced lookup table might provide greater accuracy. You need to consider the needs of your algorithm to determine whether you can forgo some accuracy with an evenly spaced table in order to reduce ROM and execution time. Also note that this decision applies only to lookup tables with nontunable input axes, because tables with tunable input axes always have the potential to be unevenly spaced.

Minimize the Variety of Similar Fixed-Point Utility Functions

The Embedded Coder product generates fixed-point utility functions that are designed to handle specific situations efficiently. The Simulink Coder product can generate multiple versions of these optimized utility functions depending on what a specific model requires. For example, the division of `long` integers can, in theory, require eight varieties that are combinations of the output and the two inputs being signed or unsigned. A model that uses all these combinations can generate utility functions for all these combinations.

In some cases, it is possible to make small adjustments to a model that reduce the variety of required utility functions. For example, suppose that across most of a model signed data types are used, but in a small part of a model, a local decision to use unsigned data types is made. If it is possible to switch that portion of the model to use signed data types, then the overall variety of generated utility functions can potentially be reduced.

The best way to identify these opportunities is to inspect the generated code. For each utility function that appears in the generated code, you can search for all the call sites. If relatively few calls to the function are made, then trace back from the call site to the Simulink model. By modifying those places in the Simulink model, it is possible for you to eliminate the few cases that need a rarely used utility function.

Handle Net Slope Correction

The Fixed-Point Designer software provides an optimization parameter, **Use integer division to handle net slopes that are reciprocals of integers**, that controls how the software handles net slope correction. To learn how to enable this optimization, see "Use Integer Division to Handle Net Slope Correction" on page 36-15.

When a change of fixed-point slope is not a power of two, net slope correction is necessary. Normally, net slope correction 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 accurately represented as the reciprocal of an integer. Under this condition, the division implementation gives 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 for the division implementation might require less ROM or improve model execution time.

When to Use Integer Division to Handle Net Slope Correction

This optimization works if:

- The net slope is a 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 Integer Division to Handle Net Slope Correction

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 integer division to handle net slope correction 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

  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 Integer Division to Handle Net Slope Correction

To enable this optimization:

1. Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

   For more information, see “Use integer division to handle net slopes that are reciprocals of integers”.

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`.

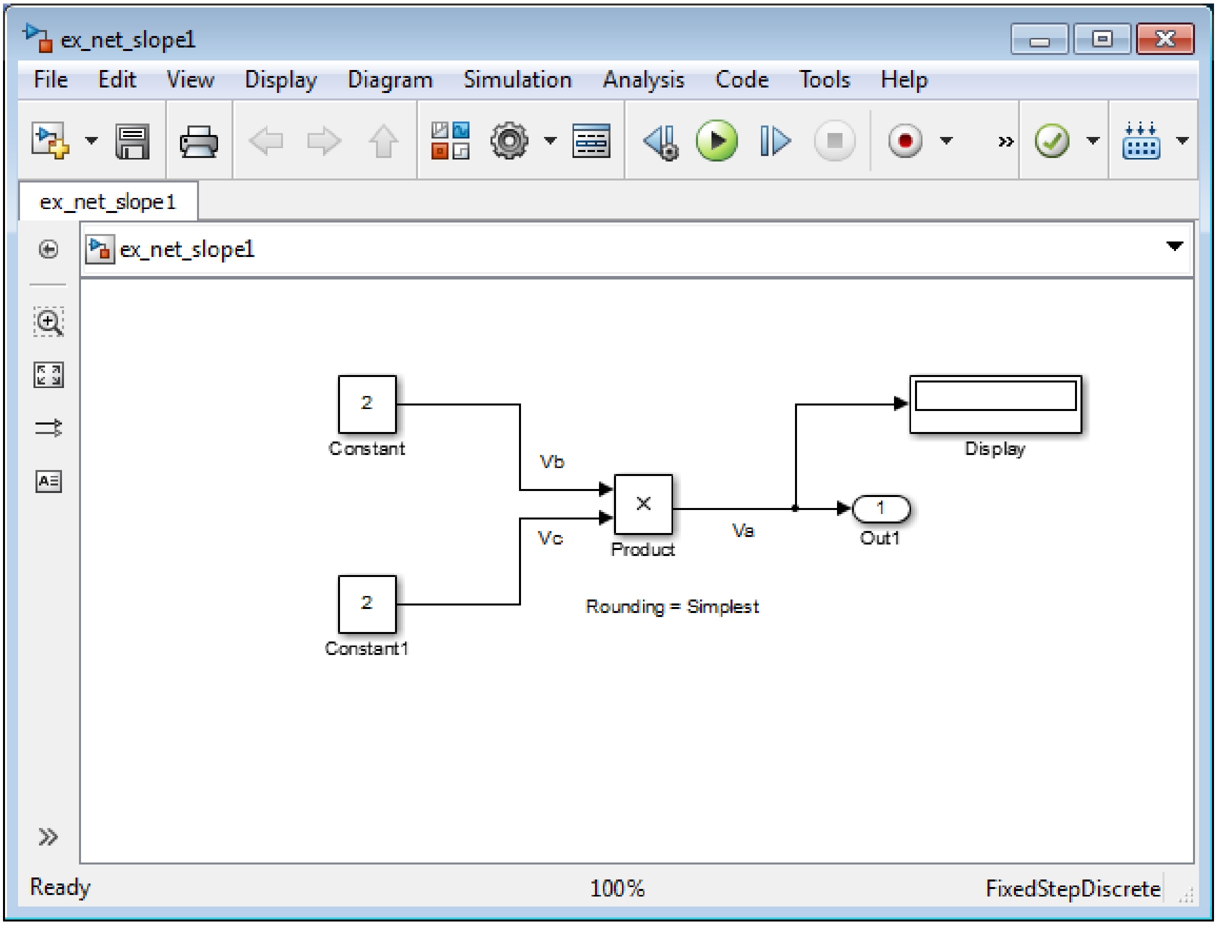
Note You must set this parameter to a value that is appropriate for the target hardware. Failure to do so might result in division operations that comply with the definition on the **Hardware Implementation** pane, but are inappropriate for the target hardware.

3. Set the **Integer rounding mode** of the blocks that require net slope correction (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 "Use the Model Advisor to Optimize Fixed-Point Operations in Generated Code" on page 36-42.

Improve Numerical Accuracy of Simulation Results with Integer Division to Handle Net Slope

This example illustrates how selecting the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter 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 27-6: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$S_a Q_a = S_b Q_b . S_c Q_c$, 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 `1/1000`. Because the net slope is the reciprocal of an integer, you can use the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter if your model and hardware configuration are suitable. For more information, see "When to Use Integer Division to Handle Net Slope Correction" on page 36-14.

To set up the model and run the simulation:

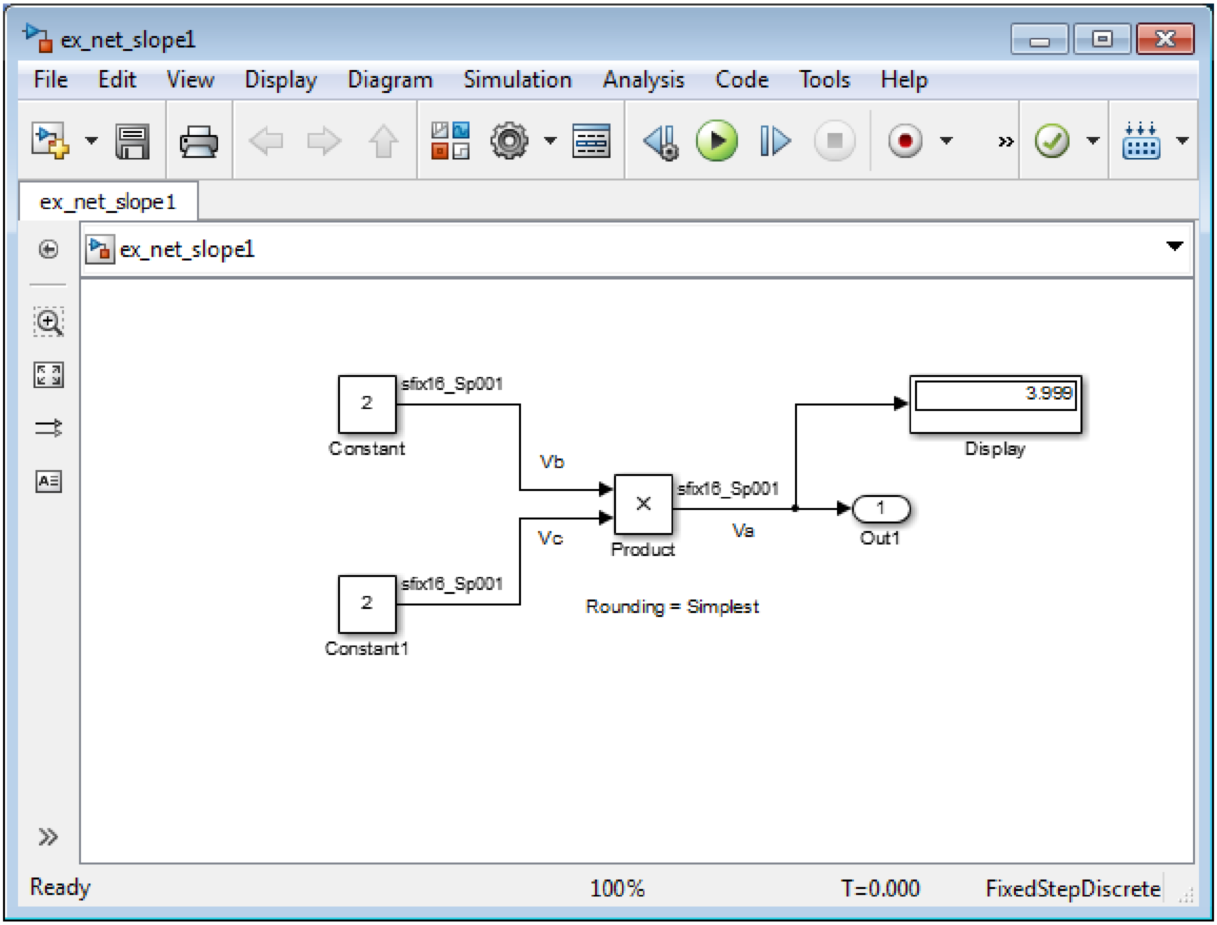
1. For the two Constant blocks, set the **Output data type** to `fixdt(1, 16, 1/1000, 0)`.

2. For the Product block, set the **Output data type** to `fixdt(1, 16, 1/1000, 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. Clear the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

5. In your Simulink model window, select **Simulation > Run**.

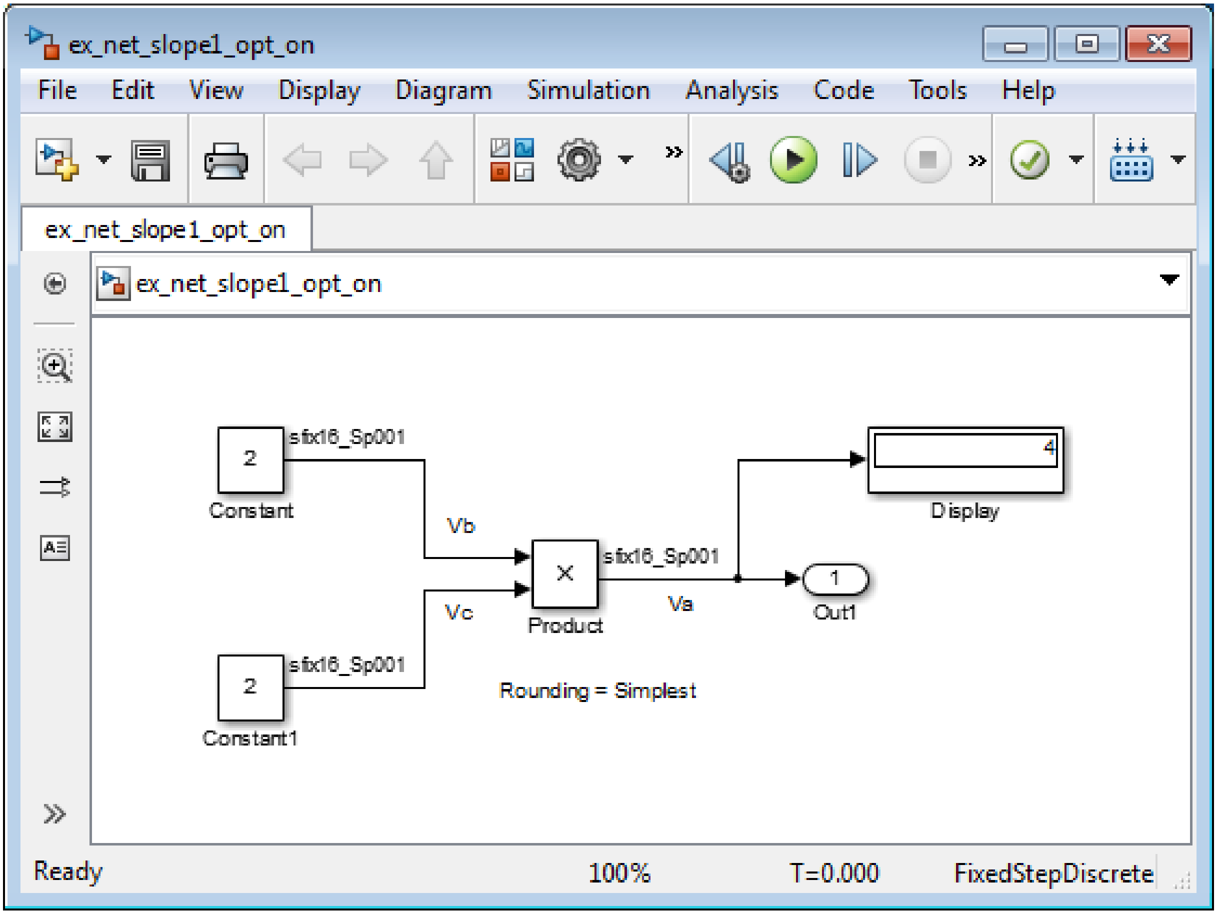


Because the simulation uses multiplication followed by shifts to handle the net slope correction, net slope precision loss occurs. This precision loss results in numerical inaccuracy: the calculated product is `3.999`, not `4`, as you expect.

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 28-21.

6 Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter, save your model, and simulate again.

The software implements the net slope correction using division instead of multiplication followed by shifts. The calculated product is 4, as you expect.



The optimization works for this model because:

- The net slope is a reciprocal of an integer.

- 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.

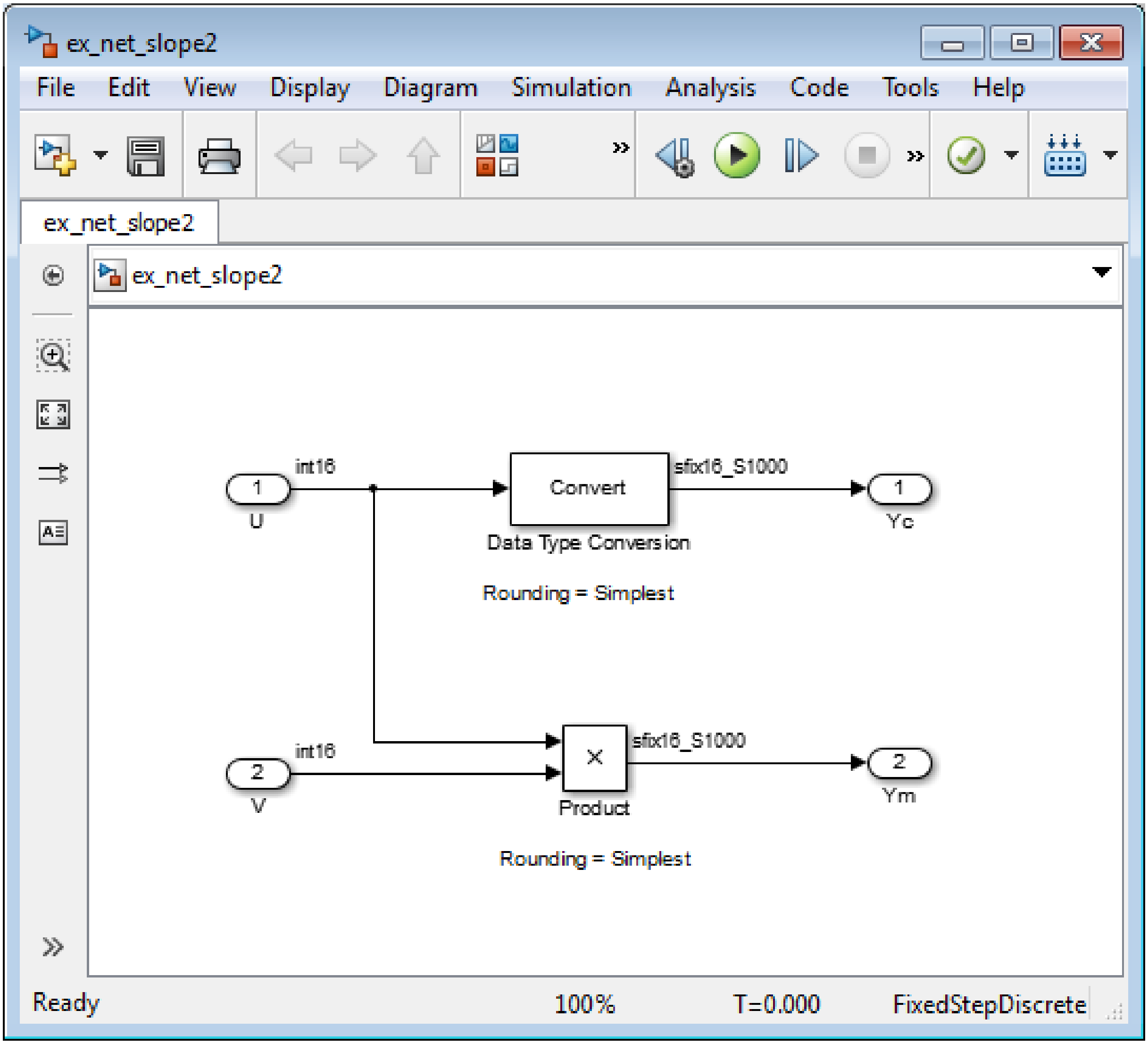
- The **Integer rounding mode** of the Product block in the model is set to `Simplest`.
- The model does not use multiword operations.

Improve Efficiency of Generated Code with Integer Division to Handle Net Slope

This example illustrates how selecting the **Use integer division to handle net slope correction** optimization parameter 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,

$$V_a = V_b \times V_c$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 27-6: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$S_a Q_a = S_b Q_b . S_c Q_c$, 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 `1/1000`.

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 `1/1000`.

Because the net slope is the reciprocal of an integer, you can use the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter if your model and hardware configuration are suitable. For more information, see “When to Use Integer Division to Handle Net Slope Correction” on page 36-14.

To set up the model and generate code:

1. For the two Inport blocks, `U` and `V`, set the **Data type** to `int16`.

2. For the Data Type Conversion block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.

3. For the Product block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.

4 Set the **Hardware Implementation > Production hardware > Signed integer division rounds to** configuration parameter to `Zero`.

5 Clear the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

6 From the Simulink model menu, select **Tools > Code > C/C++ Code > Build Model**.

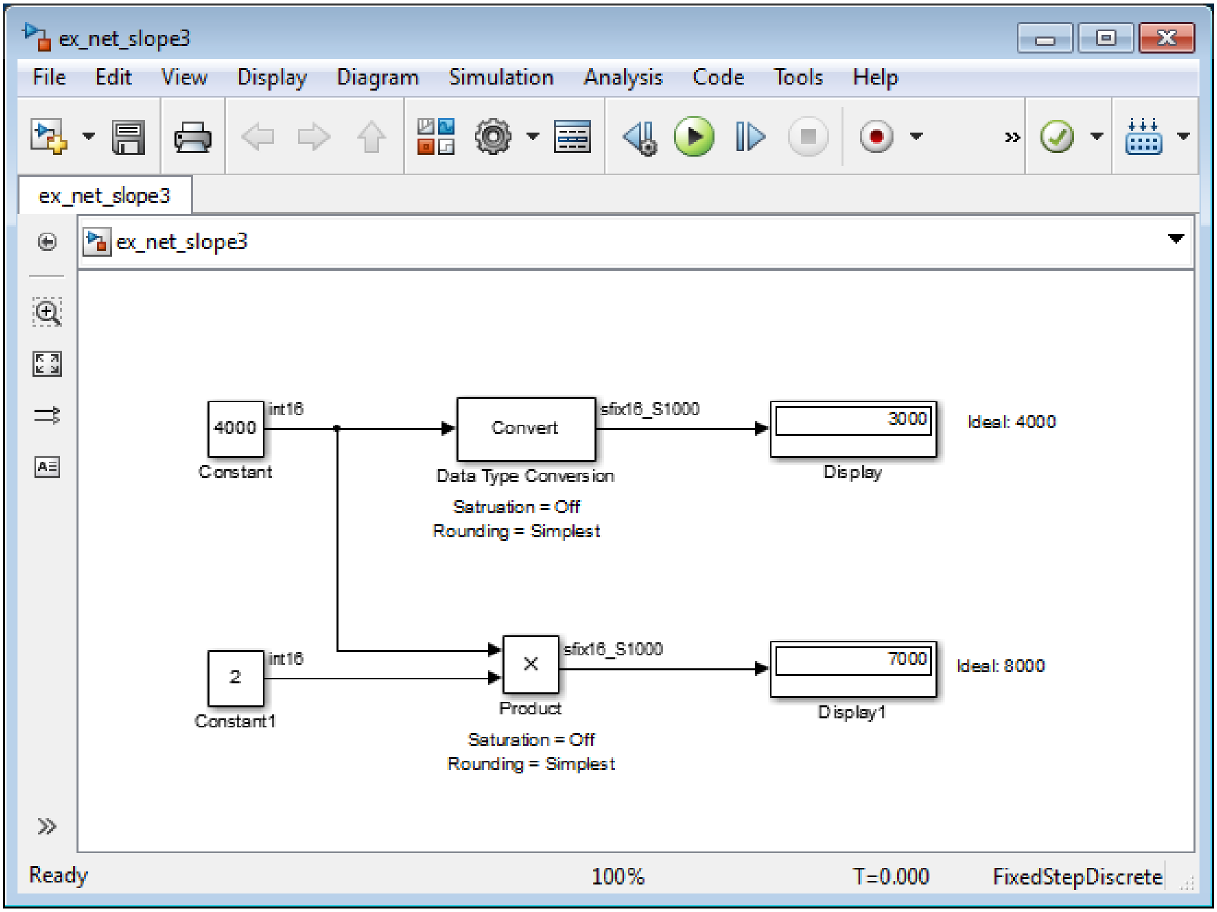
Conceptually, the net slope correction is `1/1000` or `0.001`:

```
Yc = 0.001 * U;
Ym = 0.001 * U * V;
```

The generated code uses multiplication with shifts:

```
% For the conversion
Yc = (int16_T)U * 16777 >> 24;
% For the multiplication
Ym = (int16_T)((int16_T)(U * V >> 10) * 16777 >> 14);
```

The ideal value of the net slope correction is `0.001`. In the generated code, the approximate value of the net slope correction is `16777L >> 24 = 16777/2^24 = 0.000999987125396729`. This approximation introduces numerical inaccuracy. For example, using the same model with constant inputs produces the following results.

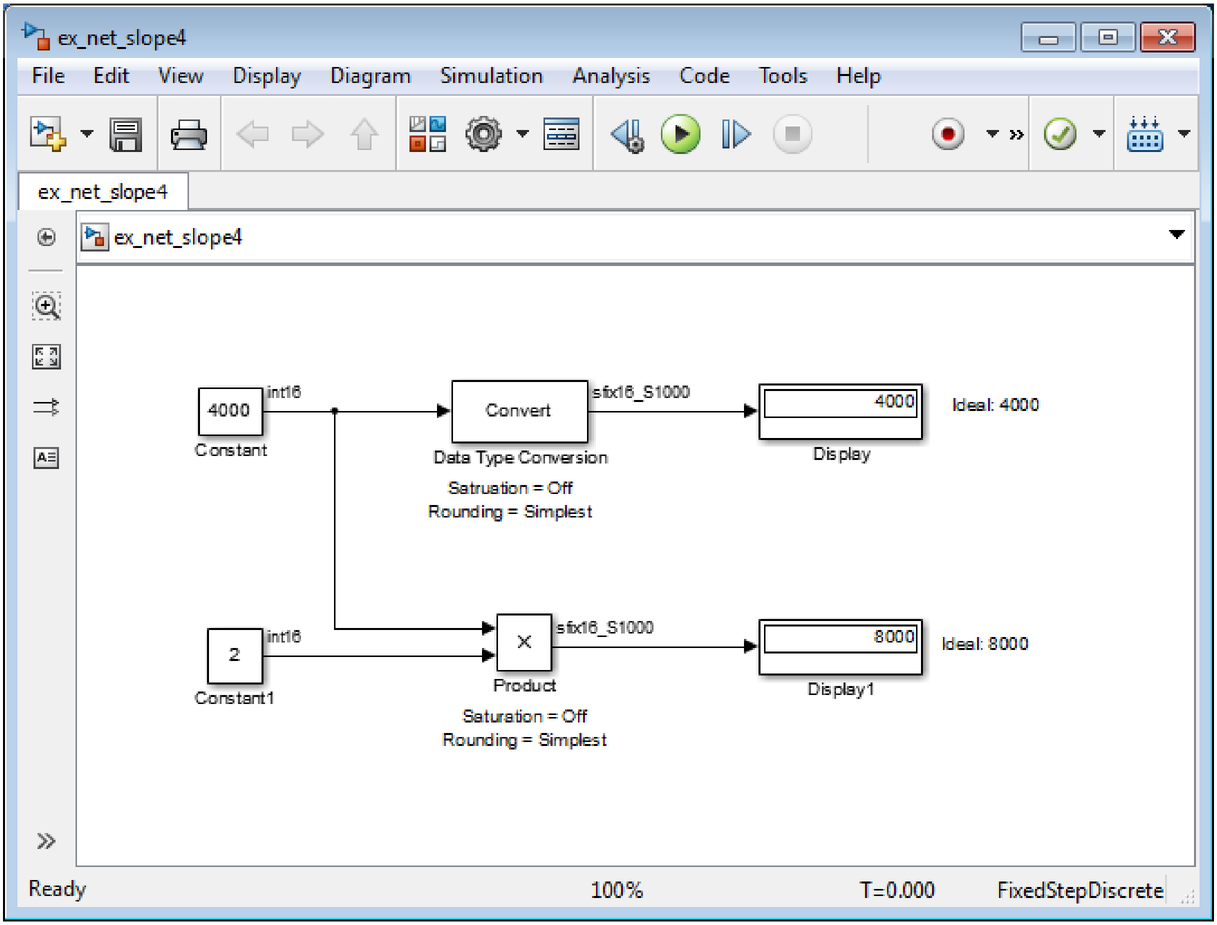


7 Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** optimization parameter, update diagram, and generate code again.

The generated code now uses integer division instead of multiplication followed by shifts:

```
% For the conversion
Yc = (int16_T)(U / 1000);
% For the multiplication
Ym = (int16_T)(U * V / 1000);
```

8 In the generated code, the value of the net slope correction is now the ideal value of `0.001`. Using division, the results are numerically accurate.



The optimization works for this model because the:

- Net slope is a reciprocal of an integer.
- **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.

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 36-30.

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 36-27.

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".

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 Outport block.

    Outport 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);
```

Optimizing Your Generated Code with the Model Advisor

In this section...

“Optimize Generated Code with Model Advisor” on page 36-34

“Identify Blocks that Generate Expensive Fixed-Point and Saturation Code” on page 36-35

“Identify Questionable Fixed-Point Operations” on page 36-38

“Identify Blocks that Generate Expensive Rounding Code” on page 36-41

“Use the Model Advisor to Optimize Fixed-Point Operations in Generated Code” on page 36-42

Optimize Generated Code with Model Advisor

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 Model Advisor left pane, expand the **By Product** node and select **Embedded Coder**.

3. From the Model Advisor **Edit** menu, select **Select All** to enable all Model Advisor checks associated with the selected node. 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**.

4. 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** > **Use initial condition as initial and reset value for** parameter to `State only (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 33-25. 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.

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 correction

When a change of fixed-point slope is not a power of two, net slope correction is necessary. Normally, net slope correction 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 select the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter, but your model configuration is not compatible with this selection.
- Your model configuration is suitable for using integer division to handle net slope correction, but you do not select the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter.

For more information, see “Handle Net Slope Correction” on page 36-14.

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 cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

Check for expensive multiplication code

- "Targeting an Embedded Processor" on page 29-4 discusses the capabilities and limitations of embedded processors. "Design Rules" on page 29-5 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 28-49 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** 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 33-25. 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.

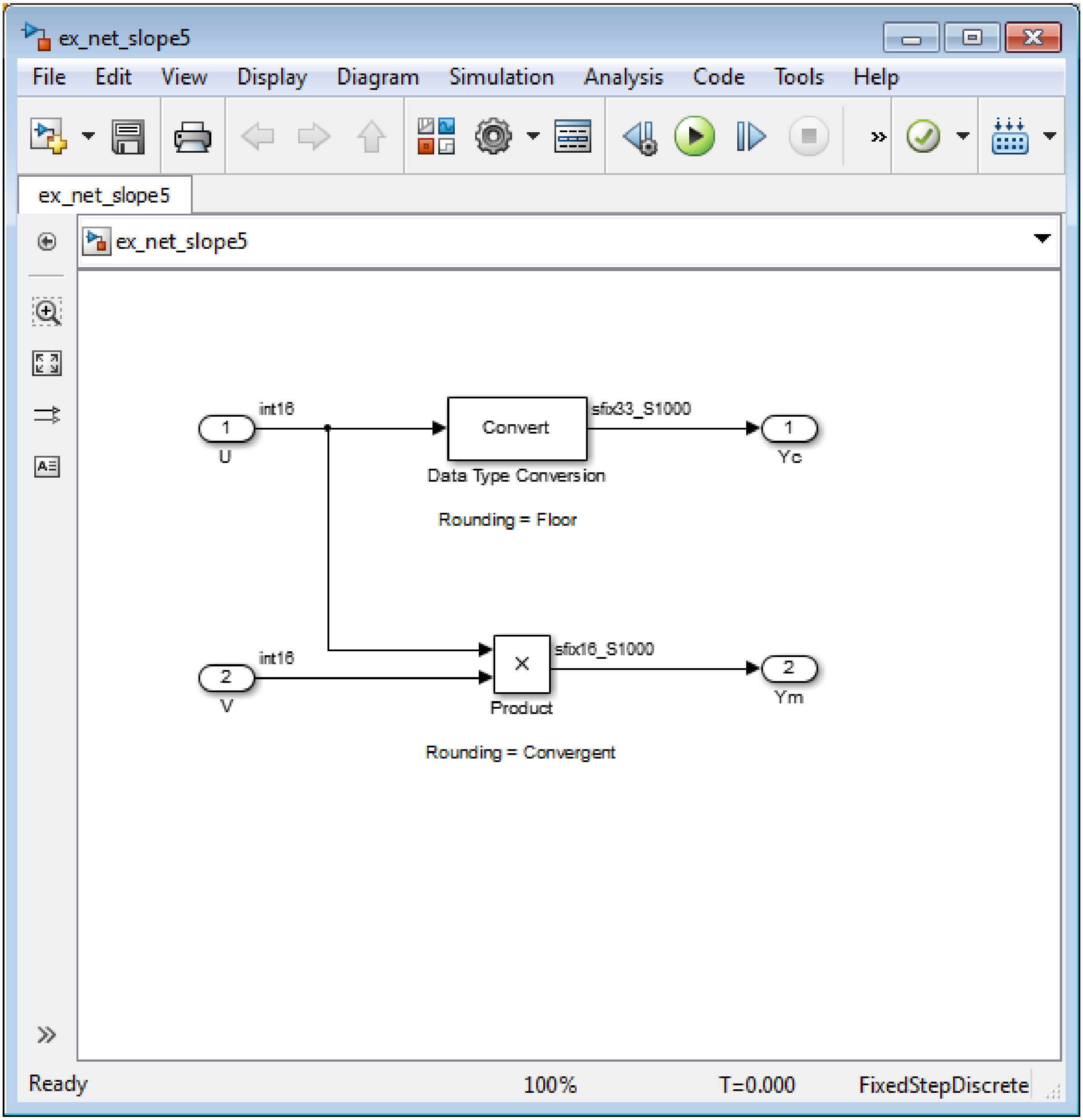
The primary exception to this rule is the rounding behavior of signed integer division. The C standard leaves this rounding behavior unspecified, but for most production targets the “no effort” mode is to round to zero. For unsigned division, everything is nonnegative, so rounding to floor and rounding 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 1-7.

Use the Model Advisor to Optimize Fixed-Point Operations in Generated Code

This example uses the following model.


ex_net_slope5
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
ex_net_slope5
ex_net_slope5
int16
1
U
Convert
Data Type Conversion
sfix33_S1000
1
Yc
Rounding = Floor
int16
2
V
×
Product
sfix16_S1000
2
Ym
Rounding = Convergent
Ready
100%
T=0.000
FixedStepDiscrete

Open the `ex_net_slope5` model. At the MATLAB command line, enter:

```
addpath(fullfile(docroot,'toolbox','fixpoint','examples'))
ex_net_slope5
```

In this model, the net slope for the Data Type Conversion and Product blocks is `1/1000`.

This model has the following properties:

- The **Data type** of the two Inport blocks, `U` and `V`, is set to `int16`.
- The **Integer rounding mode** of the Data Type Conversion block, is set to `Floor`, and the **Output data type** is set to `fixdt(1, 33, 1000, 0)`.

  Note Setting the **Output data type** word length greater than the length of the `long` data type results in multiword operations.

- The **Integer rounding mode** of the Product block, is set to `Convergent`, and the **Output data type** is set to `fixdt(1, 16, 1000, 0)`.
- The **Simulation > Model Configuration Parameters > Hardware Implementation > Production hardware > Signed integer division rounds to** configuration parameter is set to `Zero`.
- The checkbox under **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter is selected.

To run the Model Advisor checks:

1. From the model menu, select **Analysis > Model Advisor > Model Advisor**.
2. In the Model Advisor left pane, expand the **By Product** node and then expand the **Embedded Coder** node.

3 Select **Identify blocks that generate expensive fixed-point and saturation code**.

4 Click **Run this check**.

   The Model Advisor warns that your model contains a block that will invoke net slope correction.

5 Make the suggested changes

   a Double click the Data Type Conversion block and change the **Integer rounding mode** to `Simplest`.

   b Change the **Output data type** from `fixdt(1,33,1000,0)` to `fixdt(1,16,1000,0)`, to avoid multiword operations.

   c Save the model.

6 Rerun the check.

   The check now passes.

7 Run the Model Advisor **Identify questionable fixed-point operations** check.

   The check passes.

8 Select and run **Identify blocks that generate expensive rounding code**.

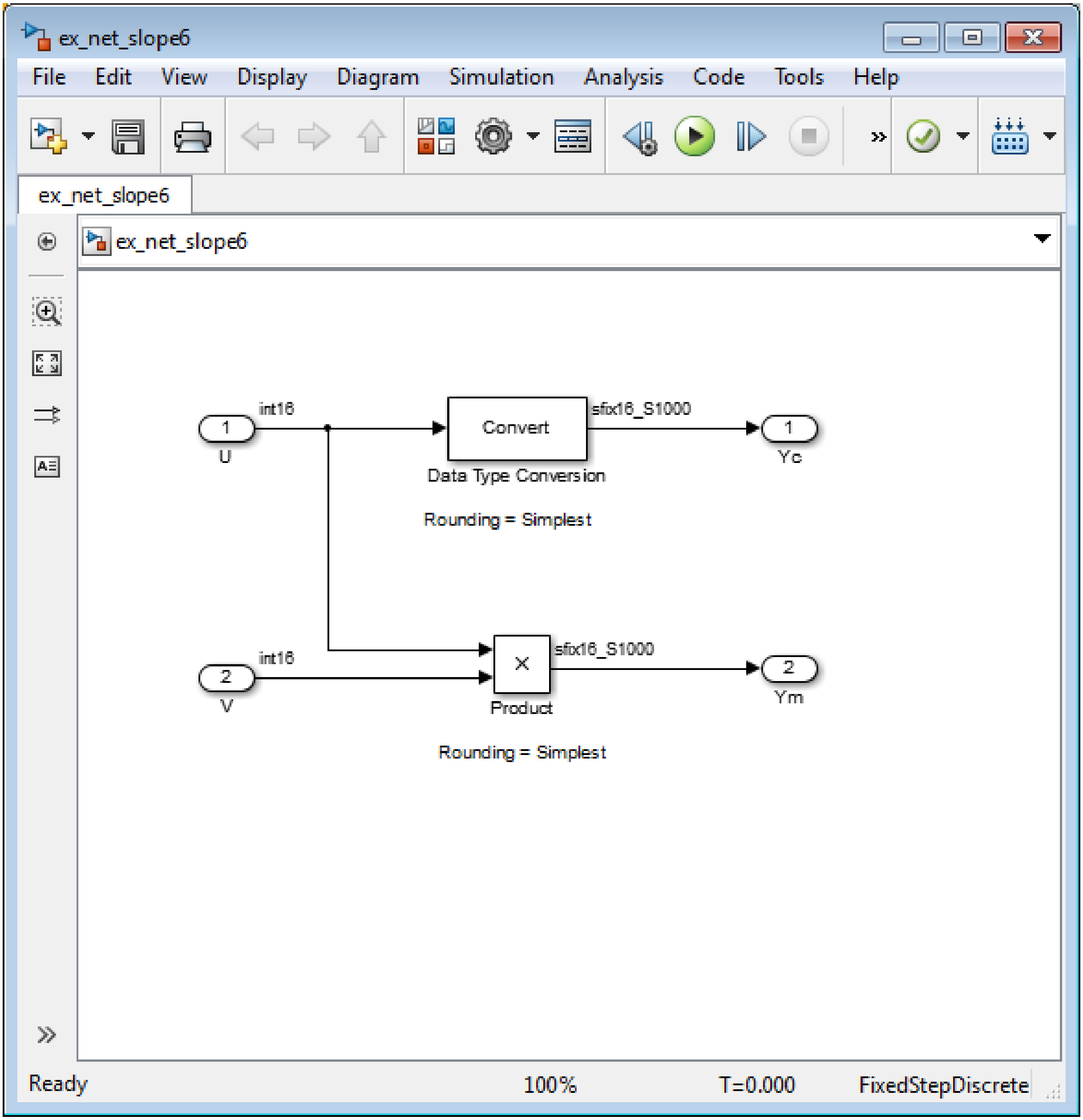
   The Model Advisor warns that your model contains blocks that will generate expensive rounding code for multiplication and division. It also provides recommendations on how to change your model configuration to make it more efficient.

9 Make the suggested changes:

   a Double click on the Product block, and change the **Integer rounding mode** to `Simplest`.

   b Save the model.

   This is your model configuration.


ex_net_slope6
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
ex_net_slope6
ex_net_slope6
int16
1
U
Convert
Data Type Conversion
sfix16_S1000
1
Yc
Rounding = Simplest
int16
2
V
×
Product
sfix16_S1000
2
Ym
Rounding = Simplest
Ready
100%
T=0.000
FixedStepDiscrete

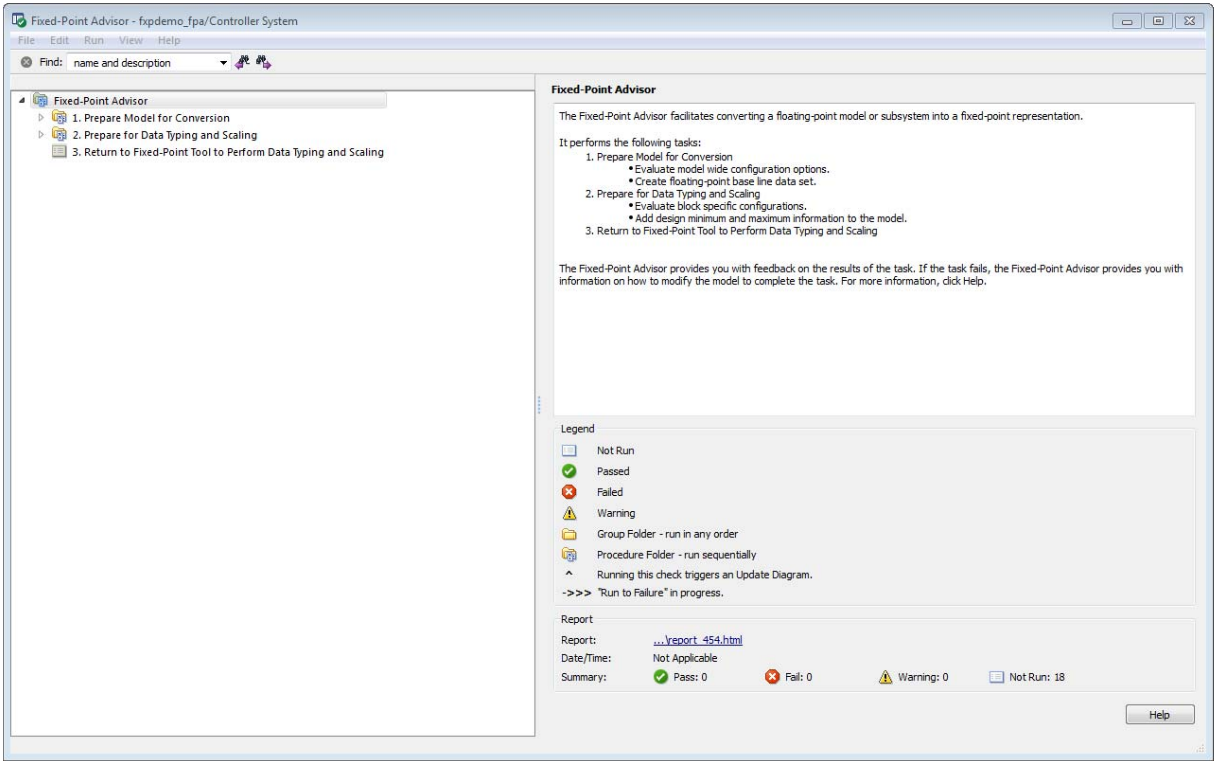
10 Rerun the Model Advisor **Identify blocks that generate expensive rounding code** check.

The check passes.

Fixed-Point Advisor Reference

- “Fixed-Point Advisor” on page 37-2
- “Preparing Model for Conversion” on page 37-6
- “Preparing for Data Typing and Scaling” on page 37-23
- “Return to the Fixed-Point Tool to Perform Data Typing and Scaling” on page 37-38

Fixed-Point Advisor


Fixed-Point Advisor - fxpdemo_fpa/Controller System
File Edit Run View Help
Find: name and description
Help

- Fixed-Point Advisor
  - 1. Prepare Model for Conversion
  - 2. Prepare for Data Typing and Scaling
  - 3. Return to Fixed-Point Tool to Perform Data Typing and Scaling

Fixed-Point Advisor

The Fixed-Point Advisor facilitates converting a floating-point model or subsystem into a fixed-point representation.

It performs the following tasks:

1. Prepare Model for Conversion
   - Evaluate model wide configuration options.
   - Create floating-point base line data set.
2. Prepare for Data Typing and Scaling
   - Evaluate block specific configurations.
   - Add design minimum and maximum information to the model.
3. Return to Fixed-Point Tool to Perform Data Typing and Scaling

The Fixed-Point Advisor provides you with feedback on the results of the task. If the task fails, the Fixed-Point Advisor provides you with information on how to modify the model to complete the task. For more information, click Help.

Legend

□	Not Run
✓	Passed
☒	Failed
⚠	Warning
🗀	Group Folder - run in any order
🗀	Procedure Folder - run sequentially
^	Running this check triggers an Update Diagram.
->>>	"Run to Failure" in progress.

Report

Report:	...\report_454.html			
Date/Time:	Not Applicable			
Summary:	✓ Pass: 0	☒ Fail: 0	⚠ Warning: 0	□ Not Run: 18

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 and removal of inheritance rules. It configures the model for autoscaling by the Fixed-Point Tool. Therefore, even if your model uses only fixed-point data types, it is 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 to Failure** 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 the run to failure 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 30-8. 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 30-2
- "Preparation for Fixed-Point Conversion" on page 30-2

Preparing Model for Conversion

In this section...

“Prepare Model for Conversion Overview” on page 37-7

“Verify model simulation settings” on page 37-8

“Verify update diagram status” on page 37-10

“Address unsupported blocks” on page 37-11

“Set up signal logging” on page 37-13

“Create simulation reference data” on page 37-14

“Verify Fixed-Point Conversion Guidelines Overview” on page 37-16

“Check model configuration data validity diagnostic parameters settings” on page 37-17

“Implement logic signals as Boolean data” on page 37-18

“Check bus usage” on page 37-19

“Simulation range checking” on page 37-20

“Check for implicit signal resolution” on page 37-21

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 30-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 30-11

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 following 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 off.	Set to on
The fiprefDataTypeOverride property is not set to `ForceOff'.	Set DataTypeOverride to 'ForceOff'
A Model Reference block is not in 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.	Configure all referenced models that are in non-Normal mode to use Normal mode. Use **Modify All** or manually configure each Model Reference block to use Normal mode: **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
	Tip Ignore this warning for protected models or any other models that you do not want the Fixed-Point Advisor to check.

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.

See Also

- “Data Type Override” on page 34-49
- “Signal logging”
- “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 a Block Diagram" 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
Blocks that do not support fixed-point data types and cannot be converted exist in model.	• Replace the block with the block specified in the Result pane by right-clicking the block and selecting the replacement from the context menu. **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. • Isolate the block by right-clicking the block and selecting **Insert Data Type Conversion** > **All Ports**. 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.

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.	If you are using simulation minimum and maximum values, specify at least one signal to be logged. Otherwise, ignore this warning. The Fixed-Point Advisor provides separate results for each referenced model. Specify at least one signal to be logged for each referenced model.

Tips

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 instrumentation 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`	If you are using simulation minimum and maximum values, set **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`. Otherwise, ignore this warning. 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.

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.

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” and “Stop time” 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 30-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 30-11

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`. **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`.	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.

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.

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`.	Set **Model Configuration Parameters** > **Optimization** > **Implement logic signals as Boolean data** to `on`. The Fixed-Point Advisor provides separate results for each referenced model. Use **Modify All** or manually configure this option for each referenced model.

Action Results

Clicking **Modify All** selects the model **Model Configuration Parameters** > **Optimization** > **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.

Check bus usage

Identify any Mux block used as a bus creator and any bus signal treated as a vector.

Description

This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Results and Recommended Actions

Conditions	Recommended Action
The Fixed-Point Advisor is not operating on a top-level model.	If this task is important to your conversion, start the Fixed-Point Advisor on the top-level model. Otherwise, you can ignore this warning.
The model is not configured to detect future changes that might result in improper bus usage.	Set **Model Configuration Parameters** > **Diagnostics** > **Connectivity** > **Buses** > **Bus signal treated as vector** to `error`. The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, manually configure this option for each referenced model.

Note This task is a Simulink task. For more information, see “Check bus usage” in the Simulink documentation.

Simulation range checking

Verify that **Model Configuration Parameters** > **Diagnostics** > **Simulation range checking** is not set to `none`.

Description

If **Model Configuration Parameters** > **Diagnostics** > **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** > **Simulation range checking** is set to `none`.	Set **Model Configuration Parameters** > **Diagnostics** > **Simulation range checking** to `warning`. 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.

Action Results

Clicking **Modify All** sets **Model Configuration Parameters** > **Diagnostics** > **Simulation range checking** to `warning`.

If your model contains referenced models, modifies this setting on all referenced models.

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.	• 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**. 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.

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.

See Also

“Resolve Signal Objects for Output Data” in the Simulink documentation

Preparing for Data Typing and Scaling

In this section...

"Prepare for Data Typing and Scaling Overview" on page 37-24

"Review locked data type settings" on page 37-25

"Remove output data type inheritance" on page 37-26

"Relax input data type settings" on page 37-29

"Verify Stateflow charts have strong data typing with Simulink" on page 37-31

"Remove redundant specification between signal objects and blocks" on page 37-32

"Verify hardware selection" on page 37-34

"Specify block minimum and maximum values" on page 37-36

Prepare for Data Typing and Scaling Overview

Configure blocks with data type inheritance or 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 30-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 30-11

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.

Remove output data type inheritance

Identify blocks with an inherited output signal data type.

Description

Inherited data types might lead to data type propagation errors.

For floating-point inheritance blocks with floating-point inputs or outputs, the Fixed-Point Advisor replaces the inheritance with the fixed-point data type specified by the user. For floating-point inheritance blocks with fixed-point output and other Simulink and DSP System Toolbox and Communications System Toolbox blocks, the Fixed-Point Advisor now detects inheritance and replaces it with the compiled data type.

What are Floating-Point Inheritance Blocks?

For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point.

Note This task is preparation for automatic data typing, not actual automatic data typing.

Input Parameters

Data type for blocks with floating-point inheritance

Enter a default fixed-point data type to use for floating-point inheritance blocks, or select one from the list:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Results and Recommended Actions

Conditions	Recommended Action
An input parameter is invalid.	Enter or select a valid value for the **Data type for blocks with floating-point inheritance** input parameter. The value of this parameter applies to the entire model reference hierarchy.
The system or subsystems contain floating-point inheritance blocks that have floating-point inputs.	Set the block output data type to the recommended data type. Remove floating-point inheritance for these blocks by explicitly configuring the **Output data type** or **Output data type mode** parameter to the recommended value where possible. The recommended value is based on the value that you specify for the **Data type for blocks with floating-point inheritance** input parameter. The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, remove floating-point inheritance for all blocks in all referenced models.
Blocks or Stateflow output data in the current system or subsystems have inherited output data types.	Remove output data type inheritance for blocks by explicitly configuring the **Output data type** or **Output data type mode** parameter to the recommended value where possible. Remove output data type inheritance for Logical Operator blocks by clearing the **Require all inputs and outputs to have the same data type parameter** parameter. Remove Stateflow output data type inheritance by explicitly configuring the output data **Type** property. The Fixed-Point Advisor provides separate results for each referenced model. For this task to pass, remove output data type inheritance in all referenced models.

Action Results

Clicking **Modify All** explicitly configures the output data types to the recommended values where possible. Tables list the previous and current data types for the reconfigured blocks.

If your model contains referenced models, where possible, explicitly configures the output data types to the recommended values for all referenced models.

Relax input data type settings

Identify blocks with input data type constraints.

Description

Blocks that have input data type constraints might lead to data type propagation errors.

Note This task is preparation for automatic data typing, not actual automatic data typing.

Results and Recommended Actions

Conditions	Recommended Action
The input data types of blocks or Stateflow charts in the current system or subsystems have constraints.	Explicitly configure flexible input data types for blocks by setting the `InputSameDT` parameter to `off` where possible. Explicitly configure Logical Operator blocks to have flexible input data types by setting the `AllPortsSameDT` parameter to `off`. Explicitly configure flexible Stateflow chart input data types by setting the `Type` method to `Inherited`. Select the **Use Strong Data Typing with Simulink I/O** chart property. 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.

Action Results

Clicking **Modify All** explicitly configures the specified settings to the recommended value where possible. A table lists the previous and current settings for the reconfigured blocks.

If your model contains referenced models, where possible, explicitly configures the output data types to the recommended values for all referenced models.

Tip

Removing unnecessary data setting restrictions makes it more likely that the **Propose data types** task will succeed downstream.

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.	Select the **Use Strong Data Typing with Simulink I/O** check box in the chart properties dialog. 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.

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.

Remove redundant specification between signal objects and blocks

Identify and remove redundant data type specification originating from blocks and Simulink signal objects.

Description

This task prepares your model for automatic data typing by identifying and removing redundant data type specification originating from blocks and Simulink signal objects.

Note You must rerun this task whenever you delete or manipulate a Simulink signal object in the base workspace.

Input Parameters

Remove redundant specification from

Select from the list:

`Blocks`
Identify and remove redundant data type specification from blocks.

`Signal objects`
Identify and remove redundant data type specification from Simulink signal objects.

Results and Recommended Actions

Conditions	Recommended Action
Blocks associated with Simulink signal objects do not have their data type specification set to a passive mode.	Set the data type specification of these blocks to a passive mode, such as `Inherit via back propagation`. The Fixed-Point Advisor provides separate results for each referenced model. For this task

Conditions	Recommended Action
	to pass, use **Modify All** or manually configure this setting in each referenced model.
Simulink signal objects associated with blocks do not have their data type specification set to a passive mode.	Set the data type specification of these Simulink signal objects to `Auto`. 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.

Action Results

Clicking **Modify All** explicitly configures the properties of the blocks or Simulink signal objects to the recommended value where possible. A table displays the current and previous settings.

If your model contains referenced models, where possible, configures these settings for all referenced models.

Verify hardware selection

Verify target hardware setting.

Description

Review the hardware device settings and verify they are the settings you intend to use.

Input Parameters

Default type of all floating-point signals

Enter a default fixed-point data type to use for all floating-point signals, or select one from the list. For FPGA/ASIC targets, specify the type explicitly.

`Remain floating-point`
Use this setting if you are converting only part of the model to fixed point and want to leave the rest of the model as floating point.

`Same as embedded hardware integer`
Use this setting if the hardware device specified is a microprocessor.

`int8`

`int16`

`int32`

`fixdt(1,16,4)`

Results and Recommended Actions

Conditions	Recommended Action
The model's **Model Configuration Parameters** > **Hardware Implementation** device parameters are not specified.	Provide values for **Model Configuration Parameters** > **Hardware Implementation** > **Device vendor** and **Device type** parameters. The Fixed-Point Advisor provides results for each referenced model. For this task to pass, manually configure this setting in each referenced model. All referenced models must use the same hardware settings.
Default data type of all floating-point signals is set to `Remain floating-point`.	For microprocessors, set to `Same as embedded hardware integer`. For FPGA/ASIC, set the data type explicitly. The Fixed-Point Advisor uses the sign and word length of this data type.

See Also

- "Device type"
- "Device vendor"

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" 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 30-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 30-11

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/fxpdemos/`

- "Data Type Support" on page A-2
- "Structure of the S-Function" on page A-5
- "Storage Containers" on page A-7
- "Data Type IDs" on page A-13
- "Overflow Handling and Rounding Methods" on page A-20
- "Create MEX-Files" on page A-23
- "Fixed-Point S-Function Examples" on page A-24
- "API Function Reference" on page A-33

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 27-18.

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

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

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:

$$\textit{real-world value} = \textit{integer} \times 2^{-\textit{fraction length}} = \textit{integer} \times 2^{0}.$$

In either case, trivial scaling means that the real-world value is equal to the stored integer value:

$$\textit{real-world value} = \textit{integer}.$$

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`.

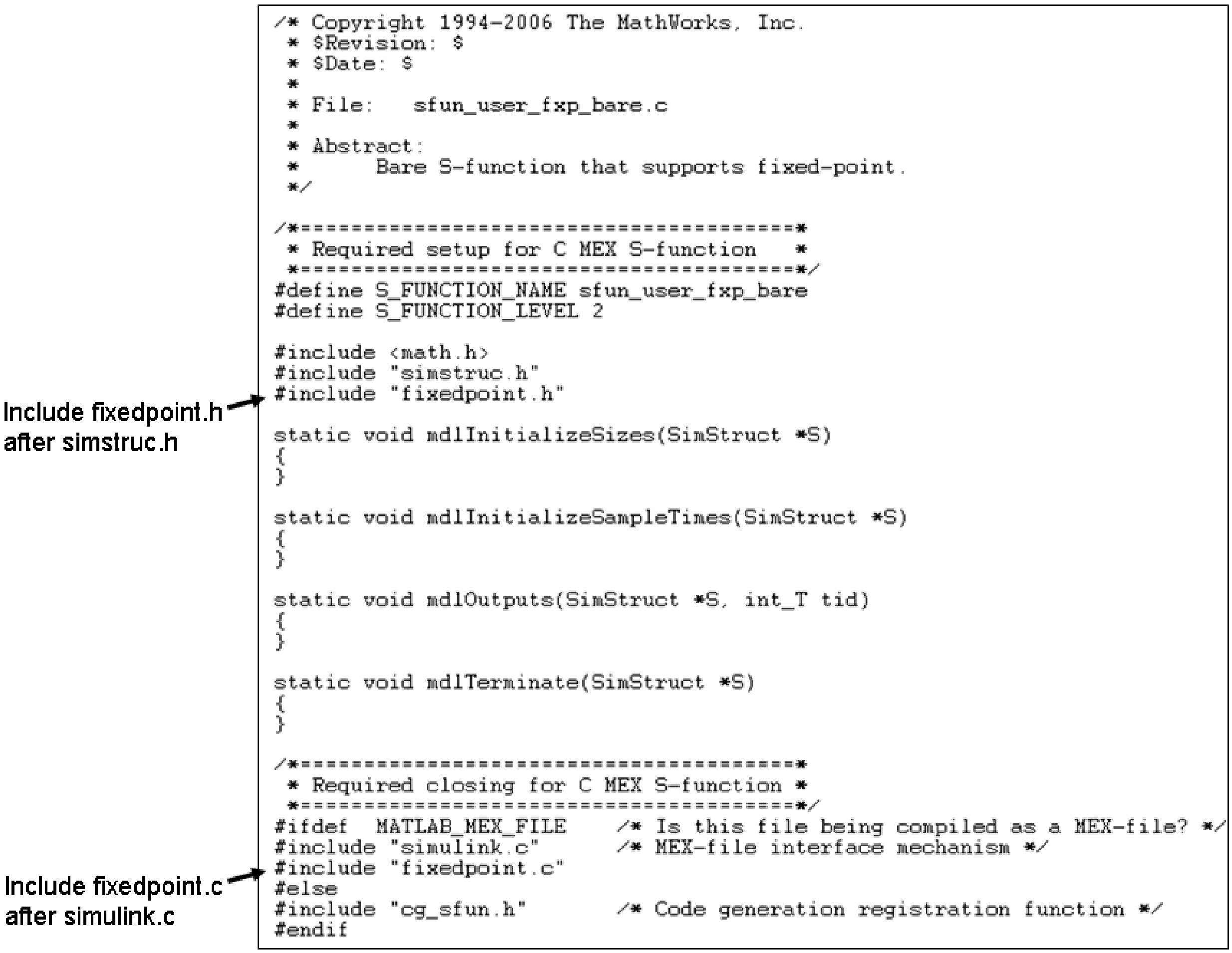
- A signal that has been overridden with `Scaled double` is treated as being of data type `ScaledDouble`.

`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 31-2 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.



```
/* Copyright 1994-2006 The MathWorks, Inc.
 * $Revision: $
 * $Date: $
 *
 * File:   sfun_user_fxp_bare.c
 *
 * Abstract:
 *      Bare S-function that supports fixed-point.
 */

/*=======================================*
 * Required setup for C MEX S-function   *
 *=======================================*/
#define S_FUNCTION_NAME sfun_user_fxp_bare
#define S_FUNCTION_LEVEL 2

#include <math.h>
#include "simstruc.h"
#include "fixedpoint.h"

static void mdlInitializeSizes(SimStruct *S)
{
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
}

static void mdlTerminate(SimStruct *S)
{
}

/*=======================================*
 * Required closing for C MEX S-function *
 *=======================================*/
#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#include "fixedpoint.c"
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Include fixedpoint.h after simstruc.h

Include fixedpoint.c after simulink.c

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-23.

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

Fixed-Point Storage Containers (Continued)

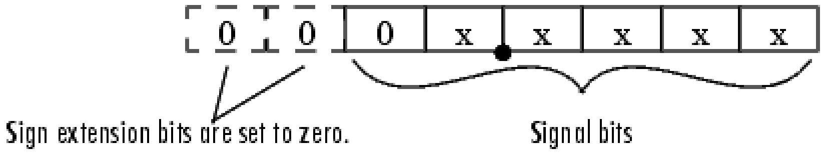
Container Category	Signal Word Length	Container Word Length	Container Size
FXP_STORAGE_OTHER_SINGLE_WORD	33 to word length of long data type	Length of long data type	Length of long data type
FXP_STORAGE_MULTIWORD	Greater than the word length of long data type to 128 bits	Multiples of length of long data 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:

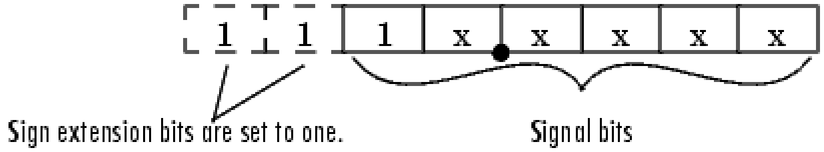
- If the data type is unsigned, the sign extension bits must be cleared to zero.
- If the data type is signed, the sign extension bits must be set to one for strictly negative numbers, and cleared to zero otherwise.

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

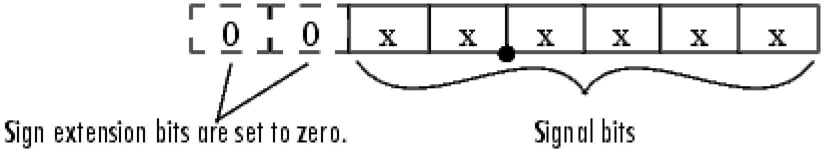


8-bit container for a signed, 6-bit signal that is negative



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 `ssGetDataTypeFxpWordLength` and `ssGetDataTypeFxpContainWordLen` functions, respectively. The storage container size is returned by the `ssGetDataTypeStorageContainerSize` function. The container category is returned by the `ssGetDataTypeStorageContainCat` function, which in addition to those in the table above, 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	The container type for a Simulink single
FXP_STORAGE_DOUBLE	The container type for a Simulink double
FXP_STORAGE_SCALEDDOUBLE	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,

- ssGetDataTypeStorageContainCat returns FXP_STORAGE_INT32.
- ssGetDataTypeStorageContainerSize or sizeof( ) returns 4, which is the storage container size in bytes.
- ssGetDataTypeFxpContainWordLen returns 32, which is the storage container word length in bits.
- ssGetDataTypeFxpWordLength 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 typdef'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,

$$\text{sizeof(char)} \le \text{sizeof(short)} \le \text{sizeof(int)} \le \text{sizeof(long)}.$$

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-13

"Registering Data Types" on page A-14

"Setting and Getting Data Types" on page A-16

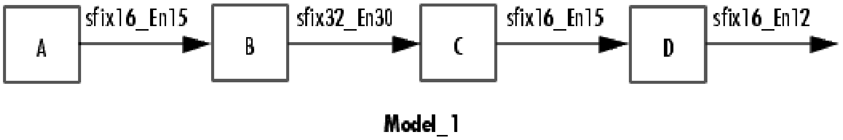
"Getting Information About Data Types" on page A-17

"Converting Data Types" on page A-19

The Assignment of Data Type IDs

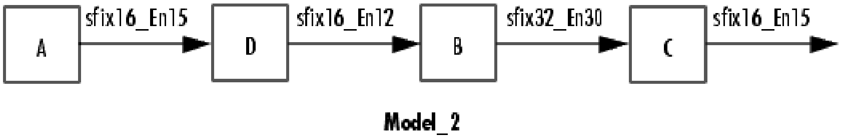
Each data type used in your S-function is assigned a data type ID. You should always use data type IDs to get and set information about data types in your S-function.

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
`ssRegisterDataTypeFxpBinaryPoint`	Register a fixed-point data type with binary-point-only scaling and return its data type ID	`sfun_user_fxp_asr.c` Line 252
`ssRegisterDataTypeFxpFSlopeFixExpBias`	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
`ssRegisterDataTypeFxpScaledDouble`	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
`ssRegisterDataTypeFxpSlopeBias`	Register a data type with [Slope Bias] scaling and return its data type ID	`sfun_user_fxp_dtprop.c` 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 on page A-15. 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`
- `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

`ssGetDataTypeFxpContainWordLen`	Return the word length of the storage container of a registered data type	`sfun_user_fxp_ContainWordLenProbe.c`
		`Line 181`

`ssGetDataTypeStorageContainCat`	Return the storage container category of a registered data type	`sfun_user_fxp_asr.c` Line 294
`ssGetDataTypeStorageContainerSize`	Return the storage container size of a registered data type	`sfun_user_fxp_StorageContainSizeProbe.c` Line 171

Signal Data Type Information Functions

Function	Description	Example of Use
ssGetDataTypeFxpIsSigned	Determine whether a fixed-point registered data type is signed or unsigned	sfun_user_fxp_asr.c Line 254
ssGetDataTypeFxpWordLength	Return the word length of a fixed-point registered data type	sfun_user_fxp_asr.c Line 255
ssGetDataTypeIsFixedPoint	Determine whether a registered data type is a fixed-point data type	sfun_user_fxp_const.c Line 127
ssGetDataTypeIsFloatingPoint	Determine whether a registered data type is a floating-point data type	sfun_user_fxp_ IsFloatingPointProbe.c Line 176
ssGetDataTypeIsFxpFltApiCompat	Determine whether a registered data type is supported by the API for user-written fixed-point S-functions	sfun_user_fxp_asr.c Line 184
ssGetDataTypeIsScalingPow2	Determine whether a registered data type has power-of-two scaling	sfun_user_fxp_asr.c Line 203
ssGetDataTypeIsScalingTrivial	Determine whether the scaling of a registered data type is slope = 1, bias = 0	sfun_user_fxp_ IsScalingTrivialProbe.c Line 171

Signal Scaling Information Functions

Function	Description	Example of Use
ssGetDataTypeBias	Return the bias of a registered data type	sfun_user_fxp_dtprop.c Line 243
ssGetDataTypeFixedExponent	Return the exponent of the slope of a registered data type	sfun_user_fxp_dtprop.c Line 237

Signal Scaling Information Functions (Continued)

Function	Description	Example of Use
`ssGetDataTypeFracSlope`	Return the fractional slope of a registered data type	`sfun_user_fxp_dtprop.c` Line 234
`ssGetDataTypeFractionLength`	Return the fraction length of a registered data type with power-of-two scaling	`sfun_user_fxp_asr.c` Line 256
`ssGetDataTypeTotalSlope`	Return the total slope of the scaling of a registered data type	`sfun_user_fxp_dtprop.c` 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
`ssFxpConvert`	Convert a value from one data type to another data type.	Not Available
`ssFxpConvertFromRealWorldValue`	Convert a value of data type `double` to another data type.	Not Available
`ssFxpConvertToRealWorldValue`	Convert a value of any data type to a `double`.	Not Available

Overflow Handling and Rounding Methods

In this section...
"Tokens for Overflow Handling and Rounding Methods" on page A-20
"Overflow Logging Structure" on page A-21

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	Example of Use
`FXP_OVERFLOW_SATURATE`	Saturate overflows	Not Available
`FXP_OVERFLOW_WRAP`	Wrap overflows	Not Available

Rounding Method Tokens

Token	Description	Example of Use
`FXP_ROUND_CEIL`	Round to the closest representable number in the direction of positive infinity	Not Available
`FXP_ROUND_CONVERGENT`	Round toward nearest integer with ties rounding to nearest even integer	Not Available
`FXP_ROUND_FLOOR`	Round to the closest representable number in the direction of negative infinity	Not Available

Rounding Method Tokens (Continued)

Token	Description	Example of Use
`FXP_ROUND_NEAR`	Round to the closest representable number, with the exact midpoint rounded in the direction of positive infinity	Not Available
`FXP_ROUND_NEAR_ML`	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers	Not Available
`FXP_ROUND_SIMPLEST`	Automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible	Not Available
`FXP_ROUND_ZERO`	Round to the closest representable number in the direction of zero	Not Available

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

In this section...

"List of Fixed-Point S-Function Examples" on page A-24

"Get the Input Port Data Type" on page A-25

"Set the Output Port Data Type" on page A-27

"Interpret an Input Value" on page A-28

"Write an Output Value" on page A-30

"Determine Output Type Using the Input Type" on page A-31

List of Fixed-Point S-Function Examples

The following files in *`matlabroot`*`/toolbox/simulink/fixedandfloat/fxpdemos/` 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

The sections that follow present smaller portions of code that focus on specific kinds of tasks you might want to perform within your S-function.

Get the Input Port Data Type

Within your S-function, you might need to know the data types of different ports, run-time 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 = ssGetDataTypeStorageContainCat( S,
     dataTypeIdU0 );
storageContainerU1 = ssGetDataTypeStorageContainCat( 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 = ssGetDataTypeFxpIsSigned( S, dataTypeIdU0 );
    isSignedU1 = ssGetDataTypeFxpIsSigned( S, dataTypeIdU1 );

    wordLengthU0 = ssGetDataTypeFxpWordLength( S, dataTypeIdU0 );
    wordLengthU1 = ssGetDataTypeFxpWordLength( S, dataTypeIdU1 );

    fracSlopeU0 = ssGetDataTypeFracSlope( S, dataTypeIdU0 );
    fracSlopeU1 = ssGetDataTypeFracSlope( S, dataTypeIdU1 );

    fixedExponentU0 = ssGetDataTypeFixedExponent( S,dataTypeIdU0 );
    fixedExponentU1 = ssGetDataTypeFixedExponent( S,dataTypeIdU1 );
    totalSlopeU0 = ssGetDataTypeTotalSlope( S, dataTypeIdU0 );
    totalSlopeU1 = ssGetDataTypeTotalSlope( S, dataTypeIdU1 );

    biasU0 = ssGetDataTypeBias( S, dataTypeIdU0 );
    biasU1 = ssGetDataTypeBias( 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.

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 on page A-15. 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 )
    {
        DTypeId 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 );
    }
```

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 =
     ssGetDataTypeStorageContainCat( 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;
```

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>
```

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

Purpose

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`

ssFxpConvertFromRealWorldValue

Purpose

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.

ssFxpConvertFromRealWorldValue

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`

ssFxpConvertToRealWorldValue

Purpose

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`

ssFxpGetU32BitRegion

Purpose

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`

Purpose

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`

Purpose

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`

Purpose

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`

Purpose

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_DataTypeBias`

See Also

`ssGetDataTypeFixedExponent`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

ssGetDataTypeFixedExponent

Purpose

Return exponent of slope of registered data type

Syntax

```
extern int ssGetDataTypeFixedExponent (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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

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 `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_DataTypeFixedExponent`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

ssGetDataTypeFracSlope

Purpose

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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

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`

ssGetDataTypeFractionLength

Purpose

Return fraction length of registered data type with power-of-two scaling

Syntax

```
extern int ssGetDataTypeFractionLength (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 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 `ssGetDataTypeIsScalingPow2` returns `FALSE`.

This function also 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_DataTypeFractionLength`

See Also

`ssGetDataTypeFxpWordLength`

Purpose

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`

Purpose

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_DataTypeFxpIsSigned`

ssGetDataTypeFxpWordLength

Purpose

Return word length of fixed-point registered data type

Syntax

```
extern int ssGetDataTypeFxpWordLength (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 `ssGetDataTypeFxpContainWordLen`:

- 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 `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

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- ssGetDataTypeFxpWordLength returns 24, which is the data type word length in bits.
- ssGetDataTypeFxpContainWordLen returns 32, which is the storage container word length in bits.
- ssGetDataTypeStorageContainerSize or sizeof( ) returns 4, which is the storage container size in bytes.

TLC Functions

FixPt_DataTypeFxpWordLength

See Also

ssGetDataTypeFxpContainWordLen, ssGetDataTypeFractionLength, ssGetDataTypeStorageContainerSize

ssGetDataTypeIsFixedPoint

Purpose

Determine whether registered data type is fixed-point data type

Syntax

```
extern int ssGetDataTypeIsFixedPoint(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`

Purpose

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`

Purpose

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.

ssGetDataTypeIsScalingPow2

Purpose

Determine whether registered data type has power-of-two scaling

Syntax

```
extern int ssGetDataTypeIsScalingPow2 (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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

When *bias* = 0 and *fractional slope* = 1, the only scaling factor that remains is a power of two:

$$\textit{real-world value} = (2^{\textit{exponent}} \times \textit{integer}) = (2^{\textit{-fraction length}} \times \textit{integer}).$$

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 `ssGetDataTypeIsScalingPow2` 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`

Purpose

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

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

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:

$$\textit{real-world value} = \textit{integer} \times 2^{-\textit{fraction length}} = \textit{integer} \times 2^{0}.$$

In either case, trivial scaling means that the real-world value is simply equal to the stored integer value:

$$\textit{real-world value} = \textit{integer}.$$

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`.

ssGetDataTypeIsScalingTrivial

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`

Purpose

Return number of chunks in multiword storage container of registered data type

Syntax

```
extern int ssGetDataTypeNumberOfChunks(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 `ssGetDataTypeStorageContainCat` 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

`ssGetDataTypeStorageContainCat`

ssGetDataTypeStorageContainCat

Purpose

Return storage container category of registered data type

Syntax

```
extern fxpStorageContainerCategory
ssGetDataTypeStorageContainCat(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

Fixed-Point Storage Containers (Continued)

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 ssGetDataTypeStorageContainCat 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 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

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-11 for more information:

- FixPt_DataTypeNativeType
- FixPt_DataTypeStorageDouble
- FixPt_DataTypeStorageSingle
- FixPt_DataTypeStorageScaledDouble
- FixPt_DataTypeStorageSInt
- FixPt_DataTypeStorageUInt
- FixPt_DataTypeStorageSLong

- `FixPt_DataTypeStorageULong`
- `FixPt_DataTypeStorageSShort`
- `FixPt_DataTypeStorageUShort`

See Also

`ssGetDataTypeStorageContainerSize`

ssGetDataTypeStorageContainerSize

Purpose

Return storage container size of registered data type

Syntax

```
extern size_t ssGetDataTypeStorageContainerSize
                                                (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 `ssGetDataTypeFxpContainWordLen`. To get the word length of the data type, use `ssGetDataTypeFxpWordLength`.

The container of the size returned by this function stores input and output signals, run-time parameters, and DWorks during Simulink simulations. It is also the appropriate size measurement to pass to functions like `memcpy( )`.

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

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeStorageContainerSize` or `sizeof( )` returns `4`, which is the storage container size in bytes.
- `ssGetDataTypeFxpContainWordLen` returns `32`, which is the storage container word length in bits.
- `ssGetDataTypeFxpWordLength` returns `24`, which is the data type word length in bits.

TLC Functions

`FixPt_GetDataTypeStorageContainerSize`

See Also

`ssGetDataTypeFxpContainWordLen`, `ssGetDataTypeFxpWordLength`, `ssGetDataTypeStorageContainCat`

ssGetDataTypeTotalSlope

Purpose

Return total slope of scaling of registered data type

Syntax

```
extern double ssGetDataTypeTotalSlope (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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

This function returns the total slope, rather than the fractional slope, of the data type designated by `dataTypeId`. To get the fractional slope, use `ssGetDataTypeFracSlope`:

- 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 `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

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`

ssLogFixptInstrumentation

Purpose

Record information collected during simulation

Syntax

```
extern void ssLogFixptInstrumentation
                              (SimStruct *S,
                               double minValue,
                               double maxValue,
                               int countOverflows,
                               int countSaturations,
                               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.

`countSaturations`
: 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

ssRegisterDataTypeFxpBinaryPoint

Purpose

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-13.

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`

ssRegisterDataTypeFxpFSlopeFixExpBias

Purpose

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-13.

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`

ssRegisterDataTypeFxpScaledDouble

Purpose

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-13.

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`

Purpose

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-13.

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`

Index

A

accumulations
  scaling recommendations 28-37
  slope/bias encoding 28-37
accumulator data types 29-4
  feedback controller example 34-46
addition
  fixed-point block rules 28-50
  scaling recommendations 28-34
  slope/bias encoding 28-34
Additional Math and Discrete Library
  support for blocks in 35-43
ALUs 28-49
ANSI C
  compared with `fi` objects 1-22
API
  fixed-point A-1
API function reference A-33
  `ssFxpConvert` A-34
  `ssFxpConvertFromRealWorldValue` A-36
  `ssFxpConvertToRealWorldValue` A-38
  `ssFxpGetU32BitRegion` A-39
  `ssFxpGetU32BitRegionCompliant` A-41
  `ssFxpSetU32BitRegion` A-43
  `ssFxpSetU32BitRegionCompliant` A-45
  `ssGetDataTypeBias` A-47
  `ssGetDataTypeFixedExponent` A-48
  `ssGetDataTypeFracSlope` A-50
  `ssGetDataTypeFractionLength` A-52
  `ssGetDataTypeFxpContainWordLen` A-53
  `ssGetDataTypeFxpIsSigned` A-55
  `ssGetDataTypeFxpWordLength` A-56
  `ssGetDataTypeIsFixedPoint` A-58
  `ssGetDataTypeIsFloatingPoint` A-59
  `ssGetDataTypeIsFxpFltApiCompat` A-60
  `ssGetDataTypeIsScalingPow2` A-61
  `ssGetDataTypeIsScalingTrivial` A-63
  `ssGetDataTypeNumberOfChunks` A-65
  `ssGetDataTypeStorageContainCat` A-66
  `ssGetDataTypeStorageContainerSize` A-70
  `ssGetDataTypeTotalSlope` A-72
  `ssLogFixptInstrumentation` A-74
  `ssRegisterDataTypeFxpBinaryPoint` A-76
  `ssRegisterDataTypeFxpFSlopeFixExpBias` A-79
  `ssRegisterDataTypeFxpScaledDouble` A-82
  `ssRegisterDataTypeFxpSlopeBias` A-85
arguments
  limit on number for code generation from
    MATLAB 11-19
arithmetic
  fixed-point 4-11
  with [Slope Bias] signals 4-17
arithmetic logic units (ALUs) 28-49
arithmetic operations
  fixed-point 1-10
arithmetic shifts 28-65
automatic data typing
  feedback controller example 34-51
  using simulation data 34-11
automatic data typing using derived data
  workflow 34-25
automatic data typing using simulation data
  workflow 34-11

B

base data type 29-4
  feedback controller example 34-46
binary conversions 1-25
binary point 27-4
binary-point-only scaling 27-7
bits 27-3
  hidden 27-29
  multipliers 27-9
  shifts 28-64
block configurations
  selecting a data type 26-22
Bode plots 34-42

C

C/C++ code generation for supported functions 21-1
casts
  fixed-point 1-19
ceil function 28-7
ceiling
  rounding 28-7
chopping 28-18
chunk arrays A-7
chunks A-7
clone method 25-8
code generation 36-2
  signal conversions 28-48
  summation 28-52
Code generation
  fixed-point 9-1
code generation from MATLAB
  benefits of 16-2
  best practices
    generate code generation report 9-47
    preserving your code 9-50
    separating test bench from function code 9-50
    specifying input properties 9-47
    using build scripts 9-48
    using file naming convention 9-50
    using the MATLAB code analyzer 9-49
  best practices for working with variables 15-3
  calling local functions 11-9
  calling MATLAB functions 11-11
  calling MATLAB functions using feval 11-16
  characters 13-7
  communications system toolbox System objects 24-7
  compilation directive %#codegen 11-8
  compiler options for MEX code generation 9-30
  computer vision system toolbox System objects 24-2
  controlling run-time checks 9-71
  converting mxArrays to known types 11-18
  declaring MATLAB functions as extrinsic functions 11-12
  defining persistent variables 15-10
  defining variables 15-2
  defining variables by assignment 15-3
  dsp system toolbox System objects 24-13
  eliminating redundant copies of function inputs 19-7
  eliminating redundant copies of uninitialized variables 15-7
  how it resolves function calls 11-2
  how to disable run-time checks 9-72
  image acquisition toolbox System objects 24-23
  initializing persistent variables 15-10
  inlining functions 19-9
  limit on number of function arguments 11-19
  phased array system toolbox System objects 24-19
  pragma 11-8
  recommended options for fiaccel 9-47
  resolving extrinsic function calls during simulation 11-16
  resolving extrinsic function calls in generated code 11-17
  rules for defining uninitialized variables 15-7
  setting properties of indexed variables 15-6
  supported toolbox functions 11-10
  using Code Analyzer 9-26
  using type cast operators in variable definitions 15-6
  variables, complex 13-4
  when not to use 16-2
  when to disable run-time checks 9-72
  when to use 16-2
  which features to use 16-4

working with mxArrays 11-17
code optimization 36-9
`coder.extrinsic` 11-12
coder.nullcopy
uninitialized variables 15-7
Commonly Used Blocks Library
support for blocks in 35-43
communications system toolbox System objects
supported for code generation from MATLAB 24-7
compiler options parameters
for MEX code generation from MATLAB 9-30
compilers
supported for generating MEX functions with fiaccel 9-16
complex multiplication
fixed-point 1-13
computational noise 28-2
rounding 28-4
computational units 28-49
computer vision system toolbox System objects
supported for code generation from MATLAB 24-2
configuring fixed-point blocks 26-21
constant scaling for best precision 27-15
limitations for code generation 36-7
containers
fixed-point API A-7
contiguous bits 27-28
Continuous Library
support for blocks in 35-43
controlling run-time checks
code generation from MATLAB 9-71
convergent
rounding 28-8
`convergent` function 28-9
conversions 28-46
parameter 28-45
signal 28-46
See also online conversion, offline conversion

D

data type IDs A-13
for built-in data types A-15
data type override 5-12
data types 26-22
fractional numbers 26-23
generalized fixed-point numbers 26-23
IEEE numbers 26-24
integers 26-23
parameters 27-11
registering fixed-point A-14
defining uninitialized variables
rules 15-7
defining variables
for C/C++ code generation 15-3
denormalized numbers 27-33
design considerations
when writing MATLAB Code for code generation 16-7
development cycle 26-19
digital controllers 34-43
digital filters 29-2
direct form realization 29-8
feedback controller example 34-45
Discontinuities Library
support for blocks in 35-44
Discrete Library
support for blocks in 35-44
display preferences
setting 5-5
division
fixed-point block rules 28-62
scaling recommendations 28-41
slope/bias encoding 28-41
double bits 28-55
double-precision formats 27-30
dsp system toolbox System objects
supported for code generation from MATLAB 24-13

E

eliminating redundant copies of function
    inputs 19-7
encoding schemes 27-6
eps function 27-32
examples
    casting from doubles to fixed-point 26-40
    conversions and arithmetic operations 28-67
    division process 28-63
    fixed-point format 27-9
    limitations on precision and errors 28-20
    limitations on range 28-31
    maximizing precision 28-21
    multiplication process 28-60
    port data type display 27-25
    saturation and wrapping 28-28
    scaled doubles 27-22
    selecting a measurement scale 26-8
    summation process 28-52
exceptional arithmetic 27-33
exponents
    IEEE numbers 27-29
external mode 36-7
extrinsic functions 11-12

F

feedback designs 34-40
fi objects
    constructing 2-2
fiaccel
    recommended options 9-47
    supported compilers 9-16
filters
    digital 29-2
fimath objects
    properties
        setting in the Model Explorer 4-8
    setting properties in the Model Explorer 4-8
fimath objects 1-16
    constructing 4-2
fipref objects
    constructing 5-2
fix function 28-17
Fixed-Point Advisor
    example
        converting a model from floating-point to
            fixed-point 30-11
    fixing a task failure 30-5
    introduction 30-2
    running 30-4
fixed-point arithmetic 4-11
fixed-point blocks
    configuring 26-21
fixed-point data
    reading from workspace 10-2 26-35
    writing to workspace 10-2 26-35
fixed-point data types
    addition 1-12
    arithmetic operations 1-10
    casts 1-19
    complex multiplication 1-13
    modular arithmetic 1-10
    multiplication 1-13
    overflow handling 1-5
    precision 1-5
    range 1-5
    registering A-14
    rounding 1-6
    saturation 1-5
    scaling 1-4
    subtraction 1-12
    two’s complement 1-11
    wrapping 1-5
fixed-point math 4-11
Fixed-Point MATLAB code 9-1
fixed-point numbers
    general format 27-3
    scaling 27-6
fixed-point run-time API 10-7 26-38

fixed-point signal logging 10-6 26-38
Fixed-Point Tool
 applying proposed data types 34-22 34-37
 automatic data typing Simulink signal objects 34-23
 examining results 34-17 34-33
 feedback controller example 34-46
 opening 31-2
 overview 31-2
 proposing data types 34-15 34-30
 tutorial 34-40
floating-point numbers 27-28
floor
 rounding 28-10
`floor` function 28-10
fraction
 IEEE numbers 27-29
fractional numbers 26-23
 guard bits 28-31
fractional slope 27-6
functions
 limit on number of arguments for code generation 11-19
Functions supported for C/C++ code
 generation 21-1
 alphabetical list 21-2
 arithmetic operator functions 21-90
 bit-wise operation functions 21-90
 casting functions 21-91
 Communications System Toolbox functions 21-91
 complex number functions 21-91
 Computer Vision System Toolbox functions 21-92
 data and file management functions 21-94
 data type functions 21-94
 derivative and integral functions 21-95
 discrete math functions 21-95
 error handling functions 21-96
 exponential functions 21-96
 filtering and convolution functions 21-97
 Fixed-Point Designer functions 21-97
 histogram functions 21-106
 Image Processing Toolbox functions 21-106
 input and output functions 21-108
 interpolation and computational geometry functions 21-109
 linear algebra functions 21-109
 logical operator functions 21-109
 MATLAB Compiler functions 21-110
 MATLAB Desktop environment functions 21-110
 matrix/array functions 21-110
 nonlinear numerical methods 21-114
 Phased Array System Toolbox functions 21-115
 polynomial functions 21-118
 programming utilities 21-118
 relational operator functions 21-118
 rounding and remainder functions 21-119
 set functions 21-119
 signal processing functions 21-120
 Signal Processing Toolbox functions 21-120
 special value functions 21-125
 specialized math functions 21-125
 statistical functions 21-126
 Statistics Toolbox functions 21-126
 string functions 21-132
 structure functions 21-133
 trigonometric functions 21-133
Functions supported for MEX and C/C++ code
 generation
 categorized list 21-88

G

gain
 scaling recommendations 28-40
 using slope/bias encoding 28-39
generalized fixed-point numbers 26-23

getDiscreteState method 25-9
getNumInputs method 25-9
getNumOutputs method 25-9
global overrides with doubles 34-49
guard bits 28-31

H

hidden bits 27-29
how to disable run-time checks
    code generation from MATLAB 9-72

I

IEEE floating-point numbers
    formats
        double-precision 27-30
        exponent 27-29
        fraction 27-29
        sign bit 27-28
        single-precision 27-29
    precision 27-32
    range 27-31
image acquisition toolbox System objects
    supported for code generation from
        MATLAB 24-23
indexed variables
    setting properties for code generation from
        MATLAB 15-6
infinity 27-33
info method 25-8
initialization
    persistent variables 15-10
installation 26-3
integers
    data types 26-23
interoperability
    fi objects with DSP System Toolbox 10-8
    fi objects with Filter Design Toolbox 10-12
    fi objects with Simulink 10-2
isChangedProperty method 25-8
isDone method 25-8
isLocked method 25-8

L

least significant bit (LSB) 27-4
limit cycles 28-2
    feedback controller example 34-56
locked vs. unlocked mode 25-10
logging
    overflows and underflows 5-7
logging modes
    setting 5-7
Logic and Bit Operations Library
    support for blocks in 35-45
logical shifts 28-65
Lookup Table Library
    support for blocks in 35-45
LSB (least significant bit) 27-4

M

MACs 28-49
math
    with [Slope Bias] signals 4-17
Math Operations Library
    support for blocks in 35-46
MATLAB
    features not supported for code
        generation 16-14
MATLAB Coder
    best practices
        using the MATLAB code analyzer 9-131
    combining property specifications 23-14
    specifying general properties of primary
        inputs 23-14
MATLAB for code generation
    variable types 15-18
MATLAB Function block

using with Model Explorer and fixed-point models 9-79
MATLAB functions
and generating code for mxArrays 11-17
measurement scales 26-5
MEX-files
creating fixed-point A-23
fixed-point A-23
Model Advisor
code optimization 36-34
Model Explorer
setting `embedded.fimath` properties 4-8
setting `embedded.numerictype` properties 6-9
using with fixed-point code generation for MATLAB 9-79
Model Verification Library
support for blocks in 35-48
Model-Wide Utilities Library
support for blocks in 35-48
modeling the system 26-19
modular arithmetic 1-10
most significant bit (MSB) 27-4
MSB (most significant bit) 27-4
multiplication
fixed-point 1-13
fixed-point block rules 28-55
scaling recommendations 28-38
slope/bias encoding 28-37
multiply and accumulate units 28-49
mxArrays
converting to known types 11-18
for code generation from MATLAB 11-17

N

NaNs 27-33
nearest
rounding 28-11
`nearest` function 28-11
`numerictype` objects
properties
setting in the Model Explorer 6-9
setting properties in the Model Explorer 6-9
`numerictype` objects
constructing 6-2

O

offline conversions
addition and subtraction 28-51
multiplication with zero bias and matching fractional slopes 28-59
multiplication with zero bias and mismatched fractional slopes 28-58
parameter conversions 28-46
signals 28-47
one’s complement 1-11
online conversions
addition and subtraction 28-51
multiplication with zero bias and mismatched fractional slopes 28-58
multiplication with zero biases and matching fractional slopes 28-59
signals 28-47
optimization
code 36-9
using Model Advisor 36-34
optimizing generated code
unrolling for-loops 19-41
overflow handling 1-5
compared with ANSI C 1-28
overflows
code generation 36-3
definition 28-2
logging 5-7
overrides with doubles
global override 34-49

P

padding 1-19
padding with trailing zeros
  definition 28-19
  feedback controller example 34-45
parallel form realization 29-15
parameter conversions 28-45
  See also conversions 28-45
persistent variables
  defining for code generation from MATLAB 15-10
  initializing for code generation from MATLAB 15-10
phased array system toolbox System objects
  supported for code generation from MATLAB 24-19
Ports & Subsystems Library
  support for blocks in 35-48
precision
  fixed-point data types 1-5
  fixed-point numbers 27-11
  fixed-point parameters 28-46
  IEEE floating-point numbers 27-32
property values 25-4
  `quantizer` objects 7-3

Q

quantization 28-2
  effects of fixed-point arithmetic 26-43
  feedback controller example 34-49
  real-world value 27-8
  rounding 28-4
`quantizer` objects
  constructing 7-2
  property values 7-3

R

radix point 27-4
range
  fixed-point data types 1-5
  fixed-point numbers 27-10
  IEEE floating-point numbers 27-31
rapid simulation (rsim) target 36-8
reading fixed-point data from workspace 10-2 26-35
real-world values 27-6
realizations
  design constraints 29-7
  direct form 29-8
  parallel form 29-15
  series cascade form 29-13
registering fixed-point data types A-14
release method 25-8
reset method 25-8
round
  rounding 28-12
`round` function 28-13
rounding
  fixed-point data types 1-6
rounding modes 28-4
  code generation 36-3
  convergent 28-8
  round 28-12
  simplest 28-14
  toward ceiling 28-7
  toward floor 28-10
  toward nearest 28-11
  toward zero 28-17
rsim target 36-8
run-time API
  fixed-point data 10-7 26-38

S

S-functions
  examples
    fixed-point A-24
  fixed-point A-1

fixed-point examples A-24
structure for fixed-point A-5
writing fixed-point A-1
saturation 1-5 28-28
scaling 1-4
accumulation 28-37
addition 28-34
binary-point-only 27-7
code generation 36-4
constant scaling for best precision 27-15
division 28-41
gain 28-39
multiplication 28-37
slope/bias 27-8
trivial A-3
scientific notation 27-28
series cascade form realizations 29-13
sharing fixed-point models 26-3
shifts 28-64
sign
extension 28-31
sign bit for IEEE numbers 27-28
Signal Attributes Library
support for blocks in 35-50
signal conversions 28-46
signal logging
fixed-point 10-6 26-38
signal processing functions
for C/C++ code generation 21-120
Signal Routing Library
support for blocks in 35-50
simplest
rounding 28-14
Simulink acceleration modes 36-5
Simulink Coder
external mode 36-7
rapid simulation (rsim) target 36-8
Simulink signal objects
automatic data typing using Fixed-Point Tool 34-23
single-precision format 27-29
Sinks Library
support for blocks in 35-51
[Slope Bias] arithmetic 4-17
slope/bias scaling 27-8
Sources Library
support for blocks in 35-52
`ssFxpConvert` A-34
`ssFxpConvertFromRealWorldValue` A-36
`ssFxpConvertToRealWorldValue` A-38
`ssFxpGetU32BitRegion` A-39
`ssFxpGetU32BitRegionCompliant` A-41
`ssFxpSetU32BitRegion` A-43
`ssFxpSetU32BitRegionCompliant` A-45
`ssGetDataTypeBias` A-47
`ssGetDataTypeFixedExponent` A-48
`ssGetDataTypeFracSlope` A-50
`ssGetDataTypeFractionLength` A-52
`ssGetDataTypeFxpContainWordLen` A-53
`ssGetDataTypeFxpIsSigned` A-55
`ssGetDataTypeFxpWordLength` A-56
`ssGetDataTypeIsFixedPoint` A-58
`ssGetDataTypeIsFloatingPoint` A-59
`ssGetDataTypeIsFxpFltApiCompat` A-60
`ssGetDataTypeIsScalingPow2` A-61
`ssGetDataTypeIsScalingTrivial` A-63
`ssGetDataTypeNumberOfChunks` A-65
`ssGetDataTypeStorageContainCat` A-66
`ssGetDataTypeStorageContainerSize` A-70
`ssGetDataTypeTotalSlope` A-72
`ssLogFixptInstrumentation` A-74
`ssRegisterDataTypeFxpBinaryPoint` A-76
`ssRegisterDataTypeFxpFSlopeFixExpBias` A-79
`ssRegisterDataTypeFxpScaledDouble` A-82
`ssRegisterDataTypeFxpSlopeBias` A-85
step method 25-8
storage containers
fixed-point API A-7
stored integers 26-30

streaming data
- using System objects 25-9

subtraction
- *See* addition 28-36

System object
- clone method 25-8
- creating 25-4
- description 25-2
- getDiscreteState method 25-9
- getNumInputs method 25-9
- getNumOutputs method 25-9
- info method 25-8
- isChangedProperty method 25-8
- isDone method 25-8
- isLocked 25-8
- locked vs. unlocked mode 25-10
- methods 25-7
- properties 25-4
- property values 25-4
- release method 25-8
- reset method 25-8
- step method 25-8
- tunable property 25-11
- using with MATLAB code generation 25-13
- value-only input 25-5

T

targeting an embedded processor
- design rules 29-5
- operation assumptions 29-4
- size assumptions 29-4

trivial scaling A-3

truncation 28-18

tunable 25-11

two's complement 1-11 27-4

type cast operators
- using in variable definitions 15-6

U

unary conversions 1-24

underflow 27-31

underflows
- logging 5-7

uninitialized variables
- eliminating redundant copies in generated code 15-7

unsupported features
- Simulink 35-41

User-Defined Functions Library
- support for blocks in 35-53

V

value-only input 25-5

variable types supported for code generation
- from MATLAB 15-18

variables
- eliminating redundant copies in C/C++ code generated from MATLAB 15-7

Variables
- defining by assignment for code generation from MATLAB 15-3
- defining for code generation from MATLAB 15-2

W

when to disable run-time checks
- code generation from MATLAB 9-72

wrapping 28-28
- fixed-point data types 1-5

writing fixed-point data to workspace 10-2 26-35

Z

zero
- rounding 28-17