

# Fixed-Point Designer™

Reference



# MATLAB®

R2019b



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Fixed-Point Designer™ Reference*

© COPYRIGHT 2013–2019 by The MathWorks, Inc.

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

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

## **Trademarks**

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

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (R2013b)
March 2014	Online only	Revised for Version 4.2 (R2014a)
October 2014	Online Only	Revised for Version 4.3 (R2014b)
March 2015	Online Only	Revised for Version 5.0 (R2015a)
September 2015	Online Only	Revised for Version 5.1 (R2015b)
October 2015	Online only	Rereleased for Version 5.0.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 5.2 (R2016a)
September 2016	Online only	Revised for Version 5.3 (R2016b)
March 2017	Online only	Revised for Version 5.4 (R2017a)
September 2017	Online only	Revised for Version 6.0 (R2017b)
March 2018	Online only	Revised for Version 6.1 (R2018a)
September 2018	Online only	Revised for Version 6.2 (R2018b)
March 2019	Online only	Revised for Version 6.3 (R2019a)
September 2019	Online only	Revised for Version 6.4 (R2019b)



## Apps — Alphabetical List

1

## Blocks — Alphabetical List

2

## Property Reference

3

<b>fi Object Properties</b> .....	<b>3-2</b>
bin .....	3-2
data .....	3-2
dec .....	3-2
double .....	3-2
fimath .....	3-2
hex .....	3-3
int .....	3-3
NumericType .....	3-3
oct .....	3-3
Value .....	3-3
<b>fipref Object Properties</b> .....	<b>3-4</b>
DataTypeOverride .....	3-4
DataTypeOverrideAppliesTo .....	3-4
FimathDisplay .....	3-4
LoggingMode .....	3-5
NumericTypeDisplay .....	3-5
NumberDisplay .....	3-6

<b>quantizer Object Properties</b> .....	<b>3-7</b>
DataMode .....	3-7
Format .....	3-7
OverflowMode .....	3-8
RoundingMode .....	3-9

## Fixed-Point Tool

# 4

<b>Fixed-Point Tool Parameters and Dialog Box</b> .....	<b>4-2</b>
Main Toolbar .....	4-2
Model Hierarchy Pane .....	4-5
Contents Pane .....	4-5
Customizing the Contents Pane View .....	4-8
Dialog Pane .....	4-10
Fixed-Point Advisor .....	4-12
Configure model settings .....	4-13
Run name .....	4-14
Simulate .....	4-15
Merge instrumentation results from multiple simulations . . .	4-15
Derive ranges for selected system .....	4-16
Propose .....	4-16
Propose for .....	4-17
Default fraction length .....	4-17
Default word length .....	4-18
When proposing types use .....	4-18
Safety margin for simulation min/max (%) .....	4-18
 <b>Advanced Settings</b> .....	 <b>4-20</b>
Advanced Settings Overview .....	4-20
Fixed-point instrumentation mode .....	4-20
Data type override .....	4-21
Data type override applies to .....	4-23
Name of shortcut .....	4-25
Allow modification of fixed-point instrumentation settings . . .	4-25
Allow modification of data type override settings .....	4-26
Allow modification of run name .....	4-27
Run name .....	4-27
Capture system settings .....	4-27
Fixed-point instrumentation mode .....	4-27
Data type override .....	4-28

**5** | **Functions – Alphabetical List**

**6** | **Classes – Alphabetical List**

**7** | **Methods – Alphabetical List**

| **Glossary**

**A** | **Selected Bibliography**





# Apps — Alphabetical List

---

## Fixed-Point Converter

Convert MATLAB code to fixed point

### Description

The Fixed-Point Converter app converts floating-point MATLAB® code to fixed-point MATLAB code.

Using the app, you can:

- Propose data types based on simulation range data, static range data, or both.
- Propose fraction lengths based on default word lengths or propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- View a histogram of bits used by each variable.
- Specify replacement functions or generate approximate functions for functions in the original MATLAB algorithm that do not support fixed point.
- Test the numerical behavior of the fixed-point code. You can then compare its behavior against the floating-point version of your algorithm using either the Simulation Data Inspector or your own custom plotting functions.

If your end goal is to generate fixed-point C code, use the MATLAB Coder™ app instead. See “Convert MATLAB Code to Fixed-Point C Code” (MATLAB Coder).



If your end goal is to generate HDL code, use the HDL Coder™ workflow advisor instead. See “Floating-Point to Fixed-Point Conversion” (HDL Coder).

### Open the Fixed-Point Converter App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- MATLAB command prompt: Enter `fixedPointConverter`.
- To open an existing Fixed-Point Converter app project, either double-click the `.prj` file or open the app and browse to the project file.

Creating a project or opening an existing project causes any other Fixed-Point Converter or MATLAB Coder projects to close.

- A MATLAB Coder project opens in the MATLAB Coder app. To convert the project to a Fixed-Point Converter app project, in the MATLAB Coder app:

- 1  Click  and select **Reopen project as**.
- 2 Select Fixed-Point Converter.

## Examples

- “Propose Data Types Based on Simulation Ranges”
- “Propose Data Types Based on Derived Ranges”

## Programmatic Use

`fixedPointConverter` opens the Fixed-Point Converter app.

`fixedPointConverter -tocode projectname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. It writes the script to the Command Window.

`fixedPointConverter -tocode projectname -script scriptname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. The script is named `scriptname.m`.

- If `scriptname` already exists, `fixedPointConverter` overwrites it.
- The script contains the MATLAB commands to:
  - Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
  - Run the `fiaccel` command to convert the floating-point MATLAB function to a fixed-point MATLAB function.

Before converting the project to a script, you must complete the **Test** step of the fixed-point conversion process.

## See Also

### Functions

fiaccel

### Topics

*“Propose Data Types Based on Simulation Ranges”*

*“Propose Data Types Based on Derived Ranges”*

*“Fixed-Point Conversion Workflows”*

*“Automated Fixed-Point Conversion”*

*“Generated Fixed-Point Code”*

*“Automated Conversion”*

### Introduced in R2014b

# Fixed-Point Tool

Convert floating-point model to fixed-point

## Description

In conjunction with Fixed-Point Designer software, the Fixed-Point Tool provides convenient access to:

- An interactive automatic data typing feature that proposes fixed-point data types for appropriately configured objects in your model, and then allows you to selectively accept and apply the data type proposals
- Model and subsystem parameters that control the signal logging, fixed-point instrumentation mode, and data type override
- Plotting capabilities that enable you to plot data that resides in the MATLAB workspace

Most of the functionality in the Fixed-Point Tool is for use with the Fixed-Point Designer software. However, even if you do not have Fixed-Point Designer software, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, the Simulink® software temporarily overrides fixed-point data types with floating-point data types when simulating the model.

## Open the Fixed-Point Tool

- From the Simulink **Analysis** menu, select **Data Type Design > Fixed-Point Tool**.
- From a subsystem context (right-click) menu, select **Fixed-Point Tool**.
- From the MATLAB command prompt, enter `fxptdlg(system_name)` where `system_name` is the name of the model or system you want to convert, specified as a string.

## Examples

- “Convert Floating-Point Model to Fixed Point”

- “Convert a Model to Fixed Point Using the Command Line”

## **See Also**

“Autoscaling Using the Fixed-Point Tool” | “The Command-Line Interface for the Fixed-Point Tool” | `fxptdlg`

## **Topics**

“Convert Floating-Point Model to Fixed Point”

“Convert a Model to Fixed Point Using the Command Line”

**Introduced before R2006a**

# Lookup Table Optimizer

Optimize an existing lookup table or approximate a function with a lookup table

## Description

Use the Lookup Table Optimizer to obtain an optimized (memory-efficient) lookup table that approximates an existing lookup table, math function, or a function handle. The optimizer supports any combination of floating-point and fixed-point data types. The original input and output data types can be kept or changed as desired. To minimize memory used, the optimizer selects the data types of breakpoints and table data, as well as the number and spacing of breakpoints.

## Open the Lookup Table Optimizer

- In a Simulink model, select **Analysis > Data Type Design > Lookup Table Optimizer**

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

`FunctionApproximation.LUTMemoryUsageCalculator` |  
`FunctionApproximation.LUTSolution` | `FunctionApproximation.Options` |  
`FunctionApproximation.Problem`

### Functions

`approximate` | `compare` | `displayallsolutions` | `displayfeasiblesolutions` |  
`solutionfromID` | `solve` | `totalmemoryusage`

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**



# Single Precision Converter

Convert double-precision system to single precision

## Description

The Single Precision Converter automatically converts a double-precision system to single precision.

During the conversion process, the converter replaces all user-specified double-precision data types, as well as output data types that compile to double precision, with single-precision data types. The converter does not change built-in integer, Boolean, or fixed-point data types.

## Open the Single Precision Converter App

- From the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**.

## Examples

- “Convert a System to Single Precision”

## Programmatic Use

`report = DataTypeWorkflow.Single.convertToSingle(systemToConvert)`  
converts the system specified by `systemToConvert` to single-precision and returns a report. The `systemToConvert` must be open before you begin the conversion.

## See Also

### Functions

`DataTypeWorkflow.Single.convertToSingle`

## **Topics**

“Convert a System to Single Precision”

“Getting Started with Single Precision Converter”

**Introduced in R2016b**

# Blocks — Alphabetical List

---

## Complex Burst Matrix Solve Using QR Decomposition

Compute the value of  $x$  in the equation  $Ax = B$  for complex-valued matrices using QR decomposition

### Description

The Complex Burst Matrix Solve Using QR Decomposition block solves the system of linear equations  $Ax = B$  using QR decomposition, where  $A$  and  $B$  are complex-valued matrices. To compute  $x = A^{-1}B$ , set  $B$  to be the identity matrix.

### Creation

`model = fixed.getMatrixSolveModel(A, B)` generates a model named `model` containing a Matrix Solve QR Decomposition block and data input matrices,  $A$  and  $B$ .

### Ports

#### Input

##### **A(i, :)** — Rows of matrix $A$

vector

Rows of complex matrix  $A$ , where  $A$  is a  $m$ -by- $n$  matrix with  $m \geq 2$  and  $m \geq n$ . If  $B$  is single or double,  $A$  must be the same data type as  $B$ . If  $A$  is fixed point,  $A$  must be signed, use binary point scaling, and have the same word length as  $B$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

##### **B(i, :)** — Rows of matrix $B$

vector

Rows of complex matrix  $B$ , where  $B$  is a  $m$ -by- $p$  matrix with  $m \geq 2$ . If  $A$  is single or double,  $B$  must be the same data type as  $A$ . If  $B$  is fixed point,  $B$  must be signed, use binary point

scaling, and have the same word length as  $A$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

### **validIn — Indicates valid inputs**

Boolean scalar

Indicates valid inputs, specified as a Boolean scalar. Control signal that indicates when the data from the  $A(i, :)$  and  $B(i, :)$  input ports is valid. When this value is 1 (`true`) and “ready” on page 2-0 is 1 (`true`), the block captures the values on the  $A(i, :)$  and  $B(i, :)$  input ports. When this value is 0 (`false`), the block ignores the input samples.

Data Types: Boolean

### **restart — Clear internal states**

Boolean scalar

Clears internal state, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the “validIn” on page 2-0 value is 1 (`true`), the block begins a new subframe.

Data Types: Boolean

## **Output**

### **$X(i, :)$ — Rows of matrix $X$**

scalar | vector

Rows of the matrix  $X$ .

Data Types: `single` | `double` | `fixed point`

### **validOut — Indicates valid output data**

Boolean scalar

Control signal that indicates when the data at the output port  $X(i, :)$  is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix  $X$ . When this value is 0 (`false`), the output data is not valid.

Data Types: Boolean

### **ready — Indicates block is ready**

Boolean scalar

Control signal that indicates when the block is ready for new input data. When this value is 1 (`true`) and “validIn” on page 2-0 is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: Boolean

## Parameters

### **Number of rows in matrices A and B — Number of rows in matrices A and B**

4 (default) | integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *m*

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

### **Number of columns in matrix A — Number of columns in matrix A**

4 (default) | integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *n*

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

### **Number of columns in matrix B — Number of columns in matrix B**

1 (default) | integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *p*

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '1'

**Output datatype — Data type of the output matrix X**

`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as a string. The type be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

**Programmatic Use**

**Block Parameter:** `OutputType`

**Type:** character vector

**Values:** `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`

**Default:** `fixdt(1,18,14)`

## See Also

[Real Burst Matrix Solve Using QR Decomposition](#) | [Real Burst QR Decomposition](#) | [Complex Burst QR Decomposition](#)

## Complex Burst QR Decomposition

QR decomposition for complex-valued matrices

### Description

The Complex Burst QR Decomposition block uses QR decomposition to compute  $R$  and  $C = Q'B$ , where  $QR = A$ , and  $A$  and  $B$  are complex-valued matrices. The least-squares solution to  $Ax = B$  is  $x = R \setminus C$ .  $R$  is an upper triangular matrix and  $Q$  is an orthogonal matrix. To compute  $C = Q'$ , set  $B$  to be the identity matrix.

### Creation

`model = fixed.getQRFactorizationModel(A, B)` generates a model named `model` containing a QR Decomposition block and data input matrices,  $A$  and  $B$ .

### Ports

#### Input

##### **A(i, :)** — Rows of matrix $A$

vector

Rows of matrix  $A$ , where  $A$  is a  $m$ -by- $n$  matrix with  $m, n \geq 2$ . If  $B$  is single or double,  $A$  must be the same data type as  $B$ . If  $A$  is fixed point,  $A$  must be signed, use binary point scaling, and have the same word length as  $B$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

##### **B(i, :)** — Rows of matrix $B$

vector

Rows of matrix  $B$ , where  $B$  is an  $m$ -by- $p$  matrix with  $m \geq 2$ . If  $A$  is single or double,  $B$  must be the same data type as  $A$ . If  $B$  is fixed point,  $B$  must be signed, use binary point scaling,



and have the same word length as  $A$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

### **validIn** — Indicates valid inputs

Boolean scalar

Indicates valid inputs, specified as a Boolean scalar. Control signal that indicates when the data from the  $A(i, :)$  and  $B(i, :)$  input ports is valid. When this value is 1 (`true`) and “ready” on page 2-0 is 1 (`true`), the block captures the values on the  $A(i, :)$  and  $B(i, :)$  input ports. When this value is 0 (`false`), the block ignores the input samples.

Data Types: `Boolean`

### **restart** — Clear internal states

Boolean scalar

Clears internal state, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`), and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

## **Output**

### **$R(i, :)$ — Rows of upper-triangular matrix $R$**

scalar | vector

Rows of the economy sized QR decomposition matrix  $R$ .  $R$  is an upper triangular matrix.  $R$  has the same data type as  $A$ .

Data Types: `single` | `double` | `fixed point`

### **$C(i, :)$ — Rows of matrix $C=Q'B$**

scalar | vector

Rows of the economy sized QR decomposition matrix  $C=Q'B$ .  $C$  has the same number of rows as  $R$ .  $C$  has the same data type as  $B$ .

Data Types: `single` | `double` | `fixed point`

### **validOut** — Indicates valid output data

Boolean scalar

Control signal that indicates when the data at output ports  $R(i, :)$  and  $C(i, :)$  is valid. When this value is 1 (`true`), the block has successfully computed the  $R$  and  $C$  matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: Boolean

### **ready — Indicates block is ready**

Boolean scalar

Control signal that indicates when the block is ready for new input data. When this value is 1 (`true`), and `validIn` is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: Boolean

## Parameters

### **Number of rows in matrices A and B — Number of rows in matrices A and B**

4 (default) | integer-valued scalar

The number of rows in matrices  $A$  and  $B$ , specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:**  $m$

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

### **Number of columns in matrix A — Number of columns in matrix A**

4 (default) | integer-valued scalar

The number of columns in input matrix  $A$ , specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:**  $n$

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

### **Number of columns in matrix B — Number of columns in matrix B**

4 (default) | integer-valued scalar

The number of columns in input matrix  $B$ , specified as a positive integer-valued scalar.

**Programmatic Use**

**Block Parameter:** p

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '1'

**See Also**

Complex Burst QR Decomposition | Real Burst Matrix Solve Using QR Decomposition |  
Complex Burst Matrix Solve Using QR Decomposition

**Introduced in R2019b**

## Real Burst Matrix Solve Using QR Decomposition

Compute the value of  $x$  in the equation  $Ax = B$  for real-valued matrices using QR decomposition

### Description

The Real Burst Matrix Solve Using QR Decomposition block solves the system of linear equations  $Ax = B$  using QR decomposition, where  $A$  and  $B$  are real-valued matrices. To compute  $x = A^{-1}B$ , set  $B$  to be the identity matrix.

### Creation

`model = fixed.getMatrixSolveModel(A, B)` generates a model named `model` containing a Matrix Solve QR Decomposition block and data input matrices,  $A$  and  $B$ .

### Ports

#### Input

##### **A(i, :)** — Rows of matrix $A$

vector

Rows of real matrix  $A$ , where  $A$  is a  $m$ -by- $n$  matrix with  $m \geq 2$  and  $m \geq n$ . If  $B$  is single or double,  $A$  must be the same data type as  $B$ . If  $A$  is fixed point,  $A$  must be signed, use binary point scaling, and have the same word length as  $B$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

##### **B(i, :)** — Rows of matrix $B$

vector

Rows of real matrix  $B$ , where  $B$  is a  $m$ -by- $p$  matrix with  $m \geq 2$ . If  $A$  is single or double,  $B$  must be the same data type as  $A$ . If  $B$  is fixed point,  $B$  must be signed, use binary point

scaling, and have the same word length as  $A$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

### **validIn — Indicates valid inputs**

Boolean scalar

Indicates valid inputs, specified as a Boolean scalar. Control signal that indicates when the data from the  $A(i, :)$  and  $B(i, :)$  input ports is valid. When this value is 1 (`true`) and “ready” on page 2-0 is 1 (`true`), the block captures the values on the  $A(i, :)$  and  $B(i, :)$  input ports. When this value is 0 (`false`), the block ignores the input samples.

Data Types: `Boolean`

### **restart — Clear internal states**

Boolean scalar

Clears internal state, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the “validIn” on page 2-0 value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

## **Output**

### **$X(i, :)$ — Rows of matrix $X$**

scalar | vector

Rows of the matrix  $X$ .

Data Types: `single` | `double` | `fixed point`

### **validOut — Indicates valid output data**

Boolean scalar

Control signal that indicates when the data at the output port  $X(i, :)$  is valid. When this value is 1 (`true`), the block has successfully computed a row of  $X$ . When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

### **ready — Indicates block is ready**

Boolean scalar

Control signal that indicates when the block is ready for new input data. When this value is 1 (`true`) and “validIn” on page 2-0 is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: Boolean

## Parameters

**Number of rows in matrices A and B — Number of rows in matrices A and B**  
4 (default) | integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *m*

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

**Number of columns in matrix A — Number of columns in matrix A**  
4 (default) | integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *n*

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

**Number of columns in matrix B — Number of columns in matrix B**  
1 (default) | integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:** *p*

**Type:** character vector  
**Values:** integer-valued scalar  
**Default:** '1'

**Output datatype — Data type of the output matrix X**

`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as a string. The type be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

**Programmatic Use**

**Block Parameter:** `OutputType`

**Type:** character vector

**Values:** `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`

**Default:** `fixdt(1,18,14)`

## See Also

[Complex Burst Matrix Solve Using QR Decomposition](#) | [Complex Burst QR Decomposition](#) | [Real Burst QR Decomposition](#)

**Introduced in R2019b**

## Real Burst QR Decomposition

QR decomposition for real-valued matrices

### Description

The Real Burst QR Decomposition block uses QR decomposition to compute  $R$  and  $C = Q'B$ , where  $QR = A$ , and  $A$  and  $B$  are real-valued matrices. The least-squares solution to  $Ax = B$  is  $x = R \setminus C$ .  $R$  is an upper triangular matrix and  $Q$  is an orthogonal matrix. To compute  $C = Q'$ , set  $B$  to be the identity matrix.

### Creation

`model = fixed.getQRFactorizationModel(A, B)` generates a model named `model` containing a QR Decomposition block and data input matrices,  $A$  and  $B$ .

### Ports

#### Input

**A(i, :)** — Rows of matrix  $A$

vector

Rows of real matrix  $A$ , where  $A$  is a  $m$ -by- $n$  matrix with  $m, n \geq 2$ . If  $B$  is single or double,  $A$  must be the same data type as  $B$ . If  $A$  is fixed point,  $A$  must be signed, use binary point scaling, and have the same word length as  $B$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

**B(i, :)** — Rows of matrix  $B$

vector

Rows of real matrix  $B$ , where  $B$  is a  $m$ -by- $p$  matrix with  $m \geq 2$ . If  $A$  is single or double,  $B$  must be the same data type as  $A$ . If  $B$  is fixed point,  $B$  must be signed, use binary point



scaling, and have the same word length as  $A$ . Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

### **validIn — Indicates valid inputs**

Boolean scalar

Indicates valid inputs, specified as a Boolean scalar. Control signal that indicates when the data from the  $A(i, :)$  and  $B(i, :)$  input ports is valid. When this value is 1 (`true`) and “ready” on page 2-0 is 1 (`true`), the block captures the values on the  $A(i, :)$  and  $B(i, :)$  input ports. When this value is 0 (`false`), the block ignores the input samples.

Data Types: `Boolean`

### **restart — Clear internal states**

Boolean scalar

Clears internal state, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the “validIn” on page 2-0 value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

## **Output**

### **$R(i, :)$ — Rows of matrix $R$**

scalar | vector

Rows of the economy sized QR decomposition matrix  $R$ .  $R$  is an upper triangular matrix.  $R$  has the same data type as  $A$ .

Data Types: `single` | `double` | `fixed point`

### **$C(i, :)$ — Rows of matrix $C = Q'B$**

scalar | vector

Rows of the economy sized QR decomposition matrix  $C=Q'B$ .  $C$  has the same number of rows as  $R$ .  $C$  has the same data type as  $B$ .

Data Types: `single` | `double` | `fixed point`

### **validOut — Indicates valid output data**

Boolean scalar

Control signal that indicates when the data at output ports  $R(i, :)$  and  $C(i, :)$  is valid. When this value is 1 (`true`), the block has successfully computed the  $R$  and  $C$  matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: Boolean

### **ready — Indicates block is ready**

Boolean scalar

Control signal that indicates when the block is ready for new input data. When this value is 1 (`true`) and “validIn” on page 2-0 is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: Boolean

## Parameters

### **Number of rows in matrices A and B — Number of rows in matrices A and B**

4 (default) | integer-valued scalar

Number of rows in input matrices  $A$  and  $B$ , specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:**  $m$

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

### **Number of columns in matrix A — Number of columns in matrix A**

4 (default) | integer-valued scalar

Number of columns in input matrix  $A$ , specified as a positive integer-valued scalar.

#### **Programmatic Use**

**Block Parameter:**  $n$

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '4'

**Number of columns in matrix B — Number of columns in matrix *B***

1 (default) | integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

**Programmatic Use**

**Block Parameter:** p

**Type:** character vector

**Values:** integer-valued scalar

**Default:** '1'

## See Also

Complex Burst QR Decomposition | Real Burst Matrix Solve Using QR Decomposition |  
Complex Burst Matrix Solve Using QR Decomposition



# Property Reference

---

- “fi Object Properties” on page 3-2
- “fipref Object Properties” on page 3-4
- “quantizer Object Properties” on page 3-7

# fi Object Properties

The properties associated with `fi` objects are described in the following sections in alphabetical order.

You can set these properties when you create a `fi` object. For example, to set the stored integer value of a `fi` object:

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

---

**Note** The `fimath` properties and `numericType` properties are also properties of the `fi` object. Refer to “`fimath` Object Properties” and “`numericType` Object Properties” for more information.

---

## **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.

## **fimath**

`fimath` properties associated with a `fi` object. `fimath` properties determine the rules for performing fixed-point arithmetic operations on `fi` objects. `fi` objects get their `fimath` properties from a local `fimath` object or from default values. The factory-default `fimath` values have the following settings:

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

To learn more about `fi` objects, refer to “`fi` Object Construction”. For more information about each of the `fi` object properties, refer to “`fi` Object Properties”.

## **hex**

Stored integer value of a `fi` object in hexadecimal.

## **int**

Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type.

## **NumericType**

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. For a table of the possible settings of each field of the structure, see “Valid Values for `numericType` Object Properties” in the Fixed-Point Designer User's Guide.

---

**Note** You cannot change the `numericType` properties of a `fi` object after `fi` object creation.

---

## **oct**

Stored integer value of a `fi` object in octal.

## **Value**

Full-precision real world value of a `fi` object, stored as a character vector.

# fipref Object Properties

The properties associated with `fipref` objects are described in the following sections in alphabetical order.

## **DataTypeOverride**

Data type override options for `fi` objects

- `ForceOff` — No data type override
- `ScaledDoubles` — Override with scaled doubles
- `TrueDoubles` — Override with doubles
- `TrueSingles` — Override with singles

Data type override only occurs when the `fi` constructor function is called.

The default value of this property is `ForceOff`.

## **DataTypeOverrideAppliesTo**

Data type override application to `fi` objects

- `AllNumericTypes` — Apply data type override to all `fi` data types. This setting does not override builtin integer types.
- `Fixed-Point` — Apply data type override only to fixed-point data types
- `Floating-Point` — Apply data type override only to floating-point `fi` data types

`DataTypeOverrideAppliesTo` displays only if `DataTypeOverride` is not set to `ForceOff`.

The default value of this property is `AllNumericTypes`.

## **FimathDisplay**

Display options for the `fimath` attributes of a `fi` object

- `full` — Displays all of the `fimath` attributes of a fixed-point object



- `none` — None of the `fimath` attributes are displayed

The default value of this property is `full`.

## LoggingMode

Logging options for operations performed on `fi` objects

- `off` — No logging
- `on` — Information is logged for future operations

Overflows and underflows for assignment, plus, minus, and multiplication operations are logged as warnings when `LoggingMode` is set to `on`.

When `LoggingMode` is `on`, you can also 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`.

The default value of this property is `off`.

## NumericTypeDisplay

Display options for the `numericType` attributes of a `fi` object

- `full` — Displays all the `numericType` attributes of a fixed-point object
- `none` — None of the `numericType` attributes are displayed.
- `short` — Displays an abbreviated notation of the fixed-point data type and scaling of a fixed-point object in the format `xWL, FL` where
  - `x` is `s` for signed and `u` for unsigned.
  - `WL` is the word length.

- FL is the fraction length.

The default value of this property is `full`.

## NumberDisplay

Display options for the value of a `fi` object

- `bin` — Displays the stored integer value in binary format
- `dec` — Displays the stored integer value in unsigned decimal format
- `RealWorldValue` — Displays the stored integer value in the format specified by the MATLAB format function
- `hex` — Displays the stored integer value in hexadecimal format
- `int` — Displays the stored integer value in signed decimal format
- `none` — No value is displayed.

The default value of this property is `RealWorldValue`. In this mode, the value of a `fi` object is displayed in the format specified by the MATLAB format function: `+`, `bank`, `compact`, `hex`, `long`, `long e`, `long g`, `loose`, `rat`, `short`, `short e`, or `short g`. `fi` objects in `rat` format are displayed according to

$$\frac{1}{(2^{\text{fixed} - \text{pointexponent}})} \times \text{storedinteger}$$

## quantizer Object Properties

The properties associated with `quantizer` objects are described in the following sections in alphabetical order.

### DataMode

Type of arithmetic used in quantization. This property can have the following values:

- `fixed` — Signed fixed-point calculations
- `float` — User-specified floating-point calculations
- `double` — Double-precision floating-point calculations
- `single` — Single-precision floating-point calculations
- `ufixed` — Unsigned fixed-point calculations

The default value of this property is `fixed`.

When you set the `DataMode` property value to `double` or `single`, the `Format` property value becomes read only.

### Format

Data format of a `quantizer` object. The interpretation of this property value depends on the value of the `DataMode` property.

For example, whether you specify the `DataMode` property with fixed- or floating-point arithmetic affects the interpretation of the data format property. For some `DataMode` property values, the data format property is read only.

The following table shows you how to interpret the values for the `Format` property value when you specify it, or how it is specified in read-only cases.

<b>DataMode Property Value</b>	<b>Interpreting the Format Property Values</b>
fixed or ufixed	<p>You specify the Format property value as a vector. The number of bits for the quantizer object word length is the first entry of this vector, and the number of bits for the quantizer object fraction length is the second entry.</p> <p>The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to one less than the word length.</p>
float	<p>You specify the Format property value as a vector. The number of bits you want for the quantizer object word length is the first entry of this vector, and the number of bits you want for the quantizer object exponent length is the second entry.</p> <p>The word length can range from 2 to the limits of memory on your PC. The exponent length can range from 0 to 11.</p>
double	<p>The Format property value is specified automatically (is read only) when you set the DataMode property to double. The value is [64 11], specifying the word length and exponent length, respectively.</p>
single	<p>The Format property value is specified automatically (is read only) when you set the DataMode property to single. The value is [32 8], specifying the word length and exponent length, respectively.</p>

## OverflowMode

Overflow-handling mode. The value of the OverflowMode property can be one of the following:

- Saturate — Overflows saturate.

When the values of data to be quantized lie outside the range of the largest and smallest representable numbers (as specified by the data format properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.

- Wrap — Overflows wrap to the range of representable values.

When the values of data to be quantized lie outside the range of the largest and smallest representable numbers (as specified by the data format properties), these

values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

The default value of this property is `Saturate`. This property becomes a read-only property when you set the `DataMode` property to `float`, `double`, or `single`.

---

**Note** Floating-point numbers that extend beyond the dynamic range overflow to  $\pm\text{inf}$ .

---

## RoundingMode

Rounding method. The value of the `RoundingMode` property can be one of the following:

- `Ceiling` — Round up to the next allowable quantized value.
- `Convergent` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 0.
- `Zero` — Round negative numbers up and positive numbers down to the next allowable quantized value.
- `Floor` — Round down to the next allowable quantized value.
- `Nearest` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.

The default value of this property is `Floor`.



# Fixed-Point Tool

---

- “Fixed-Point Tool Parameters and Dialog Box” on page 4-2
- “Advanced Settings” on page 4-20

## Fixed-Point Tool Parameters and Dialog Box











The Fixed-Point Tool includes the following components:

- **Main** toolbar
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane


### Main Toolbar

The Fixed-Point Tool's main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool's menu.

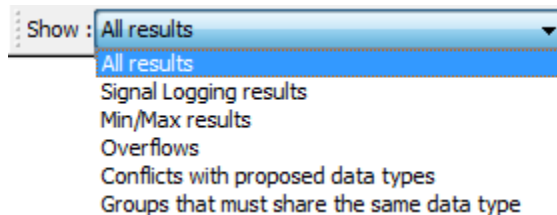
The toolbar contains the following buttons that execute commonly used Fixed-Point Tool commands:

Button	Usage
	Open the Fixed-Point Advisor to prepare the model for conversion to fixed point.
	Simulate a model and store the run results.
	Pause a simulation.
	Stop a simulation.
	Analyze model and store derived minimum and maximum results.
	Propose data types. Propose fraction lengths for specified word lengths or propose word lengths for specified fraction lengths.
	Apply accepted data types.
	Compare selected runs.
	Create a difference plot for the selected signals.
	Plot the selected signal.






Button	Usage
	Create a histogram plot for the selected signal.

The toolbar also contains the **Show** option:



The **Show** option specifies the type of results to display in the **Contents** pane. The **Contents** pane displays information only after you simulate a system or propose fraction lengths. If there are no results that satisfy a particular filter option, the list will be blank.

Show Option	Result
<b>All results</b>	Displays all results for the selected tree node.
<b>Signal Logging results</b>	For the selected tree node, displays blocks whose output ports have logged signal data. The Fixed-Point tool marks these blocks with the logged signal icon  .  <b>Note</b> You can plot simulation results associated with logged signal data using the Simulation Data Inspector.
<b>Min/Max results</b>	For the selected tree node, displays blocks that record design Min/Max, simulation Min/Max, and overflow data.  <b>Prerequisites:</b> <b>Fixed-point instrumentation mode</b> should not be set to Force Off.
<b>Overflows</b>	For the selected tree node, displays blocks that have non-zero overflows recorded. If a block has its <b>Saturate on integer overflow</b> option selected, overflow information appears in the <b>Saturations</b> column, otherwise it appears in the <b>OverflowWraps</b> column.

Show Option	Result
<p><b>Conflicts with proposed data types</b></p>	<p>For the selected tree node, displays results that have potential data typing or scaling issues.</p> <p><b>Prerequisites:</b> This information is available only after you propose data types.</p> <p>The Fixed-Point Tool marks these results with a yellow or red icon, as shown here:</p> <div style="border: 1px solid black; padding: 5px;"> <p> The proposed data type poses potential issues for this object. Open the Result Details tab to review these issues.</p> <p> The proposed data type will introduce errors if applied to this object. Open the Result Details tab for details about how to resolve these issues.</p> </div>
<p><b>Groups that must share the same data type</b></p>	<p>For the selected tree node, displays blocks that must share the same data type because of data type propagation rules.</p> <p><b>Prerequisites:</b> This information is available only after you propose fraction lengths.</p> <p>The Fixed-Point Tool allocates an identification tag to blocks that must share the same data type. This identification tag is displayed in the <b>DTGroup</b> column as follows:</p> <ul style="list-style-type: none"> <li>• If the selected tree node is the model root           <p>All results for the model are listed. The <b>DTGroup</b> column is sorted by default so that you can easily view all blocks in a group.</p> </li> <li>• If the selected tree node is a subsystem           <p>The identification tags have a suffix that indicates the total number of results in each group. For example, G2 (2) means group G2 has 2 members. This information enables you to see how many members of a group belong to the selected subsystem and which groups share data types across subsystem boundaries.</p> </li> </ul>

## Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy. The first node in the pane represents a Simulink model. Expanding the root node displays subnodes that represent the model's subsystems, MATLAB Function blocks, Stateflow® charts, and referenced models.

The Fixed-Point Tool's **Contents** pane displays elements that comprise the object selected in the **Model Hierarchy** pane. The **Dialog** pane provides parameters for specifying the selected object's data type override and fixed-point instrumentation mode. You can also specify an object's data type override and fixed-point instrumentation mode by right-clicking on the object. The **Model Hierarchy** pane indicates the value of these parameters by displaying the following abbreviations next to the object name:

Abbreviation	Parameter Value
<b>Fixed-point instrumentation mode</b>	
mmo	Minimums, maximums and overflows
o	Overflows only
fo	Force off
<b>Data type override</b>	
scl	Scaled double
dbl	Double
sgl	Single
off	Off

## Contents Pane

The **Contents** pane displays a tabular view of objects that log fixed-point data in the system or subsystem selected in the **Model Hierarchy** pane. The table rows correspond to model objects, such as blocks, block parameters, and Stateflow data. The table columns correspond to attributes of those objects, such as the data type, design minimum and maximum values, and simulation minimum and maximum values.

The **Contents** pane displays information only after you simulate a system, analyze the model to derive minimum and maximum values, or propose fraction lengths.

You can control which of the following columns the Fixed-Point Tool displays in this pane. For more information, see “Customizing the Contents Pane View” on page 4-8.

Column Label	Description
<b>Accept</b>	Check box that enables you to selectively accept the Fixed-Point Tool's data type proposal.
<b>CompiledDesignMax</b>	Compile-time information for <b>DesignMax</b> .
<b>CompiledDesignMin</b>	Compile-time information for <b>DesignMin</b> .
<b>CompiledDT</b>	Compile-time data type. This data type appears on the signal line in <code>sfix</code> format. See “Fixed-Point Data Type and Scaling Notation”.
<b>DerivedMax</b>	Maximum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.
<b>DerivedMin</b>	Minimum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.
<b>DesignMax</b>	Maximum value the block specifies in its parameter dialog box, for example, the value of its <b>Output maximum</b> parameter.
<b>DesignMin</b>	Minimum value the block specifies in its parameter dialog box, for example, the value of its <b>Output minimum</b> parameter.
<b>DivByZero</b>	Number of divide-by-zero instances that occur during simulation.
<b>DTGroup</b>	Identification tag associated with objects that share data types.
<b>InitValueMax</b>	<p>Maximum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a <b>Constant value</b> that initializes the block output signal.</p> <hr/> <p><b>Note</b> The Fixed-Point Tool uses this parameter when it proposes data types.</p>

Column Label	Description
<b>InitValueMin</b>	<p>Minimum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a <b>Constant value</b> that initializes the block output signal.</p> <hr/> <p><b>Note</b> The Fixed-Point Tool uses this parameter when it proposes data types.</p>
<b>LogSignal</b>	<p>Check box that allows you to enable or disable signal logging for an object.</p>
<b>ModelRequiredMin</b>	<p>Minimum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the <b>Breakpoints</b> and <b>Table data</b> parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets <b>ModelRequiredMin</b> to the minimum of the minimum values of all these parameters.</p> <hr/> <p><b>Note</b> The Fixed-Point Tool uses this parameter when it proposes data types.</p>
<b>ModelRequiredMax</b>	<p>Maximum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the <b>Breakpoints</b> and <b>Table data</b> parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets <b>ModelRequiredMax</b> to the maximum of the maximum values of all these parameters.</p> <hr/> <p><b>Note</b> The Fixed-Point Tool uses this parameter when it proposes data types.</p>
<b>Name</b>	<p>Identifies path and name of block.</p>
<b>OverflowWraps</b>	<p>Number of overflows that wrap during simulation.</p>
<b>ProposedDT</b>	<p>Data type that the Fixed-Point Tool proposes.</p>

Column Label	Description
<b>ProposedMax</b>	Maximum value that results from the data type the Fixed-Point Tool proposes.
<b>ProposedMin</b>	Minimum value that results from the data type the Fixed-Point Tool proposes.
<b>Run</b>	Indicates the run name for these results.
<b>Saturations</b>	Number of overflows that saturate during simulation.
<b>SimDT</b>	Data type the block uses during simulation. This data type appears on the signal line in <code>sfix</code> format. See “Fixed-Point Data Type and Scaling Notation”.
<b>SimMax</b>	Maximum value that occurs during simulation.
<b>SimMin</b>	Minimum value that occurs during simulation.
<b>SpecifiedDT</b>	Data type the block specifies in its parameter dialog box, for example, the value of its <b>Output data type</b> parameter.

## Customizing the Contents Pane View

You can customize the **Contents** pane in the following ways:

- “Using Column Views” on page 4-8
- “Changing Column Order and Width” on page 4-10
- “Sorting by Columns” on page 4-10

### Using Column Views

The Fixed-Point Tool provides the following standard Column Views:

View Name	Columns Provided	When Does the Fixed-Point Tool Display this View?
Simulation View (default)	<b>Name, Run, CompiledDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations</b>	After a simulating minimum and maximum values.

<b>View Name</b>	<b>Columns Provided</b>	<b>When Does the Fixed-Point Tool Display this View?</b>
Automatic Data Typing View	<b>Name, Run, CompiledDT, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DesignMin, DesignMax, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations, ProposedMin, ProposedMax</b>	After proposing data types if proposal is based on simulation, derived, and design min/max.
Automatic Data Typing With Simulation Min/Max View	<b>Name, Run, CompiledDT, Accept, ProposedDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations, ProposedMin, ProposedMax</b>	After proposing data types if the proposal is based on simulation and design min/max.
Automatic Data Typing With Derived Min/Max View	<b>Name, Run, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DerivedMin, DerivedMax, ProposedMin, ProposedMax</b>	After proposing data types if the proposal is based on design min/max and/or derived min/max.
Data Collection View	<b>Name, Run, CompiledDT, SpecifiedDT, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations</b>	After simulating or deriving minimum and maximum values if the results have simulation min/max, derived min/max, and design min/max.
Derived Min/Max View	<b>Name, Run, CompiledDesignMax, CompiledDesignMin, DerivedMin, DerivedMax</b>	After deriving minimum and maximum values.

By selecting **Show Details**, you can:

- Customize the standard column views
- Create your own column views

- Export and import column views saved in MAT-files, which you can share with other users
- Reset views to factory settings

If you upgrade to a new release of Simulink, and the column views available in the Fixed-Point Tool do not match the views described in the documentation, reset your views to factory settings. When you reset all views, the Model Explorer removes all the custom views you have created. Before you reset views to factory settings, export any views that you will want to use in the future.

You can prevent the Fixed-Point Tool from automatically changing the column view of the contents pane by selecting **View > Lock Column View** in the Fixed-Point Tool menu. For more information on controlling views, see “Customize Model Explorer Views” (Simulink).

### Changing Column Order and Width

You can alter the order and width of columns that appear in the **Contents** pane as follows:

- To move a column, click and drag the head of a column to a new location among the column headers.
- To make a column wider or narrower, click and drag the right edge of a column header. If you double-click the right edge of a column header, the column width changes to fit its contents.

### Sorting by Columns

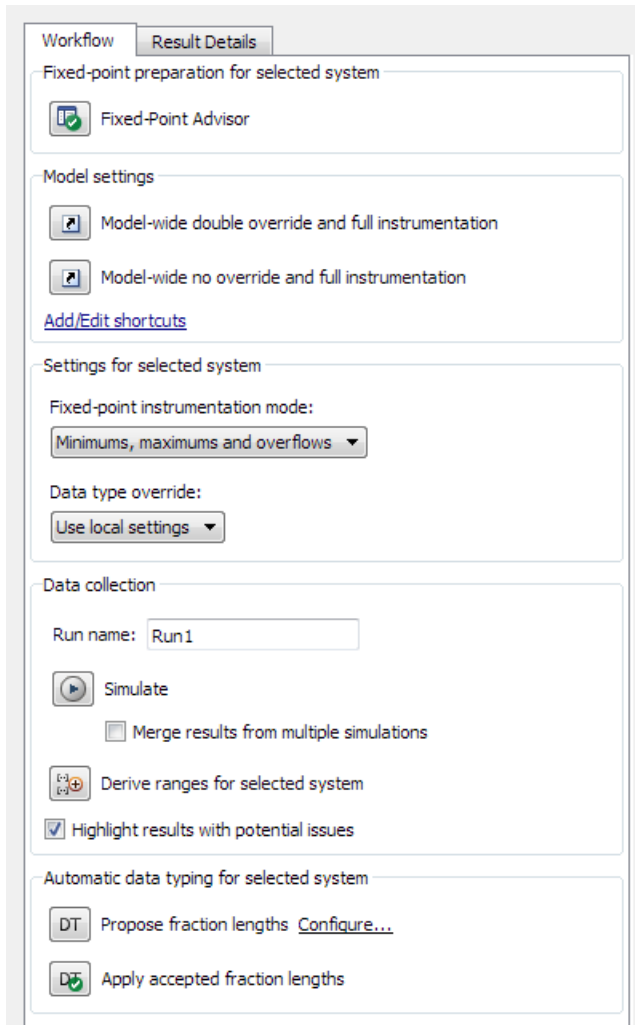
By default, the **Contents** pane displays its contents in ascending order of the **Name** column. You can alter the order in which the **Contents** pane displays its rows as follows:

- To sort all the rows in ascending order of another column, click the head of that column.
- To change the order from ascending to descending, simply click again on the head of that column.

### Dialog Pane

Use the Dialog pane to view and change properties associated with the system under design.





The Dialog pane includes the following components:

Component	Description
<b>System under design</b>	Displays the system under design for conversion. You can change the selected system by clicking <b>Change</b> .

Component	Description
<b>Fixed-point preparation</b>	Contains the <b>Fixed-Point Advisor</b> button. Use this button to open the Fixed-Point Advisor to guide you through the tasks to prepare your floating-point model for conversion to fixed point. For more information, see “Fixed-Point Advisor” on page 4-12.
<b>Configure model settings</b>	Contains default configurations that set up run parameters, such as the run name and data type override settings, by clicking a button. For more information, see “Configure model settings” on page 4-13.
<b>Range collection</b>	Contains controls to collect simulation or derived minimum and maximum data for your model.
<b>Automatic data typing</b>	Contains controls to propose and, optionally, accept data type proposals.
<b>Result Details</b> tab	Use this tab to view data type information about the object selected in the <b>Contents</b> pane.

### Tips

From the Fixed-Point Tool **View** menu, you can customize the layout of the **Dialog** pane. Select:

- **Show Fixed-Point Preparation** to show/hide the **Fixed-Point Advisor** button. By default, the Fixed-Point Tool displays this button.
- **Show Dialog View** to show/hide the **Dialog** pane. By default, the Fixed-Point Tool displays this pane.
- **Settings for selected system** to show/hide the **Settings for selected system** pane. By default, the Fixed-Point Tool displays this pane.

### Fixed-Point Advisor

Open the Fixed-Point Advisor to guide you through the tasks to prepare a floating-point model for conversion to fixed point. Use the Fixed-Point Advisor if your model contains blocks that do not support fixed-point data types.

## Configure model settings

Use the configurations to set up model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently-used factory default configurations
- The ability to add and edit custom configurations

---

**Note** The factory default configurations apply to the whole model. You cannot use these shortcuts to configure subsystems.

---

### Factory Defaults

Factory Default Configuration	Description
<b>Range collection using double override</b>	<p>Use this configuration to observe ideal numeric behavior of the model and collect ranges for data type proposals.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> <li>• <b>Run name</b> to DoubleOverride</li> <li>• <b>Fixed-point instrumentation mode</b> to Minimums, maximums and overflows</li> <li>• <b>Data type override</b> to Double</li> <li>• <b>Data type override applies to</b> to All numeric types</li> </ul> <p>By default, a button for this configuration appears in the <b>Configure model settings</b> pane.</p>

Factory Default Configuration	Description
<p><b>Range collection with specified data types</b></p>	<p>Use this configuration to collect ranges of actual model and to validate current behavior.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> <li>• <b>Run name</b> to NoOverride</li> <li>• <b>Fixed-point instrumentation mode</b> to Minimums, maximums and overflows</li> <li>• <b>Data type override</b> to Use local settings</li> </ul> <p>By default, a button for this shortcut appears in the <b>Configure model settings</b> pane.</p>
<p><b>Remove overrides and disable range collection</b></p>	<p>Use this configuration to cleanup settings after finishing fixed-point conversion and to restore maximum simulation speed.</p> <p>This configuration sets:</p> <ul style="list-style-type: none"> <li>• <b>Fixed-point instrumentation mode</b> to Off</li> <li>• <b>Data type override</b> to Use local settings</li> </ul> <p>By default, a button for this shortcut appears in the <b>Configure model settings</b> pane.</p>

**Advanced settings**

Use **Advanced settings** to add new configurations or edit existing user-defined configurations.

**Run name**

Specifies the run name

If you use a default configuration to set up a run, the Fixed-Point Tool uses the run name associated with this configuration. You can override the run name by entering a new name in this field.

## Tips

- To store data for multiple runs, provide a different run name for each run. Running two simulations with the same run name overwrites the original run unless you select **Merge results from multiple simulations**.
- You can edit the run name in the Contents pane **Run** column.

For more information, see “Run Management”.

## Simulate

Simulates model and stores results.

### Action

Simulates the model and stores the results with the run name specified in **Run name**. The Fixed-Point Tool displays the run name in the **Run** column of the **Contents** pane.

## Merge instrumentation results from multiple simulations

Control how simulation results are stored

### Settings

**Default:** Off

On

Merges new simulation minimum and maximum results with existing simulation results in the run specified by the run name parameter. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

Off

Clears all existing simulation results from the run specified by the run name parameter before displaying new simulation results.

### Command-Line Alternative

**Parameter:** 'MinMaxOverflowArchiveMode'

**Type:** string

**Value:** 'Overwrite' | 'Merge'

**Default:** 'Overwrite'

### Tip

Select this parameter to log simulation minimum and maximum values captured over multiple simulations. For more information, see “Propose Data Types For Merged Simulation Ranges”.

## Derive ranges for selected system

Derive minimum and maximum values for signals for the selected system.

The Fixed-Point Tool analyzes the selected system to compute derived minimum and maximum values based on design minimum and maximum values specified on blocks. For example, using the **Output minimum** and **Output maximum** for block outputs.

### Action

Analyzes the selected system to compute derived minimum and maximum information based on the design minimum and maximum values specified on blocks.

By default, the Fixed-Point Tool displays the **Derived Min/Max View** with the following information in the **Contents** pane.

### Command-Line Alternative

No command line alternative available.

### Dependencies

Range analysis:

- Requires a Fixed-Point Designer license.

## Propose

### Signedness

Select whether you want The Fixed-Point Tool to propose signedness for results in your model. The Fixed-Point Tool proposes signedness based on collected range data and block constraints. By default, the **Signedness** check box is selected.

When the check box is selected, signals that are always strictly positive get an unsigned data type proposal. If you clear the check box, the Fixed-Point Tool proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the Fixed-Point Tool will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

### **Word length or fraction length**

Select whether you want the Fixed-Point Tool to propose word lengths or fraction lengths for the objects in your system.

- If you select **Word length**, the Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length to avoid overflows.
- If you select **Fraction length**, the Fixed-Point Tool proposes a data type with the specified word length and best-precision fraction length while avoiding overflows.

If a result currently specifies a fixed-point data type, that information will be used in the proposal. If a result specifies a floating-point or inherited output data type, and the **Inherited** and **Floating point** check boxes are selected, the Fixed-Point Tool uses the settings specified under **Automatic data typing** to make a data type proposal.

## **Propose for**

### **Inherited**

Propose data types for results that specify one of the inherited output data types.

### **Floating-point**

Propose data types for results that specify floating-point output data types.

## **Default fraction length**

Specify the default fraction length for objects in your model. The Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length that avoids overflows.

### **Command-Line Alternative**

No command line alternative available.

### **Default word length**

Specify the default word length for objects in your model. The Fixed-Point Tool will propose best-precision fraction lengths based on the specified default word length.

### **Command-Line Alternative**

No command line alternative available.

### **When proposing types use**

Specify the types of ranges to use for data type proposals.

### **Design and derived ranges**

The Fixed-Point Tool uses the design ranges in conjunction with derived ranges to propose data types. Design ranges take precedence over derived ranges.

### **Design and simulation ranges**

The Fixed-Point Tool uses the design ranges in conjunction with collected simulation ranges to propose data types. Design ranges take precedence over simulation ranges.

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 (%)” on page 4-18

### **All collected ranges**

The Fixed-Point Tool uses design ranges in addition to derived and simulation ranges to propose data types.

Design minimum and maximum values take precedence over simulation and derived ranges.

### **Command-Line Alternative**

No command line alternative available.

### **Safety margin for simulation min/max (%)**

Specify safety factor for simulation minimum and maximum values.



## Settings

### Default: 0

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. The specified safety margin must be a real number greater than -100. For example, a value of 55 specifies that a range *at least* 55 percent larger is desired. A value of -15 specifies that a range *up to* 15 percent smaller is acceptable.

### Dependencies

Before performing automatic data typing, you must specify design minimum and maximum values or run a simulation to collect simulation minimum and maximum data, or collect derived minimum and maximum values.

### Command-Line Alternative

No command line alternative available.

## Advanced Settings

In this section...
“Advanced Settings Overview” on page 4-20
“Fixed-point instrumentation mode” on page 4-20
“Data type override” on page 4-21
“Data type override applies to” on page 4-23
“Name of shortcut” on page 4-25
“Allow modification of fixed-point instrumentation settings” on page 4-25
“Allow modification of data type override settings” on page 4-26
“Allow modification of run name” on page 4-27
“Run name” on page 4-27
“Capture system settings” on page 4-27
“Fixed-point instrumentation mode” on page 4-27
“Data type override” on page 4-28
“Data type override applies to” on page 4-29

### Advanced Settings Overview

Use the Advanced Settings dialog to control the fixed-point instrumentation mode, and data type override settings. You can also use the Advanced Settings dialog to add or edit user-defined configurations. You cannot modify the factory default configurations. If you add a new configuration and want it to appear as a button on the Fixed-Point Tool **Configure model settings** pane, use the controls in the **Shortcuts** tab.

### Fixed-point instrumentation mode

Control which objects log minimum, maximum and overflow data during simulation.

#### Settings

**Default:** Use local settings

### Use local settings

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

### Minimums, maximums and overflows

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

### Overflows only

Logs only overflow data for all blocks in the current system or subsystem.

### Force off

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

## Tips

- You cannot change the instrumentation mode for linked subsystems or referenced models.

## Dependencies

The value of this parameter for parent systems controls min/max logging for all child subsystems, unless `Use local settings` is selected.

## Command-Line Alternative

**Parameter:** 'MinMaxOverflowLogging'

**Type:** string

**Value:** 'UseLocalSettings' | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'

**Default:** 'UseLocalSettings'

## Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

## Settings

**Default:** Use local settings

The value of this parameter for parent systems controls data type override for all child subsystems, unless `Use local settings` is selected.

### Use local settings

Overrides data types according to the setting of this parameter for each subsystem.

### Scaled double

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

### Double

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

### Single

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

### Off

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

## Tips

- Set this parameter to `Double` or `Single` and the **Data type override applies to** parameter to `All numeric types` to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.
- You cannot change the **Data type override** setting on linked subsystems or referenced models.
- Data type override never applies to `boolean` data types.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local settings`, you can set the **Data type override** parameter for individual children.
- Use this parameter with the **Data type override applies to** parameter. The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double
Single	All numeric types	Single	Single
	Floating-point	Single	Unchanged
	Fixed-point	Unchanged	Single
Scaled double	All numeric types	Double	Scaled double equivalent of fixed-point type
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Scaled double equivalent of fixed-point type

### Dependencies

- The following Simulink blocks allow you to set data types in their block masks, but ignore the **Data type override** setting:
  - Probe
  - Trigger
  - Width

### Command-Line Alternative

**Parameter:** 'DataTypeOverride'

**Type:** string

**Value:** 'UseLocalSettings' | 'ScaledDouble' | 'Double' | 'Single' | 'Off'

**Default:** 'UseLocalSettings'

### Data type override applies to

Specifies which data types the Fixed-Point Tool overrides

## Settings

**Default:** All numeric types

All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to `boolean` or enumerated data types.

Floating-point

Data type override applies only to floating-point data types, that is, `double` and `single`.

Fixed-point

Data type override applies only to fixed-point data types, for example, `uint8`, `fixdt`.

## Tips

- Use this parameter with the **Data type override** parameter.
- Data type override never applies to `boolean` or enumerated data types or to buses.
- When you set the **Data type override** parameter of a parent system to `Double`, `Single`, `Scaled double` or `Off`, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is `Use local setting`, you can set the **Data type override** parameter for individual children.
- The following table details how these two parameters affect the data types in your model.

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Use local settings/Off	N/A	Unchanged	Unchanged
Double	All numeric types	Double	Double
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Double
Single	All numeric types	Single	Single
	Floating-point	Single	Unchanged
	Fixed-point	Unchanged	Single

Fixed-Point Tool Settings		Block Local Settings	
Data type override	Data type override applies to	Floating-point types	Fixed-point types
Scaled double	All numeric types	Double	Scaled double equivalent of fixed-point type
	Floating-point	Double	Unchanged
	Fixed-point	Unchanged	Scaled double equivalent of fixed-point type

### Dependencies

This parameter is enabled only when **Data type override** is set to Scaled double, Double or Single.

### Command-Line Alternative

**Parameter:** 'DataTypeOverrideAppliesTo'

**Type:** string

**Value:** 'AllNumericTypes' | 'Floating-point' | 'Fixed-point'

**Default:** 'AllNumericTypes'

### Name of shortcut

Enter a unique name for your shortcut. By default, the Fixed-Point Tool uses this name as the **Run name** for this shortcut.

If the shortcut name already exists, the new settings overwrite the existing settings.

### See Also

- “Run Management”

### Allow modification of fixed-point instrumentation settings

Select whether to change the model fixed-point instrumentation settings when you apply this shortcut to the model.

### Settings

**Default:** On

On

When you apply this shortcut to the model, changes the fixed-point instrumentation settings of the model and its subsystems to the setting defined in this shortcut.

Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

### Tip

If you want to control data type override settings without altering the fixed-point instrumentation settings on your model, clear this option.

### See Also

- “Run Management”

## Allow modification of data type override settings

Select whether to change the model data type override settings when you apply this shortcut to the model

### Settings

**Default:** On

On

When you apply this shortcut to the model, changes the data type override settings of the model and its subsystems to the settings defined in this shortcut .

Off

Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.



## Allow modification of run name

Select whether to change the run name on the model when you apply this shortcut to the model

### Settings

**Default:** On

On

Changes the run name to the setting defined in this shortcut when you apply this shortcut to the model.

Off

Does not change the run name when you apply this shortcut to the model.

## Run name

Specify the run name to use when you apply this shortcut.

By default, the run name uses the name of the shortcut. Run names are case sensitive.

### Dependency

**Allow modification of run name** enables this parameter.

## Capture system settings

Copy the model and subsystem fixed-point instrumentation mode and data type override settings into the Shortcut editor.

## Fixed-point instrumentation mode

Control which objects in the shortcut editor log minimum, maximum and overflow data during simulation.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

### Settings

**Default:** Same as model setting

#### Use local settings

Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

#### Minimums, maximums and overflows

Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

#### Overflows only

Logs only overflow data for all blocks in the current system or subsystem.

#### Force off

Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

### Dependency

**Allow modification of fixed-point instrumentation settings** enables this parameter.

## Data type override

Control data type override of objects that allow you to specify data types in their dialog boxes.

This information is stored in the shortcut. To use the current model settings, click **Capture system settings**.

### Settings

**Default:** Same as model

The value of this parameter for parent systems controls data type override for all child subsystems, unless Use local settings is selected.

#### Use local settings

Overrides data types according to the setting of this parameter for each subsystem.

### Scaled double

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

### Double

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

### Single

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

### Off

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

## Dependency

**Allow modification of data type override settings** enables this parameter.

## Data type override applies to

Specifies which data types to override when you apply this shortcut.

This information is stored in the shortcut. To use the current model setting, click **Capture system settings**.

## Settings

**Default:** All numeric types

### All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to boolean or enumerated data types.

### Floating-point

Data type override applies only to floating-point data types, that is, double and single.

### Fixed-point

Data type override applies only to fixed-point data types, for example, uint8, fixdt.

**Dependency**

**Allow modification of data type override settings** enables this parameter.

# Functions — Alphabetical List

---

## abs

Absolute value of `fi` object

### Syntax

```
c = abs(a)
c = abs(a,T)
c = abs(a,F)
c = abs(a,T,F)
```

### Description

`c = abs(a)` returns the absolute value of `fi` object `a` with the same `numericType` object as `a`. Intermediate quantities are calculated using the `fiMath` associated with `a`. The output `fi` object `c` has the same local `fiMath` as `a`.

`c = abs(a,T)` returns a `fi` object with a value equal to the absolute value of `a` and `numericType` object `T`. Intermediate quantities are calculated using the `fiMath` associated with `a` and the output `fi` object `c` has the same local `fiMath` as `a`. See “Data Type Propagation Rules” on page 5-3.

`c = abs(a,F)` returns a `fi` object with a value equal to the absolute value of `a` and the same `numericType` object as `a`. Intermediate quantities are calculated using the `fiMath` object `F`. The output `fi` object `c` has no local `fiMath`.

`c = abs(a,T,F)` returns a `fi` object with a value equal to the absolute value of `a` and the `numericType` object `T`. Intermediate quantities are calculated using the `fiMath` object `F`. The output `fi` object `c` has no local `fiMath`. See “Data Type Propagation Rules” on page 5-3.

---

**Note** When the Signedness of the input `numericType` object `T` is `Auto`, the `abs` function always returns an `Unsigned fi` object.

---

`abs` only supports `fi` objects with [Slope Bias] scaling when the bias is zero and the fractional slope is one. `abs` does not support complex `fi` objects of data type `Boolean`.

When the object `a` is real and has a signed data type, the absolute value of the most negative value is problematic since it is not representable. In this case, the absolute value saturates to the most positive value representable by the data type if the `OverflowAction` property is set to `saturate`. If `OverflowAction` is `wrap`, the absolute value of the most negative value has no effect.

## Data Type Propagation Rules

For syntaxes for which you specify a `numerictype` object `T`, the `abs` function follows the data type propagation rules listed in the following table. In general, these rules can be summarized as “floating-point data types are propagated.” This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input <code>fi</code> Object <code>a</code>	Data Type of <code>numerictype</code> object <code>T</code>	Data Type of Output <code>c</code>
<code>fi Fixed</code>	<code>fi Fixed</code>	Data type of <code>numerictype</code> object <code>T</code>
<code>fi ScaledDouble</code>	<code>fi Fixed</code>	<code>ScaledDouble</code> with properties of <code>numerictype</code> object <code>T</code>
<code>fi double</code>	<code>fi Fixed</code>	<code>fi double</code>
<code>fi single</code>	<code>fi Fixed</code>	<code>fi single</code>
Any <code>fi</code> data type	<code>fi double</code>	<code>fi double</code>
Any <code>fi</code> data type	<code>fi single</code>	<code>fi single</code>

## Examples

### Example 1

The following example shows the difference between the absolute value results for the most negative value representable by a signed data type when `OverflowAction` is `saturate` or `wrap`.

```
P = fipref('NumericTypeDisplay','full',...
          'FimathDisplay','full');
```

```
a = fi(-128)
a =
    -128
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 8
abs(a)
ans =
    127.9961
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 8
a.OverflowAction = 'Wrap'
a =
    -128
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 8
        RoundingMethod: Nearest
        OverflowAction: Wrap
        ProductMode: FullPrecision
        SumMode: FullPrecision
abs(a)
ans =
    -128
```



```

    DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
    FractionLength: 8

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

```

## Example 2

The following example shows the difference between the absolute value results for complex and real `fi` inputs that have the most negative value representable by a signed data type when `OverflowAction` is `wrap`.

```
re = fi(-1,1,16,15)
```

```
re =
```

```
-1
```

```

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

```

```
im = fi(0,1,16,15)
```

```
im =
```

```
0
```

```

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

```

```
a = complex(re,im)
```

```
a =
```

```
-1
```

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

abs(a, re.numericitytype, fimath('OverflowAction', 'Wrap'))

ans =

    1.0000

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

abs(re, re.numericitytype, fimath('OverflowAction', 'Wrap'))

ans =

    -1

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

### Example 3

The following example shows how to specify `numericitytype` and `fimath` objects as optional arguments to control the result of the `abs` function for real inputs. When you specify a `fimath` object as an argument, that `fimath` object is used to compute intermediate quantities, and the resulting `fi` object has no local `fimath`.

```
a = fi(-1,1,6,5, 'OverflowAction', 'Wrap')

a =

    -1

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 6
```

```
FractionLength: 5

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

abs(a)

ans =

-1

  DataTypeMode: Fixed-point: binary point scaling
  Signedness: Signed
  WordLength: 6
  FractionLength: 5

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

f = fimath('OverflowAction','Saturate')

f =

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

abs(a,f)

ans =

0.9688

  DataTypeMode: Fixed-point: binary point scaling
  Signedness: Signed
  WordLength: 6
  FractionLength: 5

t = numerictype(a.numerictype, 'Signed', false)
```

```
t =  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 6  
    FractionLength: 5  
  
abs(a,t,f)  
  
ans =  
  
    1  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 6  
    FractionLength: 5
```

## Example 4

The following example shows how to specify `numericType` and `fimath` objects as optional arguments to control the result of the `abs` function for complex inputs.

```
a = fi(-1-i,1,16,15,'OverflowAction','Wrap')  
  
a =  
  
-1.0000 - 1.0000i  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 15  
  
    RoundingMethod: Nearest  
    OverflowAction: Wrap  
    ProductMode: FullPrecision  
    SumMode: FullPrecision  
  
t = numericType(a.numericType,'Signed',false)
```

```
t =  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 16  
    FractionLength: 15  
  
abs(a,t)  
  
ans =  
  
    1.4142  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 16  
    FractionLength: 15  
  
    RoundingMethod: Nearest  
    OverflowAction: Wrap  
    ProductMode: FullPrecision  
    SumMode: FullPrecision  
  
f = fimath('OverflowAction','Saturate','SumMode',...  
    'KeepLSB','SumWordLength',a.WordLength,...  
    'ProductMode','specifyprecision',...  
    'ProductWordLength',a.WordLength,...  
    'ProductFractionLength',a.FractionLength)  
  
f =  
  
    RoundingMethod: Nearest  
    OverflowAction: Saturate  
    ProductMode: SpecifyPrecision  
    ProductWordLength: 16  
    ProductFractionLength: 15  
    SumMode: KeepLSB  
    SumWordLength: 16  
    CastBeforeSum: true  
  
abs(a,t,f)
```

```
ans =
```

```
1.4142
```

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

## Algorithms

The absolute value  $y$  of a real input  $a$  is defined as follows:

$$y = \begin{cases} a & \text{if } a \geq 0 \\ -a & \text{if } a < 0 \end{cases}$$

The absolute value  $y$  of a complex input  $a$  is related to its real and imaginary parts as follows:

$$y = \sqrt{\text{real}(a)^2 + \text{imag}(a)^2}$$

The `abs` function computes the absolute value of complex inputs as follows:

- 1 Calculate the real and imaginary parts of  $a$  using the following equations:

$$\begin{aligned} \text{re} &= \text{real}(a) \\ \text{im} &= \text{imag}(a) \end{aligned}$$

- 2 Compute the squares of `re` and `im` using one of the following objects:

- The `fimath` object  $F$  if  $F$  is specified as an argument.
- The `fimath` associated with  $a$  if  $F$  is not specified as an argument.

- 3 Cast the squares of `re` and `im` to unsigned types if the input is signed.

- 4 Add the squares of `re` and `im` using one of the following objects:

- The `fimath` object  $F$  if  $F$  is specified as an argument.
- The `fimath` object associated with  $a$  if  $F$  is not specified as an argument.

- 5 Compute the square root of the sum computed in step four using the `sqrt` function with the following additional arguments:
- The `numericType` object `T` if `T` is specified, or the `numericType` object of `a` otherwise.
  - The `fimath` object `F` if `F` is specified, or the `fimath` object associated with `a` otherwise.

---

**Note** Step three prevents the sum of the squares of the real and imaginary components from being negative. This is important because if either `re` or `im` has the maximum negative value and the `OverflowAction` property is set to `wrap` then an error will occur when taking the square root in step five.

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**Introduced before R2006a**

## accumneg

Subtract two `fi` objects or values

### Syntax

```
c = accumneg(a,b)
c = accumneg(a,b,RoundingMethod)
c = accumneg(a,b,RoundingMethod,OverflowAction)
```

### Description

`c = accumneg(a,b)` subtracts `b` from `a` using `a`'s data type. `b` is cast into `a`'s data type. If `a` is a `fi` object, the default 'Floor' rounding method and default 'Wrap' overflow action are used. The `fi`math properties of `a` and `b` are ignored.

`c = accumneg(a,b,RoundingMethod)` uses the rounding method specified in `RoundingMethod`.

`c = accumneg(a,b,RoundingMethod,OverflowAction)` uses the overflow action specified in `OverflowAction`.

### Input Arguments

#### **a**

Number from which to subtract. `a` can be `fi` object or double, single, logical, or integer value. The data type of `a` is used to compute the output data type.

#### **b**

Number to subtract. `b` can be `fi` object or double, single, logical, or integer value. .



### **RoundingMethod**

Rounding method to use if `a` is a `fi` object. Valid values are 'Ceiling', 'Convergent', 'Floor', 'Nearest', 'Round' and 'Zero'.

**Default:** Floor

### **OverflowAction**

Overflow action to take if `a` is a `fi` object. Valid values are 'Saturate' and 'Wrap',

**Default:** Wrap

## **Output Arguments**

`c`

Result of subtracting input `b` from input `a`.

## **Examples**

Subtract `fi` numbers using default `accumneg` settings and then, using non-default rounding method and overflow action.

```
a = fi(pi,1,16,13);  
b = fi(1.5,1,16,14);  
subtr_default = accumneg(a,b);  
subtr_custom = accumneg(a,b,'Nearest','Saturate');
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

### **See Also**

accumneg

### **Topics**

“Avoid Multiword Operations in Generated Code”

**Introduced in R2012a**

# accumpos

Add two `fi` objects or values

## Syntax

```
c = accumpos(a,b)
c = accumpos(a,b,RoundingMethod)
c = accumpos(a,b,RoundingMethod,OverflowAction)
```

## Description

`c = accumpos(a,b)` adds `a` and `b` using the `a`'s data type. `b` is cast into `a`'s data type. If `a` is a `fi` object, the default 'Floor' rounding method and default 'Wrap' overflow action are used. The `fi`math properties of `a` and `b` are ignored.

`c = accumpos(a,b,RoundingMethod)` uses the rounding method specified in `RoundingMethod`.

`c = accumpos(a,b,RoundingMethod,OverflowAction)` uses the overflow action specified in `OverflowAction`.

## Input Arguments

### **a**

Number to add. `a` can be `fi` object or double, single, logical, or integer value. The data type of `a` is used to compute the output data type.

### **b**

Number to add. `b` can be `fi` object or double, single, logical, or integer value.

**RoundingMethod**

Rounding method to use if `a` is a `fi` object. Valid values are 'Ceiling', 'Convergent', 'Floor', 'Nearest', 'Round', and 'Zero'.

**Default:** Floor

**OverflowAction**

Overflow action to take if `a` is a `fi` object. Valid values are 'Saturate' and 'Wrap'.

**Default:** Wrap

## Output Arguments

**c**

Result of adding the `a` and `b` inputs.

## Examples

Add two `fi` numbers using default `accumpos` settings and then, using nondefault rounding method and overflow action.

```
a = fi(pi,1,16,13);  
b = fi(1.5,1,16,14);  
add_default = accumpos(a,b);  
add_custom = accumpos(a,b,'Nearest','Saturate');
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## **GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

### **See Also**

accumneg

### **Topics**

“Avoid Multiword Operations in Generated Code”

**Introduced in R2012a**

## add

Add two objects using `fimath` object

## Syntax

```
c = add(F,a,b)
```

## Description

`c = add(F,a,b)` adds objects `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` has no local `fimath`.

`a` and `b` must both be `fi` objects and must have the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object.

## Examples

### Add Two Fixed-Point Numbers

In this example, `c` is the 32-bit sum of `a` and `b` with fraction length 16.

```
a = fi(pi);  
b = fi(exp(1));  
F = fimath('SumMode','SpecifyPrecision',...  
    'SumWordLength',32,'SumFractionLength',16);  
c = add(F,a,b)
```

```
c =  
    5.8599
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 32  
    FractionLength: 16
```

## Algorithms

`c = add(F, a, b)` is similar to

```
a.fimath = F;
b.fimath = F;
c = a + b
c =
```

5.8599

```
    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
```

but not identical. When you use `add`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` has no local `fimath`. When you use the syntax `c = a + b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See “`fimath` Rules for Fixed-Point Arithmetic” in the Fixed-Point Designer User's Guide for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The syntax `F.add(a,b)` is not supported. You must use the syntax `add(F,a,b)`.

### **See Also**

`divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide` | `sub` | `sum`

**Introduced before R2006a**



# assignmentquantizer

Assignment quantizer object of `fi` object

## Syntax

```
q = assignmentquantizer(a)
```

## Description

`q = assignmentquantizer(a)` returns the quantizer object `q` that is used in assignment operations for the `fi` object `a`.

## See Also

`quantize` | `quantizer`

**Introduced in R2008a**

## atan2

Four-quadrant inverse tangent of fixed-point values

### Syntax

```
z = atan2(y,x)
```

### Description

`z = atan2(y,x)` returns the four-quadrant arctangent on page 5-24 of fi input `y/x` using a table-lookup algorithm.

### Input Arguments

**y,x**

`y` and `x` can be real-valued, signed or unsigned scalars, vectors, matrices, or N-dimensional arrays containing fixed-point angle values in radians. The lengths of `y` and `x` must be the same. If they are not the same size, at least one input must be a scalar value. Valid data types of `y` and `x` are:

- fi single
- fi double
- fi fixed-point with binary point scaling
- fi scaled double with binary point scaling

### Output Arguments

**z**

`z` is the four-quadrant arctangent of `y/x`. The numeric type of `z` depends on the signedness of `y` and `x`:

- If either  $y$  or  $x$  is signed,  $z$  is a signed, fixed-point number in the range  $[-\pi, \pi]$ . It has a 16-bit word length and 13-bit fraction length (`numerictype(1, 16, 13)`).
- If both  $y$  and  $x$  are unsigned,  $z$  is an unsigned, fixed-point number in the range  $[0, \pi/2]$ . It has a 16-bit word length and 15-bit fraction length (`numerictype(0, 16, 15)`).

This arctangent calculation is accurate only to within the top 16 most-significant bits of the input.

## Examples

Calculate the arctangent of unsigned and signed fixed-point input values. The first example uses unsigned, 16-bit word length values. The second example uses signed, 16-bit word length values.

```
y = fi(0.125,0,16);
x = fi(0.5,0,16);
z = atan2(y,x)
```

z =

0.2450

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

```
y = fi(-0.1,1,16);
x = fi(-0.9,1,16);
z = atan2(y,x)
```

z =

-3.0309

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

## More About

### Four-Quadrant Arctangent

The four-quadrant arctangent is defined as follows, with respect to the atan function:

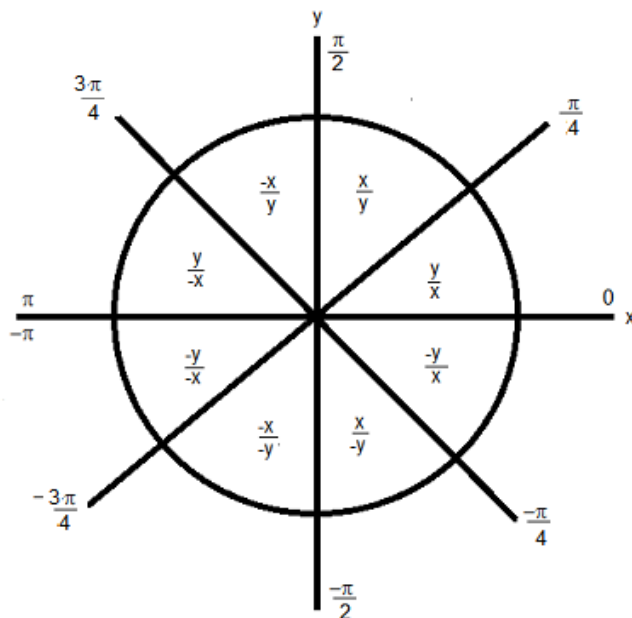
$$\text{atan2}(y, x) = \begin{cases} \text{atan}\left(\frac{y}{x}\right) & x > 0 \\ \pi + \text{atan}\left(\frac{y}{x}\right) & y \geq 0, x < 0 \\ -\pi + \text{atan}\left(\frac{y}{x}\right) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ 0 & y = 0, x = 0 \end{cases}$$

## Algorithms

The `atan2` function computes the four-quadrant arctangent of fixed-point inputs using an 8-bit lookup table as follows:

- 1 Divide the input absolute values to get an unsigned, fractional, fixed-point, 16-bit ratio between 0 and 1. The absolute values of `y` and `x` determine which value is the divisor.

The signs of the `y` and `x` inputs determine in what quadrant their ratio lies. The input with the larger absolute value is used as the denominator, thus producing a value between 0 and 1.



- 2 Compute the table index, based on the 16-bit, unsigned, stored integer value:
  - a Use the 8 most-significant bits to obtain the first value from the table.
  - b Use the next-greater table value as the second value.
- 3 Use the 8 least-significant bits to interpolate between the first and second values using nearest neighbor linear interpolation. This interpolation produces a value in the range  $[0, \pi/4)$ .
- 4 Perform octant correction on the resulting angle, based on the values of the original  $y$  and  $x$  inputs.

### fimath Propagation Rules

The `atan2` function ignores and discards any `fimath` attached to the inputs. The output,  $z$ , is always associated with the default `fimath`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

[angle](#) | [atan2](#) | [cordicatan2](#) | [cos](#) | [sin](#)

### **Topics**

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2012a**

# autofixexp

Automatically change scaling of fixed-point data types

## Syntax

```
autofixexp
```

## Description

The `autofixexp` script automatically changes the scaling for model objects that specify fixed-point data types. However, if an object's **Lock output data type setting against changes by the fixed-point tools** parameter is selected, the script refrains from scaling that object.

This script collects range data for model objects, either from design minimum and maximum values that objects specify explicitly, or from logged minimum and maximum values that occur during simulation. Based on these values, the tool changes the scaling of fixed-point data types in a model so as to maximize precision and cover the range.

You can specify design minimum and maximum values for model objects using parameters typically titled **Output minimum** and **Output maximum**. See “Blocks That Allow Signal Range Specification” (Simulink) for a list of Simulink blocks that permit you to specify these values. In the autoscaling procedure that the `autofixexp` script executes, design minimum and maximum values take precedence over the simulation range.

If you intend to scale fixed-point data types using simulation minimum and maximum values, the script yields meaningful results when exercising the full range of values over which your design is meant to run. Therefore, the simulation you run prior to using `autofixexp` must simulate your design over its full intended operating range. It is especially important that you use simulation inputs with appropriate speed and amplitude profiles for dynamic systems. The response of a linear dynamic system is frequency dependent. For example, a bandpass filter will show almost no response to very slow and very fast sinusoid inputs, whereas the signal of a sinusoid input with a frequency in the passband will be passed or even significantly amplified. The response of nonlinear dynamic systems can have complicated dependence on both the signal speed and amplitude.

---

**Note** If you already know the simulation range you need to cover, you can use an alternate autoscaling technique described in the `fixptbestprec` reference page.

---

To control the parameters associated with automatic scaling, such as safety margins, use the Fixed-Point Tool.

To learn how to use the Fixed-Point Tool, refer to “Propose Fraction Lengths Using Simulation Range Data”.

### **See Also**

`fxptdlg`

**Introduced before R2006a**



## bin

Binary representation of stored integer of `fi` object

## Syntax

`bin(a)`

## Description

`bin(a)` returns the stored integer of `fi` object `a` in unsigned binary format as a character vector. `bin(a)` is equivalent to `a.bin`.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

## Examples

The following code

```
a = fi([-1 1],1,8,7);
y = bin(a)
z = a.bin
```

returns

y =

```
10000000  01111111
```

z =

10000000 01111111

## **See Also**

dec | hex | oct | storedInteger

**Introduced before R2006a**

## bin2num

Convert two's complement binary string to number using quantizer object

### Syntax

```
y = bin2num(q,b)
```

### Description

`y = bin2num(q,b)` uses the properties of `quantizer` object `q` to convert the binary character vector `b` to a numeric array `y`. When `b` is a cell array containing binary representations, `y` is a cell array of the same dimension containing numeric arrays. The fixed-point binary representation is two's complement. The floating-point binary representation is in IEEE® Standard 754 style.

`bin2num` and `num2bin` are inverses of one another. Note that `num2bin` always returns the binary representations in a column.

### Examples

Create a `quantizer` object and an array of numeric character vectors. Convert the numeric character vectors to binary, then use `bin2num` to convert them back to numeric character vectors.

```
q=quantizer([4 3]);  
[a,b]=range(q);  
x=(b:-eps(q):a)';  
b = num2bin(q,x)
```

```
b =
```

```
0111  
0110  
0101  
0100
```

```
0011
0010
0001
0000
1111
1110
1101
1100
1011
1010
1001
1000
```

`bin2num` performs the inverse operation of `num2bin`.

```
y=bin2num(q,b)
```

```
y =
```

```
 0.8750
 0.7500
 0.6250
 0.5000
 0.3750
 0.2500
 0.1250
  0
-0.1250
-0.2500
-0.3750
-0.5000
-0.6250
-0.7500
-0.8750
-1.0000
```

### See Also

`hex2num` | `num2bin` | `num2hex` | `num2int`

**Introduced before R2006a**

## bitand

Bitwise AND of two `fi` objects

### Syntax

```
c = bitand(a, b)
```

### Description

`c = bitand(a, b)` returns the bitwise AND of `fi` objects `a` and `b`.

The `numericType` properties associated with `a` and `b` must be identical. If both inputs have a local `fiMath` object, the `fiMath` objects must be identical. If the `numericType` is signed, then the bit representation of the stored integer is in two's complement representation.

`a` and `b` must have the same dimensions unless one is a scalar.

`bitand` only supports `fi` objects with fixed-point data types.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

### See Also

`bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor`

**Introduced before R2006a**

# bitandreduce

Reduce consecutive slice of bits to one bit by performing bitwise AND operation

## Syntax

```
c = bitandreduce(a)
c = bitandreduce(a, lidx)
c = bitandreduce(a, lidx, ridx)
```

## Description

`c = bitandreduce(a)` performs a bitwise AND operation on the entire set of bits in the fixed-point input, `a`, and returns the result as an unsigned integer of word length 1.

`c = bitandreduce(a, lidx)` performs a bitwise AND operation on a consecutive range of bits, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitandreduce(a, lidx, ridx)` performs a bitwise AND operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitandreduce` arguments must satisfy the following condition:

$$a.\text{WordLength} \geq lidx \geq ridx \geq 1$$

## Examples

### Perform Bitwise AND Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise AND operation on the entire set of bits in `a`.

```
c = bitandreduce(a)
```

```
c =  
    0
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Unsigned  
        WordLength: 1  
        FractionLength: 0
```

Because the bits of `a` do not all have a value of 1, the output has a value of 0.

### **Perform Bitwise AND Operation on a Range of Bits in a Vector**

Create a fixed-point vector.

```
a = fi([12, 4, 8, 15],0,8,0);  
disp(bin(a))
```

```
00001100  00000100  00001000  00001111
```

Perform a bitwise AND operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c = bitandreduce(a, fi(4))
```

```
c=1x4 object
```

```
    0    0    0    1
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Unsigned  
        WordLength: 1  
        FractionLength: 0
```

The only element in output `c` with a value of 1 is the 4th element. This is because it is the only element of `a` that had only 1's between positions `fi(4)` and 1.



## Perform Bitwise AND Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7, 8, 1; 5, 9, 5; 8, 37, 2], 0, 8, 0);
disp(bin(a))
```

```
00000111    00001000    00000001
00000101    00001001    00000101
00001000    00100101    00000010
```

Perform a bitwise AND operation on the bits of each element of matrix `a` beginning at position 3 and ending at position 1.

```
c = bitandreduce(a, 3, 1)
```

```
c=3×3 object
```

```
 1    0    0
 0    0    0
 0    0    0
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 1
FractionLength: 0
```

There is only one element in output `c` with a value of 1. This condition occurs because the corresponding element in `a` is the only element with only 1's between positions 3 and 1.

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

`bitandreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitandreduce` performs the operation on a two's complement bit representation of the stored integer.

**Data Types:** fixed-point `fi`

### **lidx — Start position of range**

scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

**Data Types:** `fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

### **ridx — End position of range**

scalar

End position of range specified as a scalar of built-in type. `ridx` represents the position in the range closest to the LSB (the bit at position 1).

**Data Types:** `fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

## **Output Arguments**

### **c — Output array**

scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `c` is unsigned with word length 1.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`bitconcat` | `bitorreduce` | `bitsliceget` | `bitxorreduce`

**Introduced in R2007b**

## bitcmp

Bitwise complement of `fi` object

### Syntax

```
c = bitcmp(a)
```

### Description

`c = bitcmp(a)` returns the bitwise complement of `fi` object `a`. If `a` has a signed `numericType`, the bit representation of the stored integer is in two's complement representation.

`bitcmp` only supports `fi` objects with fixed-point data types. `a` can be a scalar `fi` object or a vector `fi` object.

### Examples

This example shows how to get the bitwise complement of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value of 10, word length 4, and fraction length 0:

```
a = fi(10,0,4,0);  
disp(bin(a))
```

```
1010
```

Complement the values of the bits in `a`:

```
c = bitcmp(a);  
disp(bin(c))
```

```
0101
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitand` | `bitget` | `bitor` | `bitset` | `bitxor`

**Introduced before R2006a**

## bitconcat

Concatenate bits of `fi` objects

### Syntax

```
y = bitconcat(a)
y = bitconcat (a, b, ...)
```

### Description

`y = bitconcat(a)` concatenates the bits of the elements of fixed-point `fi` input array, `a`.

`y = bitconcat (a, b, ...)` concatenates the bits of the fixed-point `fi` inputs.

### Examples

#### Concatenate the Elements of a Vector

Create a fixed-point vector.

```
a = fi([1,2,5,7],0,4,0);
disp(bin(a))

0001  0010  0101  0111
```

Concatenate the bits of the elements of `a`.

```
y = bitconcat(a)

y =
    4695
```

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

```

        WordLength: 16
        FractionLength: 0

```

```

disp(bin(y))
0001001001010111

```

The word length of the output, *y*, equals the sum of the word lengths of each element of *a*.

### Concatenate the Bits of Two fi Objects

Create two fixed-point numbers.

```

a = fi(5,0,4,0);
disp(bin(a))

```

```

0101

```

```

b = fi(10,0,4,0);
disp(bin(b))

```

```

1010

```

Concatenate the bits of the two inputs.

```

y = bitconcat(a,b)

```

```

y =
    90

```

```

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

```

```

disp(bin(y))
01011010

```

The output, *y*, is unsigned with a word length equal to the sum of the word lengths of the two inputs, and a fraction length of 0.

### Perform Element-by-Element Concatenation of Two Vectors

When `a` and `b` are both vectors of the same size, `bitconcat` performs element-wise concatenation of the two vectors and returns a vector.

Create two fixed-point vectors of the same size.

```
a = fi([1,2,5,7],0,4,0);  
disp(bin(a))
```

```
0001  0010  0101  0111
```

```
b = fi([7,4,3,1],0,4,0);  
disp(bin(b))
```

```
0111  0100  0011  0001
```

Concatenate the elements of `a` and `b`.

```
y = bitconcat(a,b)
```

```
y=1x4 object  
    23    36    83   113
```

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

```
disp(bin(y))
```

```
00010111  00100100  01010011  01110001
```

The output, `y`, is a vector of the same length as the input vectors, and with a word length equal to the sum of the word lengths of the two input vectors.

### Perform Element-by-Element Concatenation of Two Matrices

When the inputs are both matrices of the same size, `bitconcat` performs element-wise concatenation of the two matrices and returns a matrix of the same size.



Create two fixed-point matrices.

```
a = fi([1,2,5;7,4,5;3,1,12],0,4,0);
disp(bin(a))
```

```
0001  0010  0101
0111  0100  0101
0011  0001  1100
```

```
b = fi([6,1,7;7,8,1;9,7,8],0,4,0);
disp(bin(b))
```

```
0110  0001  0111
0111  1000  0001
1001  0111  1000
```

Perform element-by-element concatenation of the bits of a and b.

```
y = bitconcat(a,b)
```

```
y=3x3 object
```

```
    22    33    87
   119    72    81
    57    23   200
```

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

```
disp(bin(y))
```

```
00010110  00100001  01010111
01110111  01001000  01010001
00111001  00010111  11001000
```

The output, *y*, is a matrix with word length equal to the sum of the word lengths of *a* and *b*.

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `bitconcat` accepts `varargin` number of inputs for concatenation.

**Data Types:** fixed-point `fi`

### **b** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `b` is nonscalar, it must have the same dimension as the other inputs.

**Data Types:** fixed-point `fi`

## Output Arguments

### **y** — Output array

scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of unsigned fixed-point `fi` objects.

The output array has word length equal to the sum of the word lengths of the inputs and a fraction length of zero. The bit representation of the stored integer is in two's complement representation. Scaling does not affect the result type and value.

If the inputs are all scalar, then `bitconcat` concatenates the bits of the inputs and returns a scalar.

If the inputs are all arrays of the same size, then `bitconcat` performs element-wise concatenation of the bits and returns an array of the same size.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitreplicate` | `bitset` | `bitsliceget` | `bitxor`

**Introduced in R2007b**

## bitget

Get bits at certain positions

### Syntax

```
c = bitget(a, bit)
```

### Description

`c = bitget(a, bit)` returns the values of the bits at the positions specified by `bit` in `a` as unsigned integers of word length 1.

### Examples

#### Get Bit When Input and Index Are Both Scalar

Consider the following unsigned fixed-point `fi` number with a value of 85, word length 8, and fraction length 0:

```
a = fi(85,0,8,0);  
disp(bin(a))
```

```
01010101
```

Get the binary representation of the bit at position 4:

```
c = bitget(a,4);
```

`bitget` returns the bit at position 4 in the binary representation of `a`.

#### Get Bit When Input Is a Matrix and the Index Is a `fi`

Begin with a signed fixed-point 3-by-3 matrix with word length 4 and fraction length 0.

```
a = fi([2 3 4;6 8 2;3 5 1],0,4,0);
disp(bin(a))
```

```
0010  0011  0100
0110  1000  0010
0011  0101  0001
```

Get the binary representation of the bits at a specified position.

```
c = bitget(a,fi(2))
```

```
c=3x3 object
```

```
 1     1     0
 1     0     1
 1     0     0
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness:   Unsigned
      WordLength:   1
      FractionLength: 0
```

MATLAB® returns a matrix of the bits in position `fi(2)` of `a`. The output matrix has the same dimensions as `a`, and a word length of 1.

### Get Bit When Both Input and Index Are Vectors

Begin with a signed fixed-point vector with word length 16, fraction length 4.

```
a = fi([86 6 53 8 1],0,16,4);
disp(bin(a))
```

```
0000010101100000  0000000001100000  0000001101010000  0000000010000000  0000000000
```

Create a vector that specifies the positions of the bits to get.

```
bit = [1,2,5,7,4]
```

```
bit = 1x5
```

```
 1     2     5     7     4
```

Get the binary representation of the bits of `a` at the positions specified in `bit`.

```
c = bitget(a,bit)
```

```
c=1x5 object
```

```
0 0 1 0 0
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned  
WordLength: 1  
FractionLength: 0
```

`bitget` returns a vector of the bits of `a` at the positions specified in `bit`. The output vector has the same length as inputs, `a` and `bit`, and a word length of 1.

### Get Bit When Input Is Scalar and Index Is a Vector

Create a default `fi` object with a value of `pi`.

```
a = fi(pi);  
disp(bin(a))
```

```
0110010010001000
```

The default object is signed with a word length of 16.

Create a vector of the positions of the bits you want to get in `a`, and get the binary representation of those bits.

```
bit = fi([15,3,8,2]);  
c = bitget(a,bit)
```

```
c=1x4 object
```

```
1 0 1 0
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned  
WordLength: 1  
FractionLength: 0
```

MATLAB® returns a vector of the bits in `a` at the positions specified by the index vector, `bit`.

## Input Arguments

### `a` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `a` and `bit` are both nonscalar, they must have the same dimension. If `a` has a signed `numericType`, the bit representation of the stored integer is in two's complement representation.

**Data Types:** fixed-point `fi`

### `bit` — Bit index

scalar | vector | matrix | multidimensional array

Bit index, specified as a scalar, vector, matrix or multidimensional array of `fi` objects or built-in data types. If `a` and `bit` are both nonscalar, they must have the same dimension. `bit` must contain integer values between 1 and the word length of `a`, inclusive. The LSB (right-most bit) is specified by bit index 1 and the MSB (left-most bit) is specified by the word length of `a`. `bit` does not need to be a vector of sequential bit positions; it can also be a variable index value.

```
a = fi(pi,0,8);
a.bin
```

```
11001001
```

	MSB							LSB
bit index	8	7	6	5	4	3	2	1
value	1	1	0	0	1	0	0	1

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **c** — Output array

scalar | vector | matrix | multidimensional array

Output array, specified as an unsigned scalar, vector, matrix, or multidimensional array with `WordLength` 1.

If `a` is an array and `bit` is a scalar, `c` is an unsigned array with word length 1. This unsigned array comprises the values of the bits at position `bit` in each fixed-point element in `a`.

If `a` is a scalar and `bit` is an array, `c` is an unsigned array with word length 1. This unsigned array comprises the values of the bits in `a` at the positions specified in `bit`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitand` | `bitcmp` | `bitor` | `bitset` | `bitxor`

**Introduced before R2006a**



## bitor

Bitwise OR of two `fi` objects

### Syntax

```
c = bitor(a,b)
```

### Description

`c = bitor(a,b)` returns the bitwise OR of `fi` objects `a` and `b`. The output is determined as follows:

- Elements in the output array `c` are assigned a value of 1 when the corresponding bit in either input array has a value of 1.
- Elements in the output array `c` are assigned a value of 0 when the corresponding bit in both input arrays has a value of 0.

The `numericType` properties associated with `a` and `b` must be identical. If both inputs have a local `fiMath`, their local `fiMath` properties must be identical. If the `numericType` is signed, then the bit representation of the stored integer is in two's complement representation.

`a` and `b` must have the same dimensions unless one is a scalar.

`bitor` only supports `fi` objects with fixed-point data types.

### Examples

The following example finds the bitwise OR of `fi` objects `a` and `b`.

```
a = fi(-30,1,6,0);  
b = fi(12, 1, 6, 0);  
c = bitor(a,b)
```

```
c =
```

-18

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 6
FractionLength: 0
```

You can verify the result by examining the binary representations of  $a$ ,  $b$  and  $c$ .

```
binary_a = a.bin
binary_b = b.bin
binary_c = c.bin
```

```
binary_a =
```

```
100010
```

```
binary_b =
```

```
001100
```

```
binary_c =
```

```
101110
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

### See Also

`bitand` | `bitcmp` | `bitget` | `bitset` | `bitxor`

**Introduced before R2006a**

## bitorreduce

Reduce consecutive slice of bits to one bit by performing bitwise OR operation

### Syntax

```
c = bitorreduce(a)
c = bitorreduce(a, lidx)
c = bitorreduce(a, lidx, ridx)
```

### Description

`c = bitorreduce(a)` performs a bitwise OR operation on the entire set of bits in the fixed-point input, `a`, and returns the result as an unsigned integer of word length 1.

`c = bitorreduce(a, lidx)` performs a bitwise OR operation on a consecutive range of bits, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitorreduce(a, lidx, ridx)` performs a bitwise OR operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitorreduce` arguments must satisfy the following condition:

```
a.WordLength      >=      lidx      >=      ridx      >=      1
```

### Examples

#### Perform Bitwise OR Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise OR operation on the entire set of bits in `a`.

```
c = bitorreduce(a)
```

```
c =  
    1
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 1  
    FractionLength: 0
```

Because there is at least one bit in `a` with a value of 1, the output has a value of 1.

### Perform Bitwise OR Operation on a Range of Bits in a Vector

Create a fixed-point vector.

```
a=fi([12,4,8,15],0,8,0);  
disp(bin(a))
```

```
00001100    00000100    00001000    00001111
```

Perform a bitwise OR operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c=bitorreduce(a,fi(4))
```

```
c=1x4 object  
    1    1    1    1
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 1  
    FractionLength: 0
```

All of the entries of output `c` have a value of 1 because all of the entries of `a` have at least one bit with a value of 1 between the positions `fi(4)` and 1.

## Perform Bitwise OR Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7,8,1;5,9,5;8,37,2],0,8,0);  
disp(bin(a))
```

```
00000111    00001000    00000001  
00000101    00001001    00000101  
00001000    00100101    00000010
```

Perform a bitwise OR operation on the bits of each element of matrix `a` beginning at position 5, and ending at position 2.

```
c = bitorreduce(a,5,2)
```

```
c=3×3 object
```

```
    1    1    0  
    1    1    1  
    1    1    1
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 1  
    FractionLength: 0
```

There is only one element in output `c` that does not have a value of 1. This condition occurs because the corresponding element in `a` is the only element of `a` that does not have any bits with a value of 1 between positions 5 and 2.

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects.

`bitorreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitorreduce` performs the operation on a two's complement bit representation of the stored integer.

**Data Types:** fixed-point `fi`

**lidx — Start position of range**

scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ridx — End position of range**

scalar

End position of range specified as a scalar of built-in type. `ridx` represents the position in the range closest to the LSB (the bit at position 1).

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

**c — Output array**

scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `c` is unsigned with word length 1.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitandreduce` | `bitconcat` | `bitsliceget` | `bitxorreduce`

**Introduced in R2007b**



# bitreplicate

Replicate and concatenate bits of `fi` object

## Syntax

```
c = bitreplicate(a,n)
```

## Description

`c = bitreplicate(a,n)` concatenates the bits in `fi` object `a` `n` times and returns an unsigned fixed-point value. The word length of the output `fi` object `c` is equal to `n` times the word length of `a` and the fraction length of `c` is zero. The bit representation of the stored integer is in two's complement representation.

The input `fi` object can be signed or unsigned. `bitreplicate` concatenates signed and unsigned bits the same way.

`bitreplicate` only supports `fi` objects with fixed-point data types.

`bitreplicate` does not support inputs with complex data types.

Sign and scaling of the input `fi` object does not affect the result type and value.

## Examples

The following example uses `bitreplicate` to replicate and concatenate the bits of `fi` object `a`.

```
a = fi(14,0,6,0);  
a_binary = a.bin  
c = bitreplicate(a,2);  
c_binary = c.bin
```

MATLAB returns the following:

```
a_binary =
```

```
001110
```

```
c_binary =
```

```
001110001110
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[bitand](#) | [bitconcat](#) | [bitget](#) | [bitor](#) | [bitset](#) | [bitsliceget](#) | [bitxor](#)

**Introduced in R2008a**

## bitrol

Bitwise rotate left

### Syntax

```
c = bitrol(a, k)
```

### Description

`c = bitrol(a, k)` returns the value of the fixed-point `fi` object, `a`, rotated left by `k` bits. `bitrol` rotates bits from the most significant bit (MSB) side into the least significant bit (LSB) side. It performs the rotate left operation on the stored integer bits of `a`.

`bitrol` does not check overflow or underflow. It ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

`a` and `c` have the same `fimath` and `numericType` properties.

### Examples

#### Rotate the Bits of a `fi` Object Left

Create an unsigned fixed-point `fi` object with a value of 10, word length 4, and fraction length 0.

```
a = fi(10,0,4,0);  
disp(bin(a))
```

```
1010
```

Rotate `a` left 1 bit.

```
disp(bin(bitrol(a,1)))
```

```
0101
```

Rotate a left 2 bits.

```
disp(bin(bitrol(a,2)))  
1010
```

### **Rotate Bits in a Vector Left**

Create a vector of `fi` objects.

```
a = fi([1,2,5,7],0,4,0)
```

```
a=1x4 object
```

```
    1     2     5     7
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 4  
    FractionLength: 0
```

```
disp(bin(a))
```

```
0001  0010  0101  0111
```

Rotate the bits in vector `a` left 1 bit.

```
disp(bin(bitrol(a,1)))
```

```
0010  0100  1010  1110
```

### **Rotate Bits Left Using `fi` to Specify Number of Bits to Rotate**

Create an unsigned fixed-point `fi` object with a value 10, word length 4, and fraction length 0.

```
a = fi(10,0,4,0);
```

```
disp(bin(a))
```

```
1010
```

Rotate `a` left 1 bit where `k` is a `fi` object.

```
disp(bin(bitrol(a,fi(1))))
```

```
0101
```

## Input Arguments

### **a** — Data that you want to rotate

scalar | vector | matrix | multidimensional array

Data that you want to rotate, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects. `a` can be signed or unsigned.

**Data Types:** fixed-point `fi`

**Complex Number Support:** Yes

### **k** — Number of bits to rotate

non-negative, integer-valued scalar

Number of bits to rotate, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type. `k` can be greater than the word length of `a`. This value is always normalized to `mod(a.WordLength,k)`.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitconcat` | `bitror` | `bitshift` | `bitsliceget` | `bitsll` | `bitsra` | `bitsrl`

**Introduced in R2007b**

## bitror

Bitwise rotate right

### Syntax

```
c = bitror(a, k)
```

### Description

`c = bitror(a, k)` returns the value of the fixed-point `fi` object, `a`, rotated right by `k` bits. `bitror` rotates bits from the least significant bit (LSB) side into the most significant bit (MSB) side. It performs the rotate right operation on the stored integer bits of `a`.

`bitror` does not check overflow or underflow. It ignores `fi`math properties such as `RoundingMode` and `OverflowAction`.

`a` and `c` have the same `fi`math and `numericType` properties.

### Examples

#### Rotate Bits of a `fi` Object Right

Create an unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);  
disp(bin(a))
```

```
0101
```

Rotate `a` right 1 bit.

```
disp(bin(bitror(a,1)))
```

```
1010
```

Rotate a right 2 bits.

```
disp(bin(bitror(a,2)))
```

```
0101
```

### **Rotate Bits in a Vector Right**

Create a vector of `fi` objects.

```
a = fi([1,2,5,7],0,4,0);  
disp(bin(a))
```

```
0001  0010  0101  0111
```

Rotate the bits in vector `a` right 1 bit.

```
disp(bin(bitror(a,fi(1))))
```

```
1000  0001  1010  1011
```

### **Rotate Bits Right Using `fi` to Specify Number of Bits to Rotate**

Create an unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);  
disp(bin(a))
```

```
0101
```

Rotate `a` right 1 bit where `k` is a `fi` object.

```
disp(bin(bitror(a,fi(1))))
```

```
1010
```



## Input Arguments

### **a** — Data that you want to rotate

scalar | vector | matrix | multidimensional array

Data that you want to rotate, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects. `a` can be signed or unsigned.

**Data Types:** fixed-point `fi`

**Complex Number Support:** Yes

### **k** — Number of bits to rotate

non-negative, integer-valued scalar

Number of bits to rotate, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type. `k` can be greater than the word length of `a`. This value is always normalized to `mod(a.WordLength, k)`.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitconcat` | `bitrol` | `bitshift` | `bitsliceget` | `bitsll` | `bitsra` | `bitsrl`

**Introduced in R2007b**

# bitset

Set bits at certain positions

## Syntax

```
c = bitset(a, bit)
c = bitset(a, bit, v)
```

## Description

`c = bitset(a, bit)` returns the value of `a` with position `bit` set to 1 (on).

`c = bitset(a, bit, v)` returns the value of `a` with position `bit` set to `v`.

## Examples

### Set the Bit at a Certain Position

Begin with an unsigned fixed-point `fi` number with a value of 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Set the bit at position 4 to 1 (on).

```
c = bitset(a,4);
disp(bin(c))
```

```
1101
```

### Set the Bit at a Certain Position in a Vector

Consider the following fixed-point vector with word length 4 and fraction length 0.

```
a = fi([0 1 8 2 4],0,4,0);
disp(bin(a))
```

```
0000    0001    1000    0010    0100
```

In each element of vector `a`, set the bits at position 2 to 1.

```
c = bitset(a,2,1);
disp(bin(c))
```

```
0010    0011    1010    0010    0110
```

### Set the Bit at a Certain Position with Fixed Point Index

Consider the following fixed-point scalar with a value of 5.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Set the bit at position `fi(2)` to 1.

```
c = bitset(a,fi(2),1);
disp(bin(c))
```

```
0111
```

### Set the Bit When Index Is a Vector

Create a `fi` object with a value of `pi`.

```
a = fi(pi);
disp(bin(a))
```

```
0110010010001000
```

In this case, `a` is signed with a word length of 16.

Create a vector of the bit positions in `a` that you want to set to on. Then, get the binary representation of the resulting `fi` vector.

```
bit = fi([15,3,8,2]);  
c = bitset(a,bit);  
disp(bin(c))
```

```
0110010010001000   0110010010001100   0110010010001000   0110010010001010
```

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `a` has a signed `numericType`, the bit representation of the stored integer is in two's complement representation.

**Data Types:** fixed-point `fi`

### **bit** — Bit index

scalar | vector | matrix | multidimensional array

Bit index, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. `bit` must be a number between 1 and the word length of `a`, inclusive. The LSB (right-most bit) is specified by bit index 1 and the MSB (left-most bit) is specified by the word length of `a`.

```
a = fi(pi,0,8);  
a.bin
```

```
11001001
```

	MSB							LSB
bit index	8	7	6	5	4	3	2	1
value	1	1	0	0	1	0	0	1

**Data Types:** `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **v — Bit value**

scalar | vector | matrix | multidimensional array

Bit value of `a` at index `bit`, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. `v` can have values of `0`, or `1`. Any value other than `0` is automatically set to `1`. When `v` is nonscalar, it must have the same dimensions as one of the other inputs.

**Data Types:** `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **c — Output array**

scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitxor`

**Introduced before R2006a**

# bitshift

Shift bits specified number of places

## Syntax

```
c = bitshift(a, k)
```

## Description

`c = bitshift(a, k)` returns the value of `a` shifted by `k` bits. The input `fi` object `a` may be a scalar value or a vector and can be any fixed-point numeric type. The output `fi` object `c` has the same numeric type as `a`. `k` must be a scalar value and a MATLAB built-in numeric type.

The `OverflowAction` property of `a` is obeyed, but the `RoundingMethod` is always `Floor`. If obeying the `RoundingMethod` property of `a` is important, try using the `pow2` function.

When the overflow action is `Saturate` the sign bit is always preserved. The sign bit is also preserved when the overflow action is `Wrap`, and `k` is negative. When the overflow action is `Wrap` and `k` is positive, the sign bit is not preserved.

- When `k` is positive, 0-valued bits are shifted in on the right.
- When `k` is negative, and `a` is unsigned, or a signed and positive `fi` object, 0-valued bits are shifted in on the left.
- When `k` is negative and `a` is a signed and negative `fi` object, 1-valued bits are shifted in on the left.

## Examples

This example highlights how changing the `OverflowAction` property of the `fimath` object can change the results returned by the `bitshift` function. Consider the following signed fixed-point `fi` object with a value of 3, word length 16, and fraction length 0:

```
a = fi(3,1,16,0);
```

By default, the `OverflowAction` `fimath` property is `Saturate`. When `a` is shifted such that it overflows, it is saturated to the maximum possible value:

```
for k=0:16,b=bitshift(a,k);...  
disp([num2str(k,'%02d'),' ' ,bin(b)]);end
```

```
00. 0000000000000011  
01. 0000000000000110  
02. 0000000000001100  
03. 0000000000011000  
04. 0000000000110000  
05. 0000000001100000  
06. 0000000011000000  
07. 0000000110000000  
08. 0000001100000000  
09. 0000011000000000  
10. 0000110000000000  
11. 0001100000000000  
12. 0011000000000000  
13. 0110000000000000  
14. 0111111111111111  
15. 0111111111111111  
16. 0111111111111111
```

Now change `OverflowAction` to `Wrap`. In this case, most significant bits shift off the “top” of `a` until the value is zero:

```
a = fi(3,1,16,0,'OverflowAction','Wrap');  
for k=0:16,b=bitshift(a,k);...  
disp([num2str(k,'%02d'),' ' ,bin(b)]);end
```

```
00. 0000000000000011  
01. 0000000000000110  
02. 0000000000001100  
03. 0000000000011000  
04. 0000000000110000  
05. 0000000001100000  
06. 0000000011000000  
07. 0000000110000000  
08. 0000001100000000  
09. 0000011000000000  
10. 0000110000000000  
11. 0001100000000000
```



```
12. 0011000000000000
13. 0110000000000000
14. 1100000000000000
15. 1000000000000000
16. 0000000000000000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitsll` | `bitsra` | `bitsrl` | `bitxor` | `pow2`

**Introduced before R2006a**

## bitsliceget

Get consecutive slice of bits

### Syntax

```
c = bitsliceget(a)
c = bitsliceget(a, lidx)
c = bitsliceget(a, lidx, ridx)
```

### Description

`c = bitsliceget(a)` returns the entire set of bits in the fixed-point input `a`.

`c = bitsliceget(a, lidx)` returns a consecutive slice of bits from `a`, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitsliceget(a, lidx, ridx)` returns a consecutive slice of bits from `a`, starting at position `lidx` and ending at position `ridx`.

The `bitsliceget` arguments must satisfy the following condition:

$$a.\text{WordLength} \geq lidx \geq ridx \geq 1$$

### Examples

#### Get Entire Set of Bits

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);
disp(bin(a))
```

```
01010101
```

Get the entire set of bits of `a`.

```
c = bitsliceget(a);  
disp(bin(c))
```

```
01010101
```

### **Get a Slice of Consecutive Bits with Unspecified Endpoint**

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);  
disp(bin(a))
```

```
01010101
```

Get the binary representation of the consecutive bits, starting at position 6.

```
c = bitsliceget(a,6);  
disp(bin(c))
```

```
010101
```

### **Get a Slice of Consecutive Bits with Fixed-Point Indexes**

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);  
disp(bin(a))
```

```
01010101
```

Get the binary representation of the consecutive bits from `fi(6)` to `fi(2)`.

```
c = bitsliceget(a,fi(6),fi(2));  
disp(bin(c))
```

```
01010
```

### Get a Specified Set of Consecutive Bits from Each Element of a Matrix

Begin with the following unsigned fixed-point 3-by-3 matrix.

```
a = fi([2 3 4;6 8 2;3 5 1],0,4,0);  
disp(bin(a))
```

```
0010    0011    0100  
0110    1000    0010  
0011    0101    0001
```

Get the binary representation of a consecutive set of bits of matrix `a`. For each element, start at position 4 and end at position 2.

```
c = bitsliceget(a,4,2);  
disp(bin(c))
```

```
001    001    010  
011    100    001  
001    010    000
```

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `a` has a signed `numericType`, the bit representation of the stored integer is in two's complement representation.

**Data Types:** fixed-point `fi`

### **lidx** — Start position for slice

scalar

Start position of slice specified as a scalar of built-in type. `lidx` represents the position in the slice closest to the MSB.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ridx — End position for slice**

scalar

End position of slice specified as a scalar of built-in type. `ridx` represents the position in the slice closest to the LSB (the bit at position 1).

**Data Types:** `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

**c — Output array**

scalar | vector | matrix | multidimensional array

Fixed-point `fi` output, specified as a scalar, vector, matrix, or multidimensional array with no scaling. The word length is equal to slice length, `lidx - ridx + 1`.

If `lidx` and `ridx` are equal, `bitsliceget` only slices one bit, and `bitsliceget(a, lidx, ridx)` is the same as `bitget(a, lidx)`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor`

**Introduced in R2007b**

## bitsll

Bit shift left logical

### Syntax

```
c = bitsll(a, k)
```

### Description

`c = bitsll(a, k)` returns the result of a logical left shift by `k` bits on input `a` for fixed-point operations. `bitsll` shifts zeros into the positions of bits that it shifts left. The function does not check overflow or underflow. For floating-point operations, `bitsll` performs a multiply by  $2^k$ .

`bitsll` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numericType` objects.

### Examples

#### Shift Left a Signed fi Input

Shift a signed `fi` input left by 1 bit.

Create a `fi` object, and display its binary value.

```
a = fi(10,0,4,0);  
disp(bin(a))
```

```
1010
```

Shift `a` left by 1 bit, and display its binary value.

```
disp(bin(bitsll(a,1)))
```

```
0100
```

Shift `a` left by 1 more bit.

```
disp(bin(bitsll(a,2)))
```

```
1000
```

### Shift Left Using a fi Shift Value

Shift left a built-in `int8` input using a `fi` shift value.

```
k = fi(2);  
a = int8(16);  
bitsll(a,k)
```

```
ans = int8  
    64
```

### Shift Left a Built-in int8 Input

Use `bitsll` to shift an `int8` input left by 2 bits.

```
a = int8(4);  
bitsll(a,2)
```

```
ans = int8  
    16
```

### Shift Left a Floating-Point Input

Scale a floating-point `double` input by  $2^3$ .

```
a = double(16);  
bitsll(a,3)
```

```
ans = 128
```

## Input Arguments

### **a** — Data that you want to shift

scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in numeric types.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

### **k** — Number of bits to shift

non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

### **GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:



- Generated code might not handle out of range shifting.

## **See Also**

`bitconcat` | `bitrol` | `bitror` | `bitshift` | `bitsra` | `bitsrl` | `pow2`

**Introduced in R2007b**

## bitsra

Bit shift right arithmetic

### Syntax

```
c=bitsra(a,k)
```

### Description

`c=bitsra(a,k)` returns the result of an arithmetic right shift by `k` bits on input `a` for fixed-point operations. For floating-point operations, it performs a multiply by  $2^{-k}$ .

If the input is unsigned, `bitsra` shifts zeros into the positions of bits that it shifts right. If the input is signed, `bitsra` shifts the most significant bit (MSB) into the positions of bits that it shifts right.

`bitsra` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numericType` objects.

### Examples

#### Shift Right a Signed fi Input

Create a signed fixed-point `fi` object with a value of `-8`, word length `4`, and fraction length `0`. Then display the binary value of the object.

```
a = fi(-8,1,4,0);  
disp(bin(a))
```

```
1000
```

Shift `a` right by 1 bit.

```
disp(bin(bitsra(a,1)))
```

```
1100
```

`bitsra` shifts the MSB into the position of the bit that it shifts right.

### Shift Right a Built-in int8 Input

Use `bitsra` to shift an `int8` input right by 2 bits.

```
a = int8(64);  
bitsra(a,2)
```

```
ans = int8  
    16
```

### Shift Right Using a fi Shift Value

Shift right a built-in `int8` input using a `fi` shift value.

```
k = fi(2);  
a = int8(64);  
bitsra(a,k)
```

```
ans = int8  
    16
```

### Shift Right a Floating-Point Input

Scale a floating-point `double` input by  $2^{-3}$ .

```
a = double(128);  
bitsra(a,3)
```

```
ans = 16
```

## Input Arguments

### **a** — Data that you want to shift

scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in numeric types.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

### **k** — Number of bits to shift

non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

## **See Also**

[bitshift](#) | [bitsll](#) | [bitsrl](#) | [pow2](#)

**Introduced in R2007b**

## bitsrl

Bit shift right logical

### Syntax

```
c = bitsrl(a, k)
```

### Description

`c = bitsrl(a, k)` returns the result of a logical right shift by `k` bits on input `a` for fixed-point operations. `bitsrl` shifts zeros into the positions of bits that it shifts right. It does not check overflow or underflow.

`bitsrl` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numericType` objects.

### Examples

#### Shift right a signed `fi` input

Shift a signed `fi` input right by 1 bit.

Create a signed fixed-point `fi` object with a value of -8, word length 4, and fraction length 0 and display its binary value.

```
a = fi(-8,1,4,0);  
disp(bin(a))
```

```
1000
```

Shift `a` right by 1 bit, and display the binary value.

```
disp(bin(bitsrl(a,1)))
```

```
0100
```

`bitsrl` shifts a zero into the position of the bit that it shifts right.

### Shift right using a fi shift value

Shift right a built-in `int8` input using a `fi` shift value.

```
k = fi(2);  
a = int8(64);  
bitsrl(a,k)
```

```
ans = int8  
    16
```

### Shift right a built-in uint8 input

Use `bitsrl` to shift a `uint8` input right by 2 bits.

```
a = uint8(64);  
bitsrl(a,2)
```

```
ans = uint8  
    16
```

## Input Arguments

### **a** — Data that you want to shift

scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array.

**Data Types:** `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

**k — Number of bits to shift**

non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar.

**Data Types:** fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

### See Also

bitconcat | bitrol | bitror | bitshift | bitsliceget | bitsll | bitsra | pow2

**Introduced in R2007b**



## bitxor

Bitwise exclusive OR of two `fi` objects

### Syntax

```
c = bitxor(a,b)
```

### Description

`c = bitxor(a,b)` returns the bitwise exclusive OR of `fi` objects `a` and `b`. The output is determined as follows:

- Elements in the output array `c` are assigned a value of 1 when exactly one of the corresponding bits in the input arrays has a value of 1.
- Elements in the output array `c` are assigned a value of 0 when the corresponding bits in the input arrays have the same value (e.g. both 1's or both 0's).

The `numericType` properties associated with `a` and `b` must be identical. If both inputs have a local `fiMath`, their local `fiMath` properties must be identical. If the `numericType` is signed, then the bit representation of the stored integer is in two's complement representation.

`a` and `b` must have the same dimensions unless one is a scalar.

`bitxor` only supports `fi` objects with fixed-point data types.

### Examples

The following example finds the bitwise exclusive OR of `fi` objects `a` and `b`.

```
a = fi(-28,1,6,0);  
b = fi(12, 1, 6, 0);  
c = bitxor(a,b)
```

```
c =
```

-24

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 6
FractionLength: 0
```

You can verify the result by examining the binary representations of  $a$ ,  $b$  and  $c$ .

```
binary_a = a.bin
binary_b = b.bin
binary_c = c.bin
```

```
binary_a =
```

```
100100
```

```
binary_b =
```

```
001100
```

```
binary_c =
```

```
101000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

### See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset`

**Introduced before R2006a**

## bitxorreduce

Reduce consecutive slice of bits to one bit by performing bitwise exclusive OR operation

### Syntax

```
c = bitxorreduce(a)
c = bitxorreduce(a, lidx)
c = bitxorreduce(a, lidx, ridx)
```

### Description

`c = bitxorreduce(a)` performs a bitwise exclusive OR operation on the entire set of bits in the fixed-point input, `a`. It returns the result as an unsigned integer of word length 1.

`c = bitxorreduce(a, lidx)` performs a bitwise exclusive OR operation on a consecutive range of bits. This operation starts at position `lidx` and ends at the LSB (the bit at position 1).

`c = bitxorreduce(a, lidx, ridx)` performs a bitwise exclusive OR operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitxorreduce` arguments must satisfy the following condition:

$$a.\text{WordLength} \quad \geq \quad \text{lidx} \quad \geq \quad \text{ridx} \quad \geq \quad 1$$

### Examples

#### Perform Bitwise Exclusive OR Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise exclusive OR operation on the entire set of bits in `a`.

```
c = bitxorreduce(a)
```

```
c =
```

```
1
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness:   Unsigned
    WordLength:   1
    FractionLength: 0
```

### Perform Bitwise Exclusive OR Operation on a Range of Bits in a Vector

Create a fixed-point vector.

```
a = fi([12,4,8,15],0,8,0);
disp(bin(a))
```

```
00001100  00000100  00001000  00001111
```

Perform a bitwise exclusive OR operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c = bitxorreduce(a,fi(4))
```

```
c=1x4 object
```

```
0  1  1  0
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness:   Unsigned
    WordLength:   1
    FractionLength: 0
```

### Perform a Bitwise Exclusive OR Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7,8,1;5,9,5;8,37,2],0,8,0);  
disp(bin(a))
```

```
00000111    00001000    00000001  
00000101    00001001    00000101  
00001000    00100101    00000010
```

Perform a bitwise exclusive OR operation on the bits of each element of matrix `a` beginning at position 5 and ending at position 2.

```
c = bitxorreduce(a,5,2)
```

```
c=3×3 object
```

```
    0     1     0  
    1     1     1  
    1     1     1
```

```
        DataTypeMode: Fixed-point: binary point scaling  
          Signedness: Unsigned  
         WordLength: 1  
    FractionLength: 0
```

## Input Arguments

### **a** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects.

`bitxorreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitxorreduce` performs the operation on a two's complement bit representation of the stored integer.

**Data Types:** fixed-point `fi`

### **lidx** — Start position of range

scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

**Data Types:** fi | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ridx — End position of range**

scalar

End position of range specified as a scalar of built-in type. ridx represents the position in the range closest to the LSB (the bit at position 1).

**Data Types:** fi | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

**c — Output array**

scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point fi objects. c is unsigned with word length 1.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

bitandreduce | bitconcat | bitorreduce | bitsliceget

**Introduced in R2007b**

## buildInstrumentedMex

Generate compiled C code function including logging instrumentation

### Syntax

```
buildInstrumentedMex fcn -options  
buildInstrumentedMex fcn_1... fcn_n -options -coder
```

### Description

`buildInstrumentedMex fcn -options` translates the MATLAB file `fcn.m` to a MEX function and enables instrumentation for logging minimum and maximum values of all named and intermediate variables. Optionally, you can enable instrumentation for log2 histograms of all named, intermediate and expression values. The general syntax and options of `buildInstrumentedMex` and `fiaccel` are the same, except `buildInstrumentedMex` has no `fi` object restrictions and supports the `'-coder'` option.

`buildInstrumentedMex fcn_1... fcn_n -options -coder` translates the MATLAB functions `fcn_1` through `fcn_n` to a MEX function and enables instrumentation for logging minimum and maximum values of all named and intermediate variables. Generating a MEX function for multiple entry-point functions requires the `'-coder'` option.

### Input Arguments

**fcn, fcn\_1... fcn\_n**

MATLAB entry-point functions to be instrumented. The entry-point functions must be suitable for code generation. For more information, see “Make the MATLAB Code Suitable for Code Generation” (MATLAB Coder).



## options

Choice of compiler options. `buildInstrumentedMex` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

- `-args example_inputs` Define the size, class, and complexity of all MATLAB function inputs. Use the values in *example\_inputs* to define these properties. *example\_inputs* must be a cell array that specifies the same number and order of inputs as the MATLAB function.
- `-coder` Use MATLAB Coder software to compile the MEX file, instead of the default Fixed-Point Designer `fiaccel` function. This option removes `fiaccel` restrictions and allows for full code generation support. You must have a MATLAB Coder license to use this option.
- `-config config_object` Specify MEX generation parameters, based on *config\_object*, defined as a MATLAB variable using `coder.mexconfig`. For example:  

```
cfg = coder.mexconfig;
```

`-d out_folder`

Store generated files in the absolute or relative path specified by *out\_folder*. If the folder specified by *out\_folder* does not exist, `buildInstrumentedMex` creates it for you.

If you do not specify the folder location, `buildInstrumentedMex` generates files in the default folder:

*fiaccel/mex/fcn*.

*fcn* is the name of the MATLAB function specified at the command line.

The function does not support the following characters in folder names: asterisk (\*), question-mark (?), dollar (\$), and pound (#).

`-g`

Compiles the MEX function in debug mode, with optimization turned off. If not specified, `buildinstrumentedMex` generates the MEX function in optimized mode.

- 
- `-global global_values` Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `buildInstrumentedMex`. If you do not provide initial values for global variables using the `-global` option, `buildInstrumentedMex` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `buildInstrumentedMex` generates an error.
- `-histogram` The generated MEX 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. Compute the log2 histogram for all named, intermediate and expression values. A histogram column appears in the code generation report table.
- `-I include_path` Add *include\_path* to the beginning of the code generation path. `buildInstrumentedMex` searches the code generation path *first* when converting MATLAB code to MEX code.
- `-launchreport` Generate and open a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-report` option.

- `-o output_file_name` Generate the MEX function with the base name *output\_file\_name* plus a platform-specific extension.
- output\_file\_name* can be a file name or include an existing path.
- If you do not specify an output file name, the base name is *fcn\_mex*, which allows you to run the original MATLAB function and the MEX function and compare the results.
- `-O optimization_option` Optimize generated MEX code, based on the value of *optimization\_option*:
- `enable:inline` — Enable function inlining
  - `disable:inline` — Disable function inlining
- If not specified, `buildInstrumentedMex` uses inlining for optimization.
- `-report` Generate a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-launchreport` option.

## Examples

### Create an Instrumented MEX Function

Create an instrumented MEX function. Run a test bench, then view logged results.

- 1 Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('buildInstrumentedMex')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
    'fidemos','fi_m_radix2fft_withscaling.m'),...
    'testfft.m','f')
```

- 2 Define prototype input arguments.

```
n = 128;
x = complex(zeros(n,1));
W = coder.Constant(fidemo.fi_radix2twiddles(n));
```

- 3 Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name. Use the `-histogram` option to compute histograms. (If you have a MATLAB Coder license, you may want to also add the `-coder` option. In this case, use `buildInstrumentedMex testfft -coder -o testfft_instrumented -args {x,W}` instead of the following line of code.)

---

**Note** Like `fiaccel`, `buildInstrumentedMex` generates a MEX function. To generate C code, see the MATLAB Coder `codegen` function.

---

```
buildInstrumentedMex testfft -o testfft_instrumented...
-args {x,W} -histogram
```

- 4 Run a test file to record instrumentation results. Call `showInstrumentationResults` to open the report. View the simulation minimum and maximum values and whole number status by pausing over a variable in the report. You can also see proposed data types for double precision numbers in the table.

```
for i=1:20
    y = testfft_instrumented(randn(size(x)));
end
```

```
showInstrumentationResults testfft_instrumented
```

## 5 Functions — Alphabetical List

The screenshot shows the MATLAB IDE with a function editor open for `testfft.m`. The code defines a function `fi_m_radix2fft_withscaling` and includes comments, a reference, and a copyright notice. A tooltip is displayed over the code, showing properties for a variable `LL2`.

**Function Editor Code:**


```
1 function x = fi_m_radix2fft_withscaling(X, W)
2 %FI_M_RADIX2FFT_WITHSCALING Radix-2 FFT example with scaling at each stage.
3 % Y = FI_M_RADIX2FFT_WITHSCALING(X, W) computes the radix-2 FFT of
4 % input vector X with twiddle-factors W with scaling by 1/2 at each stage.
5 % Input X is assumed to be complex.
6 %
7 % The length of vector X must be an exact power of two.
8 % Twiddle-factors W are computed via
9 % W = fidemo.fi_radix2twiddles(N)
10 % where N = length(X).
11 %
12 % This version of the algorithm has no scaling before the stages.
13 %
14 % See also FI_RADIX2FFT_DEMO.
15
16 % Reference:
17 % Charles Van Loan, Computational Frameworks for the Fast Fourier
18 % Transform, SIAM, Philadelphia, 1992, Algorithm 1.6.2, p. 45.
19 %
20 % Copyright 2004-2015 The MathWorks, Inc.
21 %
22 %codegen
23
24 n = length(X); t = log2(n);
25 x = fidemo.fi_bitreverse(X,n);
26
27 % Generate index variables LL2 so they are not computed in
28 % the loop.
29 LL = int32(2.^(1:t));
30 nr = int32(n./LL);
31 LL2 = int32(LL./2);
32 for q = 1:t
33     L = LL(q); nr = nr...
```

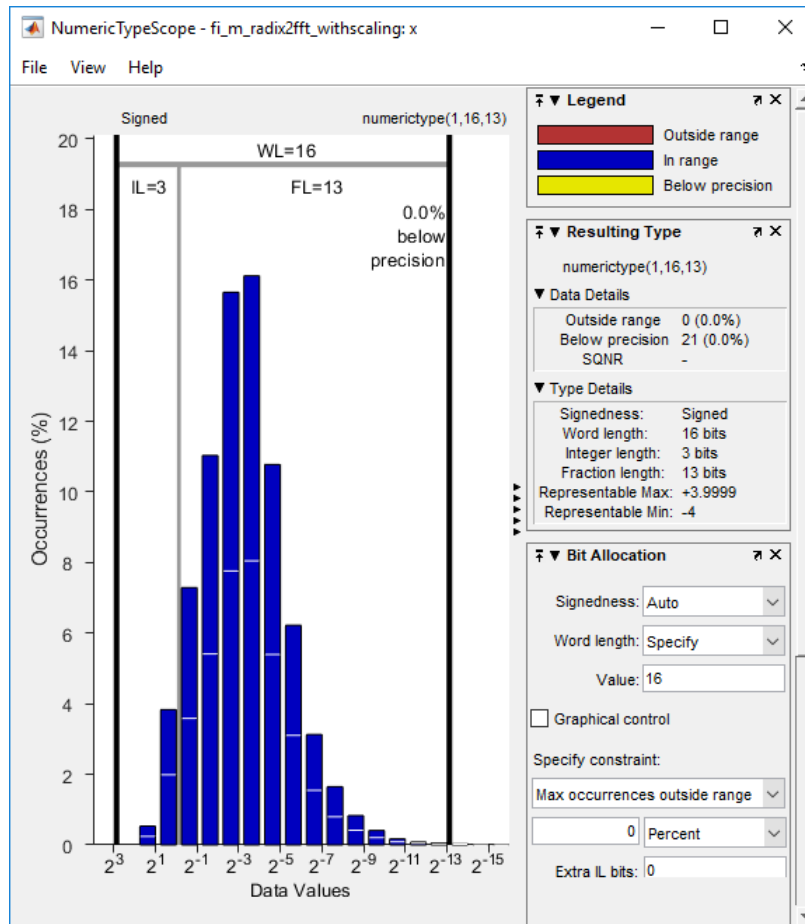
**Tooltip for LL2:**

- Size: 1 × 7
- Class: int32
- Complex: No
- Always Whole Number: Yes
- Sim Min: 1
- Sim Max: 64

**VARIABLES Table:**

Name	Type	Class	Always Whole Number	Sim Min	Sim Max
x	I/O	complex double	No	-3.722211247178794	3.325535825655795
w	Input	complex double	No	-1	1
n	Local	double	Yes	128	128
t	Local	double	Yes	7	7
LL	Local	int32	Yes	2	128
nr	Local	int32	Yes	1	64
LL2	Local	int32	Yes	1	64
temp	Local	complex double	No	-3.102703946701695	3.090313522579606
L	Local	int32	Yes	2	128
r	Local	int32	Yes	1	64

5 View the histogram for a variable by clicking  in the **Variables** tab.



For information on the figure, refer to the NumericTypeScope reference page.

- 6 Close the histogram display and then, clear the results log.

```
clearInstrumentationResults testfft_instrumented;
```

- 7 Clear the MEX function, then delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanup;
```

## Build an Instrumented MEX Function for Multiple Entry Point Functions

In a local writable folder, create the functions `ep1.m` and `ep2.m`.

```
function y1 = ep1(u) %#codegen
y1 = u;
end

function y2 = ep2(u, v) %#codegen
y2 = u + v;
end
```

Generate an instrumented MEX function for the two entry-point functions. Use the `-o` option to specify the name of the MEX function. Use the `-histogram` option to compute histograms. Use the `-coder` option to enable generating multiple entry points with the `buildInstrumentedMex` function.

```
u = 1:100;
v = 5:104;
buildInstrumentedMex -o sharedmex ...
ep1 -args {u} ... % Entry point 1
ep2 -args {u, v} ... % Entry point 2
-histogram -coder
```

Call the first entry-point function using the generated MEX function.

```
y1 = sharedmex('ep1', u);
```

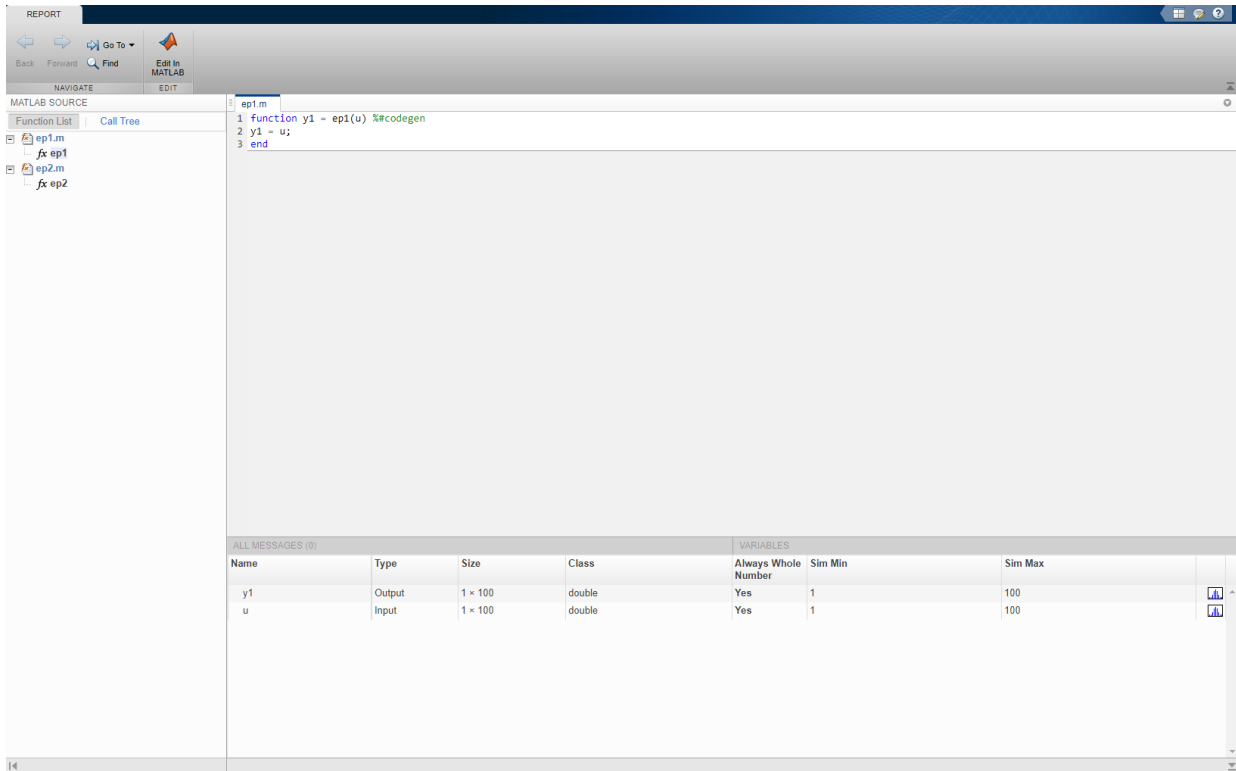
Call the second entry-point function using the generated MEX function.

```
y2 = sharedmex('ep2', u, v);
```

Show the instrumentation results.

```
showInstrumentationResults sharedmex
```






---

**Note** Generating a MEX function for multiple entry-point functions using the `buildInstrumentedMex` function requires a MATLAB Coder license.

---

## Tips

- You cannot instrument MATLAB functions provided with the software. If your top-level function is such a MATLAB function, nothing is logged. You also cannot instrument scripts.
- Instrumentation results are accumulated every time the instrumented MEX function is called. Use `clearInstrumentationResults` to clear previous results in the log.
- Some coding patterns pass a significant amount of data, but only use a small portion of that data. In such cases, you may see degraded performance when using

`buildInstrumentedMex`. In the following pattern, `subfun` only uses one element of input array, `A`. For normal execution, the amount of time to execute `subfun` once remains constant regardless of the size of `A`. The function `topfun` calls `subfun` `N` times, and thus the total time to execute `topfun` is proportional to `N`. When instrumented, however, the time to execute `subfun` once becomes proportional to  $N^2$ . This change occurs because the minimum and maximum data are calculated over the entire array. When `A` is large, the calculations can lead to significant performance degradation. Therefore, whenever possible, you should pass only the data that the function actually needs.

```
function A = topfun(A)
    N = numel(A);
    for i=1:N
        A(i) = subfun(A,i);
    end
end
function b = subfun(A,i)
    b = 0.5 * A(i);
end

function A = topfun(A)
    N = numel(A);
    for i=1:N
        A(i) = subfun(A(i));
    end
end
function b = subfun(a)
    b = 0.5 * a;
end
```

## See Also

`NumericTypeScope` | `clearInstrumentationResults` | `codegen` | `fiaccl` | `mex` | `showInstrumentationResults`

**Introduced in R2011b**

# cast

Cast variable to different data type

## Syntax

```
b = cast(a, 'like', p)
```

## Description

`b = cast(a, 'like', p)` converts `a` to the same `numericType`, complexity (real or complex), and `fi`math as `p`. If `a` and `p` are both real, then `b` is also real. Otherwise, `b` is complex.

## Examples

### Convert an int8 Value to Fixed Point

Define a scalar 8-bit integer.

```
a = int8(5);
```

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([], 1, 24, 12);
```

Convert `a` to fixed point with `numericType`, complexity (real or complex), and `fi`math of the specified `fi` object, `p`.

```
b = cast(a, 'like', p)
```

```
b =  
    5
```

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

```
WordLength: 24  
FractionLength: 12
```

### **Convert an Array to Fixed Point**

Define a 2-by-3 matrix of ones.

```
A = ones(2,3);
```

Create a signed `fi` object with word length of 16 and fraction length of 8.

```
p = fi([],1,16,8);
```

Convert `A` to the same data type and complexity (real or complex) as `p`.

```
B = cast(A, 'like', p)
```

```
B=2×3 object
```

```
    1    1    1  
    1    1    1
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 8
```

### **Write MATLAB Code That Is Independent of Data Types**

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    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
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818    0.195262145875635    0.097631072937818];
    a = [1.000000000000000    -0.942809041582063    0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);

    % Create a step input using ones with the
    % floating-point data type
    t = 0:20;
    x_float = ones(size(t),'like',T_float.data);

    % Initialize the states using zeros with the
    % floating-point data type
    z_float = zeros(1,2,'like',T_float.data);

    % Run the floating-point algorithm
    y_float = my_filter(b,a,x_float,z_float,T_float);

    % Define fixed-point types
    T_fixed.coeffs = fi([],true,8,6);
    T_fixed.data   = fi([],true,8,6);
```

```
% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t), 'like', T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2, 'like', T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf', 'subplot', 'plot', 'legend')
clf
subplot(211)
plot(t,y_float, 'co-', t,y_fixed, 'kx-')
legend('Floating-point output', 'Fixed-point output')
title('Step response')
subplot(212)
plot(t,y_float - double(y_fixed), 'rs-')
legend('Error')
figure(gcf)
end
```

## Input Arguments

### **a** — Variable that you want to cast to a different data type

fi object | numeric variable

Variable, specified as a fi object or numeric variable.

Complex Number Support: Yes

### **p** — Prototype

fi object | numeric variable

Prototype, specified as a fi object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

Complex Number Support: Yes

## Tips

Using the `b = cast(a, 'like', p)` syntax to specify data types separately 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.

## See Also

`cast` | `ones` | `zeros`

## Topics

[“Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros”](#)

[“Manual Fixed-Point Conversion Workflow”](#)

[“Manual Fixed-Point Conversion Best Practices”](#)

**Introduced in R2013a**

## ceil

Round toward positive infinity

### Syntax

```
y = ceil(a)
```

### Description

`y = ceil(a)` rounds `fi` object `a` to the nearest integer in the direction of positive infinity and returns the result in `fi` object `y`.

`y` and `a` have the same `fi` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a` plus one bit. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`ceil` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.



## Examples

### Example 1

The following example demonstrates how the `ceil` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

```
y = ceil(a)
```

```
y =
```

```
4
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 6
      FractionLength: 0
```

### Example 2

The following example demonstrates how the `ceil` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025, 1, 8, 12)
```

```
a =
```

```
0.0249
```

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

Signedness: Signed  
 WordLength: 8  
 FractionLength: 12

y = ceil(a)

y =

1

DataTypeMode: Fixed-point: binary point scaling  
 Signedness: Signed  
 WordLength: 2  
 FractionLength: 0

### Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>ceil(a)</b>	<b>fix(a)</b>	<b>floor(a)</b>
- 2.5	-2	-2	-3
-1.75	-1	-1	-2
-1.25	-1	-1	-2
-0.5	0	0	-1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

convergent | fix | floor | nearest | round

**Introduced in R2008a**

## clearInstrumentationResults

Clear results logged by instrumented, compiled C code function

### Syntax

```
clearInstrumentationResults('mex_fcn')  
clearInstrumentationResults mex_fcn  
clearInstrumentationResults all
```

### Description

`clearInstrumentationResults('mex_fcn')` clears the results logged from calling the instrumented MEX function *mex\_fcn*.

`clearInstrumentationResults mex_fcn` is alternative syntax for clearing the log.

`clearInstrumentationResults all` clears the results from all instrumented MEX functions.

### Input Arguments

**mex\_fcn**

Instrumented MEX function created using `buildInstrumentedMex`.

### Examples

Run a test bench to log instrumentation, then use `clearInstrumentationResults` to clear the log.

- 1 Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('showInstrumentationResults')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
    'fidemos','fi_m_radix2fft_withscaling.m'),...
    'testfft.m','f')
```

- 2 Define prototype input arguments.

```
n = 128;
x = complex(fi(zeros(n,1),'DataType','ScaledDouble'));
W = coder.Constant(fi(fidemo.fi_radix2twiddles(n)));
```

- 3 Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name.

```
buildInstrumentedMex testfft -o testfft_instrumented -args {x,W}
```

- 4 Run a test bench to record instrumentation results. Call `showInstrumentationResults` to open a report. View the simulation minimum and maximum values and whole number status by pausing over a variable in the report.

```
for i=1:20
    y = testfft_instrumented(cast(2*rand(size(x))-1,'like',x));
end
```

```
showInstrumentationResults testfft_instrumented
```

## 5 Functions — Alphabetical List

The screenshot shows the MATLAB R2019a environment with the Instrumentation Results window open for the `testfft.m` function. The window displays the source code of the function and a table of instrumentation results for various variables.

**Source Code:**

```

1 function x = fi_radix2fft_withscaling(x, w)
2 %FI_RADIX2FFT_WITHSCALING Radix-2 FFT example with scaling at each stage.
3 % Y = FI_RADIX2FFT_WITHSCALING(X, W) computes the radix-2 FFT of
4 % input vector X with twiddle-factors W with scaling by 1/2 at each stage.
5 % Input X is assumed to be complex.
6 %
7 % The length of vector X must be an exact power of two.
8 % Twiddle-factors W are computed via
9 % W = fidemo.fi_radix2twiddles(N)
10 % where N = length(X).
11 %
12 % This version of the algorithm has no scaling before the stages.
13 %
14 % See also FI_RADIX2FFT_DEMO.
15
16 % Reference:
17 % Charles Van Loan, Computational
18 % Transform, SIAM, Philadelphia.
19 %
20 % Copyright 2004-2015 The MathWorks
21 %
22 %codegen
23
24 n = length(x); t = log2(n);
25 x = fidemo.fi_bitreverse(0,n);
26
27 % Generate index variables as in
28 % the loop.
29 LL = int32(2.^(1:t));
30 rr = int32(n./LL);
31 LL2 = int32(LL./2);
32 for q=1:t
33     L = LL(q); r = rr(q); L2 =

```

**Instrumentation Results Table:**

Name	Type	Size	Class	DT Mode	Signedness	WL	FL	Percent of Current Range	Always Whole Number	Sim Min	Sim Max
x	I/O	128 × 1	complex embedded fi	ScaledDouble	Signed	16	15	100	No	-0.9998521328458922	0.9988979427807565
w	Input	127 × 1	complex embedded fi	-	Signed	16	14	51	No	-1	1
n	Local	1 × 1	double	-	-	-	-	-	Yes	128	128
t	Local	1 × 1	double	-	-	-	-	-	Yes	7	7
LL	Local	1 × 7	int32	-	-	-	-	-	Yes	2	128
rr	Local	1 × 7	int32	-	-	-	-	-	Yes	1	64
LL2	Local	1 × 7	int32	-	-	-	-	-	Yes	1	64
temp	Local	1 × 1	complex embedded fi	ScaledDouble	Signed	33	29	13	No	-0.9998521328458922	0.9988979427807565
L	Local	1 × 1	int32	-	-	-	-	-	Yes	2	128

1 Clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

2 Run a different test bench, then view the new instrumentation results.

```
for i=1:20
```

```
    y = testfft_instrumented(cast(rand(size(x))-0.5,'like',x));
end
```

```
showInstrumentationResults testfft_instrumented
```

```

16 % Reference:
17 % Charles Van Loan, Computational
18 % Transform, SIAM, Philadelphia,
19 %
20 % Copyright 2004-2015 The MathWorks
21 %
22 %#codegen
23
24 n = length(x); t = log2(n);
25 x = fidemo.fi_bitreverse(x,n);
26
27 % Generate index variables as in
28 % the loop.
29 LL = int32(2.^(1:t));
30 rr = int32(n./LL);
31 LL2 = int32(LL./2);
32 for q=1:t
33     L = LL(q); r = rr(q); L2 = L

```

VARIABLE INFO

<b>x</b>	
Size:	128 × 1
Class:	embedded.fi
Complex:	Yes

---

NUMERICTYPE

DataTypeMode:	'Scaled double: binary point scaling'
DataType:	'ScaledDouble'
Signedness:	'Signed'
WordLength:	16
FractionLength:	15

---

INSTRUMENTATION RESULTS

Percent of Current Range:	50
Always Whole Number:	No
Sim Min:	-0.49995165544249043
Sim Max:	0.4998392859913364

LL MESSAGES (0)

- 3 Clear the MEX function and delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanUp;
```

## See Also

[buildInstrumentedMex](#) | [codegen](#) | [fiaccel](#) | [mex](#) | [showInstrumentationResults](#)

**Introduced in R2011b**

## **coder.approximation**

Create function replacement configuration object

### **Syntax**

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

### **Description**

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the `fiaccl` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

### **Examples**

#### **Replace Log Function with Default Lookup Table**

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```



## Replace Log Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...  
'FunctionNamePrefix','log_replace');
```

## Replace Log Function with Optimized Lookup Table

Create a function replacement configuration object using the 'OptimizeLUTSize' option to specify to replace the log function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...  
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...  
'FunctionNamePrefix','log_optim','OptimizeIterations',25);
```

## Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, saturateExp, with an optimized lookup table.

Create a custom function, saturateExp.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the saturateExp function with an optimized lookup table. Because the saturateExp function is not listed as a function for which coder.approximation can generate an approximation automatically, you must specify the CandidateFunction property.

```
saturateExp = @(x) 1/(1+exp(-x));  
custAppx = coder.approximation('Function','saturateExp',...
```

```
'CandidateFunction', saturateExp,...  
'NumberOfPoints',50,'InputRange',[0,10]);
```

## Input Arguments

### **function\_name** — Name of the function to replace

'acos' | 'acosd' | 'acosh' | 'acoth' | 'asin' | 'asind' | 'asinh' | 'atan' | 'atand' | 'atanh' | 'cos' | 'cosd' | 'cosh' | 'erf' | 'erfc' | 'exp' | 'log' | 'normcdf' | 'reallog' | 'realsqrt' | 'reciprocal' | 'rsqrt' | 'sin' | 'sinc' | 'sind' | 'sinh' | 'sqrt' | 'tan' | 'tand'

Name of function to replace, specified as a string. The function must be one of the listed functions.

Example: 'sqrt'

Data Types: char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Function', 'log'

### **Architecture** — Architecture of lookup table approximation

'LookupTable' (default) | 'Flat'

Architecture of the lookup table approximation, specified as the comma-separated pair consisting of 'Architecture' and a string. Use this argument when you want to specify the architecture for the lookup table. The `Flat` architecture does not use interpolation.

Data Types: char

### **CandidateFunction** — Function handle of the replacement function

function handle | string

Function handle of the replacement function, specified as the comma-separated pair consisting of 'CandidateFunction' and a function handle or string referring to a function handle. Use this argument when the function that you want to replace is not

listed under `function_name`. Specify the function handle or string referring to a function handle of the function that you want to replace. You can define the function in a file or as an anonymous function.

If you do not specify a candidate function, then the function you chose to replace using the `Function` property is set as the `CandidateFunction`.

Example: `'CandidateFunction', @(x) (1./(1+x))`

Data Types: `function_handle` | `char`

### **ErrorThreshold — Error threshold value used to calculate optimal lookup table size**

0.001 (default) | nonnegative scalar

Error threshold value used to calculate optimal lookup table size, specified as the comma-separated pair consisting of `'ErrorThreshold'` and a nonnegative scalar. If `'OptimizeLUTSize'` is true, this argument is required.

### **Function — Name of function to replace with a lookup table approximation**

`function_name`

Name of function to replace with a lookup table approximation, specified as the comma-separated pair consisting of `'Function'` and a string. The function must be continuous and stateless. If you specify one of the functions that is listed under `function_name`, the conversion process automatically provides a replacement function. Otherwise, you must also specify the `'CandidateFunction'` argument for the function that you want to replace.

Example: `'Function','log'`

Example: `'Function','my_log','CandidateFunction',@my_log`

Data Types: `char`

### **FunctionNamePrefix — Prefix for generated fixed-point function names**

`'replacement_'` (default) | string

Prefix for generated fixed-point function names, specified as the comma-separated pair consisting of `'FunctionNamePrefix'` and a string. The name of a generated function consists of this prefix, followed by the original MATLAB function name.

Example: `'log_replace_'`

### **InputRange — Range over which to replace the function**

[ ] (default) | 2x1 row vector | 2xN matrix

Range over which to replace the function, specified as the comma-separated pair consisting of 'InputRange' and a 2-by-1 row vector or a 2-by-N matrix.

Example: [-1 1]

### **InterpolationDegree — Interpolation degree**

1 (default) | 0 | 2 | 3

Interpolation degree, specified as the comma-separated pair consisting of 'InterpolationDegree' and 1 (linear), 0 (none), 2 (quadratic), or 3 (cubic).

### **NumberOfPoints — Number of points in lookup table**

1000 (default) | positive integer

Number of points in lookup table, specified as the comma-separated pair consisting of 'NumberOfPoints' and a positive integer.

### **OptimizeIterations — Number of iterations**

25 (default) | positive integer

Number of iterations to run when optimizing the size of the lookup table, specified as the comma-separated pair consisting of 'OptimizeIterations' and a positive integer.

### **OptimizeLUTSize — Optimize lookup table size**

false (default) | true

Optimize lookup table size, specified as the comma-separated pair consisting of 'OptimizeLUTSize' and a logical value. Setting this property to true generates an area-optimal lookup table, that is, the lookup table with the minimum possible number of points. This lookup table is optimized for size, but might not be speed efficient.

### **PipelinedArchitecture — Option to enable pipelining**

false (default) | true

Option to enable pipelining, specified as the comma-separated pair consisting of 'PipelinedArchitecture' and a logical value.

## Output Arguments

**q** — Function replacement configuration object, returned as a `coder.mathfcngenerator.LookupTable` or a `coder.mathfcngenerator.Flat` configuration object

`coder.mathfcngenerator.LookupTable` configuration object |  
`coder.mathfcngenerator.Flat` configuration object

Function replacement configuration object that specifies how to create an approximation for a MATLAB function. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `fiaccel` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point MATLAB code.

Property	Default Value
Auto-replace function	''
InputRange	[]
FunctionNamePrefix	'replacement_'
Architecture	LookupTable (read only)
NumberOfPoints	1000
InterpolationDegree	1
ErrorThreshold	0.001
OptimizeLUTSize	false
OptimizeIterations	25

## See Also

### Classes

`coder.FixptConfig`

### Functions

`fiaccel`

## **Topics**

“Replace the exp Function with a Lookup Table”

“Replace a Custom Function with a Lookup Table”

“Replacing Functions Using Lookup Table Approximations”

## **Introduced in R2014b**

# coder.allowpcode

**Package:** coder

Control code generation from protected MATLAB files

## Syntax

```
coder.allowpcode('plain')
```

## Description

`coder.allowpcode('plain')` allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-file takes precedence.

`coder.allowpcode` is ignored outside of code generation.

## Examples

Generate optimized embeddable code from protected MATLAB code:

- 1 Write an function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)    %#codegen
% The directive %#codegen indicates that the function
```

```
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

- 2 Generate protected P-code. At the MATLAB prompt, enter:

```
pcode p_abs
```

The P-file, `p_abs.p`, appears in the current folder.

- 3 Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -args { int32(0) }
```

`codegen` generates a MEX function in the current folder.

- 4 Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` generates C library code in the `codegen\lib\p_abs` folder.

## See Also

`codegen` | `pcode`

**Introduced in R2011a**



# coder.ArrayType class

**Package:** coder

**Superclasses:** coder.Type

Represent set of MATLAB arrays

## Description

Specifies the set of arrays that the generated code accepts. Use only with the `fiaccel -args` option. Do not pass as an input to a generated MEX function.

## Construction

`coder.ArrayType` is an abstract class. You cannot create instances of it directly. You can create `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, and `coder.StructType` objects that derive from this class.

## Properties

### ClassName

Class of values in this set

### SizeVector

The upper-bound size of arrays in this set.

### VariableDims

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

### See Also

`coder.CellType` | `coder.ClassType` | `coder.EnumType` | `coder.FiType` |  
`coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` |  
`coder.resize` | `coder.typeof` | `fiaccel`

**Introduced in R2011a**

## coder.config

Create configuration object for fixed-point or single-precision conversion

### Syntax

```
config_obj = coder.config('fixpt')  
config_obj = coder.config('single')
```

### Description

`config_obj = coder.config('fixpt')` creates a `coder.FixptConfig` configuration object. Use this object with the `fiaccl` function when converting floating-point MATLAB code to fixed-point MATLAB code.

`config_obj = coder.config('single')` creates a `coder.SingleConfig` configuration object for use with the `convertToSingle` function when generating single-precision MATLAB code from double-precision MATLAB code.

### Examples

#### Convert Floating-Point MATLAB Code to Fixed-Point MATLAB Code

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Convert your floating-point MATLAB design to fixed point. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

### **Convert Double-Precision MATLAB Code to Single-Precision MATLAB Code**

Create a `coder.SingleConfig` object, `scfg`.

```
scfg = coder.config('single');
```

Set the test bench name. In this example, the test bench function name is `myfun_test`. Enable numerics testing and data logging for comparison plotting of input and output variables.

```
scfg.TestBenchName = 'myfun_test';  
scfg.TestNumerics = true;  
scfg.LogIOForComparisonPlotting = true;
```

Convert the double-precision MATLAB code to single-precision MATLAB code. In this example, the MATLAB function name is `myfun`.

```
convertToSingle -config scfg myfun
```

### **See Also**

`coder.FixptConfig` | `coder.SingleConfig` | `convertToSingle` | `fiaccel`

**Introduced in R2014b**

# **coder.const**

Fold expressions into constants in generated code

## **Syntax**

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle, arg1,..., argN)
```

## **Description**

`out = coder.const(expression)` evaluates `expression` and replaces `out` with the result of the evaluation in generated code.

`[out1,...,outN] = coder.const(handle, arg1,..., argN)` evaluates the multi-output function having handle `handle`. It then replaces `out1,...,outN` with the results of the evaluation in the generated code.

## **Examples**

### **Specify Constants in Generated Code**

This example shows how to specify constants in generated code using `coder.const`.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the expression `(1:10).^2` with `coder.const((1:10).^2)`, and then generate code for `AddShift` again using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int i;
    static const signed char iv[10] = { 1, 4, 9, 16, 25, 36,
                                        49, 64, 81, 100 };

    for (i = 0; i < 10; i++) {
        y[i] = (double)iv[i] + Shift;
    }
}
```

### Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```
function y = getsine(index) %#codegen
    assert(isa(index, 'int32'));
```

```

persistent tbl;
if isempty(tbl)
    tbl = gettable(1024);
end
y = tbl(index);

function y = gettable(n)
    y = zeros(1,n);
    for i = 1:n
        y(i) = sin((i-1)/(2*pi*n));
    end
end

```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

```
codegen -config:lib -launchreport getsine -args int32(0)
```

The generated code contains instructions for creating the lookup table.

Replace the statement:

```
tbl = gettable(1024);
```

with:

```
tbl = coder.const(gettable(1024));
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

The generated code contains the lookup table itself. `coder.const` forces the expression `gettable(1024)` to be evaluated during code generation. The generated code does not contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

### Specify Constants in Generated Code Using Multi-Output Function

This example shows how to specify constants in generated code using a multi-output function in a `coder.const` statement.

Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```
function [y1,y2] = MultiplyConst(factor) %#codegen
    [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
    y1=vec1.*factor;
    y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
    f1=z.^(2*n)/factorial(2*n);
    f2=z.^(2*n+1)/factorial(2*n+1);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator produces code for creating the vectors.

Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

### Read Constants by Processing XML File

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `MyParams.xml` containing the following statements:

```
<params>
    <param name="hello" value="17"/>
```



```
<param name="world" value="42"/>
</params>
```

Save `MyParams.xml` in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field name of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName('param');
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute('name'));
        paramValue = char(param.getAttribute('value'));
        paramValue = evalin('base', paramValue);
        s.(paramName) = paramValue;
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```
function y = MyFunc(u) %#codegen
    assert(isa(u, 'double'));
    coder.extrinsic('xml2struct');
    s = coder.const(xml2struct('MyParams.xml'));
    y = s.hello + s.world + u;
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:dll -launchreport MyFunc -args 0
```

The code generator executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

## Input Arguments

### **expression** — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```
function y=func(x)
    y=coder.const(log10(x));
```

To fix the error, assign `x` to a constant in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args coder.Constant(10)
```

Example: `2*pi`, `factorial(10)`

### **handle** — Function handle

function handle

Handle to built-in or user-written function.

Example: `@log`, `@sin`

Data Types: `function_handle`

### **arg1, ..., argN** — Arguments to the function with handle `handle`

function arguments that are constants

Arguments to the function with handle `handle`.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because `x` and `y` are not compile-time constants.

```
function y=func(x,y)
    y=coder.const(@nchoosek,x,y);
```

To fix the error, assign `x` and `y` to constants in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

## Output Arguments

### **out** — Value of expression

value of the evaluated expression

Value of expression. In the generated code, MATLAB Coder replaces occurrences of `out` with the value of expression.

### **out1, ..., outN** — Outputs of the function with handle handle

values of the outputs of the function with handle handle

Outputs of the function with handle handle. MATLAB Coder evaluates the function and replaces occurrences of `out1, ..., outN` with constants in the generated code.

## Tips

- When possible, the code generator constant-folds expressions automatically. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generator does not constant-fold expressions on its own.
- When constant-folding computationally intensive function calls, to reduce code generation time, make the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

```
function j = fcn(z)
zTable = coder.const(0:0.01:100);
jTable = coder.const(feval('besselj',3,zTable));
j = interp1(zTable,jTable,z);
end
```

See “Use `coder.const` with Extrinsic Function Calls” (MATLAB Coder).

- If `coder.const` is unable to constant-fold a function call, try to force constant-folding by making the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

```
function yi = fcn(xi)
y = coder.const(feval('rand',1,100));
yi = interp1(y,xi);
end
```

See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

## See Also

### Topics

“Fold Function Calls into Constants” (MATLAB Coder)

“Use coder.const with Extrinsic Function Calls” (MATLAB Coder)

**Introduced in R2013b**

# coder.Constant class

**Package:** coder

**Superclasses:** coder.Type

Represent set containing one MATLAB value

## Description

Use a `coder.Constant` object to define values that are constant during code generation. Use only with the `fiaccel -args` options. Do not pass as an input to a generated MEX function.

## Construction

`const_type=coder.Constant(v)` creates a `coder.Constant` type from the value `v`.

`const_type=coder.newtype('constant', v)` creates a `coder.Constant` type from the value `v`.

## Input Arguments

**v**

Constant value used to construct the type.

## Properties

**Value**

The actual value of the constant.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

Create a constant with value 42.

```
k = coder.Constant(42);
```

Create a new constant type for use in code generation.

```
k = coder.newtype('constant', 42);
```

## Limitations

- You cannot use `coder.Constant` on sparse matrices, or on structures, cell arrays, or classes that contain sparse matrices.

## See Also

`coder.Type` | `coder.newtype` | `fiaccl`

**Introduced in R2011a**

## coder.EnumType class

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of MATLAB enumerations

### Description

Specifies the set of MATLAB enumerations that the generated code should accept. Use only with the `fiaccel -args` options. Do not pass as an input to a generated MEX function.

### Construction

`enum_type = coder.typeof(enum_value)` creates a `coder.EnumType` object representing a set of enumeration values of class (`enum_value`).

`enum_type = coder.typeof(enum_value, sz, variable_dims)` returns a modified copy of `coder.typeof(enum_value)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the (upper bound) sizes of `v` do not change. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed; the unbounded dimensions are variable size. When `variable_dims` is a scalar, it applies to bounded dimensions that are not 1 or 0 (which are fixed).

`enum_type = coder.newtype(enum_name, sz, variable_dims)` creates a `coder.EnumType` object that has variable size with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to bounded dimensions that are not 1 or 0 (which are fixed).

## Input Arguments

### **enum\_value**

Enumeration value defined in a file on the MATLAB path.

### **sz**

Size vector specifying each dimension of type object.

**Default:** [1 1] for `coder.newtype`

### **variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

**Default:** `false(size(sz)) | sz==Inf` for `coder.newtype`

### **enum\_name**

Name of enumeration defined in a file on the MATLAB path.

## Properties

### **ClassName**

Class of values in the set.

### **SizeVector**

The upper-bound size of arrays in the set.

### **VariableDims**

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.



## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

Create a `coder.EnumType` object using a value from an existing MATLAB enumeration.

- 1 Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

- 2 Create a `coder.EnumType` object from this enumeration.

```
t = coder.typeof(MyColors.red);
```

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

- 1 Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

- 2 Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors');
```

## See Also

`coder.ArrayType` | `coder.ClassType` | `coder.Type` | `coder.newtype` |  
`coder.resize` | `coder.typeof` | `fiaccel`

## **Topics**

“Enumerations”

**Introduced in R2011a**

# coder.extrinsic

Declare extrinsic functions

## Syntax

```
coder.extrinsic(function)
coder.extrinsic(function1, ... ,functionN)

coder('-sync:on', function1, ... ,functionN)
coder('-sync:off', function1, ... ,functionN)
```

## Description

`coder.extrinsic(function)` declares `function` as an extrinsic function. The code generator does not produce code for the body of the extrinsic function and instead uses the MATLAB engine to execute the call. This functionality is available only when the MATLAB engine is available in MEX functions, Simulink simulations, or during calls at compile time.

During standalone code generation, the code generator attempts to determine whether an extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that there is no change to the output, the code generator proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, the code generator produces a compilation error.

You cannot use `coder.ceval` on functions that you declare as extrinsic by using `coder.extrinsic`.

`coder.extrinsic` is ignored outside of code generation.

`coder.extrinsic(function1, ... ,functionN)` declares `function1` through `functionN` as extrinsic functions.

`coder('-sync:on', function1, ... ,functionN)` enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions `function1` through `functionN`. If only a few extrinsic calls use or modify

global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At` MEX-function entry and exit. Use the `'-sync:on'` option to turn on synchronization for only the extrinsic calls that do modify global data.

See “Generate Code for Global Data” (MATLAB Coder).

`coder('-sync:off', function1, ... ,functionN)` disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions `function1` through `functionN`. If most extrinsic calls use or modify global data, but a few do not, use the `'-sync:off'` option to turn off synchronization for the extrinsic calls that do not modify global data.

See “Generate Code for Global Data” (MATLAB Coder).

## Examples

### Declare a MATLAB Function as Extrinsic

The MATLAB function `patch` is not supported for code generation. This example shows how you can still use the functionality of `patch` in your generated MEX function by declaring `patch` as extrinsic your MATLAB function.

This MATLAB code declares `patch` as extrinsic in the local function `create_plot`. By declaring `patch` as extrinsic, you instruct the code generator not to produce code for `patch`. Instead, the code generator dispatches `patch` to MATLAB for execution.

The code generator automatically treats many common MATLAB visualization functions, such as the function `axis` as extrinsic.

```
function c = pythagoras(a,b,color) %#codegen
% Calculate the hypotenuse of a right triangle
% and display the triangle as a patch object.
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
end

function create_plot(a, b, color)
%Declare patch as extrinsic
coder.extrinsic('patch');
```

```
x = [0;a;a];  
y = [0;0;b];  
patch(x,y,color);  
axis('equal');  
end
```

Generate a MEX function for `pythagoras`. Also, generate the code generation report.

```
codegen -report pythagoras -args {1, 1, [.3 .3 .3]}
```

In the report, view the MATLAB code for `create_plot`.

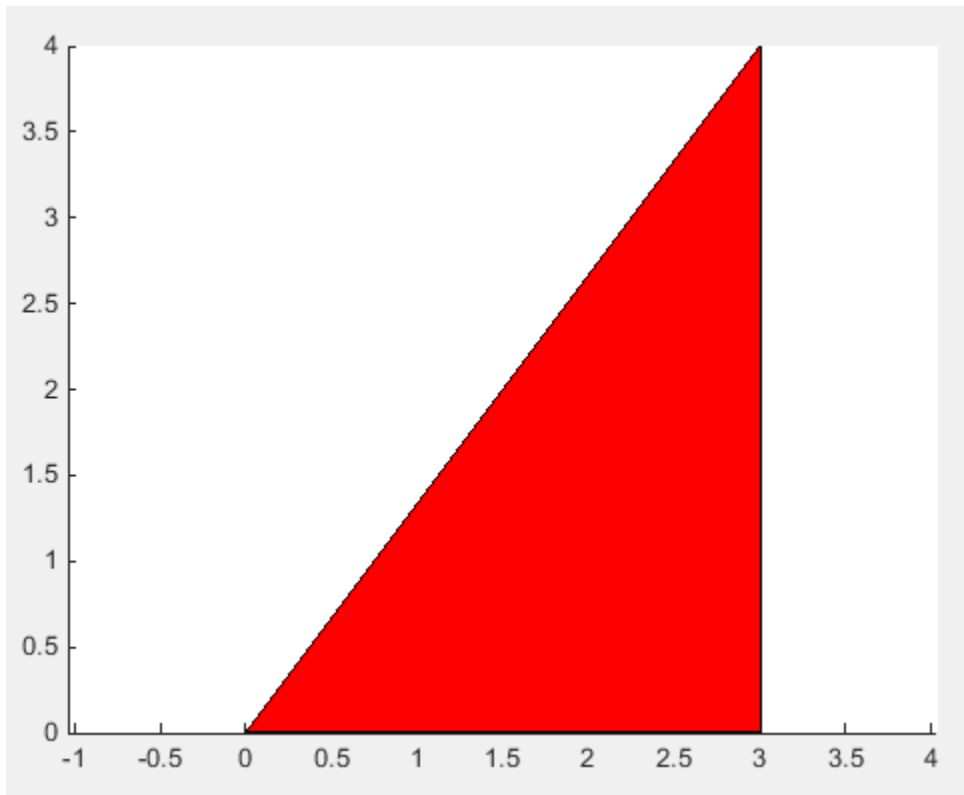
```
7 function create_plot(a, b, color)  
8 coder.extrinsic('patch');  
9 x = [0;a;a];  
10 y = [0;0;b];  
11 patch(x,y,color);  
12 axis('equal');  
13 end
```

The report highlights the `patch` and `axis` functions to indicate that they are treated as extrinsic functions.

Run the MEX function.

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays the plot of the right triangle as a red patch object.



## Input Arguments

**function** — MATLAB function name

character vector

Name of the MATLAB function that is declared as extrinsic.

Example: `coder.extrinsic('patch')`

Data Types: char

## Limitations

- Extrinsic function calls have some overhead that can affect performance. Input data that is passed in an extrinsic function call must be provided to MATLAB, which requires making a copy of the data. If the function has any output data, this data must be transferred back into the MEX function environment, which also requires a copy.
- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

## Tips

- The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`.
- Use the `coder.screener` function to detect which functions you must declare as extrinsic. This function runs the Code Generation Readiness Tool that screens the MATLAB code for features and functions that are not supported for code generation.

## See Also

`coder.screener`

## Topics

“Extrinsic Functions”

“Generate Code for Global Data” (MATLAB Coder)

“Resolution of Function Calls for Code Generation”

**Introduced in R2011a**

## **coder.FiType class**

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of MATLAB fixed-point arrays

### **Description**

Specifies the set of fixed-point array values that the generated code should accept. Use only with the `fiaccel -args` options. Do not pass as an input to the generated MEX function.

### **Construction**

`t=coder.typeof(v)` creates a `coder.FiType` object representing a set of fixed-point values whose properties are based on the fixed-point input `v`.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the (upper bound) sizes of `v` do not change. If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not 1 or 0 (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not 1 or 0 (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and additional options specified by one or more `Name, Value` pair



arguments. Name can also be a property name and Value is the corresponding value. Specify Name as a character vector or string scalar. You can specify several name-value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

## Input Arguments

### **v**

Fixed-point value used to create new coder.FiType object.

### **sz**

Size vector specifying each dimension of type object.

**Default:** [1 1] for coder.newtype

### **variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

**Default:** false(size(sz)) | sz == Inf for coder.newtype

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **complex**

Set complex to true to create a coder.Type object that can represent complex values. The type must support complex data.

**Default:** false

### **fimath**

Specify local fimath. If not, uses default fimath.

## Properties

### **ClassName**

Class of values in the set.

### **Complex**

Indicates whether fixed-point arrays in the set are real (`false`) or complex (`true`).

### **Fimath**

Local `fimath` that the fixed-point arrays in the set use.

### **NumericType**

`numericType` that the fixed-point arrays in the set use.

### **SizeVector**

The upper-bound size of arrays in the set.

### **VariableDims**

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

Create a new fixed-point type `t`.

```
t = coder.typeof(fi(1));  
% Returns  
% coder.FiType  
%   1x1 embedded.fi  
%       DataTypeMode:Fixed-point: binary point scaling
```

```
% Signedness:Signed
% WordLength:16
% FractionLength:14
```

Create a new fixed-point type for use in code generation. The fixed-point type uses the default `fimath`.

```
t = coder.newtype('embedded.fi', numerictype(1, 16, 15), [1 2])
```

```
t =
% Returns
% coder.FiType
% 1x2 embedded.fi
% DataTypeMode: Fixed-point: binary point scaling
% Signedness: Signed
% WordLength: 16
% FractionLength: 15
```

This new type uses the default `fimath`.

## See Also

[coder.ArrayType](#) | [coder.ClassType](#) | [coder.Type](#) | [coder.newtype](#) | [coder.resize](#) | [coder.typeof](#) | [fiaccel](#)

**Introduced in R2011a**

## **coder.FixptConfig class**

**Package:** coder

Floating-point to fixed-point conversion configuration object

### **Description**

A `coder.FixptConfig` object contains the configuration parameters that the `fiaccl` function requires to convert floating-point MATLAB code to fixed-point MATLAB code. Use the `-float2fixed` option to pass this object to the `fiaccl` function.

### **Construction**

`fixptcfg = coder.config('fixpt')` creates a `coder.FixptConfig` object for floating-point to fixed-point conversion.

### **Properties**

#### **ComputeDerivedRanges**

Enable derived range analysis.

Values: `true`|`false` (default)

#### **ComputeSimulationRanges**

Enable collection and reporting of simulation range data. If you need to run a long simulation to cover the complete dynamic range of your design, consider disabling simulation range collection and running derived range analysis instead.

Values: `true` (default)|`false`

#### **DefaultFractionLength**

Default fixed-point fraction length.

Values: 4 (default) | positive integer

### **DefaultSignedness**

Default signedness of variables in the generated code.

Values: 'Automatic' (default) | 'Signed' | 'Unsigned'

### **DefaultWordLength**

Default fixed-point word length.

Values: 14 (default) | positive integer

### **DetectFixptOverflows**

Enable detection of overflows using scaled doubles.

Values: true | false (default)

### **fimath**

fimath properties to use for conversion.

Values: fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'SumMode', 'FullPrecision') (default) | string

### **FixPtFileNameSuffix**

Suffix for fixed-point file names.

Values: '\_fixpt' | string

### **LaunchNumericTypesReport**

View the numeric types report after the software has proposed fixed-point types.

Values: true (default) | false

### **LogI0ForComparisonPlotting**

Enable simulation data logging to plot the data differences introduced by fixed-point conversion.

Values: `true` (default) | `false`

### **OptimizeWholeNumber**

Optimize the word lengths of variables whose simulation min/max logs indicate that they are always whole numbers.

Values: `true` (default) | `false`

### **PlotFunction**

Name of function to use for comparison plots.

`LogIOForComparisonPlotting` must be set to `true` to enable comparison plotting. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

Values: `' '` (default) | string

### **PlotWithSimulationDataInspector**

Use Simulation Data Inspector for comparison plots.

`LogIOForComparisonPlotting` must be set to `true` to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

Values: `true` | `false` (default)

### **ProposeFractionLengthsForDefaultWordLength**

Propose fixed-point types based on `DefaultWordLength`.

Values: `true` (default) | `false`

### **ProposeTargetContainerTypes**

By default (`false`), propose data types with the minimum word length needed to represent the value. When set to `true`, propose data type with the smallest word length that can

represent the range and is suitable for C code generation ( 8,16,32, 64 ... ). For example, for a variable with range [0..7], propose a word length of 8 rather than 3.

Values: true| false (default)

### **ProposeWordLengthsForDefaultFractionLength**

Propose fixed-point types based on DefaultFractionLength.

Values: false (default) | true

### **ProposeTypesUsing**

Propose data types based on simulation range data, derived ranges, or both.

Values: 'BothSimulationAndDerivedRanges' (default) |  
'SimulationRanges' | 'DerivedRanges'

### **SafetyMargin**

Safety margin percentage by which to increase the simulation range when proposing fixed-point types. The specified safety margin must be a real number greater than -100.

Values: 0 (default) | double

### **StaticAnalysisQuickMode**

Perform faster static analysis.

Values: true | false (default)

### **StaticAnalysisTimeoutMinutes**

Abort analysis if timeout is reached.

Values: '' (default) | positive integer

### **TestBenchName**

Test bench function name or names, specified as a string or cell array of strings. You must specify at least one test bench.

If you do not explicitly specify input parameter data types, the conversion uses the first test bench function to infer these data types.

Values: '' (default) | string | cell array of strings

### **TestNumerics**

Enable numerics testing.

Values: true| false (default)

## **Methods**

<code>addApproximation</code>	Replace floating-point function with lookup table during fixed-point conversion
<code>addDesignRangeSpecification</code>	Add design range specification to parameter
<code>addFunctionReplacement</code>	Replace floating-point function with fixed-point function during fixed-point conversion
<code>clearDesignRangeSpecifications</code>	Clear all design range specifications
<code>getDesignRangeSpecification</code>	Get design range specifications for parameter
<code>hasDesignRangeSpecification</code>	Determine whether parameter has design range
<code>removeDesignRangeSpecification</code>	Remove design range specification from parameter

## **Examples**

### **Convert Floating-Point MATLAB Code to Fixed Point Based On Simulation Ranges**

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`. The conversion process uses the test bench to infer input data types and collect simulation range data.

```
fixptcfg.TestBenchName = 'dti_test';
```

Select to propose data types based on simulation ranges only. By default, proposed types are based on both simulation and derived ranges.



```
fixptcfg.ProposeTypesUsing = 'SimulationRanges';
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccl -float2fixed fixptcfg dti
```

### Convert Floating-Point MATLAB Code to Fixed Point Based On Simulation and Derived Ranges

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the name of the test bench to use to infer input data types. In this example, the test bench function name is `dti_test`. The conversion process uses the test bench to infer input data types.

```
fixptcfg.TestBenchName = 'dti_test';
```

Select to propose data types based on derived ranges.

```
fixptcfg.ProposeTypesUsing = 'DerivedRanges';  
fixptcfg.ComputeDerivedRanges = true;
```

Add design ranges. In this example, the `dti` function has one scalar double input, `u_in`. Set the design minimum value for `u_in` to -1 and the design maximum to 1.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
```

Convert the floating-point MATLAB function, `dti`, to fixed-point MATLAB code.

```
fiaccl -float2fixed fixptcfg dti
```

### Enable Overflow Detection

When you select to detect potential overflows, `fiaccl` generates a scaled double version of the generated fixed-point MEX function. Scaled doubles 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.

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Enable numerics testing with overflow detection.

```
fixptcfg.TestNumerics = true;  
fixptcfg.DetectFixptOverflows = true;
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccl -float2fixed fixptcfg dti
```

## Alternatives

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Converter app. Open the app using one of these methods:

- On the **Apps** tab, in the **Code Generation** section, click **Fixed-Point Converter**.
- Use the `fixedPointConverter` command.

## See Also

`coder.MexConfig` | `coder.mexconfig` | `fiaccl`

## Topics

“Propose Data Types Based on Simulation Ranges”

“Propose Data Types Based on Derived Ranges”

“Detect Overflows”

# coder.ignoreConst

Prevent use of constant value of expression for function specializations

## Syntax

```
coder.ignoreConst(expression)
```

## Description

`coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations on page 5-169. `coder.ignoreConst(expression)` returns the value of `expression`.

## Examples

### Prevent Function Specializations Based on Constant Input Values

Use `coder.ignoreConst` to prevent function specializations for a function that is called with constant values.

Write the function `call_myfcn`, which calls `myfcn`.

```
function [x, y] = call_myfcn(n)
    %#codegen
    x = myfcn(n, 'mode1');
    y = myfcn(n, 'mode2');
end

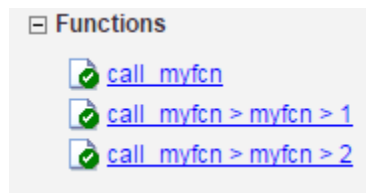
function y = myfcn(n,mode)
    coder.inline('never');
    if strcmp(mode,'mode1')
        y = n;
    else
        y = -n;
    end
end
```

```
end  
end
```

Generate standalone C code. For example, generate a static library. Enable the code generation report.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you see two function specializations for `call_myfcn`.



The code generator creates `call_myfcn>myfcn>1` for `mode` with a value of `'mode1'`. It creates `call_myfcn>myfcn>2` for `mode` with a value of `'mode2'`.

In the generated C code, you see the specializations `my_fcn` and `b_my_fcn`.

```
static double b_myfcn(double n)  
{  
    return -n;  
}
```

```
static double myfcn(double n)  
{  
    return n;  
}
```

To prevent the function specializations, instruct the code generator to ignore that values of the `mode` argument are constant.

```
function [x, y] = call_myfcn(n)  
    %#codegen  
    x = myfcn(n, coder.ignoreConst('mode1'));  
    y = myfcn(n, coder.ignoreConst('mode2'));  
end
```

```
function y = myfcn(n,mode)  
    coder.inline('never');  
    if strcmp(mode,'mode1')
```

```

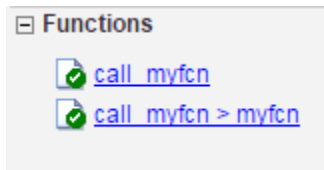
    y = n;
else
    y = -n;
end
end

```

Generate the C code.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you do not see multiple function specializations.



In the generated C code, you see one function for `my_fcn`.

## Input Arguments

**expression** — Expression whose value is to be treated as a nonconstant  
MATLAB expression

## More About

### Function Specialization

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.

### Tips

- For some recursive function calls, you can use `coder.ignoreConst` to force run-time recursion. See “Force Code Generator to Use Run-Time Recursion”.
- `coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations. It does not prevent other uses of the constant value during code generation.

### See Also

`coder.inline`

### Topics

“Force Code Generator to Use Run-Time Recursion”

“Compile-Time Recursion Limit Reached”

**Introduced in R2017a**

# coder.inline

**Package:** coder

Control inlining in generated code

## Syntax

```
coder.inline('always')  
coder.inline('never')  
coder.inline('default')
```

## Description

`coder.inline('always')` forces inlining on page 5-173 of the current function in the generated code. Place the `coder.inline` directive inside the function to which it applies. The code generator does not inline entry-point functions, inline functions into `parfor` loops, or inline functions called from `parfor` loops.

`coder.inline('never')` prevents inlining of the current function in the generated code. Prevent inlining when you want to simplify the mapping between the MATLAB source code and the generated code. You can disable inlining for all functions at the command line by using the `-O disable:inline` option of the `fiaccl` command.

`coder.inline('default')` uses internal heuristics to determine whether to inline the current function. Usually, the heuristics produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

## Examples

- “Prevent Function Inlining” on page 5-172
- “Use `coder.inline` in Control Flow Statements” on page 5-172

## 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 `coder.inline` 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 that you want to generate code for a division function used by a system with limited memory. To optimize memory use in the generated code, the `inline_division` function 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('Cannot divide by 0');
end

y = dividend / divisor;
```



## More About

### Inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.

### See Also

`fiaccl`

**Introduced in R2011a**

## coder.load

Load compile-time constants from MAT-file or ASCII file into caller workspace

### Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

### Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`coder.load` loads data at code generation time, also referred to as *compile time*. If you change the content of `filename` after you generate code, the change is not reflected in the behavior of the generated code.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename, '-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename, '-mat', var1, ..., varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

`S = coder.load(filename, '-mat', '-regex', expr1, ..., exprN)` treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

## Examples

### Load compile-time constants from MAT-file

Generate code for a function `edgeDetect1` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect1` uses `coder.load` to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
```

```
save sobel.mat k
```

Write the function `edgeDetect1`.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));
```

```
S = coder.load('sobel.mat', 'k');
H = conv2(double(originalImage), S.k, 'same');
V = conv2(double(originalImage), S.k', 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect1`.

```
codegen -report -config cfg edgeDetect1
```

codegen generates C code in the `codegen\lib\edgeDetect1` folder.

### Load compile-time constants from ASCII file

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];  
save sobel.dat k -ascii
```

Write the function `edgeDetect2`.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen  
assert(all(size(originalImage) <= [1024 1024]));  
assert(isa(originalImage, 'double'));  
assert(isa(threshold, 'double'));  
  
k = coder.load('sobel.dat');  
H = conv2(double(originalImage),k, 'same');  
V = conv2(double(originalImage),k', 'same');  
E = sqrt(H.*H + V.*V);  
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

codegen generates C code in the `codegen\lib\edgeDetect2` folder.

## Input Arguments

### **filename** — Name of file

character vector | string scalar

Name of file. `filename` must be a compile-time constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, %).

Example: `'myFile.mat'`

### **var1, ..., varN** — Names of variables to load

character vector | string scalar

Names of variables, specified as one or more character vectors or string scalars. Each variable name must be a compile-time constant. Use the `*` wildcard to match patterns.

Example: `coder.load('myFile.mat', 'A*')` loads all variables in the file whose names start with A.

### **expr1, ..., exprN** — Regular expressions indicating which variables to load

character vector | string scalar

Regular expressions indicating which variables to load specified as one or more character vectors or string scalars. Each regular expression must be a compile-time constant.

Example: `coder.load('myFile.mat', '-regexp', '^A')` loads only variables whose names begin with A.

## Output Arguments

### **S** — Loaded variables or data

structure array | m-by-n array

If `filename` is a MAT-file, `S` is a structure array.

If `filename` is an ASCII file, `S` is an m-by-n array of type `double`. `m` is the number of lines in the file and `n` is the number of values on a line.

## Limitations

- Arguments to `coder.load` must be compile-time constants.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S(i) = coder.load('myFile.mat')` is not allowed.
- You cannot use `save` to save workspace data to a file inside a function intended for code generation. The code generator does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save workspace data to a file.

## Tips

- `coder.load` loads data at compile time, not at run time. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function `load` to load run-time values.
- If the MAT-file contains unsupported constructs, use `coder.load(filename,var1,...,varN)` to load only the supported constructs.
- If you generate code in a MATLAB Coder project, the code generator practices incremental code generation for the `coder.load` function. When the MAT-file or ASCII file used by `coder.load` changes, the software rebuilds the code.

## See Also

`matfile` | `regexp` | `save`

## **Topics**

“Regular Expressions” (MATLAB)

**Introduced in R2013a**

## **coder.mexconfig**

**Package:** coder

Code acceleration configuration object

### **Syntax**

```
config_obj = coder.mexconfig
```

### **Description**

`config_obj = coder.mexconfig` creates a `coder.MexConfig` code generation configuration object for use with `fiaccl`, which generates a MEX function.

### **Output Arguments**

**config\_obj**

Code generation configuration object for use when generating MEX functions using `fiaccl`.

### **Examples**

Create a configuration object to disable run-time checks

```
cfg = coder.mexconfig
% Turn off Integrity Checks, Extrinsic Calls,
% and Responsiveness Checks
cfg.IntegrityChecks = false;
cfg.ExtrinsicCalls = false;
cfg.ResponsivenessChecks = false;
% Use fiaccl to generate a MEX function for file foo.m
fiaccl -config cfg foo
```



## See Also

`coder.ArrayType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` |  
`coder.MexConfig` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` |  
`coder.newtype` | `coder.resize` | `coder.typeof` | `fiaccel`

**Introduced in R2011a**

## **coder.newtype**

**Package:** coder

Create a coder.Type object

### **Syntax**

```
t= coder.newtype(numeric_class,sz,variable_dims)
t = coder.newtype(numeric_class,sz,variable_dims, Name,Value)
t = coder.newtype('constant',value)
t = coder.newtype('struct',struct_fields,sz,variable_dims)
t = coder.newtype('cell',cells,sz,variable_dims)
t = coder.newtype('embedded.fi',numerictype,sz,variable_dims,
Name,Value)
t = coder.newtype(enum_value,sz,variable_dims)
t = coder.newtype(class_name)
t = coder.newtype('string')
```

### **Description**

---

**Note** `coder.newtype` is an advanced function that you can use to control the `coder.Type` object. Consider using `coder.typeof` instead. `coder.typeof` creates a type from a MATLAB example.

---

`t= coder.newtype(numeric_class,sz,variable_dims)` creates a `coder.Type` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `variable_dims` is not specified, the dimensions of the type are fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to dimensions of the type that are not 1 or 0, which are fixed.

`t = coder.newtype(numeric_class,sz,variable_dims, Name,Value)` creates a `coder.Type` object with additional options specified by one or more `Name, Value` pair arguments.

`t = coder.newtype('constant',value)` creates a `coder.Constant` object representing a single value. Use this type to specify a value that must be treated as a constant in the generated code.

`t = coder.newtype('struct',struct_fields,sz,variable_dims)` creates a `coder.StructType` object for an array of structures that has the same fields as the scalar structure `struct_fields`. The structure array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`.

`t = coder.newtype('cell',cells,sz,variable_dims)` creates a `coder.CellType` object for a cell array that has the `cells` and cell types specified by `cells`. The cell array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`. You cannot change the number of cells or specify variable-size dimensions for a heterogeneous cell array.

`t = coder.newtype('embedded.fi',numeric_type,sz,variable_dims, Name,Value)` creates a `coder.FiType` object representing a set of fixed-point values with `numeric_type` and additional options specified by one or more `Name, Value` pair arguments.

`t = coder.newtype(enum_value,sz,variable_dims)` creates a `coder.Type` object representing a set of enumeration values of class `enum_value`.

`t = coder.newtype(class_name)` creates a `coder.ClassType` object for an object of the class `class_name`.

`t = coder.newtype('string')` creates a type for a string scalar. A string scalar contains one piece of text represented as a character vector. To specify the size of the character vector and whether the second dimension is variable-size, create a type for the character vector and assign it to the `Value` property of the string scalar type. For example, `t.Properties.Value = coder.newtype('char',[1 10], [0 1])` specifies that the character vector inside the string scalar is variable-size with an upper bound of 10.

## Examples

### Create Type for a Matrix

Create a type for a matrix of doubles for use in code generation.

```
t=coder.newtype('double',[2 3 4],[1 1 0])
```

```
t =
```

```
coder.PrimitiveType
    :2x:3x4 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed-size.

```
coder.newtype('double',[inf,3])
```

```
ans =
```

```
coder.PrimitiveType
    :infx3 double
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable-size with an upper bound 3.

```
coder.newtype('double',[inf, 3], [1 0])
```

```
% also returns
```

```
ans =
```

```
coder.PrimitiveType
    :infx3 double
% ':' indicates variable-size dimensions
```

### Create Type for a Structure

Create a structure type to use in code generation.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
coder.newtype('struct',struct('a',ta,'b',tb))
```

```
ans =
```

```

coder.StructType
  1x1 struct
    a: 1x1 int8
    b: :1x:2 double
% ':' indicates variable-size dimensions

```

### Create Type for a Cell Array

Create a cell array to use in code generation.

```

ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
coder.newtype('cell',{ta, tb})

% returns 1x2 heterogeneous cell
ans =

coder.CellType
  1x2 heterogeneous cell
    f1: 1x1 int8
    f2: :1x:2 double
% ':' indicates variable-size dimensions

```

### Create Type for a Constant

Create a new constant type to use in code generation.

```

k = coder.newtype('constant', 42)

k =

coder.Constant
  42

```

### Create a coder.EnumType Object

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

1. Define an enumeration `MyColors`. On the MATLAB path, create a file named `'MyColors'` containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2. Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors');
```

### Create a Fixed-Point Type

Create a fixed-point type for use in code generation.

The fixed-point type uses default `fimath` values

```
t = coder.newtype('embedded.fi', numerictype(1, 16, 15), [1 2])
```

```
t =
```

```
coder.FiType
    1x2 embedded.fi
           DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 16
           FractionLength: 15
```

### Create a Type for an Object

Create a type for an object to use in code generation.

1. Create this value class:

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
```

```

        obj.side = val;
    end
end

end
function a = calcarea(obj)
a = obj.side * obj.side;
end
end

```

2. Create a type for an object that has the same properties as mySquare.

```
t = coder.newtype('mySquare')
```

3. Change the type of the property side.

```
t.Properties.side = coder.typeof(int8(3))
```

```
t =
```

```

coder.ClassType
  1x1 mySquare
    side: 1x1 int8

```

### Create Type for a String Scalar

Create a type for a string scalar for use in code generation.

1. Create the string scalar type.

```
t = coder.newtype('string');
```

2. Specify the size.

```
t.Properties.Value = coder.newtype('char',[1, 10])
```

3. Make the string variable-size with an upper bound of 10.

```
t.Properties.Value = coder.newtype('char',[1, 10], [0, 1])
```

4. Make the string variable-size with no upper bound.

```
t.Properties.Value = coder.newtype('char',[1, inf])
```

## Input Arguments

### **numeric\_class** — Class of set of values represented by the type object

numeric (default)

Class of the set of values represented by the type object.

Example: `coder.newtype('double', [6, 3]);`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | struct | table | cell | function\_handle | categorical | datetime | duration | calendarDuration | fi  
Complex Number Support: Yes

### **struct\_fields** — Indicates fields in a new structure type

struct (default)

Scalar structure used to specify the fields in a new structure type.

Example: `coder.newtype('struct',struct('a',ta,'b',tb));`

Data Types: struct

### **cells** — Specify types of cells in a new cell array type

cell array (default)

Cell array of `coder.Type` objects that specify the types of the cells in a new cell array type.

Example: `coder.newtype('cell',{ta, tb});`

Data Types: cell

### **sz** — Dimension of type object

[1 1] (default) | row vector of integer values

Size vector specifying each dimension of type object. `sz` cannot change the number of cells for a heterogeneous cell array.

Example: `coder.newtype('int8',[1 2]);`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



Complex Number Support: Yes

### **class\_name — Name of the class**

character vector | string scalar

Name of class from which to create the `coder.ClassType`, specified as a character vector or string scalar. `class_name` must be the name of a value class.

Example: `coder.newtype('mySquare')`

Data Types: `char` | `string`

### **variable\_dims — Variable or fixed dimension**

row vector of logical values

`true` for dimensions for which `sz` specifies an upper bound of `inf`; `false` for all other dimensions.

Logical vector that specifies whether each dimension is variable size (`true`) or fixed size (`false`). You cannot specify variable-size dimensions for a heterogeneous cell array.

Example: `coder.newtype('char',[1, 10], [0, 1]);`

Data Types: `logical`

### **enum\_value — Enumeration values of class**

enum (default)

Enumeration values of class.

Example: `coder.newtype('MyColors');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`  
Complex Number Support: Yes

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `coder.newtype('embedded.fi', numericType(1, 16, 15), [1 2])`

### **complex — Type representing complex values**

true

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

### **fimath — Type representing fimath values**

numeric (default)

Specify local `fimath`. If `fimath` is not specified, uses default `fimath` values.

Use only with `t=coder.newtype('embedded.fi', numerictype,sz, variable_dims, Name, Value)`.

### **sparse — Type representing sparse data**

false (default)

Set `sparse` to `true` to create a `coder.Type` object representing sparse data. The type must support sparse data.

Not for use with `t=coder.newtype('embedded.fi', numerictype,sz, variable_dims, Name, Value)`

## **Output Arguments**

### **t — New type object**

`coder.Type` object

New `coder.Type` object.

## **Limitations**

- For sparse matrices, `coder.newtype` drops upper bounds for variable-size dimensions.

## Tips

- `coder.newtype` fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

For example, the following code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

```
t = coder.newtype('double',[1 10],1)
```

By contrast, the following code specifies a :1-by-:10 double. Both dimensions have a variable size.

```
t = coder.newtype('double',[1 10],[1 1])
```

---

**Note** For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

---

## Alternatives

`coder.typeof`

## See Also

`coder.ArrayType` | `coder.CellType` | `coder.EnumType` | `coder.FiType` |  
`coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.resize` |  
`fiaccel`

**Introduced in R2011a**

## **coder.nullcopy**

**Package:** coder

Declare uninitialized variables in code generation

### **Syntax**

`X = coder.nullcopy(A)`

### **Description**

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. The function preallocates memory for `X` without incurring the overhead of initializing memory. In code generation, the `coder.nullcopy` function declares uninitialized variables. In MATLAB, `coder.nullcopy` returns the input such that `X` is equal to `A`.

If `X` is a structure containing variable-sized arrays, then you must assign the size of each array. `coder.nullcopy` does not copy sizes of arrays or nested arrays from its argument to its result.

---

**Note** Before you use `X` in a function or a program, ensure that the data in `X` is completely initialized. Declaring a variable through `coder.nullcopy` without assigning all the elements of the variable results in nondeterministic program behavior. For more information, see “How to Eliminate Redundant Copies by Defining Uninitialized Variables”.

---

### **Examples**

#### **Declare Variables for Optimized Initialization**

Declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo %#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
```

Using `coder.nullcopy` with `zeros` lets you specify the size of vector `X` without initializing each element to zero.

## Input Arguments

### A — Variable to copy

scalar | vector | matrix | multidimensional array

Variable to copy, specified as a scalar, vector, matrix, or multidimensional array.

Example: `coder.nullcopy(A)`;

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

Complex Number Support: Yes

## Limitations

You cannot use `coder.nullcopy` on sparse matrices, structures, cell arrays, or classes that contain sparse matrices.

## See Also

### Topics

“Eliminate Redundant Copies of Variables in Generated Code”

**Introduced in R2011a**

# coder.PrimitiveType class

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of logical, numeric, or char arrays

## Description

Specifies the set of logical, numeric, or char values that the generated code should accept. Supported classes are `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `char`, and `logical`. Use only with the `fiaccl -args` option. Do not pass as an input to a generated MEX function.

## Construction

`t=coder.typeof(v)` creates a `coder.PrimitiveType` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical or char.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is `[]`, the (upper bound) sizes of `v` remain unchanged. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to bounded dimensions that are not 1 or 0 (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims)` creates a `coder.PrimitiveType` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type that are not 1 or 0 (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims, Name, Value)` creates a `coder.PrimitiveType` object with additional options specified by one or more `Name, Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. Specify `Name` as character vector or string scalar. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**v**

Input that is not a `coder.Type` object

**sz**

Size for corresponding dimension of type object. Size must be a valid size vector.

**Default:** [1 1] for `coder.newtype`

**variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

**Default:** `false(size(sz)) | sz==Inf` for `coder.newtype`

**numeric\_class**

Class of type object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**complex**

Set `complex` to `true` to create a `coder.PrimitiveType` object that can represent complex values. The type must support complex data.

**Default:** `false`



**sparse**

Set `sparse` to `true` to create a `coder.PrimitiveType` object representing sparse data. The type must support sparse data.

**Default:** `false`

## Properties

**ClassName**

Class of values in this set

**Complex**

Indicates whether the values in this set are real (`false`) or complex (`true`)

**SizeVector**

The upper-bound size of arrays in this set.

**Sparse**

Indicates whether the values in this set are sparse arrays (`true`)

**VariableDims**

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

Create a `coder.PrimitiveType` object.

```
z = coder.typeof(0,[2 3 4],[1 1 0]) % returns double :2x:3x4  
% ':' indicates variable-size dimensions
```

### See Also

[coder.ArrayType](#) | [coder.ClassType](#) | [coder.Type](#) | [coder.newtype](#) |  
[coder.resize](#) | [coder.typeof](#) | [fiaccl](#)

**Introduced in R2011a**

# coder.resize

**Package:** coder

Resize `coder.Type` object

## Syntax

```
t_out = coder.resize(t,sz)
t_out = coder.resize(t,sz,variable_dims)
t_out = coder.resize(t,[],variable_dims)
t_out = coder.resize(t,sz,variable_dims,Name,Value)
t_out = coder.resize(t,'sizelimits',limits)
```

## Description

`t_out = coder.resize(t,sz)` resizes `t` to have size `sz`.

`t_out = coder.resize(t,sz,variable_dims)` returns a modified copy of `coder.Type` `t` with (upper-bound) size `sz` and variable dimensions `variable_dims`. If `variable_dims` or `sz` are scalars, the function applies the scalars to all dimensions of `t`. By default, `variable_dims` does not apply to dimensions where `sz` is 0 or 1, which are fixed. Use the 'uniform' option to override this special case. The `coder.resize` function ignores `variable_dims` for dimensions with size `inf`. These dimensions are variable size. `t` can be a cell array of types, in which case, `coder.resize` resizes all elements of the cell array.

`t_out = coder.resize(t,[],variable_dims)` changes `t` to have variable dimensions `variable_dims` while leaving the size unchanged.

`t_out = coder.resize(t,sz,variable_dims,Name,Value)` resizes `t` by using additional options specified by one or more `Name, Value` pair arguments.

`t_out = coder.resize(t,'sizelimits',limits)` resizes `t` with dimensions becoming variable based on the `limits` vector. When the size `S` of a dimension is greater than or equal to the first threshold defined in `limits`, the dimension becomes variable

size with upper bound *S*. When the size *S* of a dimension is greater than or equal to the second threshold defined in `limits`, the dimension becomes an unbounded variable size.

## Examples

### Change Fixed-Size Array to an Unbounded, Variable-Size Array

Change a fixed-size array to an unbounded, variable-size array.

```
t = coder.typeof(ones(3,3))
t =
coder.PrimitiveType
    3×3 double

coder.resize(t,inf)

ans =
coder.PrimitiveType
    :inf×:inf double
% ':' indicates variable-size dimensions
```

### Change Fixed-Size Array to a Bounded, Variable-Size Array

Change a fixed-size array to a bounded, variable-size array.

```
t = coder.typeof(ones(3,3))
t =
coder.PrimitiveType
    3×3 double

coder.resize(t,[4 5],1)

ans =
coder.PrimitiveType
```

```
      :4×5 double  
% ':' indicates variable-size dimensions
```

### Resize Structure Field

Resize a structure field.

```
ts = coder.typeof(struct('a',ones(3, 3)))  
  
ts =  
  
coder.StructType  
  1×1 struct  
    a: 3×3 double  
  
coder.resize(ts,[5, 5],'recursive',1)  
  
ans =  
  
coder.StructType  
  5×5 struct  
    a: 5×5 double
```

### Resize Cell Array

Resize a cell array.

```
tc = coder.typeof({1 2 3})  
  
tc =  
  
coder.CellType  
  1×3 homogeneous cell  
    base: 1×1 double  
  
coder.resize(tc,[5, 5],'recursive',1)  
  
ans =  
  
coder.CellType
```

```
5x5 homogeneous cell
base: 1x1 double
```

### Change Fixed-Sized Array to Variable-Size Based on Bounded and Unbounded Thresholds

Change a fixed-sized array to a variable size based on bounded and unbounded thresholds.

```
t = coder.typeof(ones(100,200))
t =

coder.PrimitiveType
    100x200 double

coder.resize(t,'sizelimits',[99 199])

ans =

coder.PrimitiveType
    :100x:inf double
% ':' indicates variable-size dimensions
```

## Input Arguments

### **limits** — Vector that defines the threshold

row vector of integer values

A row vector of variable-size thresholds. If the value of `limits` is scalar, the threshold gets scalar-expanded. If the size `sz` of a dimension of `t` is greater than or equal to the first threshold, the dimension becomes variable size with upper bound `sz`. If the size `sz` of a dimension of `t` is greater than or equal to the second threshold, the dimension becomes an unbounded variable size.

Example: `coder.resize(t,'sizelimits',[99 199]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **sz** — New size for object type

row vector of integer values

New size for `coder.Type` object, `t_out`

Example: `coder.resize(t,[3,4]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **t — coder.Type object that you want to resize**

`coder.Type` object

If `t` is a `coder.CellType` object, the `coder.CellType` object must be homogeneous.

Example: `coder.resize(t,inf);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`  
Complex Number Support: Yes

### **variable\_dims — Variable or fixed dimension**

row vector of logical values

Specify whether each dimension of `t_out` is fixed size or variable size.

Example: `coder.resize(t,[4 5],1);`

Data Types: `logical`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `coder.resize(t,[5, 5], 'recursive', 1);`

### **recursive — Resize t and all types contained within it**

`false` (default) | `true`

Setting `recursive` to `true` resizes `t` and all types contained within it.

Data Types: `logical`

### **uniform — Resize `t` without applying the heuristic for dimensions of size one**

`false` (default) | `true`

Setting `uniform` to `true` resizes `t` but does not apply the heuristic for dimensions of size one.

Data Types: `logical`

## Output Arguments

### **`t_out` — Resized type object**

`coder.Type` object

Resized `coder.Type` object

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`  
Complex Number Support: Yes

## Limitations

- For sparse matrices, `coder.resize` drops the upper bounds for variable-size dimensions.

## See Also

`coder.newtype` | `coder.typeof` | `fiaccel`

**Introduced in R2011a**



# coder.screener

**Package:** coder

Determine if function is suitable for code generation

## Syntax

```
coder.screener(fcn)
coder.screener(fcn_1, ..., fcn_n)
```

## Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function, `fcn`. It identifies unsupported functions and language features as code generation compliance issues. It displays the code generation compliance issues in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions. It does not analyze MathWorks functions. It is possible that `coder.screener` does not detect all code generation issues. Under certain circumstances, it is possible that `coder.screener` reports false errors.

`coder.screener` does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

`coder.screener(fcn_1, ..., fcn_n)` analyzes entry-point functions (`fcn_1, ..., fcn_n`).

## Examples

### Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, `foo1`, and the function `foo2` that `foo1` calls.

Write the function `foo2` and save it in the file `foo2.m`.

```
function out = foo2(in)
    out = eval(in);
end
```

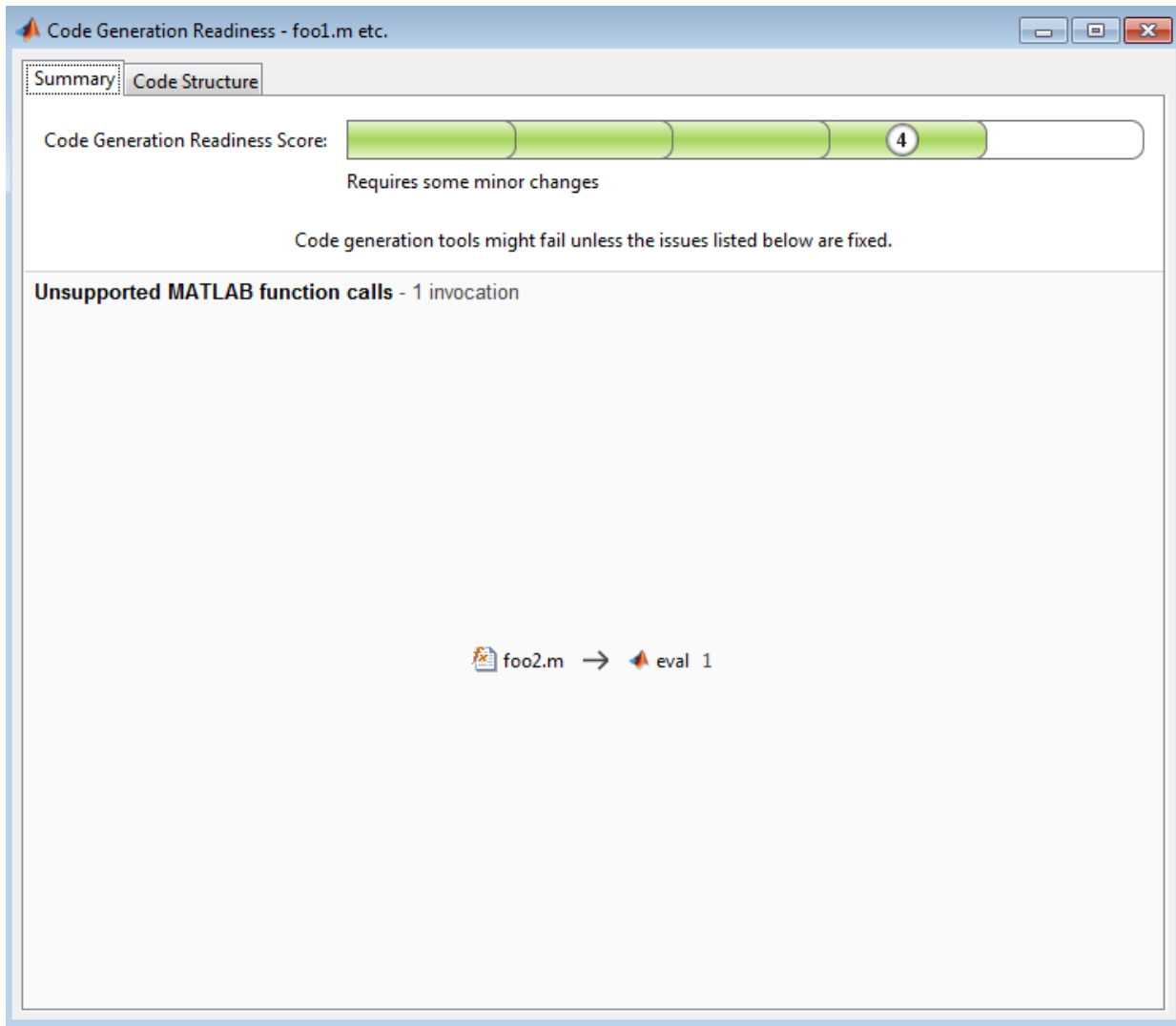
Write the function `foo1` that calls `foo2`. Save `foo1` in the file `foo1.m`.

```
function out = foo1(in)
    out = foo2(in);
    disp(out);
end
```

Analyze `foo1`.

```
coder.screener('foo1')
```

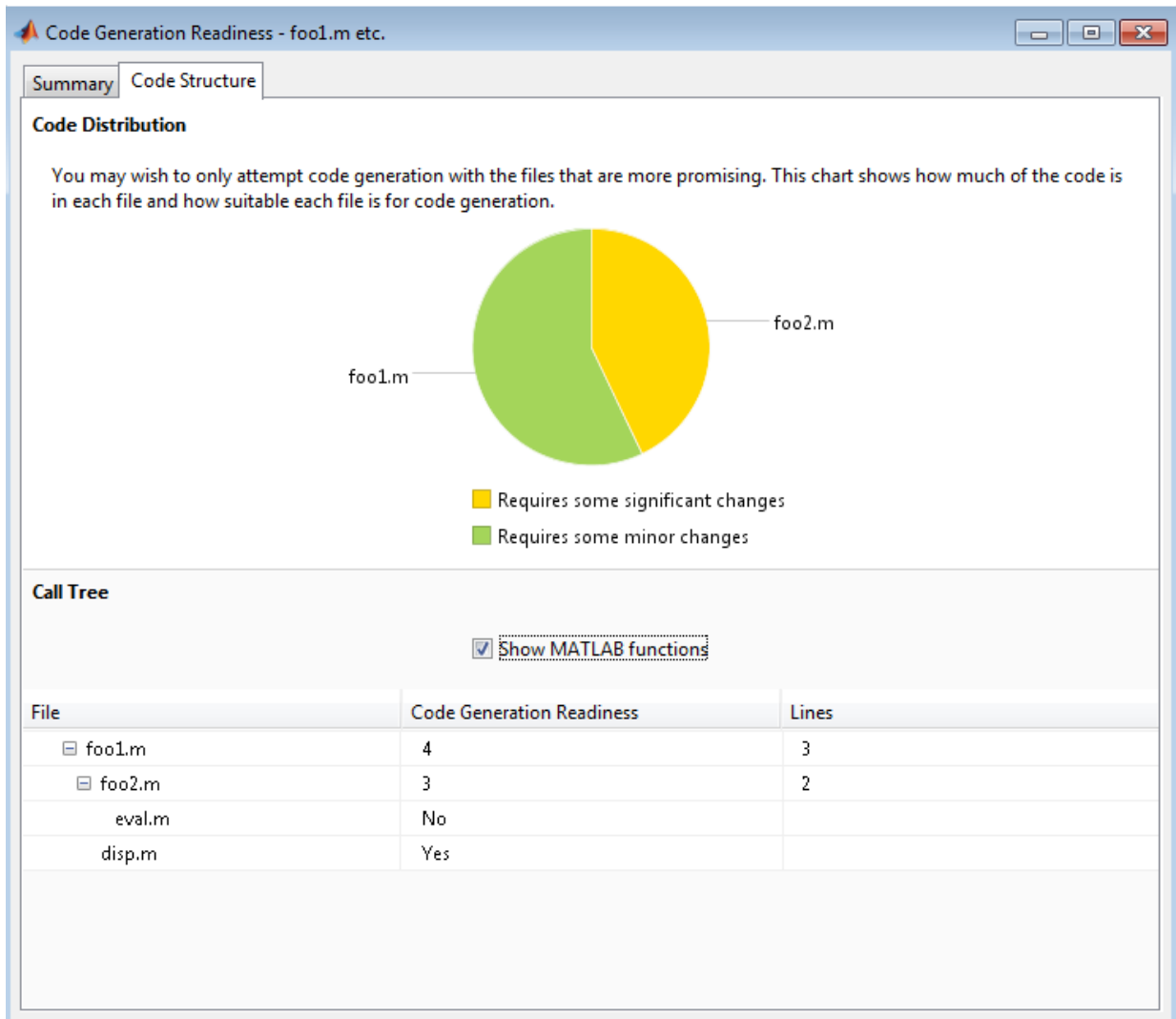
The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.



In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that it requires significant changes.
- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1-5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool does not detect issues.
- Displays a call tree.



The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function, which code generation does not support. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
function out = foo2(in)
    coder.extrinsic('eval');
```

```
    out = eval(in);  
end
```

Rerun the code generation readiness tool.

```
coder.screener('foo1')
```

The report no longer flags that code generation does not support the `eval` function. When you generate a MEX function for `foo1`, the code generator dispatches `eval` to MATLAB for execution. For standalone code generation, the code generator does not generate code for `eval`.

### Identify Unsupported Data Types

The `coder.screener` function identifies MATLAB data types that code generation does not support.

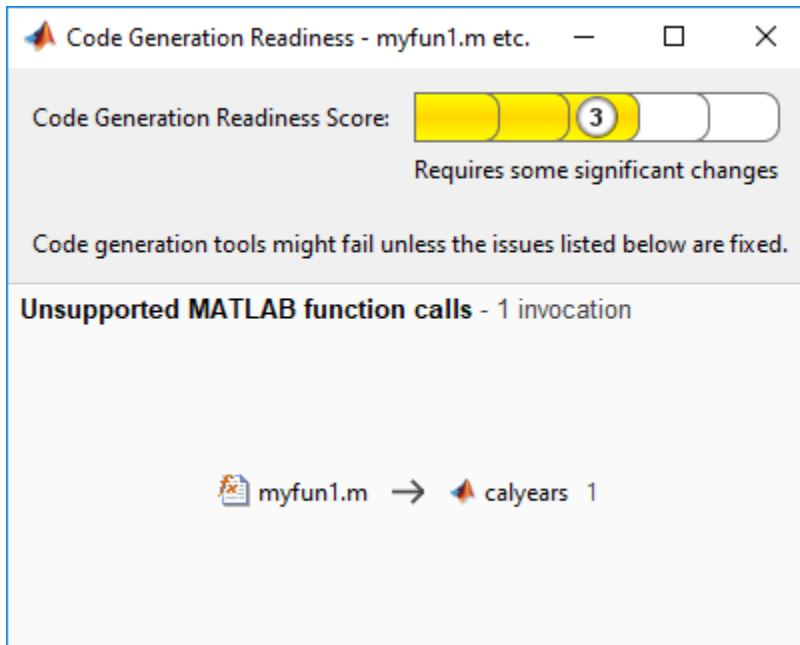
Write the function `myfun` that contains a MATLAB calendar duration array data type.

```
function out = myfun1(A)  
out = calyears(A);  
end
```

Analyze `myfun`.

```
coder.screener('myfun1');
```

The code generation readiness report indicates that the `calyears` data type is not supported for code generation.



The report assigns myfun1 a code readiness score of 3. Before generating code, you must fix the reported issues.

## Input Arguments

### **fcn** — Name of entry-point function

character vector | string scalar

Name of entry-point MATLAB function that you want to analyze. Specify as a character vector or a string scalar.

Example: `coder.screener('myfun1');`

Data Types: `char` | `string`

### **fcn\_1, ..., fcn\_n** — List of names of entry-point functions

character vector | string scalar

Comma-separated list of names of entry-point MATLAB functions that you want to analyze. Specify as character vectors or string scalars.

Example: `coder.screener('myfun1','myfun2');`

Data Types: `char` | `string`

### Tips

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.
- `coder.screener` does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

### Alternatives

- “Run the Code Generation Readiness Tool From the Current Folder Browser”

### See Also

`fiaccel`

### Topics

“Functions Supported for Code Acceleration or C Code Generation”

“Code Generation Readiness Tool”

**Introduced in R2012b**



# coder.StructType class

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of MATLAB structure arrays

## Description

Specifies the set of structure arrays that the generated code should accept. Use only with the `fiaccl -args` option. Do not pass as an input to a generated MEX function.

## Construction

`t=coder.typeof(struct_v)` creates a `coder.StructType` object for a structure with the same fields as the scalar structure `struct_v`.

`t=coder.typeof(struct_v, sz, variable_dims)` returns a modified copy of `coder.typeof(struct_v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is `[]`, the (upper bound) sizes of `struct_v` remain unchanged. If the `variable_dims` input parameter is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the bounded dimensions that are not 1 or 0 (which are assumed to be fixed).

`t=coder.newtype('struct', struct_v, sz, variable_dims)` creates a `coder.StructType` object for an array of structures with the same fields as the scalar structure `struct_v` and (upper bound) size `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type, except if the dimension is 1 or 0, which is assumed to be fixed.

## Input Arguments

### **struct\_v**

Scalar structure used to specify the fields in a new structure type.

### **sz**

Size vector specifying each dimension of type object.

**Default:** [1 1] for `coder.newtype`

### **variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

**Default:** `false(size(sz)) | sz==Inf` for `coder.newtype`

## Properties

### **Alignment**

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder® license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary so it will not be matched by CRL functions that require alignment.

Alignment must be either -1 or a power of 2 that is no more than 128.

### **ClassName**

Class of values in this set.

### **Extern**

Whether the structure type is externally defined.

## Fields

A structure giving the `coder.Type` of each field in the structure.

## HeaderFile

If the structure type is externally defined, name of the header file that contains the external definition of the structure, for example, "mystruct.h".

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty character vector or string scalar.

## SizeVector

The upper-bound size of arrays in this set.

## VariableDims

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

# Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
% coder.StructType
%    1x1 struct
%      a:   :3x:5 double
```

```
%      b: 3x3 double  
% ':' indicates variable-size dimensions
```

### See Also

[coder.ArrayType](#) | [coder.ClassType](#) | [coder.Constant](#) | [coder.EnumType](#) |  
[coder.FiType](#) | [coder.PrimitiveType](#) | [coder.Type](#) | [coder.newtype](#) |  
[coder.resize](#) | [coder.typeof](#) | [fiaccel](#)

**Introduced in R2011a**

# coder.target

Determine if code generation target is specified target

## Syntax

```
tf = coder.target(target)
```

## Description

`tf = coder.target(target)` returns true (1) if the code generation target is `target`. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

## Examples

### Use `coder.target` to Parametrize a MATLAB Function

Parametrize a MATLAB function so that it works in MATLAB or in generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
    %#codegen
    y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval(y)
    %#codegen
    % Check the target. Do not use coder.ceval if callmyabsval is
    % executing in MATLAB
    if coder.target('MATLAB')
        % Executing in MATLAB, call function myabsval
        y = myabsval(y);
    else
        % add the required include statements to generated function code
        coder.updateBuildInfo('addIncludePaths', '${START_DIR}\codegen\lib\myabsval');
        coder.cinclud('myabsval_initialize.h');
        coder.cinclud('myabsval.h');
        coder.cinclud('myabsval_terminate.h');

        % Executing in the generated code.
        % Call the initialize function before calling the
        % C function for the first time
        coder.ceval('myabsval_initialize');

        % Call the generated C library function myabsval
        y = coder.ceval('myabsval', y);

        % Call the terminate function after
        % calling the C function for the last time
        coder.ceval('myabsval_terminate');
    end
end
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects', 'myabsval.lib', '${START_DIR}\codegen\lib\myabsval\myabsval.lib');
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =
```

```
    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =
```

```
    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

## Input Arguments

### target — code generation target

```
'MATLAB' | 'MEX' | 'Sfun' | 'Rtw' | 'HDL' | 'Custom'
```

Code generation target, specified as a character vector or a string scalar. Specify one of these targets.

'MATLAB'	Running in MATLAB (not generating code)
'MEX'	Generating a MEX function
'Sfun'	Simulating a Simulink model
'Rtw'	Generating a LIB, DLL, or EXE target
'HDL'	Generating an HDL target
'Custom'	Generating a custom target

```
Example: tf = coder.target('MATLAB')
```

```
Example: tf = coder.target("MATLAB")
```

## **See Also**

**Introduced in R2011a**



# coder.Type class

**Package:** coder

Represent set of MATLAB values

## Description

Specifies the set of values that the generated code should accept. Use only with the `fiaccel -args` option. Do not pass as an input to a generated MEX function.

## Construction

`coder.Type` is an abstract class, and you cannot create instances of it directly. You can create `coder.Constant`, `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, `coder.StructType`, and `coder.CellType` objects that are derived from this class.

## Properties

**ClassName**

Class of values in this set

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

`coder.ArrayType` | `coder.CellType` | `coder.Constant` | `coder.EnumType` |  
`coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.newtype` |  
`coder.resize` | `coder.typeof` | `fiaccel`

**Introduced in R2011a**

# coder.typeof

**Package:** coder

Create `coder.Type` object to represent the type of an entry-point function input

## Syntax

```
type_obj = coder.typeof(v)
type_obj = coder.typeof(v,sz,variable_dims)
type_obj = coder.typeof(type_obj)
```

## Description

`type_obj = coder.typeof(v)` creates an object that is derived from `coder.Type` to represent the type of `v` for code generation. Use `coder.typeof` to specify only input parameter types. For example, use it with the `fiaccl` function `-args` option. Do not use it in MATLAB code from which you intend to generate a MEX function.

`type_obj = coder.typeof(v,sz,variable_dims)` returns a modified copy of `type_obj = coder.typeof(v)` with upper bound size specified by `sz` and variable dimensions specified by `variable_dims`.

`type_obj = coder.typeof(type_obj)` returns `type_obj` itself.

## Examples

### Create Type for a Matrix

Create a type for a simple fixed-size 5x6 matrix of doubles.

```
coder.typeof(ones(5,6))
```

```
ans =
```

```
coder.PrimitiveType
    5x6 double
```

```
coder.typeof(0,[5 6])
```

```
ans =
```

```
coder.PrimitiveType
    5x6 double
```

Create a type for a variable-size matrix of doubles.

```
coder.typeof(ones(3,3),[],1)
```

```
ans =
```

```
coder.PrimitiveType
    :3x:3 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix with fixed-size and variable-size dimensions.

```
coder.typeof(0,[2,3,4],[1 0 1])
```

```
ans =
```

```
coder.PrimitiveType
    :2x3x:4 double
```

```
coder.typeof(10,[1 5],1)
```

```
ans =
```

```
coder.PrimitiveType
    1x:5 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size.

```
coder.typeof(10,[inf,3])
```

```
ans =
```

```
coder.PrimitiveType
    :infx3 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size that has an upper bound of 3.

```
coder.typeof(10,[inf,3],[0 1])
ans =
```

```
coder.PrimitiveType
    :inf×:3 double
```

Convert a fixed-size matrix to a variable-size matrix.

```
coder.typeof(ones(5,5),[],1)
ans =
```

```
coder.PrimitiveType
    :5×:5 double
% ':' indicates variable-size dimensions
```

### Create Type for a Structure

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
ans =
```

```
coder.StructType
    1×1 struct
        a: :3×:5 double
        b: 3×3 double
% ':' indicates variable-size dimensions
```

Create a nested structure (a structure as a field of another structure).

```
S = struct('a',double(0),'b',single(0));
SuperS.x = coder.typeof(S);
SuperS.y = single(0);
coder.typeof(SuperS)
ans =
```

```
coder.StructType
  1x1 struct
    x: 1x1 struct
      a: 1x1 double
      b: 1x1 single
    y: 1x1 single
```

Create a structure containing a variable-size array of structures as a field.

```
S = struct('a',double(0),'b',single(0));
SuperS.x = coder.typeof(S,[1 inf],[0 1]);
SuperS.y = single(0);
coder.typeof(SuperS)
```

ans =

```
coder.StructType
  1x1 struct
    x: 1x:inf struct
      a: 1x1 double
      b: 1x1 single
    y: 1x1 single
% ':' indicates variable-size dimensions
```

### Create Type for a Cell Array

Create a type for a homogeneous cell array with a variable-size field.

```
a = coder.typeof(0,[3 5],1);
b = magic(3);
coder.typeof({a b})
```

ans =

```
coder.CellType
  1x2 homogeneous cell
    base: :3x:5 double
% ':' indicates variable-size dimensions
```

Create a type for a heterogeneous cell array.

```
a = coder.typeof('a');
b = coder.typeof(1);
coder.typeof({a b})
```

```
ans =
coder.CellType
  1×2 heterogeneous cell
    f1: 1×1 char
    f2: 1×1 double
```

Create a variable-size homogeneous cell array type from a cell array that has the same class but different sizes.

1. Create a type for a cell array that contains two character vectors with different sizes. The cell array type is heterogeneous.

```
coder.typeof({'aa', 'bbb'})
ans =
```

```
coder.CellType
  1×2 heterogeneous cell
    f1: 1×2 char
    f2: 1×3 char
```

2. Create a type by using the same cell array input. This time, specify that the cell array type has variable-size dimensions. The cell array type is homogeneous.

```
coder.typeof({'aa', 'bbb'}, [1,10], [0,1])
ans =
coder.CellType
  1×:10 locked homogeneous cell
    base: 1×:3 char
% ':' indicates variable-size dimensions
```

## Create Type for a Value Class Object

Change a fixed-size array to a bounded, variable-size array.

Create a type for a value class object.

1. Create this value class:

```
classdef mySquare
    properties
```

```
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

2. Create an object of mySquare.

```
sq_obj = coder.typeof(mySquare(4))
```

```
sq_obj =
```

```
coder.ClassType
    1x1 mySquare
        side: 1x1 double
```

3. Create a type for an object that has the same properties as sq\_obj.

```
t = coder.typeof(sq_obj)
```

```
t =
```

```
coder.ClassType
    1x1 mySquare
        side: 1x1 double
```

Alternatively, you can create the type from the class definition:

```
t = coder.typeof(mySquare(4))
```

```
t =
```

```
coder.ClassType
    1x1 mySquare
        side: 1x1 double
```



## Create Type for a String Scalar

Define a string scalar. For example:

```
s = "mystring";
```

Create a type from `s`.

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

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

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

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

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

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

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

## Input Arguments

### **v** — Set of values representing input parameter types

numeric array | character vector | string | struct | cell array

`v` can be a MATLAB numeric, logical, char, enumeration, or fixed-point array. `v` can also be a cell array, structure, or value class that contains the previous types.

When `v` is a cell array whose elements have the same classes but different sizes, if you specify variable-size dimensions, `coder.typeof` creates a homogeneous cell array type. If the elements have different classes, `coder.typeof` reports an error.

Example: `coder.typeof(ones(5,6));`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | logical | char | string | struct | table | cell |  
function\_handle | categorical | datetime | duration | calendarDuration | fi  
Complex Number Support: Yes

**sz — Dimension of type object**

row vector of integer values

Size vector specifying each dimension of type object.

If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds of `v` do not change.

If size is not specified, `sz` takes the default dimension of `v`.

Example: `coder.typeof(0,[5,6]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**variable\_dims — Variable or fixed dimension**

row vector of logical values

Logical vector that specifies whether each dimension is variable size (`true`) or fixed size (`false`). For a cell array, if the elements have different classes, you cannot specify variable-size dimensions.

If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed.

A scalar `variable_dims` applies to all dimensions. However, if `variable_dims` is `1`, the size of a singleton dimension remains fixed.

Example: `coder.typeof(0,[2,3,4],[1 0 1]);`

Data Types: `logical`

**type\_obj — Type object**

`coder.Type` object

`coder.Type` object to represent the type of `v` for code generation.

Example: `type_obj = coder.typeof(ones(5,6));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`  
Complex Number Support: Yes

## Output Arguments

### type\_obj — Type object

coder.Type object

coder.Type object to represent the type of v for code generation.

Example: `type_obj = coder.typeof(ones(5,6));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`  
 Complex Number Support: Yes

## Limitations

- For sparse matrices, `coder.typeof` drops upper bounds for variable-size dimensions.

## Tips

- `coder.typeof` fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

For example, the following code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

```
t = coder.typeof(5,[1 10],1)
```

By contrast, this code specifies a :1-by-:10 double. Both dimensions have a variable size.

```
t = coder.typeof(5,[1 10],[1 1])
```

---

**Note** For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

---

- If you are already specifying the type of an input variable by using a type function, do not use `coder.typeof` unless you also want to specify the size. For instance, instead of `coder.typeof(single(0))`, use the syntax `single(0)`.

- For cell array types, `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous.

If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type.

If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type.

For some cell arrays, classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a 1x2 heterogeneous type where the first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `coder.CellType` `makeHomogeneous` or `makeHeterogeneous` methods to make a type with the classification that you want. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as heterogeneous and homogeneous. You cannot later use one of these methods to create a copy that has a different classification.

## See Also

`coder.ArrayType` | `coder.CellType` | `coder.EnumType` | `coder.FiType` |  
`coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` |  
`coder.resize` | `fiaccel`

## Topics

“Define Input Properties by Example at the Command Line”

“Specify Cell Array Inputs at the Command Line”

“Specify Objects as Inputs”

“Define String Scalar Inputs”

**Introduced in R2011a**

## **coder.unroll**

Unroll for-loop by making a copy of the loop body for each loop iteration

### **Syntax**

```
coder.unroll()  
coder.unroll(flag)
```

### **Description**

`coder.unroll()` unrolls a for-loop. The `coder.unroll` call must be on a line by itself immediately preceding the for-loop that it unrolls.

Instead of producing a for-loop in the generated code, loop unrolling produces a copy of the for-loop body for each loop iteration. In each iteration, the loop index becomes constant. To unroll a loop, the code generator must be able to determine the bounds of the for-loop.

For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can increase code generation time significantly and generate inefficient code.

`coder.unroll` is ignored outside of code generation.

`coder.unroll(flag)` unrolls a for-loop if `flag` is `true`. `flag` is evaluated at code generation time. The `coder.unroll` call must be on a line by itself immediately preceding the for-loop that it unrolls.

### **Examples**

#### **Unroll a for-loop**

To produce copies of a for-loop body in the generated code, use `coder.unroll`.

In one file, write the entry-point function `call_getrand` and a local function `getrand`. `getrand` unrolls a `for`-loop that assigns random numbers to an `n`-by-1 array. `call_getrand` calls `getrand` with the value 3.

```
function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
coder.unroll();
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```
codegen -config:lib call_getrand -report
```

In the generated code, the code generator produces a copy of the `for`-loop body for each of the three loop iterations.

```
static void getrand(double y[3])
{
    y[0] = b_rand();
    y[1] = b_rand();
    y[2] = b_rand();
}
```

## Control for-loop Unrolling with Flag

Control loop unrolling by using `coder.unroll` with the `flag` argument.

In one file, write the entry-point function `call_getrand_unrollflag` and a local function `getrand_unrollflag`. When the number of loop iterations is less than 10, `getrand_unrollflag` unrolls the `for`-loop. `call_getrand` calls `getrand` with the value 50.

```
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
```

```

end

function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
coder.unroll(unrollflag)
for i = 1:n
    y(i) = rand();
end
end

```

Generate a static library.

```
codegen -config:lib call_getrand_unrollflag -report
```

The number of iterations is not less than 10. Therefore, the code generator does not unroll the for-loop. It produces a for-loop in the generated code.

```

static void getrand_unrollflag(double y[50])
{
    int i;
    for (i = 0; i < 50; i++) {
        y[i] = b_rand();
    }
}

```

## Use Legacy Syntax to Unroll for-Loop

```

function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
for i = coder.unroll(1:n)
    y(i) = rand();
end
end

```

## Use Legacy Syntax to Control for-Loop Unrolling

```
function z = call_getrand_unrollflag
    %#codegen
    z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
    coder.inline('never');
    unrollflag = n < 10;
    y = zeros(n, 1);
    for i = coder.unroll(1:n, unrollflag)
        y(i) = rand();
    end
end
```

## Input Arguments

### **flag** — Indicates whether to unroll the for-loop

true (default) | false

When `flag` is true, the code generator unrolls the for-loop. When `flag` is false, the code generator produces a for-loop in the generated code. `flag` is evaluated at code generation time.

## Tips

- Sometimes, the code generator unrolls a for-loop even though you do not use `coder.unroll`. For example, if a for-loop indexes into a heterogeneous cell array or into `varargin` or `varargout`, the code generator unrolls the loop. By unrolling the loop, the code generator can determine the value of the index for each loop iteration. The code generator uses heuristics to determine when to unroll a for-loop. If the heuristics fail to identify that unrolling is warranted, or if the number of loop iterations exceeds a limit, code generation fails. In these cases, you can force loop unrolling by using `coder.unroll`. See “Nonconstant Index into `varargin` or `varargout` in a for-Loop”.



## **See Also**

`coder.inline`

## **Topics**

“Nonconstant Index into varargin or varargout in a for-Loop”

**Introduced in R2011a**

## **coder.versize**

**Package:** coder

Declare variable-size data

### **Syntax**

```
coder.versize(varName1, ..., varNameN)
coder.versize(varName1, ..., varNameN, ubounds)
coder.versize(varName1, ..., varNameN, ubounds, dims)
```

### **Description**

`coder.versize(varName1, ..., varNameN)` declares that the variables named `varName1, ..., varNameN` have a variable size. The declaration instructs the code generator to allow the variables to change size during execution of the generated code. With this syntax, you do not specify the upper bounds of the dimensions of the variables or which dimensions can change size. The code generator computes the upper bounds. All dimensions, except singleton dimensions on page 5-245, are allowed to change size.

Use `coder.versize` according to these restrictions and guidelines:

- Use `coder.versize` inside a MATLAB function intended for code generation.
- The `coder.versize` declaration must precede the first use of a variable. For example:

```
...
x = 1;
coder.versize('x');
disp(size(x));
...
```

- Use `coder.versize` to declare that an output argument has a variable size or to address size mismatch errors. Otherwise, to define variable-size data, use the methods described in “Define Variable-Size Data for Code Generation”.

---

**Note** For MATLAB Function blocks, to declare variable-size input or output signals, use the Ports and Data Manager. See “Declare Variable-Size Inputs and Outputs” (Simulink). If you provide upper bounds in a `coder.versize` declaration, the upper bounds must match the upper bounds in the Ports and Data Manager.

---

For more restrictions and guidelines, see “Limitations” on page 5-243 and “Tips” on page 5-245.

`coder.versize(varName1, ..., varNameN, ubounds)` also specifies an upper bound for each dimension of the variables. All variables must have the same number of dimensions. All dimensions, except singleton dimensions on page 5-245, are allowed to change size.

`coder.versize(varName1, ..., varNameN, ubounds, dims)` also specifies an upper bound for each dimension of the variables and whether each dimension has a fixed size or a variable size. If a dimension has a fixed size, then the corresponding `ubound` element specifies the fixed size of the dimension. All variables have the same fixed-size dimensions and the same variable-size dimensions.

## Examples

### Address Size Mismatch Error by Using `coder.versize`

After a variable is used (read), changing the size of the variable can cause a size mismatch error. Use `coder.versize` to specify that the size of the variable can change.

Code generation for the following function produces a size mismatch error because `x = 1:10` changes the size of the second dimension of `x` after the line `y = size(x)` that uses `x`.

```
function [x,y] = useversize(n)
%#codegen
x = 1;
y = size(x);
if n > 10
    x = 1:10;
end
```

To declare that `x` can change size, use `coder.versize`.

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
coder.varsize('x');
y = size(x);
if n > 10
    x = 1:10;
end
```

If you remove the line `y = size(x)`, you no longer need the `coder.varsize` declaration because `x` is not used before its size changes.

### Declare Variable-Size Array with Upper Bounds

Specify that `A` is a row vector whose second dimension has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[1 20]);
...
end
```

When you do not provide `dims`, all dimensions, except singleton dimensions, have a variable size.

### Declare Variable-Size Array with a Mix of Fixed and Variable Dimensions

Specify that `A` is an array whose first dimension has a fixed size of three and whose second dimension has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[3 20], [0 1] );
...
end
```

## Declare Variable-Size Structure Fields

If a structure field belongs to an array of structures, use colon (:) as the index expression to make the field variable-size for all elements of the array.

In this function, the statement `coder.versize('data(:).values')` declares that the field `values` inside each element of `data` has a variable size.

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

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

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

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

## Declare Variable-Size Cell Array

Specify that cell array `C` has a fixed-size first dimension and variable-size second dimension with an upper bound of three. The `coder.versize` declaration must precede the first use of `C`.

```
...
C = {1 [1 2]};
coder.versize('C', [1 3], [0 1]);
y = C{1};
...
end
```

Without the `coder.versize` declaration, `C` is a heterogeneous cell array whose elements have the same class and different sizes. With the `coder.versize` declaration, `C` is a

homogeneous cell array whose elements have the same class and maximum size. The first dimension of each element is fixed at 1. The second dimension of each element has a variable size with an upper bound of 3.

### **Declare That a Cell Array Has Variable-Size Elements**

Specify that the elements of cell array `C` are vectors with a fixed-size first dimension and variable-size second dimension with an upper bound of 5.

```
...  
C = {1 2 3};  
coder.varsize('C{:}', [1 5], [0 1]);  
C = {1, 1:5, 2:3};  
...
```

## **Input Arguments**

**varName1, ..., varNameN** — Names of variables to declare as having a variable size

character vectors | string scalars

Names of variables to declare as having a variable size, specified as one or more character vectors or string scalars.

Example: `coder.varsize('x','y')`

**ubounds** — Upper bounds for array dimensions

[] (default) | vector of integer constants

Upper bounds for array dimensions, specified as a vector of integer constants.

When you do not specify `ubounds`, the code generator computes the upper bound for each variable. If the `ubounds` element corresponds to a fixed-size dimension, the value is the fixed size of the dimension.

Example: `coder.varsize('x','y',[1 2])`

**dims** — Indication of whether each dimension has a fixed size or a variable size

logical vector

Indication of whether each dimension has a fixed size or a variable size, specified as a logical vector. Dimensions that correspond to 0 or `false` in `dims` have a fixed size. Dimensions that correspond to 1 or `true` have a variable size.

When you do not specify `dims`, the dimensions have a variable size, except for the singleton dimensions.

Example: `coder.versize('x','y',[1 2], [0 1])`

## Limitations

- The `coder.versize` declaration instructs the code generator to allow the size of a variable to change. It does not change the size of the variable. Consider this code:

```
...
x = 7;
coder.versize('x', [1,5]);
disp(size(x));
...
```

After the `coder.versize` declaration, `x` is still a 1-by-1 array. You cannot assign a value to an element beyond the current size of `x`. For example, this code produces a run-time error because the index 3 exceeds the dimensions of `x`.

```
...
x = 7;
coder.versize('x', [1,5]);
x(3) = 1;
...
```

- `coder.versize` is not supported for a function input argument. Instead:
  - If the function is an entry-point function, specify that an input argument has a variable size by using `coder.typeof` at the command line. Alternatively, specify that an entry-point function input argument has a variable size by using the **Define Input Types** step of the app.
  - If the function is not an entry-point function, use `coder.versize` in the calling function with the variable that is the input to the called function.
- For sparse matrices, `coder.versize` drops upper bounds for variable-size dimensions.
- Limitations for using `coder.versize` with cell arrays:

- A cell array can have a variable size only if it is homogeneous. When you use `coder. varsize` with a heterogeneous cell array, the code generator tries to make the cell array homogeneous. The code generator tries to find a class and maximum size that apply to all elements of the cell array. For example, consider the cell array `c = {1, [2 3]}`. Both elements can be represented by a double type whose first dimension has a fixed size of 1 and whose second dimension has a variable size with an upper bound of 2. If the code generator cannot find a common class and a maximum size, code generation fails. For example, consider the cell array `c = {'a', [2 3]}`. The code generator cannot find a class that can represent both elements because the first element is `char` and the second element is `double`.
- If you use the `cell` function to define a fixed-size cell array, you cannot use `coder. varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because `x = cell(1,3)` makes `x` a fixed-size, 1-by-3 cell array.

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

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

```
...  
x = {1 2 3};  
coder.varsize('x',[1 5])  
...
```

- To create a variable-size cell array by using the `cell` function, use this code pattern:

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

See “Definition of Variable-Size Cell Array by Using `cell`”.

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

```
function mycell(n)  
%#codegen
```



```
x = cell(1,n);
for i = 1:n
    x{i} = i;
coder.versize('x',[1,20]);
end
end
```

- `coder.versize` is not supported for:
  - Global variables
  - MATLAB classes or class properties
  - String scalars

## More About

### Singleton Dimension

Dimension for which `size(A,dim) = 1`.

### Tips

- In a code generation report or a MATLAB Function report, a colon (:) indicates that a dimension has a variable size. For example, a size of `1x:2` indicates that the first dimension has a fixed size of one and the second dimension has a variable size with an upper bound of two.
- If you use `coder.versize` to specify that the upper bound of a dimension is 1, by default, the dimension has a fixed size of 1. To specify that the dimension can be 0 (empty array) or 1, set the corresponding element of the `dims` argument to `true`. For example, this code specifies that the first dimension of `x` has a fixed size of 1 and the other dimensions have a variable size of 5.

```
coder.versize('x',[1,5,5])
```

In contrast, this code specifies that the first dimension of `x` has an upper bound of 1 and has a variable size (can be 0 or 1).

```
coder.versize('x',[1,5,5],[1,1,1])
```

**Note** For a MATLAB Function block, you cannot specify that an input or output signal with size 1 has a variable size.

- If you use input variables or the result of a computation using input variables to specify the size of an array, it is declared as variable-size in the generated code. Do not re-use `coder. varsize` on the array, unless you also want to specify an upper bound for its size.
- If you do not specify upper bounds with a `coder. varsize` declaration and the code generator is unable to determine the upper bounds, the generated code uses dynamic memory allocation. Dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, specify the upper bounds by providing the `ubounds` argument.

## See Also

`coder. typeof`

## Topics

“Code Generation for Variable-Size Arrays”

“Incompatibilities with MATLAB in Variable-Size Support for Code Generation”

**Introduced in R2011a**

# colon

Create vectors, array subscripting

## Syntax

```
y = j:k
y = j:i:k
```

## Description

`y = j:k` returns a regularly-spaced vector, `[j, j+1, ..., k]`. `j:k` is empty when `j > k`.

At least one of the colon operands must be a `fi` object. All colon operands must have integer values. All the fixed-point operands must be binary-point scaled. Slope-bias scaling is not supported. If any of the operands is complex, the `colon` function generates a warning and uses only the real part of the operands.

`y = colon(j,k)` is the same as `y = j:k`.

`y = j:i:k` returns a regularly-spaced vector, `[j, j+i, j+2i, ..., j+m*i]`, where `m = fix((k-j)/i)`. `y = j:i:k` returns an empty matrix when `i == 0`, `i > 0` and `j > k`, or `i < 0` and `j < k`.

## Examples

### Use `fi` as a Colon Operator

When you use `fi` as a colon operator, all colon operands must have integer values.

```
a = fi(1,0,3,0);
b = fi(2,0,8,0);
c = fi(12,0,8,0);
x = a:b:c
```

```
x=1x6 object
      1      3      5      7      9     11
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Unsigned
      WordLength: 8
      FractionLength: 0
```

Because all the input operands are unsigned, `x` is unsigned and the word length is 8. The fraction length of the resulting vector is always 0.

### Use the colon Operator With Signed and Unsigned Operands

```
a = fi(int8(-1));
b = uint8(255);
c = a:b;
len = c.WordLength

len = 9

signedness = c.Signedness

signedness =
'Signed'
```

The word length of `c` requires an additional bit to handle the intersection of the ranges of `int8` and `uint8`. The data type of `c` is signed because the operand `a` is signed.

### Create a Vector of Decreasing Values

If the beginning and ending operands are unsigned, the increment operand can be negative.

```
x = fi(4, false):-1:1

x=1x4 object
      4      3      2      1
      DataTypeMode: Fixed-point: binary point scaling
```

```
Signedness: Unsigned
WordLength: 16
FractionLength: 0
```

### Use the colon Operator With Floating-Point and fi Operands

If any of the operands is floating-point, the output has the same word length and signedness as the `fi` operand

```
x = fi(1):10
```

```
x=1x10 object
```

```
  1    2    3    4    5    6    7    8    9   10
```

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

`x = fi(1):10` is equivalent to `fi(1:10, true, 16, 0)` so `x` is signed and its word length is 16 bits.

### Rewrite Code That Uses Non-Integer Operands

If your code uses non-integer operands, rewrite the colon expression so that the operands are integers.

The following code does not work because the colon operands are not integer values.

```
Fs = fi(100);
n = 1000;
t = (0:1/Fs:(n/Fs - 1/Fs));
```

Rewrite the colon expression to use integer operands.

```
Fs = fi(100);  
n = 1000;  
t = (0:(n-1))/Fs;
```

### All Colon Operands Must Be in the Range of the Data Type

If the value of any of the colon operands is outside the range of the data type used in the colon expression, MATLAB generates an error.

```
y = fi(1,true,8,0):256
```

MATLAB generates an error because 256 is outside the range of `fi(1,true,8,0)`. This behavior matches the behavior for built-in integers. For example, `y = int8(1):256` generates the same error.

## Input Arguments

### **j** — Beginning operand

real scalar

Beginning operand, specified as a real scalar integer-valued `fi` object or built-in numeric type.

If you specify non-scalar arrays, MATLAB interprets `j:i:k` as `j(1):i(1):k(1)`.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **i** — Increment

1 (default) | real scalar

Increment, specified as a real scalar integer-valued `fi` object or built-in numeric type. Even if the beginning and end operands, `j` and `k`, are both unsigned, the increment operand `i` can be negative.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **k** — Ending operand

real scalar

Ending operand, specified as a real scalar integer-valued `fi` object or built-in numeric type.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **y** — Regularly-spaced vector

real vector

Fixed-Point Designer determines the data type of the `y` using the following rules:

- The data type covers the union of the ranges of the fixed-point types of the input operands.
- If either the beginning or ending operand is signed, the resulting data type is signed. Otherwise, the resulting data type is unsigned.
- The word length of `y` is the smallest value such that the fraction length is 0 and the real-world value of the least-significant bit is 1.
- If any of the operands is floating-point, the word length and signedness of `y` is derived from the `fi` operand.
- If any of the operands is a scaled double, `y` is a scaled double.
- The `fimath` of `y` is the same as the `fimath` of the input operands.
- If all the `fi` objects are of data type `double`, the data type of `y` is `double`. If all the `fi` objects are of data type `single`, the data type of `y` is `single`. If there are both `double` and `single` inputs, and no fixed-point inputs, the output data type is `single`.

## See Also

`colon` | `fi`

**Introduced in R2013b**

## complex

Construct complex `fi` object from real and imaginary parts

### Syntax

```
c = complex(a,b)
c = complex(a)
c = complex(a)
```

### Description

The `complex` function constructs a complex `fi` object from real and imaginary parts.

`c = complex(a,b)` returns the complex result  $a + bi$ , where `a` and `b` are identically sized real N-D arrays, matrices, or scalars of the same data type. When `b` is all zero, `c` is complex with an all-zero imaginary part. This is in contrast to the addition of  $a + 0i$ , which returns a strictly real result.

`c = complex(a)` for a real `fi` object `a` returns the complex result  $a + bi$  with real part `a` and an all-zero imaginary part. Even though its imaginary part is all zero, `c` is complex.

`c = complex(a)` returns the complex equivalent of `a`, such that `isreal(c)` returns logical 0 (false). If `a` is real, then `c` is  $a + 0i$ . If `a` is complex, then `c` is identical to `a`.

The output `fi` object `c` has the same `numericType` and `fiMath` properties as the input `fi` object `a`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.



## **See Also**

**Introduced before R2006a**

## **conj**

Complex conjugate of `fi` object

### **Syntax**

`conj(a)`

### **Description**

`conj(a)` is the complex conjugate of `fi` object `a`.

When `a` is complex,

$$\text{conj}(a) = \text{real}(a) - i \times \text{imag}(a)$$

The `numericType` and `fiMath` properties associated with the input `a` are applied to the output.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`complex`

**Introduced before R2006a**

## conv

Convolution and polynomial multiplication of `fi` objects

### Syntax

```
c = conv(a,b)
c = conv(a,b,'shape')
```

### Description

`c = conv(a,b)` outputs the convolution of input vectors `a` and `b`, at least one of which must be a `fi` object.

`c = conv(a,b,'shape')` returns a subsection of the convolution, as specified by the `shape` parameter:

- `full` — Returns the full convolution. This option is the default shape.
- `same` — Returns the central part of the convolution that is the same size as input vector `a`.
- `valid` — Returns only those parts of the convolution that the function computes without zero-padded edges. In this case, the length of output vector `c` is  $\max(\text{length}(a) - \max(0, \text{length}(b) - 1), 0)$ .

The `fimath` properties associated with the inputs determine the `numericType` properties of output `fi` object `c`:

- If either `a` or `b` has a local `fimath` object, `conv` uses that `fimath` object to compute intermediate quantities and determine the `numericType` properties of `c`.
- If neither `a` nor `b` have an attached `fimath`, `conv` uses the default `fimath` to compute intermediate quantities and determine the `numericType` properties of `c`.

If either input is a built-in data type, `conv` casts it into a `fi` object using best-precision rules before performing the convolution operation.

The output `fi` object `c` always uses the default `fimath`.

Refer to the MATLAB `conv` reference page for more information on the convolution algorithm.

## Examples

The following example illustrates the convolution of a 22-sample sequence with a 16-tap FIR filter.

- `x` is a 22-sample sequence of signed values with a word length of 16 bits and a fraction length of 15 bits.
- `h` is the 16 tap FIR filter.

```
u = (pi/4)*[1 1 1 -1 -1 -1 1 -1 -1 1 -1];  
x = fi(kron(u,[1 1]));  
h = firls(15, [0 .1 .2 .5]*2, [1 1 0 0]);
```

Because `x` is a `fi` object, you do not need to cast `h` into a `fi` object before performing the convolution operation. The `conv` function does so using best-precision scaling.

Finally, use the `conv` function to convolve the two vectors:

```
y = conv(x,h);
```

The operation results in a signed `fi` object `y` with a word length of 36 bits and a fraction length of 31 bits. The default `fimath` properties associated with the inputs determine the `numericType` of the output. The output does not have a local `fimath`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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 might 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`.

## See Also

conv

**Introduced in R2009b**

## convergent

Round toward nearest integer with ties rounding to nearest even integer

### Syntax

```
y = convergent(a)  
y = convergent(x)
```

### Description

`y = convergent(a)` rounds `fi` object `a` to the nearest integer. In the case of a tie, `convergent(a)` rounds to the nearest even integer.

`y` and `a` have the same `fi` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`convergent` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

`y = convergent(x)` rounds the elements of `x` to the nearest integer. In the case of a tie, `convergent(x)` rounds to the nearest even integer.

## Examples

### Example 1

The following example demonstrates how the `convergent` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 8  
    FractionLength: 3
```

```
y = convergent(a)
```

```
y =
```

```
3
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 6  
    FractionLength: 0
```

### Example 2

The following example demonstrates how the `convergent` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a =
```

```
0.0249
```

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
            WordLength: 8
      FractionLength: 12
    
```

y = convergent(a)

y =

0

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
            WordLength: 2
      FractionLength: 0
    
```

### Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>convergent(a)</b>	<b>nearest(a)</b>	<b>round(a)</b>
-3.5	-4	-3	-4
-2.5	-2	-2	-3
-1.5	-2	-1	-2
-0.5	0	0	-1
0.5	0	1	1
1.5	2	2	2
2.5	2	3	3
3.5	4	4	4



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`ceil` | `fix` | `floor` | `nearest` | `round`

**Introduced before R2006a**

## convertToSingle

Convert double-precision MATLAB code to single-precision MATLAB code

### Syntax

```
convertToSingle options fcn_1, ..., fcn_n  
convertToSingle options fcn_1, -args args_1 ,..., fcn_n -args args_n
```

### Description

`convertToSingle options fcn_1, ..., fcn_n` generates single-precision MATLAB code from the specified function or functions. When you use this syntax, you must provide a test file that `convertToSingle` can use to determine the properties of the input parameters. To specify the test file, use `coder.config('single')` to create a `coder.SingleConfig` object. Specify the `TestBenchName` property.

`convertToSingle options fcn_1, -args args_1 ,..., fcn_n -args args_n` specifies the properties of the input arguments.

### Examples

#### Convert to Single Precision and Validate Using a Test File

Generate single-precision code from a double-precision function `myfun.m`. Specify a test file for determining the argument properties and for verification of the converted types. Plot the error between the double-precision and single-precision values.

```
scfg = coder.config('single');  
scfg.TestBenchName = 'myfun_test';  
scfg.TestNumerics = true;  
scfg.LogIOForComparisonPlotting = true;  
convertToSingle -config scfg myfun
```

## Convert Multiple Functions to Single Precision with the Default Configuration

Convert `myfun1.m` and `myfun2.m` to single precision. Specify that `myfun1` has a double scalar argument and `myfun2` has a 2x3 double argument.

```
convertToSingle -config cfg myfun1 -args {0} myfun2 -args {zeros(2, 3)}
```

## Specify Input Argument Properties

Generate single-precision code from a double-precision function, `myfun.m`, whose first argument is double scalar and whose second argument is 2x3 double.

```
convertToSingle myfun -args {0, zeros(2, 3)}
```

# Input Arguments

### **fcn** — Function name

character vector

MATLAB function from which to generate single-precision code.

### **args** — Argument properties

cell array of types or example values.

Definition of the size, class, and complexity of the input arguments specified as a cell array of types or example values. To create a type, use `coder.typeof`.

### **options** — options for single-precision conversion

`-config` | `-globals`

Specify one of the following single-conversion options.

`-config` *config\_object*

Specify the configuration object to use for conversion of double-precision MATLAB code to single-precision MATLAB code. To create the configuration object, use

```
coder.config('single');
```

If you do not use this option, the conversion uses a default configuration. When you omit `-config`, to specify the properties of the input arguments, use `-args`.

`-globals` *global\_values*

Specify names and initial values for global variables in MATLAB files.

`global_values` is a cell array of global variable names and initial values. The format of `global_values` is:

```
{g1, init1, g2, init2, ..., gn, initn}
```

`gn` is the name of a global variable. `initn` is the initial value. For example:

```
-globals {'g', 5}
```

Alternatively, use this format:

```
-globals {global_var, {type, initial_value}}
```

`type` is a type object. To create the type object, use `coder.typeof`.

If you do not provide initial values for global variables using the `-globals` option, `convertToSingle` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `convertToSingle` generates an error.

## See Also

`coder.SingleConfig` | `coder.config`

## Topics

“Generate Single-Precision MATLAB Code”

**Introduced in R2015b**

## copyobj

Make independent copy of quantizer object

### Syntax

```
q1 = copyobj(q)
[q1,q2,...] = copyobj(obja,objb,...)
```

### Description

`q1 = copyobj(q)` makes a copy of quantizer object `q` and returns it in `q1`.

`[q1,q2,...] = copyobj(obja,objb,...)` copies `obja` into `q1`, `objb` into `q2`, and so on.

Using `copyobj` to copy a quantizer object is not the same as using the command syntax `q1 = q` to copy a quantizer object. Quantizer objects have memory (their read-only properties). When you use `copyobj`, the resulting copy is independent of the original item; it does not share the original object's memory, such as the values of the properties `min`, `max`, `noverflows`, or `noperations`. Using `q1 = q` creates a new object that is an alias for the original and shares the original object's memory, and thus its property values.

### Examples

```
q = quantizer([8 7]);
q1 = copyobj(q)
```

### See Also

`get` | `quantizer` | `set`

**Introduced before R2006a**

# cordicabs

CORDIC-based absolute value

## Syntax

```
r = cordicabs(c)
r = cordicabs(c,niters)
r = cordicabs(c,niters,'ScaleOutput',b)
r = cordicabs(c,'ScaleOutput',b)
```

## Description

`r = cordicabs(c)` returns the magnitude of the complex elements of `C`.

`r = cordicabs(c,niters)` performs `niters` iterations of the algorithm.

`r = cordicabs(c,niters,'ScaleOutput',b)` specifies both the number of iterations and, depending on the Boolean value of `b`, whether to scale the output by the inverse CORDIC gain value.

`r = cordicabs(c,'ScaleOutput',b)` scales the output depending on the Boolean value of `b`.

## Input Arguments

**c**

`c` is a vector of complex values.

**niters**

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of

`r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

## Name-Value Pair Arguments

Optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ).

### ScaleOutput

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

**Default:** `true`

## Output Arguments

`r`

`r` contains the magnitude values of the complex input values. If the inputs are fixed-point values, `r` is also fixed point (and is always signed, with binary point scaling). All input values must have the same data type. If the inputs are signed, then the word length of `r` is the input word length + 2. If the inputs are unsigned, then the word length of `r` is the input word length + 3. The fraction length of `r` is always the same as the fraction length of the inputs.

## Examples

Compare `cordicabs` and `abs` of double values.

```
dblValues = complex(rand(5,4),rand(5,4));  
r_dbl_ref = abs(dblValues)  
r_dbl_cdc = cordicabs(dblValues)
```

Compute absolute values of fixed-point inputs.



```
fxpValues = fi(dblValues);  
r_fxp_cdc = cordicabs(fxpValues)
```

## More About

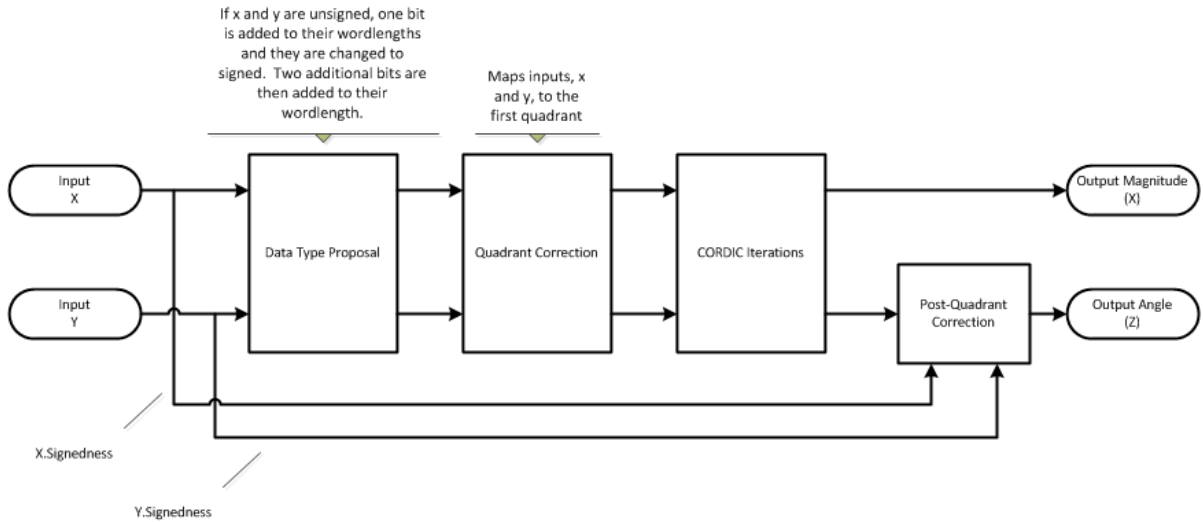
### CORDIC

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

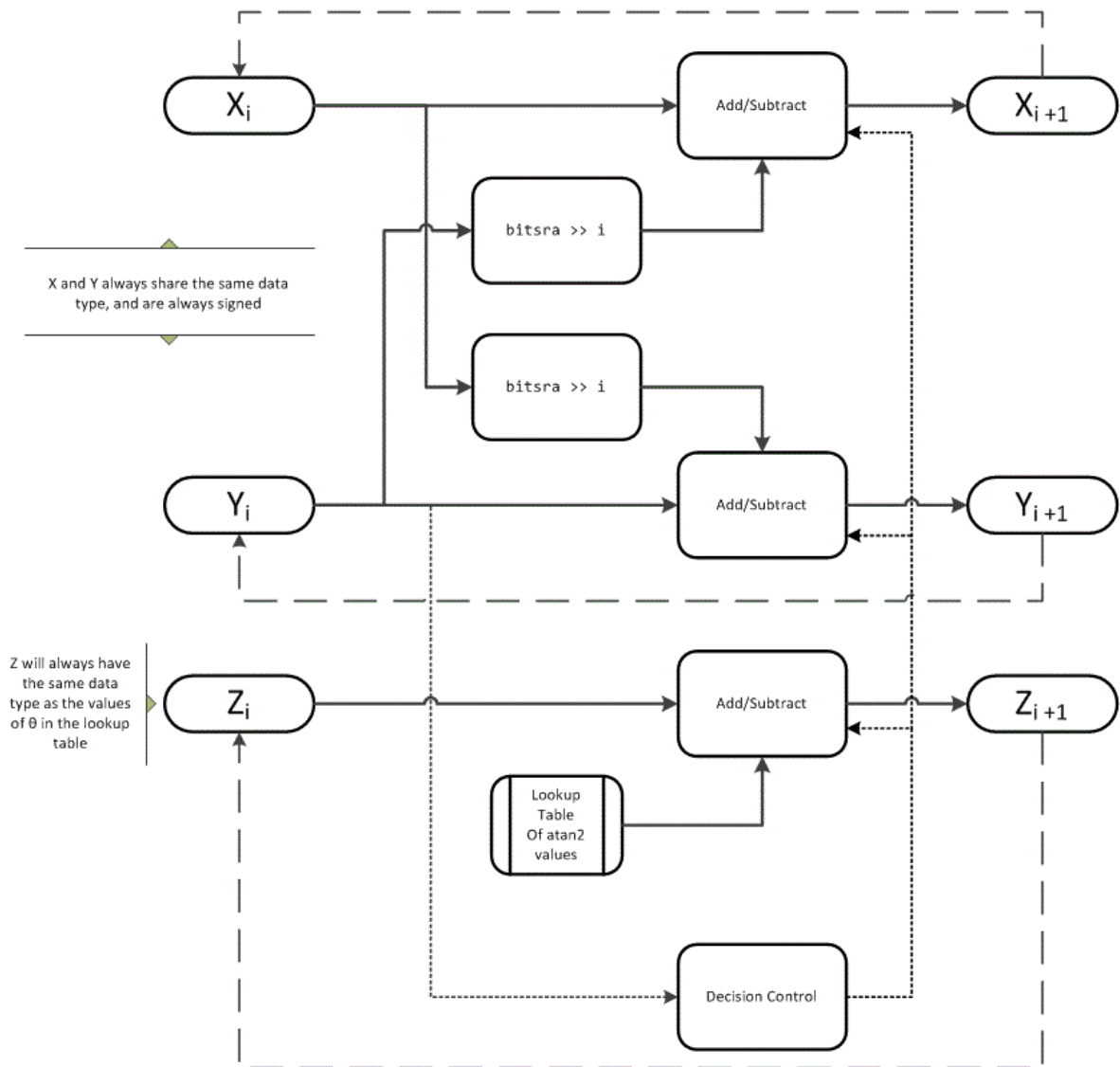
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

# Algorithms

## Signal Flow Diagrams



## CORDIC Vectoring Kernel



The accuracy of the CORDIC kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$x_0$  is initialized to the  $x$  input value

$y_0$  is initialized to the  $y$  input value

$z_0$  is initialized to 0

### **fimath Propagation Rules**

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## **References**

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`abs` | `cordicangle` | `cordiccart2pol`

**Introduced in R2011b**

## cordicacos

CORDIC-based approximation of inverse cosine

### Syntax

```
theta = cordicacos(x)
theta = cordicacos(x, niters)
```

### Description

`theta = cordicacos(x)` returns the inverse cosine of `x` based on a CORDIC approximation.

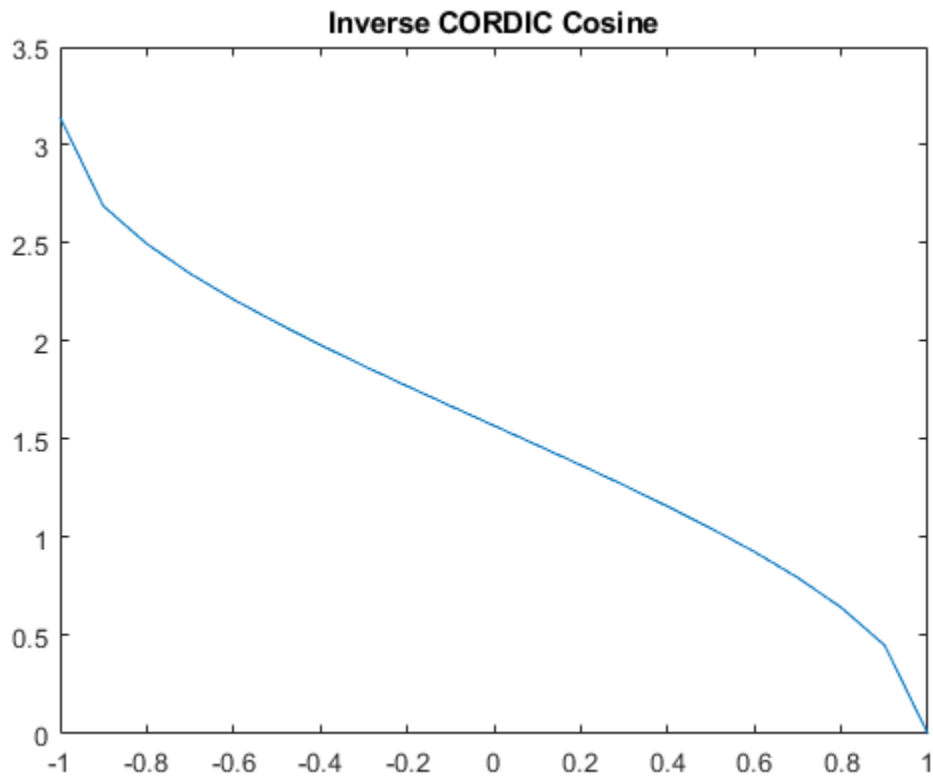
`theta = cordicacos(x, niters)` returns the inverse cosine of `x` performing `niters` iterations of the CORDIC algorithm.

### Examples

#### Calculate CORDIC Inverse Cosine

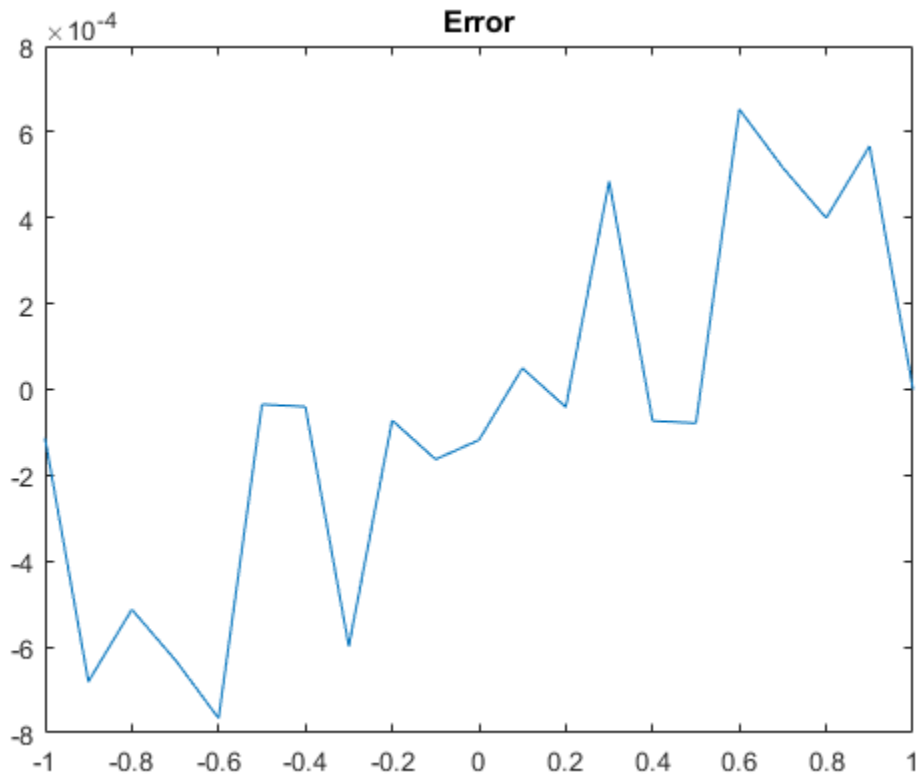
Compute the inverse cosine of a fixed-point `fi` object using a CORDIC implementation.

```
a = fi(-1:.1:1,1,16);
b = cordicacos(a);
plot(a, b);
title('Inverse CORDIC Cosine');
```



Compare the output of the cordiacos function and the acos function.

```
c = acos(double(a));  
error = double(b)-c;  
plot(a, error);  
title('Error');
```



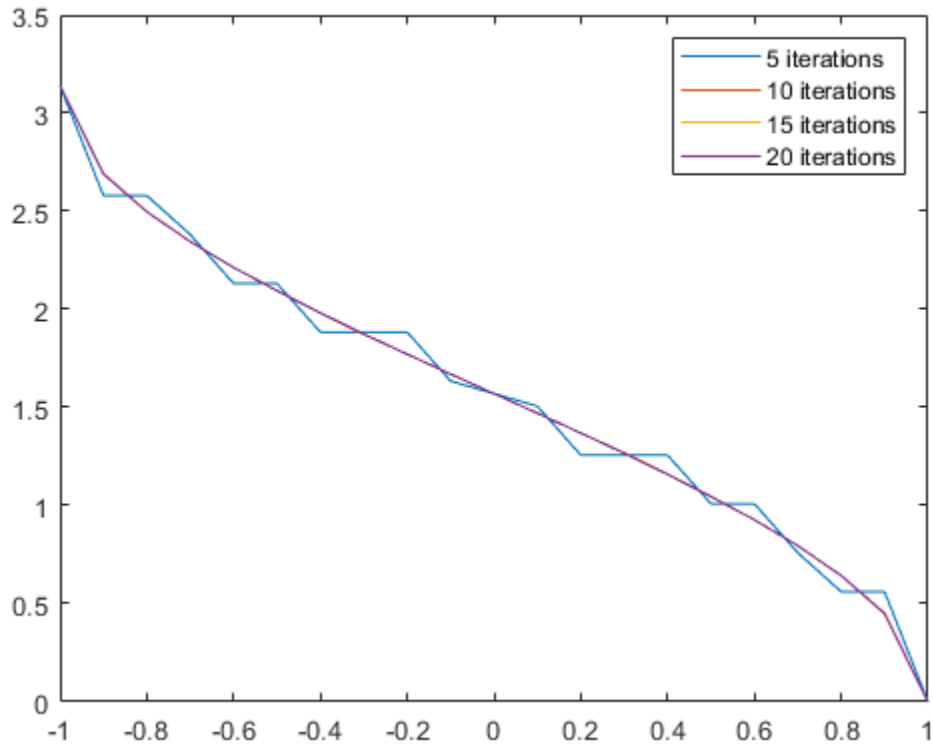
### Calculate CORDIC Inverse Cosine with Specified Number of Iterations

Find the inverse cosine of a `fi` object using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the inverse cosine with varying numbers of iterations.

```
a = fi(-1:.1:1, 1, 16);  
for i = 5:5:20  
    b = cordicacos(a,i);  
    plot(a,b);  
    hold on;
```



```
end  
legend('5 iterations', '10 iterations', '15 iterations', '20 iterations')
```



## Input Arguments

### **x** — Numeric input

scalar | vector | matrix | multidimensional array

Numeric input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | fi

Complex Number Support: Yes

### **niters** — Number of iterations

scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array, `theta`. For double-precision inputs, the default value of `niters` is 52. For single-precision inputs, the default value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **theta** — Inverse cosine angle values

scalar | vector | matrix | n-dimensional array

Inverse cosine angle values in rad.

## See Also

### Functions

`cordiccos` | `cordicsin`

### Introduced in R2018b

# cordicangle

CORDIC-based phase angle

## Syntax

```
theta = cordicangle(c)  
theta = cordicangle(c,niters)
```

## Description

`theta = cordicangle(c)` returns the phase angles, in radians, of matrix `c`, which contains complex elements.

`theta = cordicangle(c,niters)` performs `niters` iterations of the algorithm.

## Input Arguments

**c**

Matrix of complex numbers

**niters**

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

## Output Arguments

### theta

`theta` contains the polar coordinates angle values, which are in the range  $[-\pi, \pi]$  radians. If `x` and `y` are floating-point, then `theta` has the same data type as `x` and `y`. Otherwise, `theta` is a fixed-point data type with the same word length as `x` and `y` and with a best-precision fraction length for the  $[-\pi, \pi]$  range.

## Examples

Phase angle for double-valued input and for fixed-point-valued input.

```
dblRandomVals = complex(rand(5,4), rand(5,4));
theta_dbl_ref = angle(dblRandomVals);
theta_dbl_cdc = cordicangle(dblRandomVals)
fxpRandomVals = fi(dblRandomVals);
theta_fxp_cdc = cordicangle(fxpRandomVals)
```

```
theta_dbl_cdc =
```

```
    1.0422    1.0987    1.2536    0.6122
    0.5893    0.8874    0.3580    0.2020
    0.5840    0.2113    0.8933    0.6355
    0.7212    0.2074    0.9820    0.8110
    1.3640    0.3288    1.4434    1.1291
```

```
theta_fxp_cdc =
```

```
    1.0422    1.0989    1.2534    0.6123
    0.5894    0.8872    0.3579    0.2019
    0.5840    0.2112    0.8931    0.6357
    0.7212    0.2075    0.9819    0.8110
    1.3640    0.3289    1.4434    1.1289
```

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

## More About

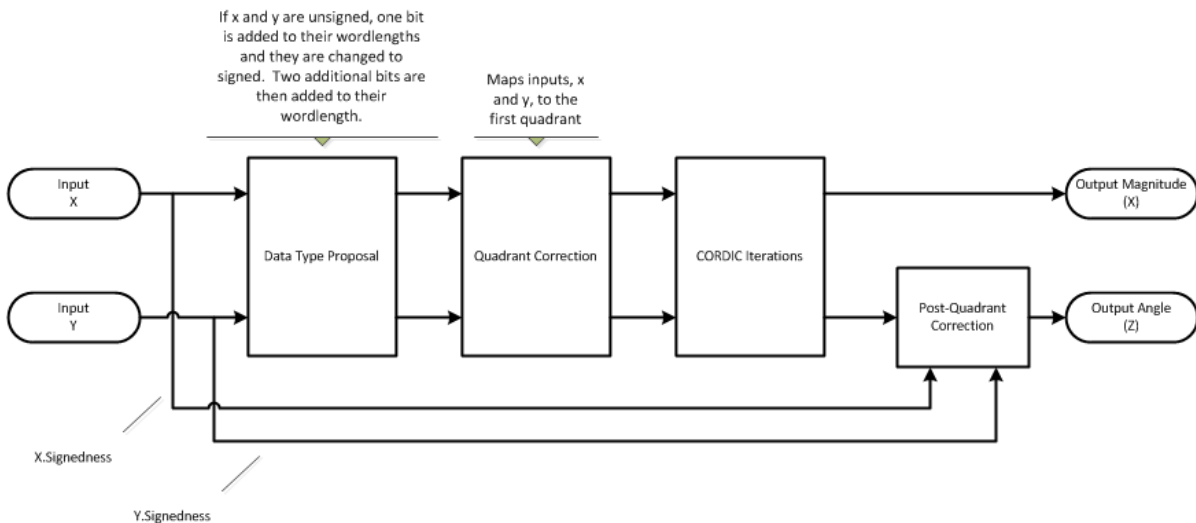
### CORDIC

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

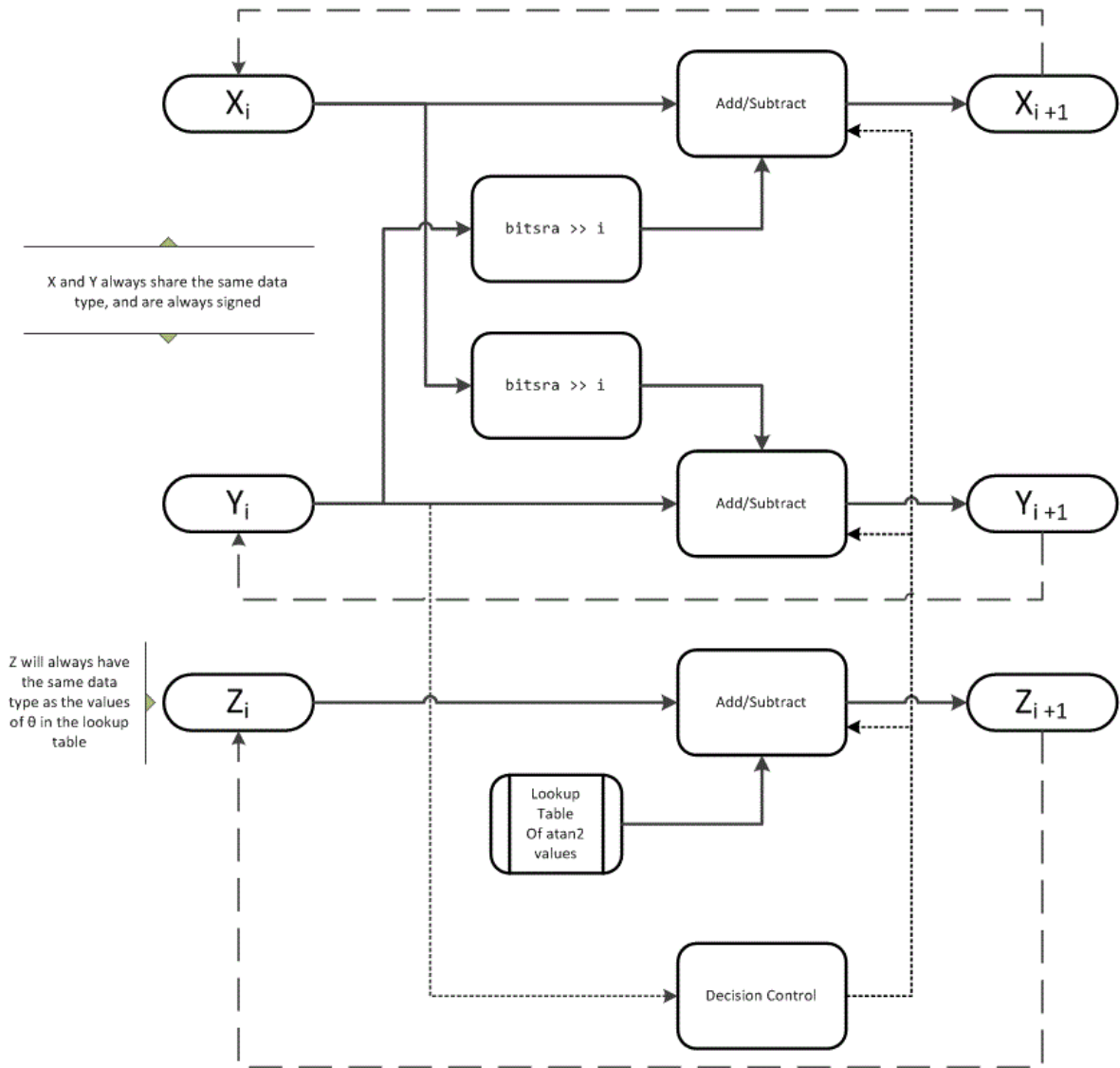
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



**CORDIC Vectorsing Kernel**



The accuracy of the CORDIC kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$x_0$  is initialized to the  $x$  input value

$y_0$  is initialized to the  $y$  input value

$z_0$  is initialized to 0

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## References

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`angle` | `cordicabs` | `cordicatan2` | `cordiccart2pol`

**Introduced in R2011b**



# cordicasin

CORDIC-based approximation of inverse sine

## Syntax

```
theta = cordicasin(x)
theta = cordicasin(x, niters)
```

## Description

`theta = cordicasin(x)` returns the inverse sine of `x` based on a CORDIC approximation.

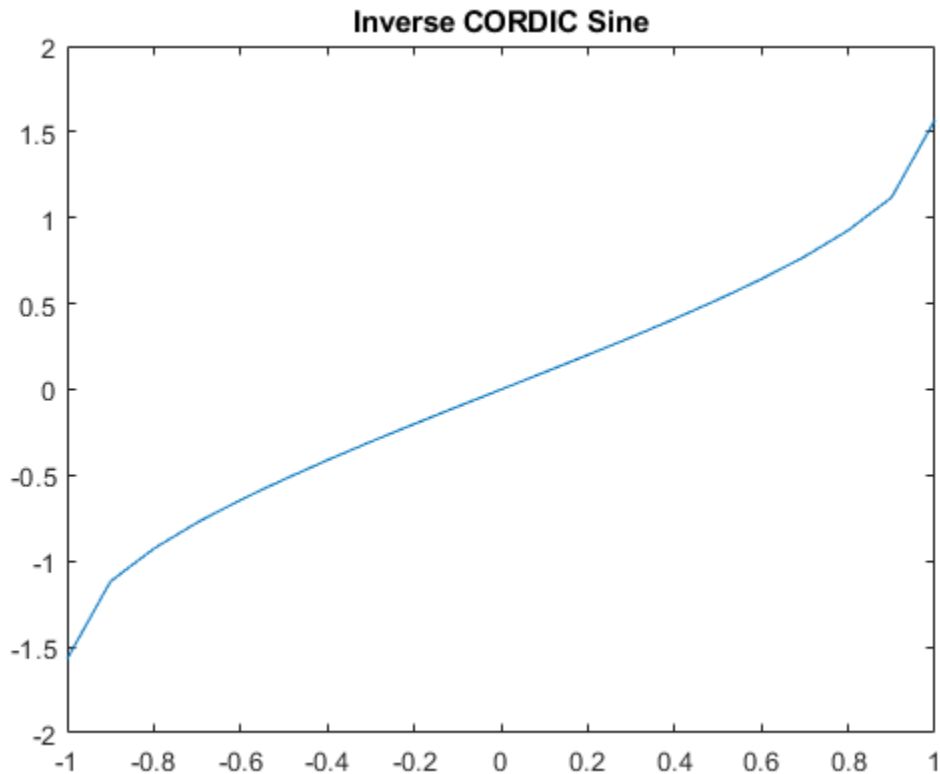
`theta = cordicasin(x, niters)` returns the inverse sine of `x` performing `niters` iterations of the CORDIC algorithm.

## Examples

### Calculate CORDIC Inverse Sine

Compute the inverse Sine of a fixed-point `fi` object using a CORDIC implementation.

```
a = fi(-1:.1:1,1,16);
b = cordicasin(a);
plot(a, b);
title('Inverse CORDIC Sine');
```

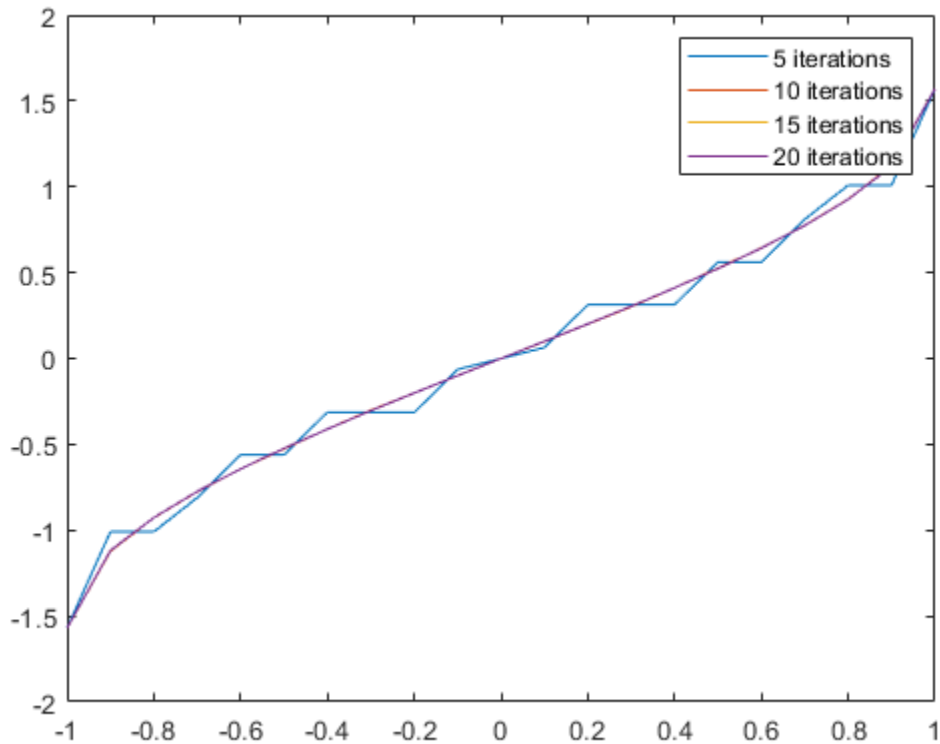


### Calculate CORDIC Inverse Sine with Specified Number of Iterations

Find the inverse sine of a `fi` object using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the inverse sine with varying numbers of iterations.

```
a = fi(-1:.1:1, 1, 16);  
for i = 5:5:20  
    b = cordicasin(a,i);  
    plot(a,b);  
    hold on;
```

```
end  
legend('5 iterations', '10 iterations', '15 iterations', '20 iterations')
```



## Input Arguments

### **x** — Numeric input

scalar | vector | matrix | multidimensional array

Numeric input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | fi

Complex Number Support: Yes

### **niters** — Number of iterations

scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array, `theta`. For double-precision inputs, the default value of `niters` is 52. For single-precision inputs, the default value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **theta** — Inverse sine angle values

scalar | vector | matrix | n-dimensional array

Inverse sine angle values in rad.

## See Also

### Functions

`cordiccos` | `cordicsin`

### Introduced in R2018b

# cordicatan2

CORDIC-based four quadrant inverse tangent

## Syntax

```
theta = cordicatan2(y,x)
theta = cordicatan2(y,x,niters)
```

## Description

`theta = cordicatan2(y,x)` computes the four quadrant arctangent of `y` and `x` using a “CORDIC” on page 5-290 algorithm approximation.

`theta = cordicatan2(y,x,niters)` performs `niters` iterations of the algorithm.

## Input Arguments

### **y,x**

`y,x` are Cartesian coordinates. `y` and `x` must be the same size. If they are not the same size, at least one value must be a scalar value. Both `y` and `x` must have the same data type.

### **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `y` or `x`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

## Output Arguments

### theta

theta is the arctangent value, which is in the range  $[-\pi, \pi]$  radians. If  $y$  and  $x$  are floating-point numbers, then `theta` has the same data type as  $y$  and  $x$ . Otherwise, `theta` is a fixed-point data type with the same word length as  $y$  and  $x$  and with a best-precision fraction length for the  $[-\pi, \pi]$  range.

## Examples

Floating-point CORDIC arctangent calculation.

```
theta_cdat2_float = cordicatan2(0.5, -0.5)
```

```
theta_cdat2_float =  
    2.3562
```

Fixed-point CORDIC arctangent calculation.

```
theta_cdat2_fixpt = cordicatan2(fi(0.5, 1, 16, 15), fi(-0.5, 1, 16, 15));
```

```
theta_cdat2_fixpt =  
    2.3562
```

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

## More About

### CORDIC

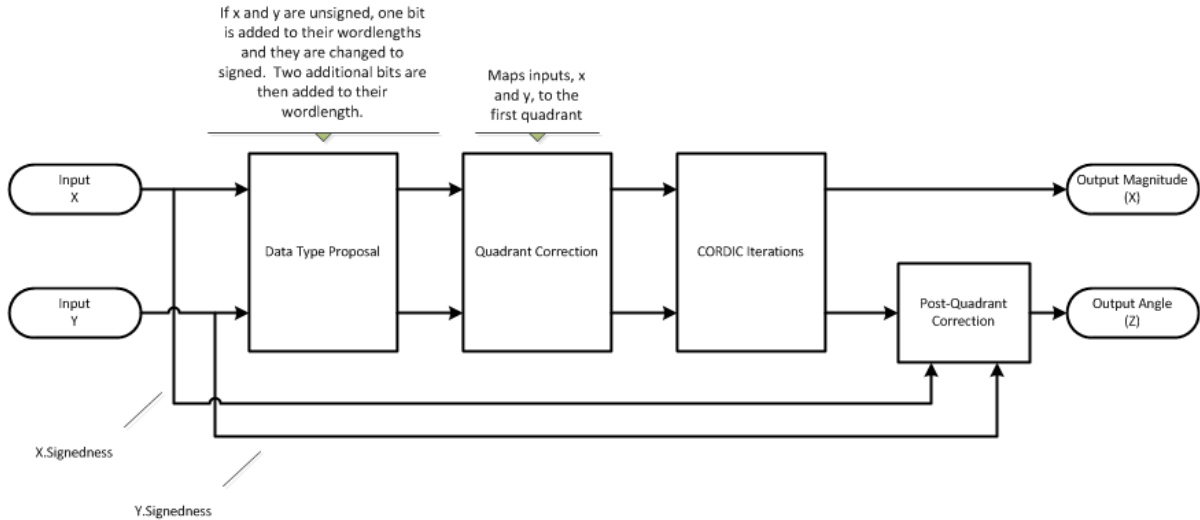
CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector

magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

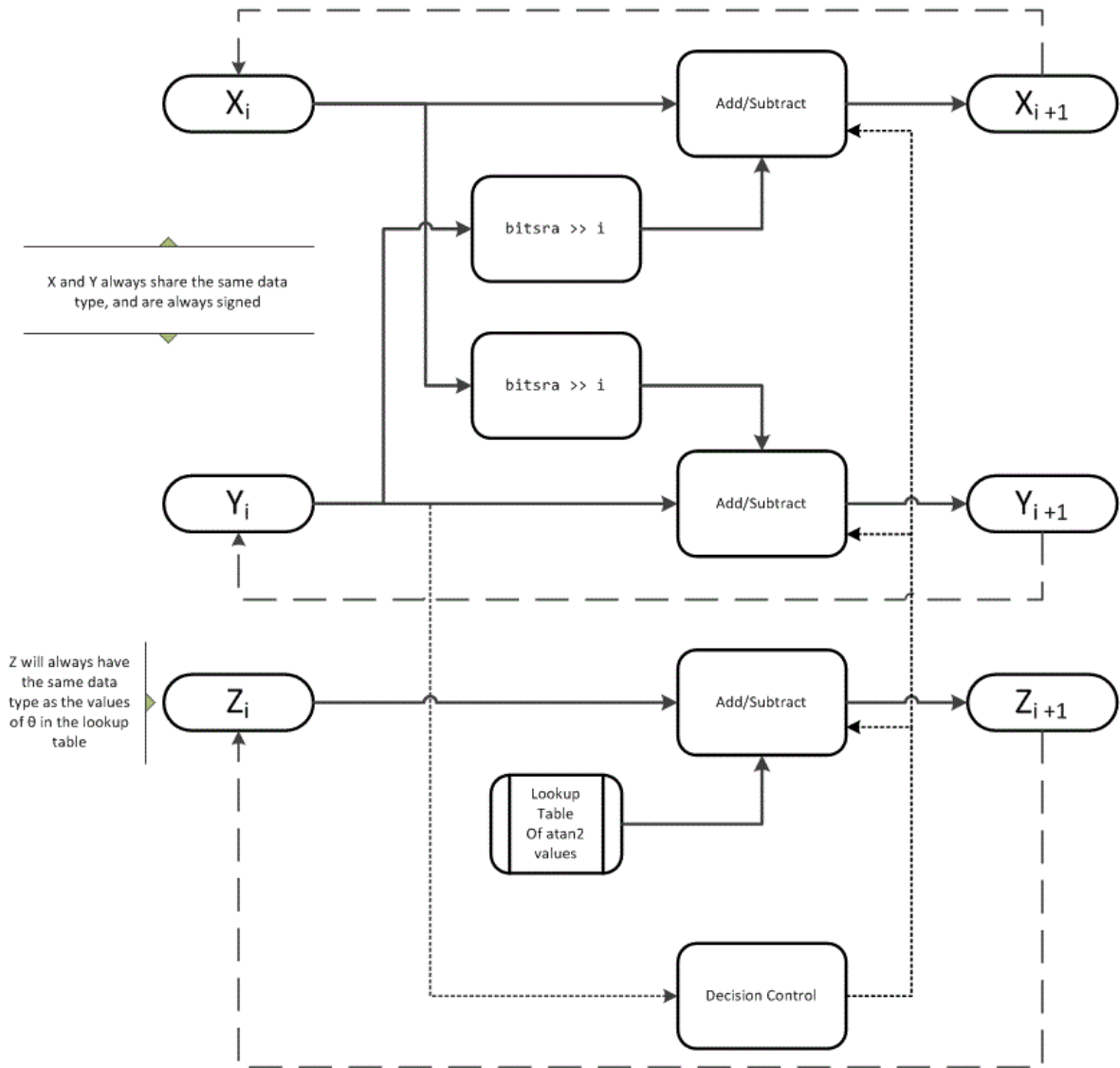
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



**CORDIC Vectorsing Kernel**





The accuracy of the CORDIC kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$x_0$  is initialized to the  $x$  input value

$y_0$  is initialized to the  $y$  input value

$z_0$  is initialized to 0

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## References

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`atan2` | `atan2` | `cordiccos` | `cordicsin`

### Topics

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2011b**

# cordiccart2pol

CORDIC-based approximation of Cartesian-to-polar conversion

## Syntax

```
[theta,r] = cordiccart2pol(x,y)
[theta,r] = cordiccart2pol(x,y, niters)
[theta,r] = cordiccart2pol(x,y, niters, 'ScaleOutput',b)
[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)
```

## Description

`[theta,r] = cordiccart2pol(x,y)` using a CORDIC algorithm approximation, returns the polar coordinates, angle `theta` and radius `r`, of the Cartesian coordinates, `x` and `y`.

`[theta,r] = cordiccart2pol(x,y, niters)` performs `niters` iterations of the algorithm.

`[theta,r] = cordiccart2pol(x,y, niters, 'ScaleOutput',b)` specifies both the number of iterations and, depending on the Boolean value of `b`, whether to scale the `r` output by the inverse CORDIC gain value.

`[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)` scales the `r` output by the inverse CORDIC gain value, depending on the Boolean value of `b`.

## Input Arguments

### **x,y**

`x,y` are Cartesian coordinates. `x` and `y` must be the same size. If they are not the same size, at least one value must be a scalar value. Both `x` and `y` must have the same data type.

**niters**

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

**Name-Value Pair Arguments**

Optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ).

**ScaleOutput**

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

**Default:** `true`

**Output Arguments****theta**

`theta` contains the polar coordinates angle values, which are in the range  $[-\pi, \pi]$  radians. If `x` and `y` are floating-point, then `theta` has the same data type as `x` and `y`. Otherwise, `theta` is a fixed-point data type with the same word length as `x` and `y` and with a best-precision fraction length for the  $[-\pi, \pi]$  range.

**r**

`r` contains the polar coordinates radius magnitude values. `r` is real-valued and can be a scalar value or have the same dimensions as `theta`. If the inputs `x`, `y` are fixed-point values, `r` is also fixed point (and is always signed, with binary point scaling). Both `x`, `y` input values must have the same data type. If the inputs are signed, then the word length

of  $r$  is the input word length + 2. If the inputs are unsigned, then the word length of  $r$  is the input word length + 3. The fraction length of  $r$  is always the same as the fraction length of the  $x, y$  inputs.

## Examples

Convert fixed-point Cartesian coordinates to polar coordinates.

```
[thPos,r]=cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(0.5,16,15))
```

thPos =

```
0.5881 0.7854 1.1072 1.5708 2.0344 2.3562 2.5535 2.6780
```

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

r =

```
0.9014 0.7071 0.5591 0.5000 0.5591 0.7071 0.9014 1.1180
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 18
    FractionLength: 15
```

```
[thNeg,r]=...
```

```
cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(-0.5,16,15))
```

thNeg =

```
-0.5881 -0.7854 -1.1072 -1.5708 -2.0344 -2.3562 -2.5535 -2.6780
```

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

r =

```
0.9014 0.7071 0.5591 0.5000 0.5591 0.7071 0.9014 1.1180
```

DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 18  
FractionLength: 15

## More About

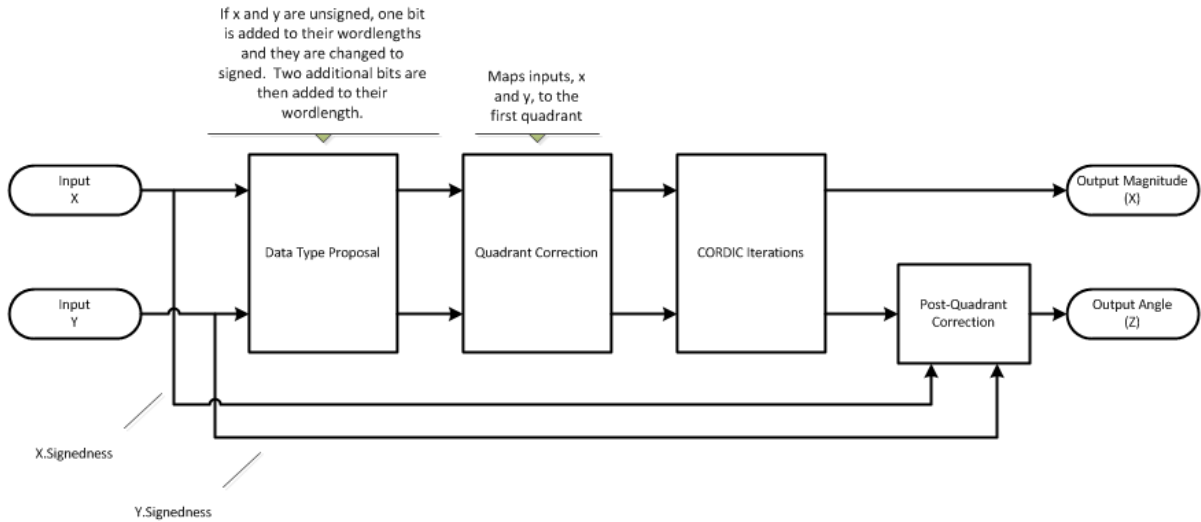
### CORDIC

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

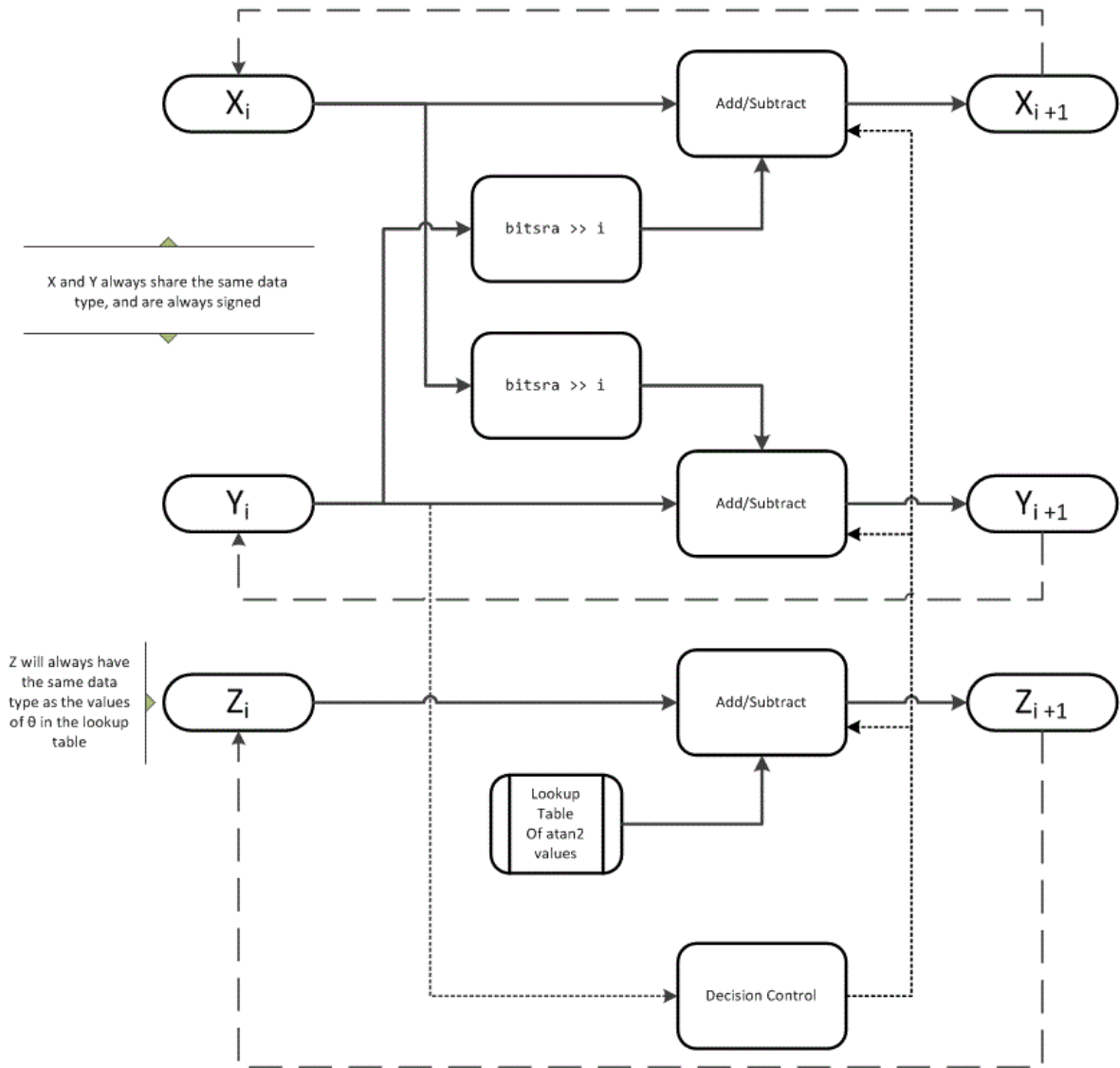
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

# Algorithms

## Signal Flow Diagrams



**CORDIC Vectorsing Kernel**





The accuracy of the CORDIC kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$x_0$  is initialized to the  $x$  input value

$y_0$  is initialized to the  $y$  input value

$z_0$  is initialized to 0

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## References

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`cart2pol` | `cordicatan2` | `cordicpol2cart`

**Introduced in R2011b**

## cordicexp

CORDIC-based approximation of complex exponential

### Syntax

```
y = cordicexp(theta,niters)
```

### Description

`y = cordicexp(theta,niters)` computes  $\cos(\theta) + j\sin(\theta)$  using a “CORDIC” on page 5-305 algorithm approximation. `y` contains the approximated complex result.

### Input Arguments

#### **theta**

`theta` can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range  $[-2\pi, 2\pi]$ .

#### **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

## Output Arguments

### y

y is the approximated complex result of the `cordiccxp` function. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength - 2`.

## Examples

The following example illustrates the effect of the number of iterations on the result of the `cordiccxp` approximation.

```

wrdLn = 8;
theta = fi(pi/2, 1, wrdLn);
fprintf('\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n');
fprintf('-----\t\t-----\t -----\t -----\t\t-----\t -----\t -----\n');
for niters = 1:(wrdLn - 1)
    cis = cordiccxp(theta, niters);
    fl = cis.FractionLength;
    x = real(cis);
    y = imag(cis);
    x_dbl = double(x);
    x_err = abs(x_dbl - cos(double(theta)));
    y_dbl = double(y);
    y_err = abs(y_dbl - sin(double(theta)));
    fprintf('%d\t\t%1.4f\t%1.4f\t%1.1f\t\t%1.4f\t%1.4f\t%1.1f\n', ...
        niters, y_dbl, y_err, (y_err*pow2(fl)), x_dbl, x_err, (x_err*pow2(fl)));
end
fprintf('\n');

```

The output table appears as follows:

NITERS	Y (SIN)	ERROR	LSBs	X (COS)	ERROR	LSBs
-----	-----	-----	-----	-----	-----	-----
1	0.7031	0.2968	19.0	0.7031	0.7105	45.5
2	0.9375	0.0625	4.0	0.3125	0.3198	20.5
3	0.9844	0.0156	1.0	0.0938	0.1011	6.5
4	0.9844	0.0156	1.0	-0.0156	0.0083	0.5
5	1.0000	0.0000	0.0	0.0312	0.0386	2.5
6	1.0000	0.0000	0.0	0.0000	0.0073	0.5
7	1.0000	0.0000	0.0	0.0156	0.0230	1.5

## More About

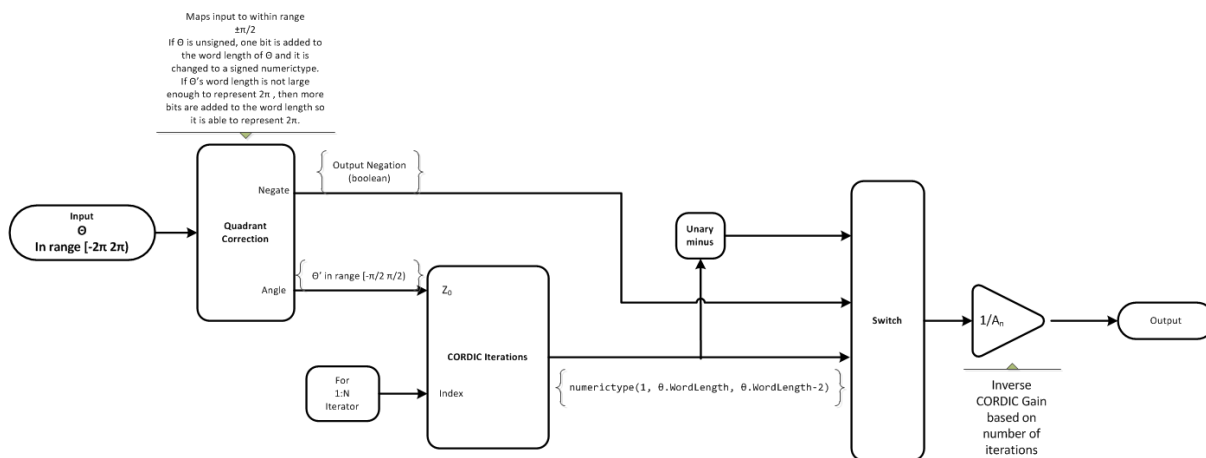
### CORDIC

CORDIC is an acronym for COordinate Rotation DIGital Computer. The Given rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

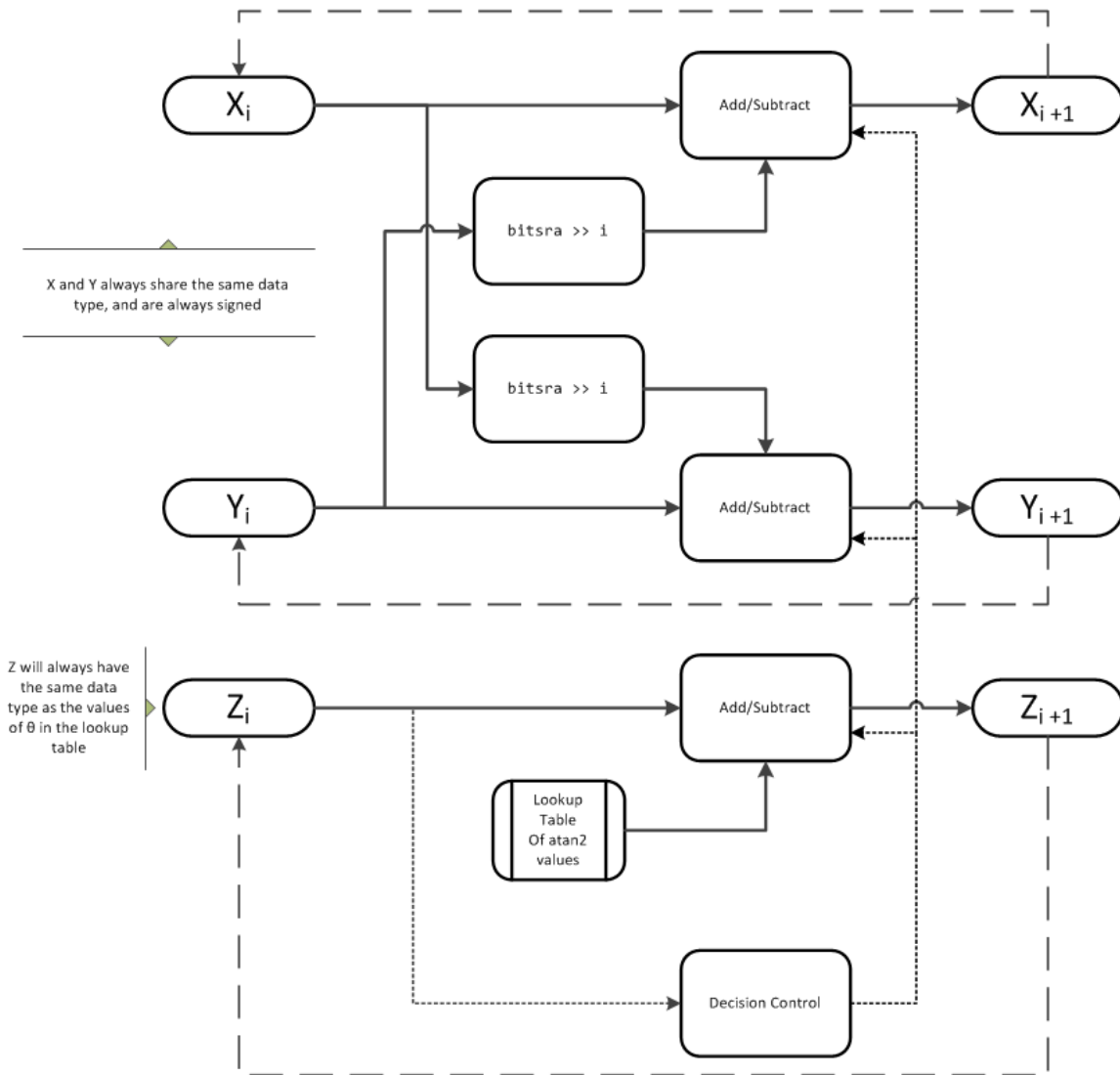
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



**CORDIC Rotation Kernel**



$X$  represents the real part,  $Y$  represents the imaginary part, and  $Z$  represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$z_0$  is initialized to the  $\theta$  input argument value

$x_0$  is initialized to  $\frac{1}{A_N}$

$y_0$  is initialized to 0

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—Wrap
- `RoundingMethod`—Floor

The output has no attached `fimath`.

## References

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`cordiccos` | `cordicsin` | `cordicsincos`

### Topics

Demo: Fixed-Point Sine and Cosine Calculation

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2010a**



# cordiccos

CORDIC-based approximation of cosine

## Syntax

```
y = cordiccos(theta, niters)
```

## Description

*y* = cordiccos(*theta*, *niters*) computes the cosine of *theta* using a “CORDIC” on page 5-312 algorithm approximation.

## Input Arguments

### **theta**

**theta** can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of **theta** must be real and in the range  $[-2\pi, 2\pi]$ .

### **niters**

**niters** is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, **niters** must be a positive, integer-valued scalar. If you do not specify **niters** or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of **theta**. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

## Output Arguments

**y**

**y** is the CORDIC-based approximation of the cosine of **theta**. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the **WordLength** - 2.

## Examples

### Compare Results of `cordiccos` and `cos` Functions

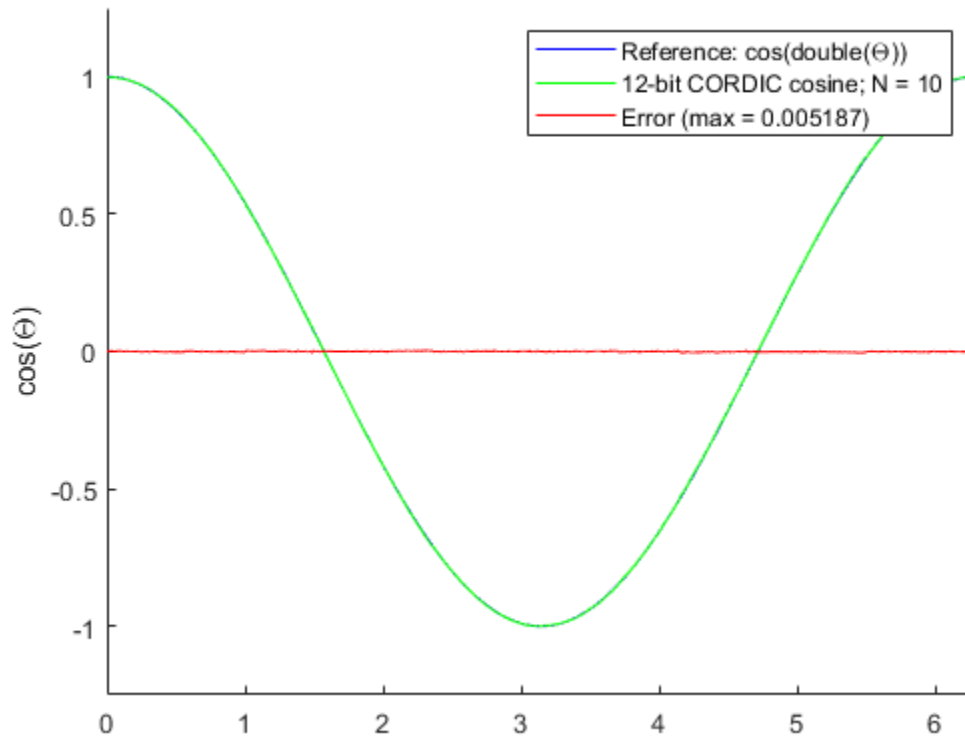
Compare the results produced by various iterations of the `cordiccos` algorithm to the results of the double-precision `cos` function.

```
% Create 1024 points between [0, 2*pi]
stepSize = pi/512;
thRadDbL = 0:stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDbL, 12); % signed, 12-bit fixed-point
cosThRef = cos(double(thRadFxp)); % reference results

% Use 12-bit quantized inputs and vary the number
% of iterations from 2 to 10.
% Compare the fixed-point CORDIC results to the
% double-precision trig function results.
for niters = 2:2:10
    cdcCosTh = cordiccos(thRadFxp, niters);
    errCdcRef = cosThRef - double(cdcCosTh);
end

figure
hold on
axis([0 2*pi -1.25 1.25]);
plot(thRadFxp, cosThRef, 'b');
plot(thRadFxp, cdcCosTh, 'g');
plot(thRadFxp, errCdcRef, 'r');
ylabel('cos(\Theta)');
gca.XTick = 0:pi/2:2*pi;
gca.XTickLabel = {'0', 'pi/2', 'pi', '3*pi/2', '2*pi'};
gca.YTick = -1:0.5:1;
```

```
gca.YTickLabel = {'-1.0', '-0.5', '0', '0.5', '1.0'};  
ref_str = 'Reference: cos(double(\Theta))';  
cdc_str = sprintf('12-bit CORDIC cosine; N = %d', niters);  
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));  
legend(ref_str, cdc_str, err_str);
```



After 10 iterations, the CORDIC algorithm has approximated the cosine of  $\theta$  to within 0.005187 of the double-precision cosine result.

## More About

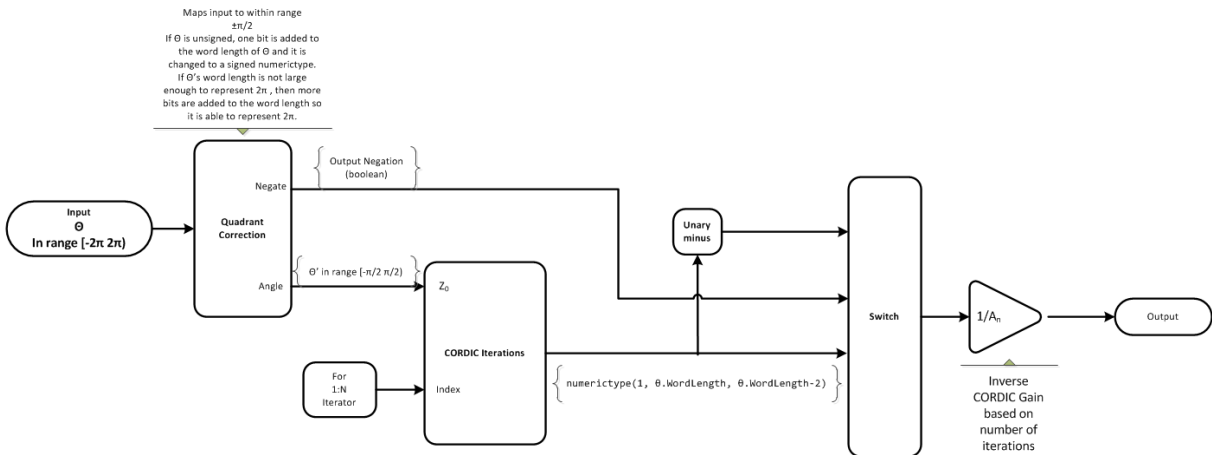
### CORDIC

CORDIC is an acronym for COordinate Rotation DIGital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

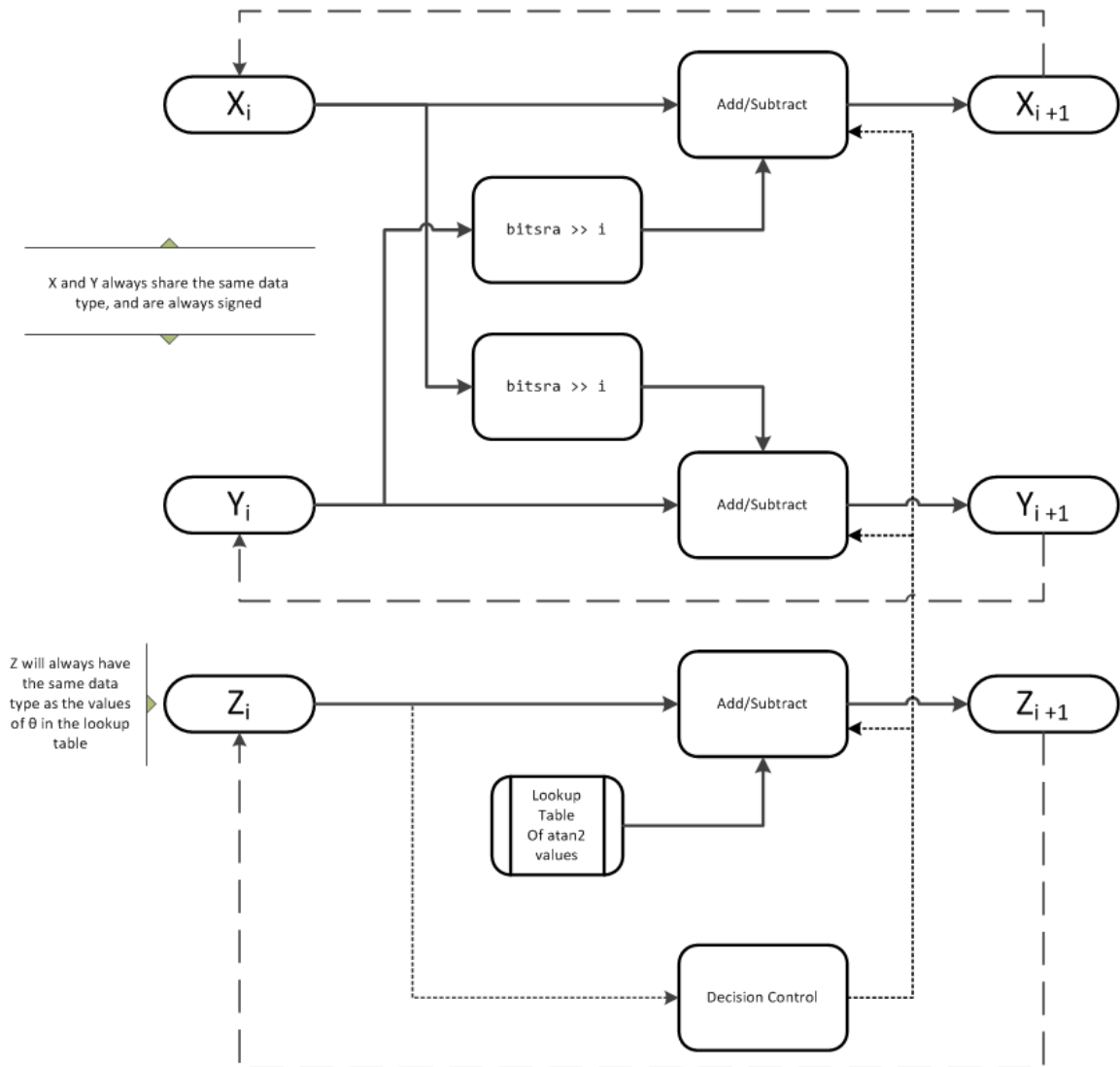
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



## CORDIC Rotation Kernel



$X$  represents the sine,  $Y$  represents the cosine, and  $Z$  represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$z_0$  is initialized to the  $\theta$  input argument value

$x_0$  is initialized to  $\frac{1}{A_N}$

$y_0$  is initialized to 0

### **fimath Propagation Rules**

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

### **References**

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`cordicexp` | `cordicsin` | `cordicsincos` | `cos` | `sin`

### Topics

Demo: Fixed-Point Sine and Cosine Calculation

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2010a**

## cordicpol2cart

CORDIC-based approximation of polar-to-Cartesian conversion

### Syntax

```
[x,y] = cordicpol2cart(theta,r)
[x,y] = cordicpol2cart(theta,r,niters)
[x,y] = cordicpol2cart(theta,r,Name,Value)
[x,y] = cordicpol2cart(theta,r,niters,Name,Value)
```

### Description

`[x,y] = cordicpol2cart(theta,r)` returns the Cartesian  $xy$  coordinates of  $r \cdot e^{j \cdot \theta}$  using a CORDIC algorithm approximation.

`[x,y] = cordicpol2cart(theta,r,niters)` performs `niters` iterations of the algorithm.

`[x,y] = cordicpol2cart(theta,r,Name,Value)` scales the output depending on the Boolean value of `b`.

`[x,y] = cordicpol2cart(theta,r,niters,Name,Value)` specifies both the number of iterations and `Name, Value` pair for whether to scale the output.

### Input Arguments

#### **theta**

`theta` can be a signed or unsigned scalar, vector, matrix, or  $N$ -dimensional array containing the angle values in radians. All values of `theta` must be in the range  $[-2\pi, 2\pi)$ .

#### **r**

`r` contains the input magnitude values and can be a scalar or have the same dimensions as `theta`. `r` must be real valued.



## **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

## **Name-Value Pair Arguments**

Optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' ' ).

### **ScaleOutput**

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

**Default:** `true`

## **Output Arguments**

### **[x,y]**

`[x,y]` contains the approximated Cartesian coordinates. When the input `r` is floating point, the output `[x,y]` has the same data type as the input.

When the input `r` is a *signed* integer or fixed point data type, the outputs `[x,y]` are signed `fi` objects. These `fi` objects have word lengths that are two bits larger than that of `r`. Their fraction lengths are the same as the fraction length of `r`.

When the input `r` is an *unsigned* integer or fixed point, the outputs `[x,y]` are signed `fi` objects. These `fi` objects have word lengths are three bits larger than that of `r`. Their fraction lengths are the same as the fraction length of `r`.

## Examples

Run the following code, and evaluate the accuracy of the CORDIC-based Polar-to-Cartesian conversion.

```

wrdLn = 16;
theta = fi(pi/3, 1, wrdLn);
u      = fi( 2.0, 1, wrdLn);

fprintf('\n\nNITERS\tX\t\t ERROR\t LSBs\t\tY\t\t ERROR\t LSBs\n');
fprintf('-----\t-----\t -----\t -----\t\t-----\t -----\t -----\n');
for niters = 1:(wrdLn - 1)
    [x_ref, y_ref] = pol2cart(double(theta),double(u));
    [x_fi, y_fi] = cordicpol2cart(theta, u, niters);
    x_dbl = double(x_fi);
    y_dbl = double(y_fi);
    x_err = abs(x_dbl - x_ref);
    y_err = abs(y_dbl - y_ref);
    fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
        niters,x_dbl,x_err,(x_err * pow2(x_fi.FractionLength)),...
        y_dbl,y_err,(y_err * pow2(y_fi.FractionLength)));
end
fprintf('\n');

```

NITERS	X	ERROR	LSBs	Y	ERROR	LSBs
1	1.4142	0.4142	3392.8	1.4142	0.3178	2603.8
2	0.6324	0.3676	3011.2	1.8973	0.1653	1354.2
3	1.0737	0.0737	603.8	1.6873	0.0448	366.8
4	0.8561	0.1440	1179.2	1.8074	0.0753	617.2
5	0.9672	0.0329	269.2	1.7505	0.0185	151.2
6	1.0214	0.0213	174.8	1.7195	0.0126	102.8
7	0.9944	0.0056	46.2	1.7351	0.0031	25.2
8	1.0079	0.0079	64.8	1.7274	0.0046	37.8
9	1.0011	0.0011	8.8	1.7313	0.0007	5.8
10	0.9978	0.0022	18.2	1.7333	0.0012	10.2
11	0.9994	0.0006	5.2	1.7323	0.0003	2.2
12	1.0002	0.0002	1.8	1.7318	0.0002	1.8
13	0.9999	0.0002	1.2	1.7321	0.0000	0.2
14	0.9996	0.0004	3.2	1.7321	0.0000	0.2
15	0.9998	0.0003	2.2	1.7321	0.0000	0.2

## More About

### CORDIC

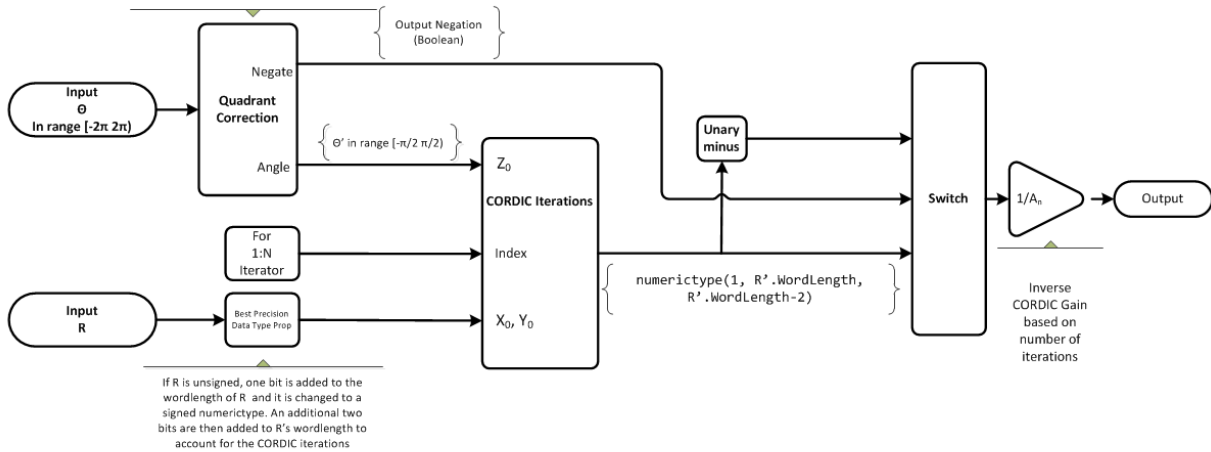
CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

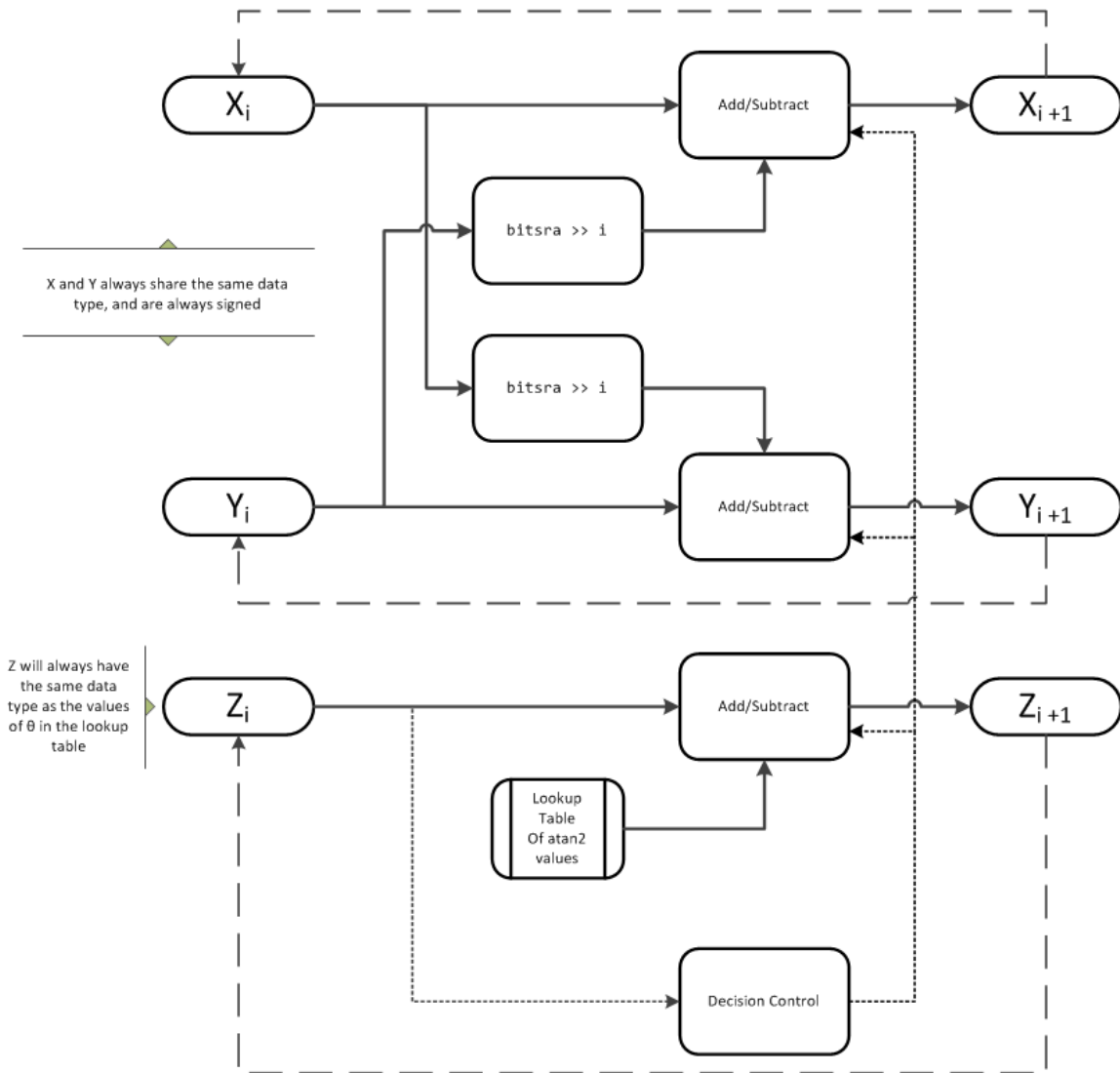
# Algorithms

## Signal Flow Diagrams

Maps input,  $\theta$ , to within range  $\pm\pi/2$ .  
 If  $\theta$  is unsigned, one bit is added to the word length of  $\theta$  and it is changed to a signed numeric type.  
 If  $\theta$ 's word length is not large enough to represent  $2\pi$ , then more bits are added to the word length so it is able to represent  $2\pi$ .



### CORDIC Rotation Kernel



$X$  represents the real part,  $Y$  represents the imaginary part, and  $Z$  represents theta. This algorithm takes its initial values for  $X$ ,  $Y$ , and  $Z$  from the inputs, `r` and `theta`.

### **fimath Propagation Rules**

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

### **References**

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

## See Also

`cordicrotate` | `cordicsincos` | `pol2cart`

**Introduced in R2011a**

## cordicrotate

Rotate input using CORDIC-based approximation

### Syntax

```
v = cordicrotate(theta,u)
v = cordicrotate(theta,u,niters)
v = cordicrotate(theta,u,Name,Value)
v = cordicrotate(theta,u,niters,Name,Value)
```

### Description

`v = cordicrotate(theta,u)` rotates the input `u` by `theta` using a CORDIC algorithm approximation. The function returns the result of `u .* e^(j*theta)`.

`v = cordicrotate(theta,u,niters)` performs `niters` iterations of the algorithm.

`v = cordicrotate(theta,u,Name,Value)` scales the output depending on the Boolean value, `b`.

`v = cordicrotate(theta,u,niters,Name,Value)` specifies both the number of iterations and the `Name,Value` pair for whether to scale the output.

### Input Arguments

#### **theta**

`theta` can be a signed or unsigned scalar, vector, matrix, or  $N$ -dimensional array containing the angle values in radians. All values of `theta` must be in the range  $[-2\pi, 2\pi]$ .

#### **u**

`u` can be a signed or unsigned scalar value or have the same dimensions as `theta`. `u` can be real or complex valued.



## **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `u` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

## **Name-Value Pair Arguments**

Optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ).

### **ScaleOutput**

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

**Default:** `true`

## **Output Arguments**

### **v**

`v` contains the approximated result of the CORDIC rotation algorithm. When the input `u` is floating point, the output `v` has the same data type as the input.

When the input `u` is a *signed* integer or fixed point data type, the output `v` is a signed `fi` object. This `fi` object has a word length that is two bits larger than that of `u`. Its fraction length is the same as the fraction length of `u`.

When the input `u` is an *unsigned* integer or fixed point, the output `v` is a signed `fi` object. This `fi` object has a word length that is three bits larger than that of `u`. Its fraction length is the same as the fraction length of `u`.

## Examples

Run the following code, and evaluate the accuracy of the CORDIC-based complex rotation.

```

wrdLn = 16;
theta = fi(-pi/3, 1, wrdLn);
u      = fi(0.25 - 7.1i, 1, wrdLn);
uTeTh = double(u) .* exp(1i * double(theta));

fprintf('\n\nNITERS\tReal\t ERROR\t LSBs\t\tImag\tERROR\tLSBs\n');
fprintf('-----\t-----\t -----\t ----\t\t-----\t-----\t----\n');
for niters = 1:(wrdLn - 1)
    v_fi = cordicrotate(theta, u, niters);
    v_dbl = double(v_fi);
    x_err = abs(real(v_dbl) - real(uTeTh));
    y_err = abs(imag(v_dbl) - imag(uTeTh));
    fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
        niters, real(v_dbl),x_err,(x_err * pow2(v_fi.FractionLength)), ...
        imag(v_dbl),y_err, (y_err * pow2(v_fi.FractionLength)));
end
fprintf('\n');

```

The output table appears as follows:

NITERS	Real	ERROR	LSBs	Imag	ERROR	LSBs
1	-4.8438	1.1800	4833.5	-5.1973	1.4306	5859.8
2	-6.6567	0.6329	2592.5	-2.4824	1.2842	5260.2
3	-5.8560	0.1678	687.5	-4.0227	0.2560	1048.8
4	-6.3098	0.2860	1171.5	-3.2649	0.5018	2055.2
5	-6.0935	0.0697	285.5	-3.6528	0.1138	466.2
6	-5.9766	0.0472	193.5	-3.8413	0.0746	305.8
7	-6.0359	0.0121	49.5	-3.7476	0.0191	78.2
8	-6.0061	0.0177	72.5	-3.7947	0.0280	114.8
9	-6.0210	0.0028	11.5	-3.7710	0.0043	17.8
10	-6.0286	0.0048	19.5	-3.7590	0.0076	31.2
11	-6.0247	0.0009	3.5	-3.7651	0.0015	6.2
12	-6.0227	0.0011	4.5	-3.7683	0.0017	6.8
13	-6.0237	0.0001	0.5	-3.7666	0.0001	0.2
14	-6.0242	0.0004	1.5	-3.7656	0.0010	4.2
15	-6.0239	0.0001	0.5	-3.7661	0.0005	2.2

## More About

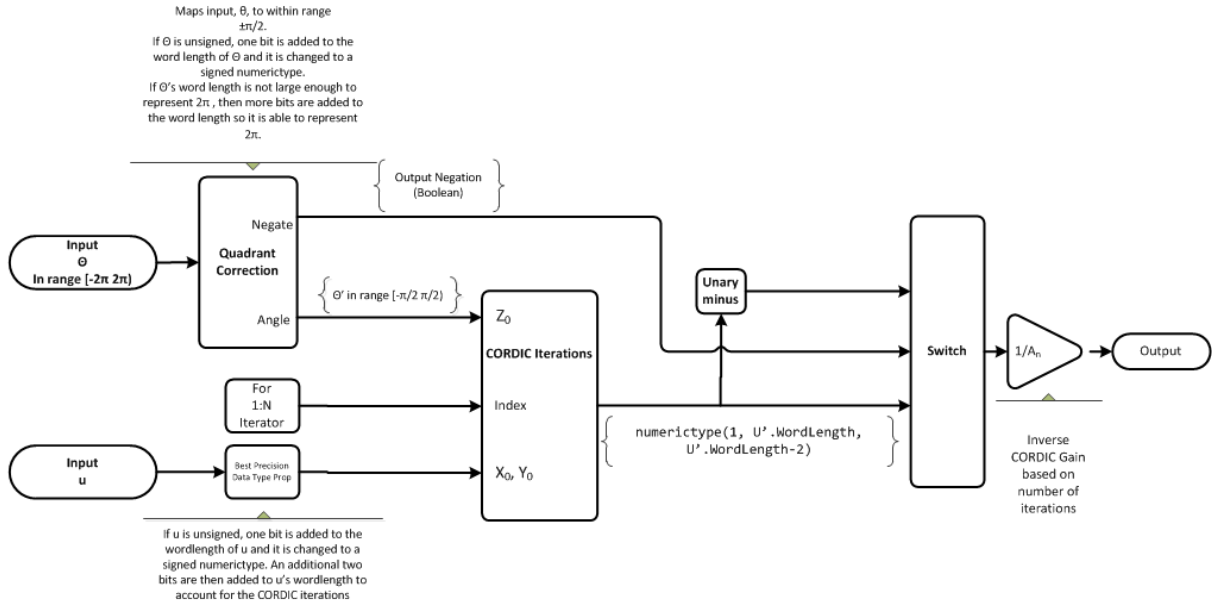
### CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

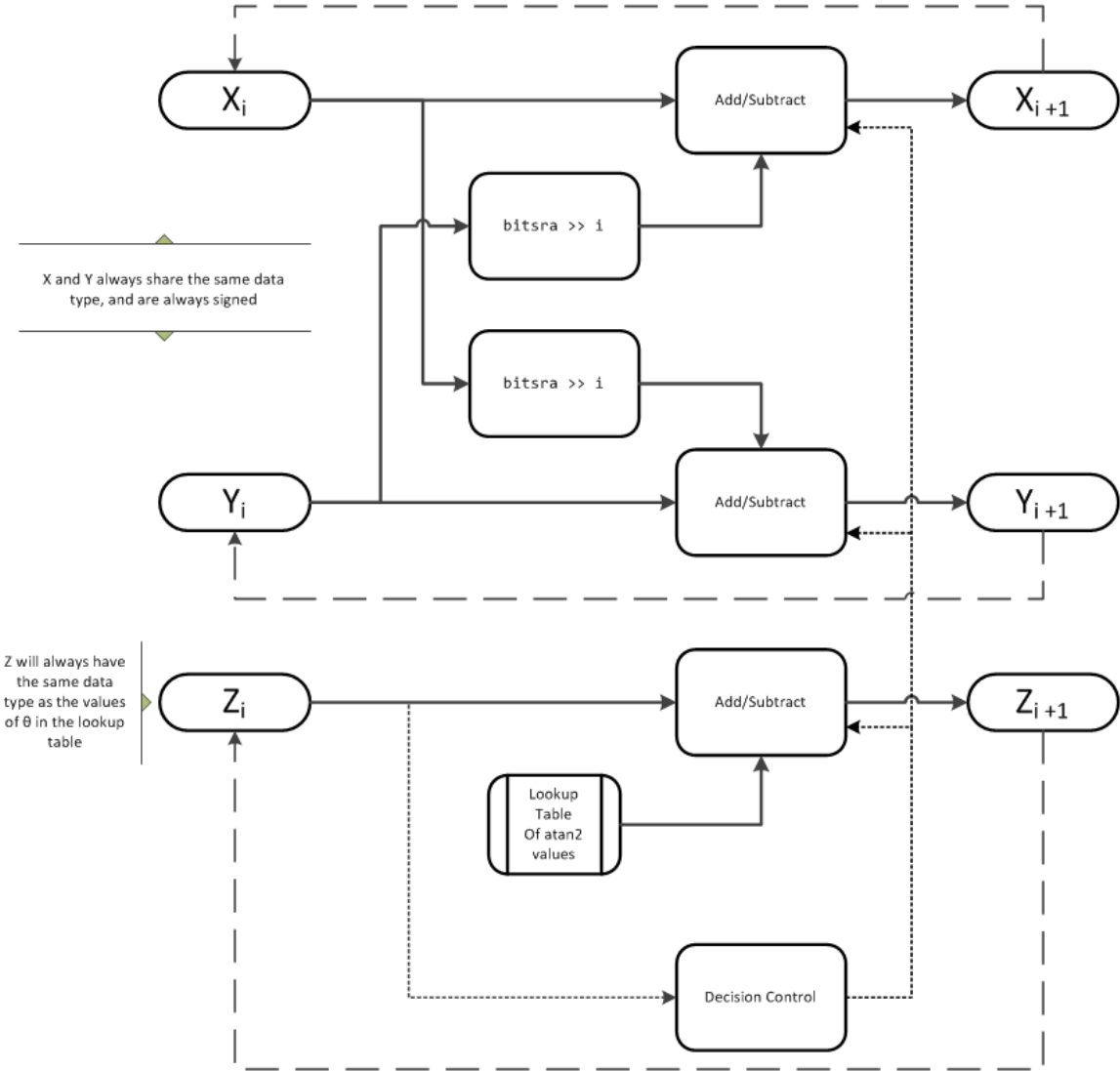
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

# Algorithms

## Signal Flow Diagrams



### CORDIC Rotation Kernel



$X$  represents the real part,  $Y$  represents the imaginary part, and  $Z$  represents theta. This algorithm takes its initial values for  $X$ ,  $Y$ , and  $Z$  from the inputs, `u` and `theta`.

### **fimath Propagation Rules**

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

### **References**

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

## See Also

`cordicexp` | `cordicpol2cart`

**Introduced in R2011a**

## cordicsin

CORDIC-based approximation of sine

### Syntax

```
y = cordicsin(theta, niters)
```

### Description

`y = cordicsin(theta, niters)` computes the sine of `theta` using a “CORDIC” on page 5-335 algorithm approximation.

### Input Arguments

#### **theta**

`theta` can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range  $[-2\pi, 2\pi]$ .

#### **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.



## Output Arguments

**y**

**y** is the CORDIC-based approximation of the sine of **theta**. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the **WordLength** - 2.

## Examples

### Compare Results of cordicsin and sin Functions

Compare the results produced by various iterations of the **cordicsin** algorithm to the results of the double-precision **sin** function.

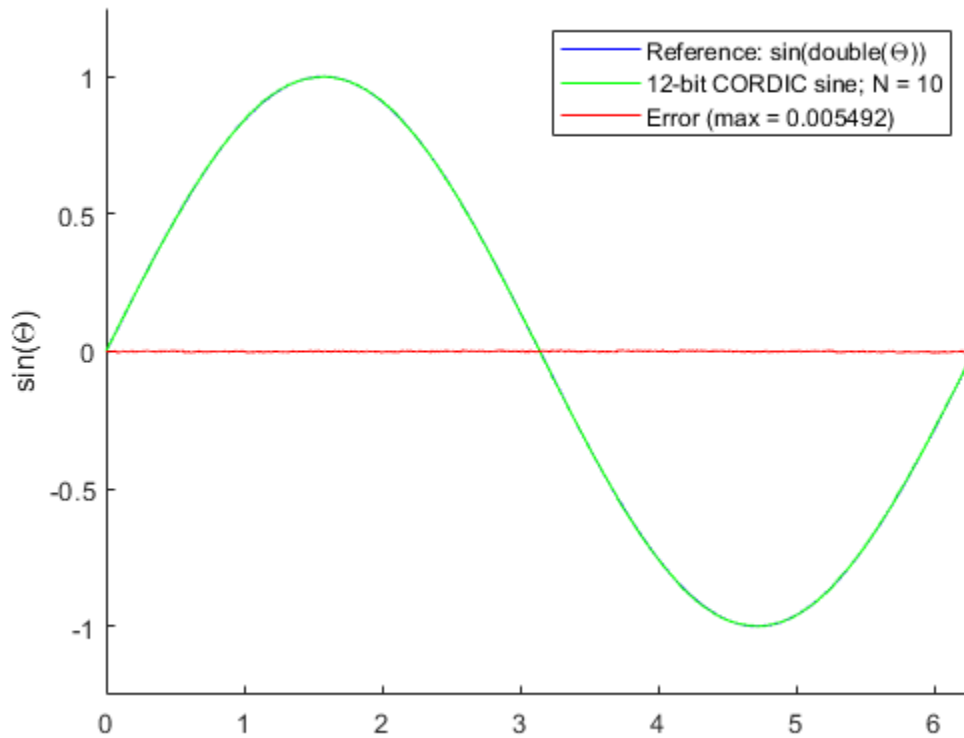
```
% Create 1024 points between [0, 2*pi]
stepSize = pi/512;
thRadDb1 = 0:stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDb1, 12);      % signed, 12-bit fixed point
sinThRef = sin(double(thRadFxp)); % reference results

% Use 12-bit quantized inputs and vary the number of iterations
% from 2 to 10.
% Compare the fixed-point cordicsin function results to the
% results of the double-precision sin function.

for niters = 2:2:10
    cdcSinTh = cordicsin(thRadFxp, niters);
    errCdcRef = sinThRef - double(cdcSinTh);
end

figure
hold on
axis([0 2*pi -1.25 1.25])
plot(thRadFxp, sinThRef, 'b');
plot(thRadFxp, cdcSinTh, 'g');
plot(thRadFxp, errCdcRef, 'r');
ylabel('sin(\Theta)');
gca.XTick = 0:pi/2:2*pi;
gca.XTickLabel = {'0', 'pi/2', 'pi', '3*pi/2', '2*pi'};
```

```
gca.YTick = -1:0.5:1;
gca.YTickLabel = {'-1.0', '-0.5', '0', '0.5', '1.0'};
ref_str = 'Reference: sin(double(\Theta))';
cdc_str = sprintf('12-bit CORDIC sine; N = %d', niters);
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
legend(ref_str, cdc_str, err_str);
```



After 10 iterations, the CORDIC algorithm has approximated the sine of  $\theta$  to within 0.005492 of the double-precision sine result.

## More About

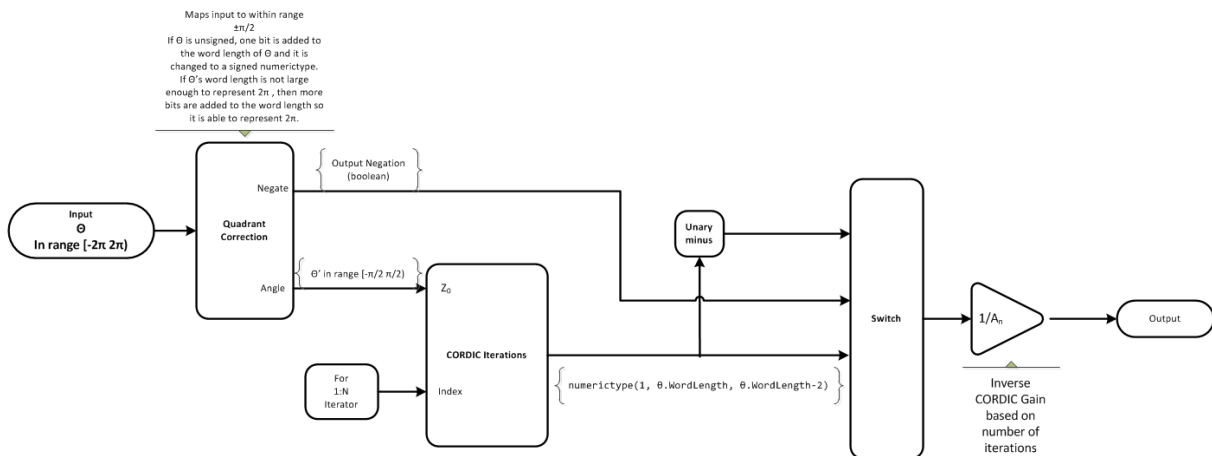
### CORDIC

CORDIC is an acronym for COordinate Rotation DIGital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

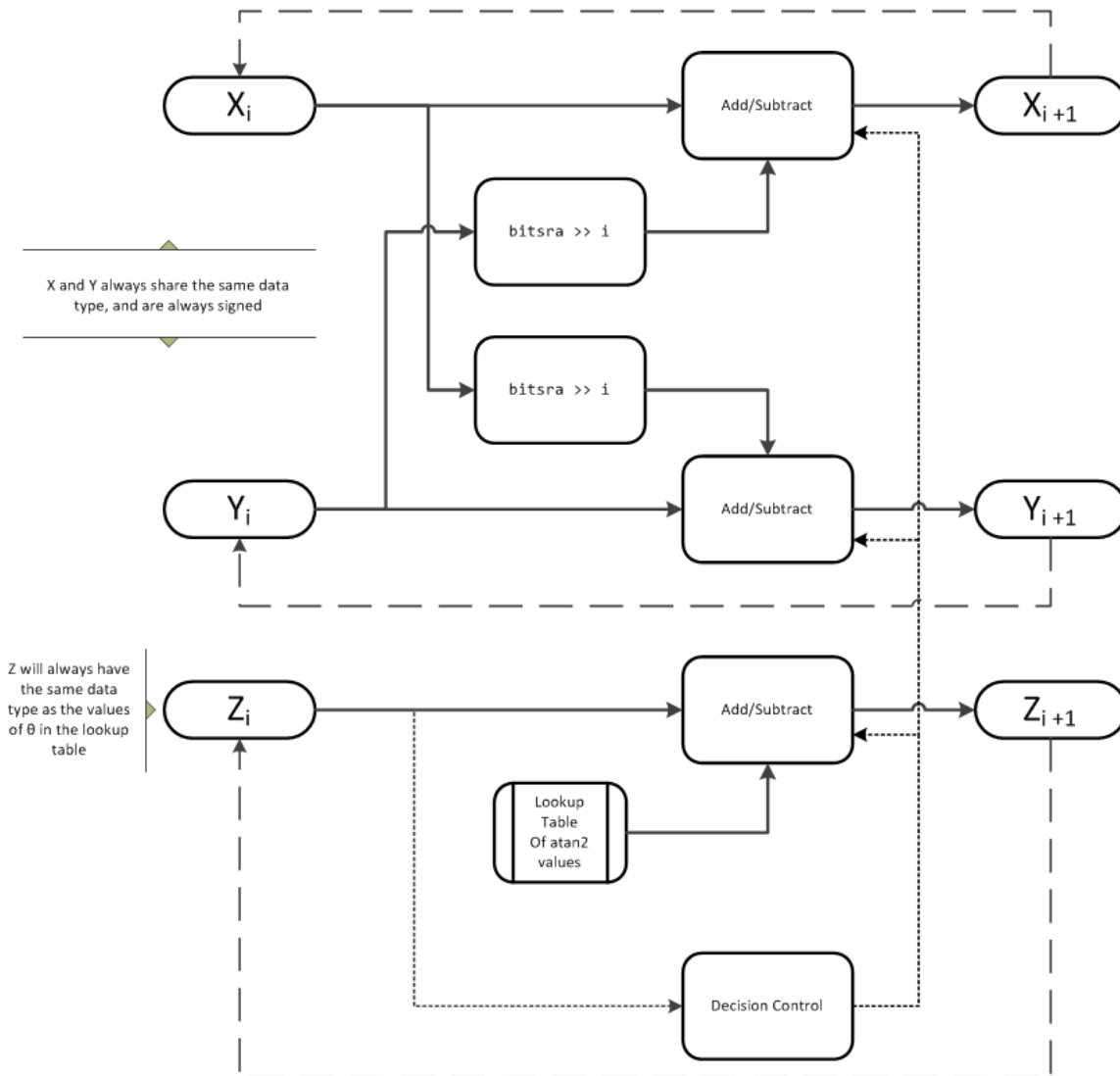
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



**CORDIC Rotation Kernel**



$X$  represents the sine,  $Y$  represents the cosine, and  $Z$  represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$z_0$  is initialized to the  $\theta$  input argument value

$x_0$  is initialized to  $\frac{1}{A_N}$

$y_0$  is initialized to 0

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## References

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`cordicexp` | `cordiccos` | `cordicsincos` | `cos` | `sin`

### Topics

Demo: Fixed-Point Sine and Cosine Calculation

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2010a**

# cordicsincos

CORDIC-based approximation of sine and cosine

## Syntax

```
[y, x] = cordicsincos(theta, niters)
```

## Description

`[y, x] = cordicsincos(theta, niters)` computes the sine and cosine of `theta` using a “CORDIC” on page 5-341 algorithm approximation. `y` contains the approximated sine result, and `x` contains the approximated cosine result.

## Input Arguments

### **theta**

`theta` can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range  $[-2\pi, 2\pi)$ . When `theta` has a fixed-point data type, it must be signed.

### **niters**

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

## Output Arguments

### y

CORDIC-based approximated sine of theta. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength - 2`.

### x

CORDIC-based approximated cosine of theta. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength - 2`.

## Examples

The following example illustrates the effect of the number of iterations on the result of the `cordicsincos` approximation.

```
wrdLn = 8;
theta = fi(pi/2, 1, wrdLn);
fprintf('\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n');
fprintf('-----\t\t-----\t -----\t -----\t\t-----\t -----\t -----\n');
for niters = 1:(wrdLn - 1)
    [y, x] = cordicsincos(theta, niters);
    y_FL = y.FractionLength;
    y_dbl = double(y);
    x_dbl = double(x);
    y_err = abs(y_dbl - sin(double(theta)));
    x_err = abs(x_dbl - cos(double(theta)));
    fprintf(' %d\t\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n', ...
        niters, y_dbl, y_err, (y_err * pow2(y_FL)), x_dbl, x_err, ...
        (x_err * pow2(y_FL)));
end
fprintf('\n');
```

The output table appears as follows:

NITERS	Y (SIN)	ERROR	LSBs	X (COS)	ERROR	LSBs
-----	-----	-----	-----	-----	-----	-----



1	0.7031	0.2968	19.0	0.7031	0.7105	45.5
2	0.9375	0.0625	4.0	0.3125	0.3198	20.5
3	0.9844	0.0156	1.0	0.0938	0.1011	6.5
4	0.9844	0.0156	1.0	-0.0156	0.0083	0.5
5	1.0000	0.0000	0.0	0.0312	0.0386	2.5
6	1.0000	0.0000	0.0	0.0000	0.0073	0.5
7	1.0000	0.0000	0.0	0.0156	0.0230	1.5

## More About

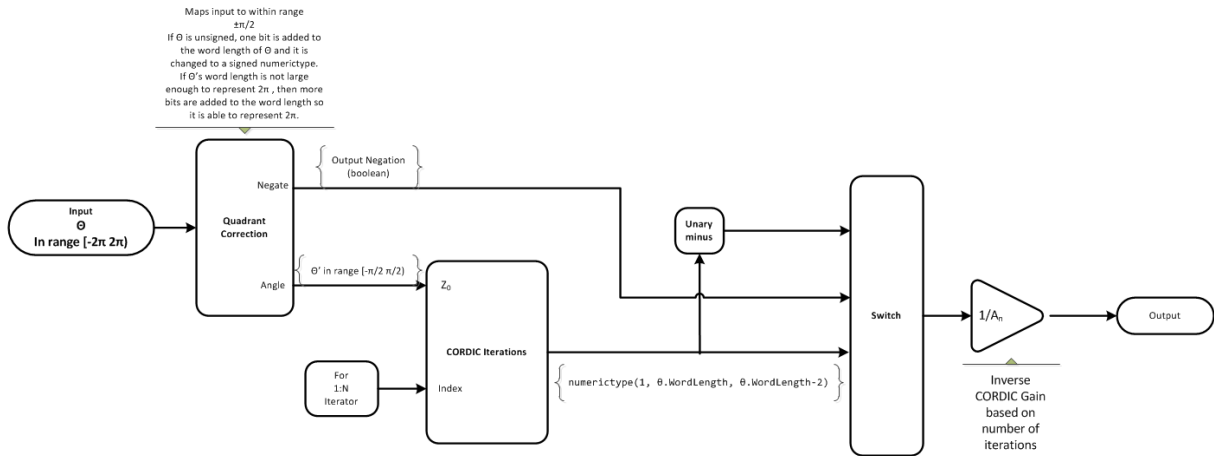
### CORDIC

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

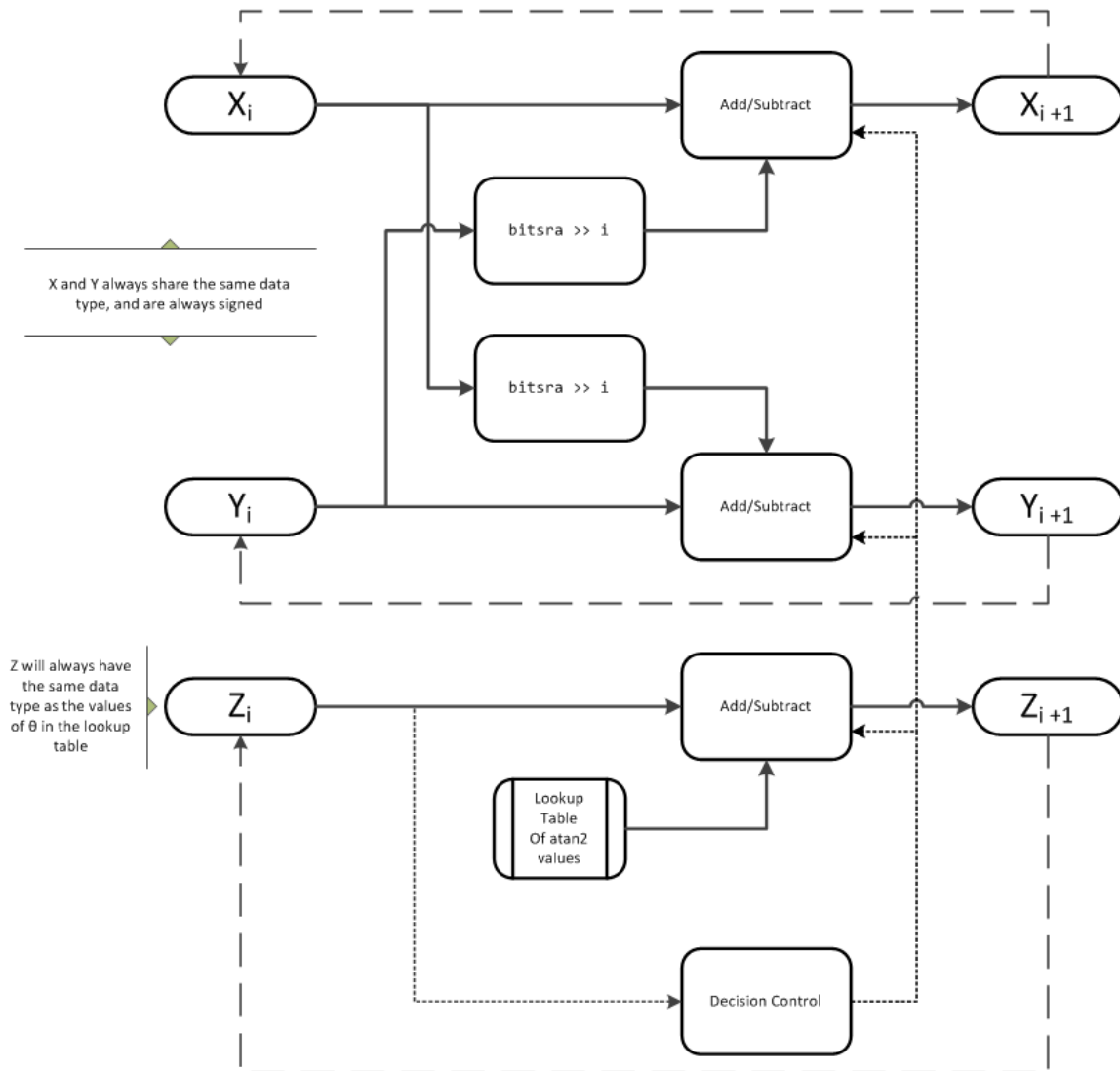
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

# Algorithms

## Signal Flow Diagrams



## CORDIC Rotation Kernel



$X$  represents the sine,  $Y$  represents the cosine, and  $Z$  represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for  $X$ ,  $Y$ , and  $Z$ . This algorithm uses the following initial values:

$z_0$  is initialized to the  $\theta$  input argument value

$x_0$  is initialized to  $\frac{1}{A_N}$

$y_0$  is initialized to 0

### **fimath Propagation Rules**

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

### **References**

- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. "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, pp. 191-200.
- [3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`cordiccxp` | `cordiccos` | `cordicsin`

### Topics

Demo: Fixed-Point Sine and Cosine Calculation

Demo: Fixed-Point Arctangent Calculation

**Introduced in R2010a**

## cordicsqrt

CORDIC-based approximation of square root

### Syntax

```
y=cordicsqrt(u)
y=cordicsqrt(u, niters)
y=cordicsqrt( ____, 'ScaleOutput', B)
```

### Description

`y=cordicsqrt(u)` computes the square root of `u` using a CORDIC algorithm implementation.

`y=cordicsqrt(u, niters)` computes the square root of `u` by performing `niters` iterations of the CORDIC algorithm.

`y=cordicsqrt( ____, 'ScaleOutput', B)` scales the output depending on the Boolean value of `B`.

### Examples

#### Calculate the CORDIC Square Root

Find the square root of `fi` object `x` using a CORDIC implementation.

```
x = fi(1.6,1,12);
y = cordicsqrt(x)
```

```
y =
    1.2646
```

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

```

    WordLength: 12
    FractionLength: 10

```

Because you did not specify `niters`, the function performs the maximum number of iterations, `x.WordLength - 1`.

Compute the difference between the results of the `cordicsqrt` function and the double-precision `sqrt` function.

```

err = abs(sqrt(double(x))-double(y))
err = 1.0821e-04

```

### Calculate the CORDIC Square Root With a Specified Number of Iterations

Compute the square root of `x` with three iterations of the CORDIC kernel.

```

x = fi(1.6,1,12);
y = cordicsqrt(x,3)

y =
    1.2646

```

```

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 12
    FractionLength: 10

```

Compute the difference between the results of the `cordicsqrt` function and the double-precision `sqrt` function.

```

err = abs(sqrt(double(x))-double(y))
err = 1.0821e-04

```

### Calculate the CORDIC Square Root Without Scaling the Output

```

x = fi(1.6,1,12);
y = cordicsqrt(x, 'ScaleOutput', 0)

```

```
y =  
    1.0479  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 12  
    FractionLength: 10
```

The output, *y*, was not scaled by the inverse CORDIC gain factor.

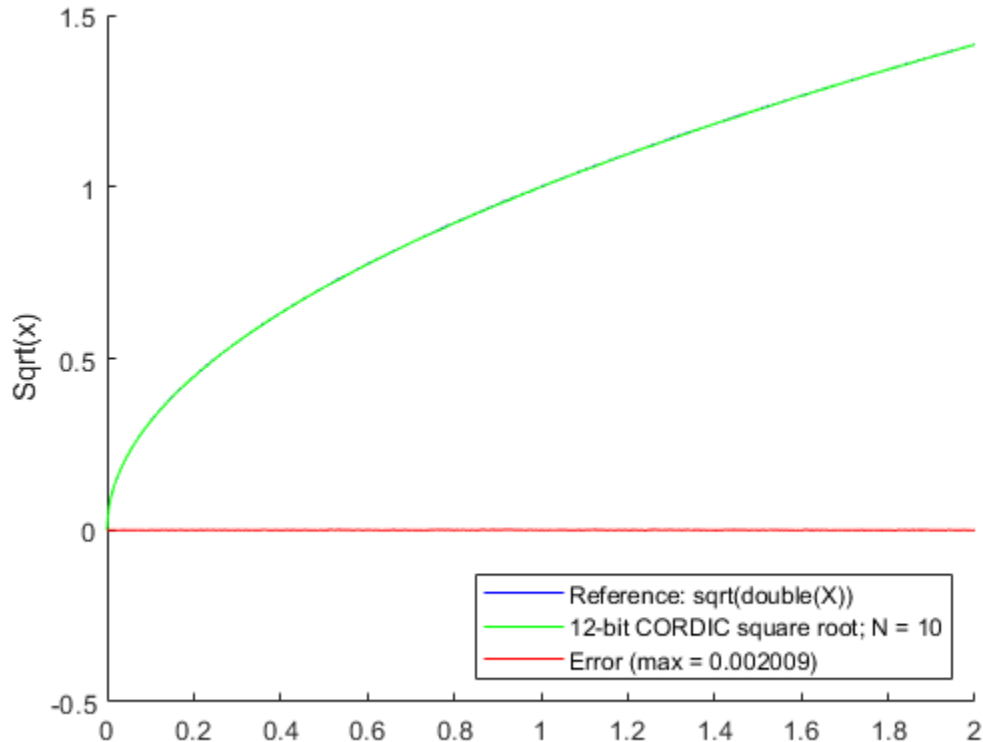
### Compare Results of `cordicsqrt` and `sqrt` Functions

Compare the results produced by 10 iterations of the `cordicsqrt` algorithm to the results of the double-precision `sqrt` function.

```
% Create 500 points between [0, 2)  
stepSize = 2/500;  
XDb1 = 0:stepSize:2;  
XFxp = fi(XDb1, 1, 12);    % signed, 12-bit fixed-point  
sqrtXRef = sqrt(double(XFxp)); % reference results  
  
% Use 12-bit quantized inputs and set the number  
% of iterations to 10.  
% Compare the fixed-point CORDIC results to the  
% double-precision sqrt function results.  
  
niters = 10;  
cdcSqrtX = cordicsqrt(XFxp, niters);  
errCdcRef = sqrtXRef - double(cdcSqrtX);  
figure  
hold on  
axis([0 2 -.5 1.5])  
plot(XFxp, sqrtXRef, 'b')  
plot(XFxp, cdcSqrtX, 'g')  
plot(XFxp, errCdcRef, 'r')  
ylabel('Sqrt(x)')  
gca.XTick = 0:0.25:2;  
gca.XTickLabel = {'0', '0.25', '0.5', '0.75', '1', '1.25', '1.5', '1.75', '2'};  
gca.YTick = -.5:.25:1.5;  
gca.YTickLabel = {'-0.5', '-0.25', '0', '0.25', '0.5', '0.75', '1', '1.25', '1.5'};  
ref_str = 'Reference: sqrt(double(X))';  
cdc_str = sprintf('12-bit CORDIC square root; N = %d', niters);
```



```
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));  
legend(ref_str, cdc_str, err_str, 'Location', 'southeast')
```



## Input Arguments

### **u** — Data input array

scalar | vector | matrix | multidimensional array

Data input array, specified as a positive scalar, vector, matrix, or multidimensional array of fixed-point or built-in data types. When the input array contains values between 0.5 and 2, the algorithm is most accurate. A pre- and post-normalization process is performed on

input values outside of this range. For more information on this process, see “Pre- and Post-Normalization” on page 5-353.

**Data Types:** `fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

### **niters — Number of iterations**

scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is `u.WordLength - 1`. For floating-point inputs, the default value of `niters` is 52 for double precision; 23 for single precision.

**Data Types:** `fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `y= cordicsqrt(x, 'ScaleOutput', 0)`

### **ScaleOutput — Whether to scale the output**

`true (default) | false`

Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

**Data Types:** `logical`

## **Output Arguments**

### **y — Output array**

`scalar | vector | matrix | multidimensional array`

Output array, returned as a scalar, vector, matrix, or multidimensional array.

## More About

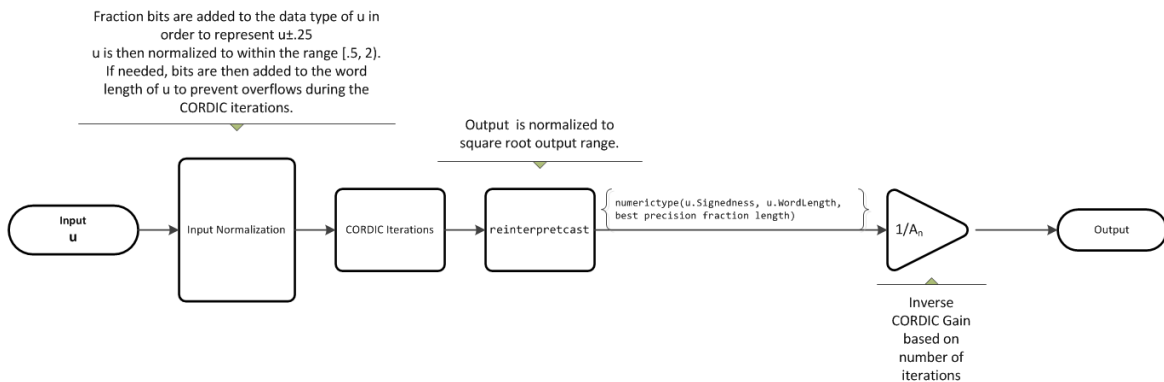
### CORDIC

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

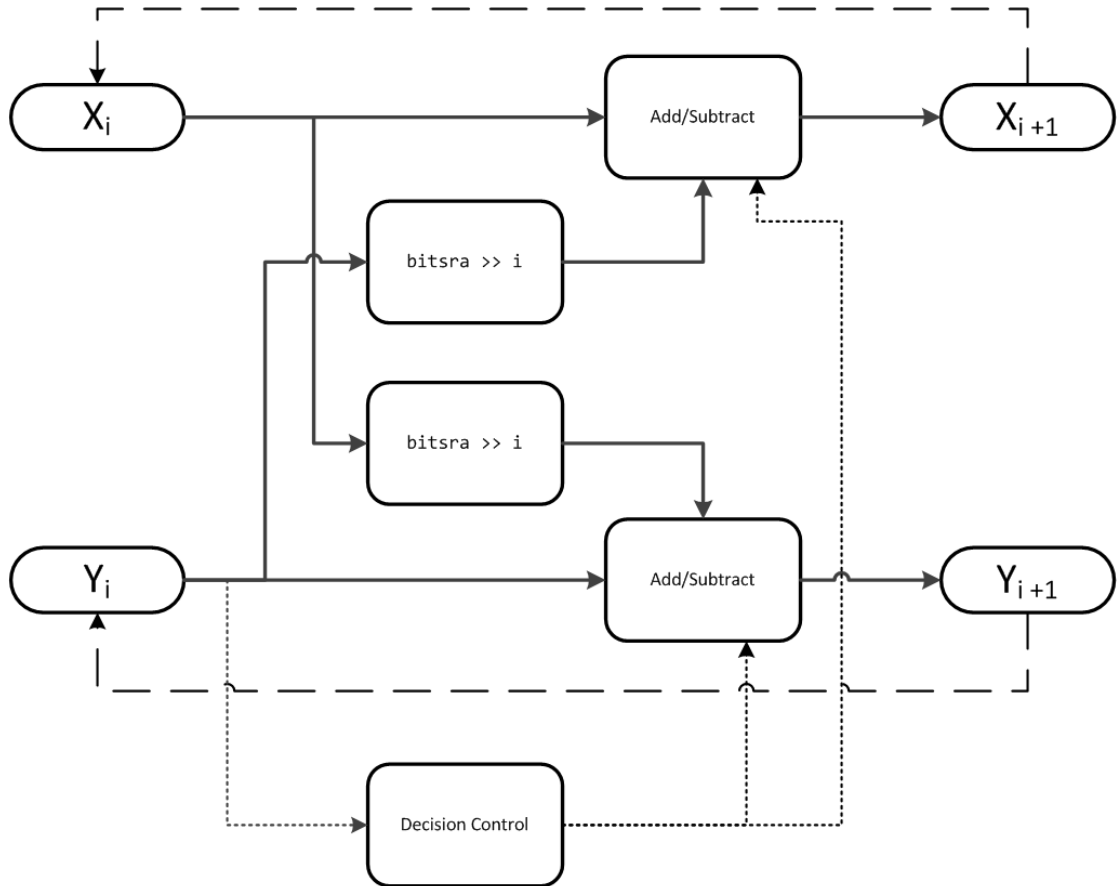
## Algorithms

### Signal Flow Diagrams



For further details on the pre- and post-normalization process, see “Pre- and Post-Normalization” on page 5-353.

**CORDIC Hyperbolic Kernel**



$X$  is initialized to  $u' + .25$ , and  $Y$  is initialized to  $u' - .25$ , where  $u'$  is the normalized function input.

With repeated iterations of the CORDIC hyperbolic kernel,  $X$  approaches  $A_N\sqrt{u}$ , where  $A_N$  represents the CORDIC gain.  $Y$  approaches 0.

## Pre- and Post-Normalization

For input values outside of the range of [0.5, 2) a pre- and post-normalization process occurs. This process performs bitshifts on the input array before passing it to the CORDIC kernel. The result is then shifted back into the correct output range during the post-normalization stage. For more details on this process see “Overcoming Algorithm Input Range Limitations” in “Compute Square Root Using CORDIC”.

## fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction—Wrap`
- `RoundingMethod—Floor`

The output has no attached `fimath`.

## References

- [1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330-334.
- [2] Andraka, R. “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, pp. 191-200.
- [3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379-386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)
- [4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317-325.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

### See Also

`sqrt`

### Topics

“Compute Square Root Using CORDIC”

**Introduced in R2014a**

# cordictanh

CORDIC-based hyperbolic tangent

## Syntax

```
T = cordictanh(theta)
T = cordictanh(theta, niters)
```

## Description

`T = cordictanh(theta)` returns the hyperbolic tangent of `theta`.

`T = cordictanh(theta, niters)` returns the hyperbolic tangent of `theta` by performing `niters` iterations of the CORDIC algorithm.

## Examples

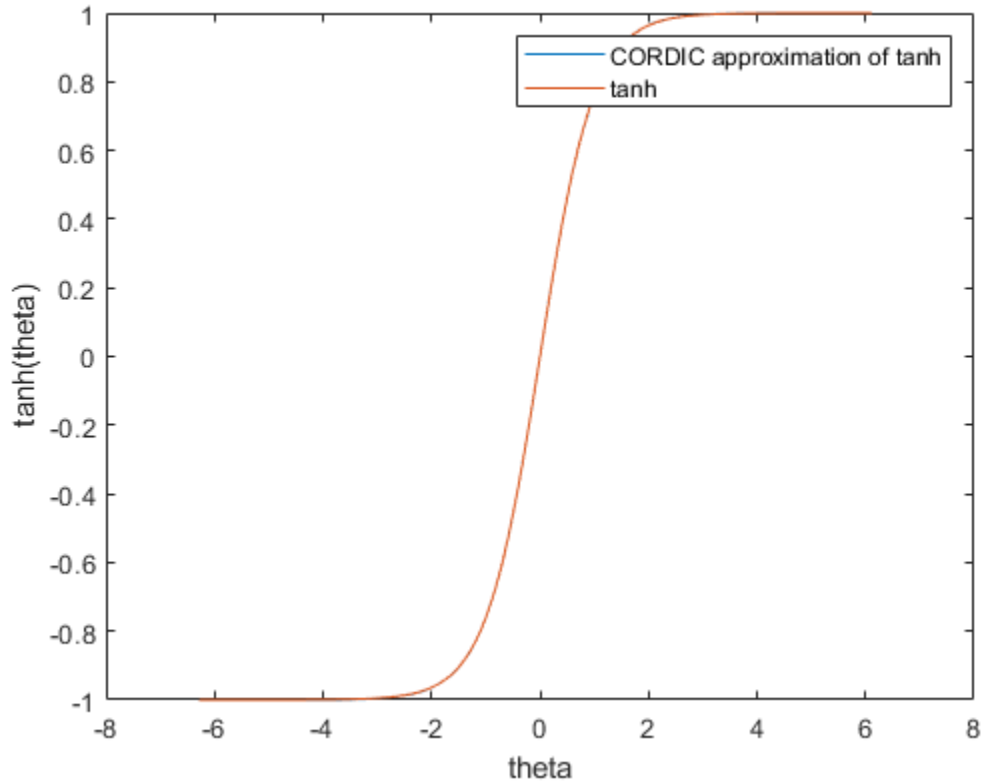
### Compute CORDIC Hyperbolic Tangent

Find the hyperbolic tangent of `fi` object `theta` using a CORDIC implementation with the default number of iterations.

```
theta = fi(-2*pi:.1:2*pi-.1);
T_cordic = cordictanh(theta);
```

Plot the hyperbolic tangent of `theta` using the `tanh` function and its CORDIC approximation.

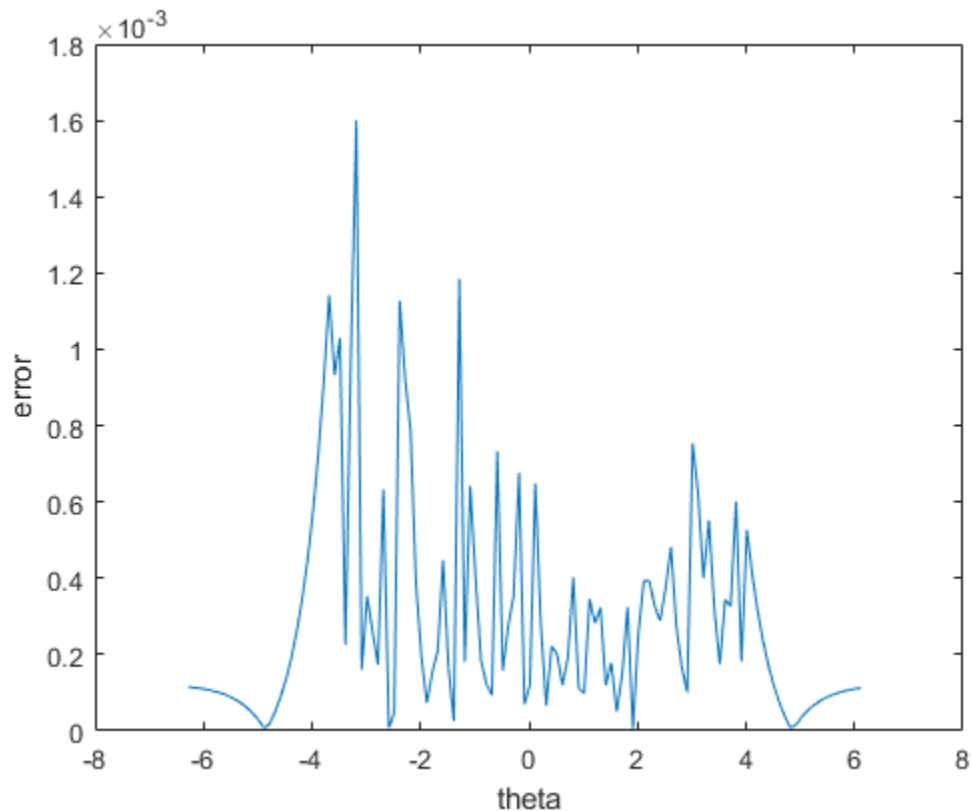
```
T = tanh(double(theta));
plot(theta, T_cordic);
hold on;
plot(theta, T);
legend('CORDIC approximation of tanh', 'tanh');
xlabel('theta');
ylabel('tanh(theta)');
```



Compute the difference between the results of the `cordictanh` function and the `tanh` function.

```
figure;  
err = abs(T - double(T_cordic));  
plot(theta, err);  
xlabel('theta');  
ylabel('error');
```



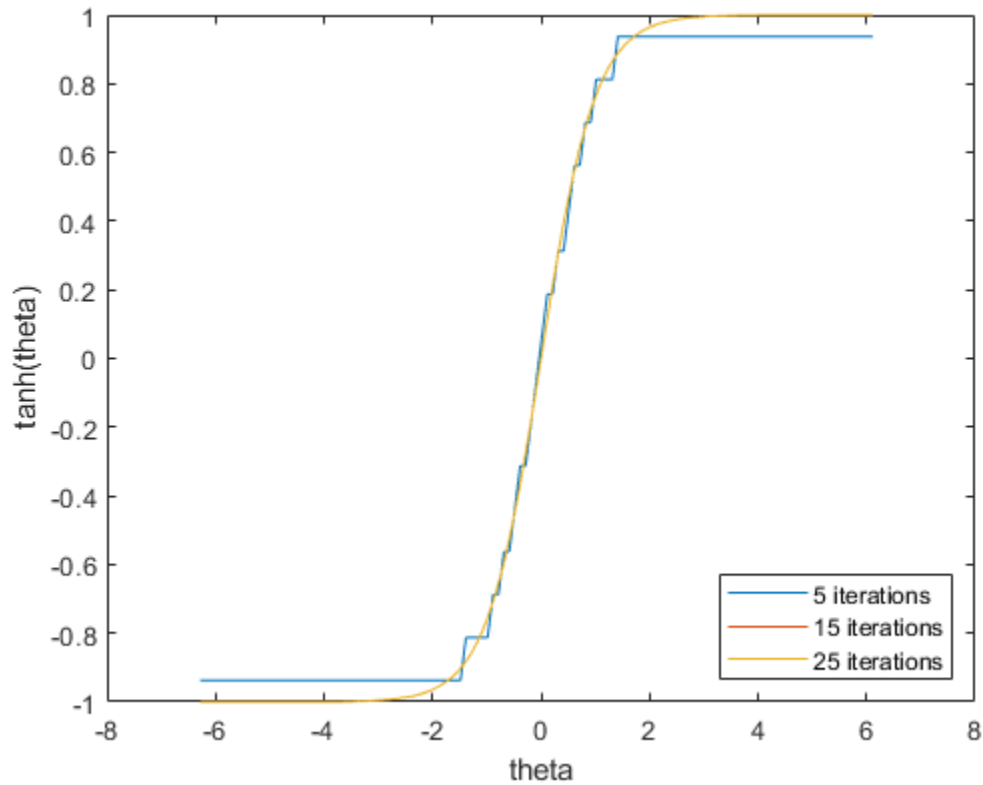


### Compute CORDIC Hyperbolic Tangent with Specified Number of Iterations

Find the hyperbolic tangent of `fi` object `theta` using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the hyperbolic tangent of `theta` with varying numbers of iterations.

```
theta = fi(-2*pi:.1:2*pi-.1);  
for niters = 5:10:25  
    T_cordic = cordictanh(theta, niters);  
    plot(theta, T_cordic);  
hold on;
```

```
end
xlabel('theta');
ylabel('tanh(theta)');
legend('5 iterations', '15 iterations', '25 iterations','Location','southeast');
```



## Input Arguments

### **theta** — angle values

scalar | vector | matrix | n-dimensional array

Angle values in radians specified as a scalar, vector, matrix, or N-dimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### **niters** — Number of iterations

scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array, `theta`. For double-precision inputs, the default value of `niters` is 52. For single-precision inputs, the default value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **T** — Output array

scalar | vector | matrix | n-dimensional array

T is the CORDIC-based approximation of the hyperbolic tangent of `theta`. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` - 2.

## See Also

`cordicatan2` | `cordiccos` | `cordicsin` | `tanh`

**Introduced in R2017b**

## **COS**

Cosine of `fi` object

### **Syntax**

```
y = cos(theta)
```

### **Description**

`y = cos(theta)` returns the cosine on page 5-361 of `fi` input `theta` using a table-lookup algorithm.

### **Input Arguments**

#### **theta**

`theta` can be a real-valued, signed or unsigned scalar, vector, matrix, or N-dimensional array containing the fixed-point angle values in radians. Valid data types of `theta` are:

- `fi single`
- `fi double`
- `fi fixed-point with binary point scaling`
- `fi scaled double with binary point scaling`

### **Output Arguments**

#### **y**

`y` is the cosine of `theta`. `y` is a signed, fixed-point number in the range `[-1,1]`. It has a 16-bit word length and 15-bit fraction length (`numericType(1,16,15)`).

## Examples

Calculate the cosine of fixed-point input values.

```
theta = fi([0,pi/4,pi/3,pi/2,(2*pi)/3,(3*pi)/4,pi])
```

```
theta =
```

```
0 0.7854 1.0472 1.5708 2.0944 2.3562 3.1416
```

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

```
y = cos(theta)
```

```
y =
```

```
1.0000 0.7072 0.4999 0.0001 -0.4999 -0.7070 -1.0000
```

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

## More About

### Cosine

The cosine of angle  $\theta$  is defined as

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}$$

## Algorithms

The `cos` function computes the cosine of fixed-point input using an 8-bit lookup table as follows:

- 1 Perform a modulo  $2\pi$ , so the input is in the range  $[0, 2\pi)$  radians.
- 2 Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
- 3 Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
- 4 Use the 8 most-significant bits to obtain the first value from the table.
- 5 Use the next-greater table value as the second value.
- 6 Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

### **fimath Propagation Rules**

The `cos` function ignores and discards any `fimath` attached to the input, `theta`. The output, `y`, is always associated with the default `fimath`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`angle` | `atan2` | `cordiccos` | `cordicsin` | `cos` | `sin`

### **Topics**

Demo: Fixed-Point Sine and Cosine Calculation

**Introduced in R2012a**

## **ctranspose**

Complex conjugate transpose of `fi` object

### **Syntax**

```
ctranspose(a)
```

### **Description**

This function accepts `fi` objects as inputs.

`ctranspose(a)` returns the complex conjugate transpose of `fi` object `a`. It is also called for the syntax `a'`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

**Introduced before R2006a**

## **DataTypeWorkflow.findDecoupledSubsystem**

### **S**

Get a list of subsystems to replace with an approximation

### **Syntax**

```
systemsToApproximate = DataTypeWorkflow.findDecoupledSubsystems(  
system)
```

### **Description**

`systemsToApproximate = DataTypeWorkflow.findDecoupledSubsystems(system)` returns a table containing all of the subsystems in the system specified by `system` created by the Fixed-Point Tool during the preparation stage of conversion.

When converting a model to fixed point using the Fixed-Point Tool, when you click **Prepare**, the tool finds any blocks that are not supported for conversion. When the tool finds these blocks, it isolates the block by placing it in a subsystem surrounded by Data Type Conversion blocks. After converting the rest of the system to fixed point, use this function to get a list of all the subsystems you must replace. You can use the Lookup Table Optimizer to generate a lookup table approximation of the subsystems containing the unsupported blocks.

### **Examples**

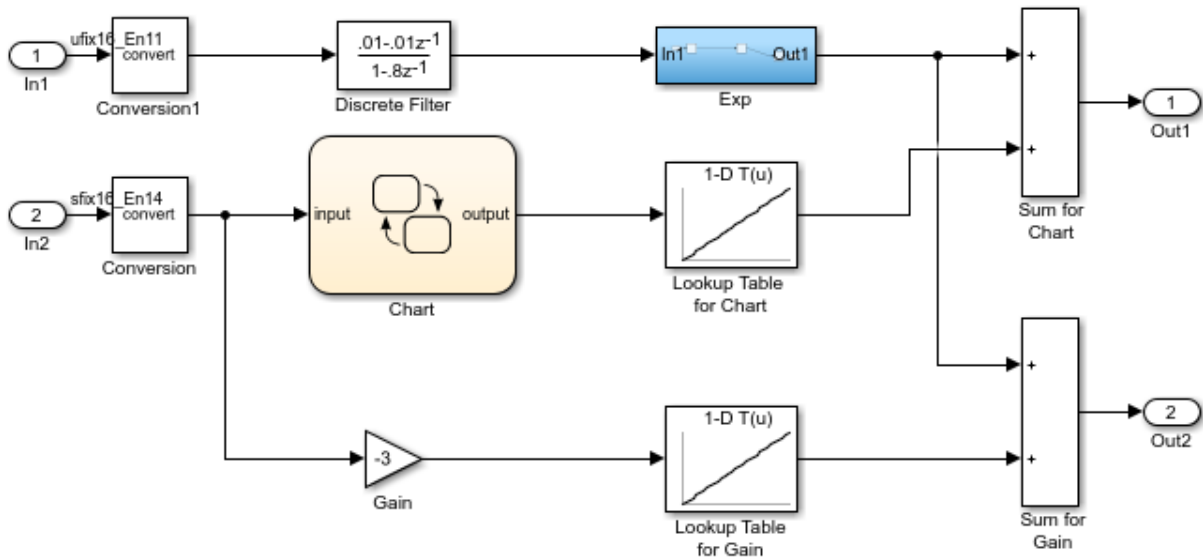
#### **Replace Unsupported Blocks with a Lookup Table Approximation**

In this example, you replace a block that is not supported for fixed-point conversion, with a lookup table approximation.

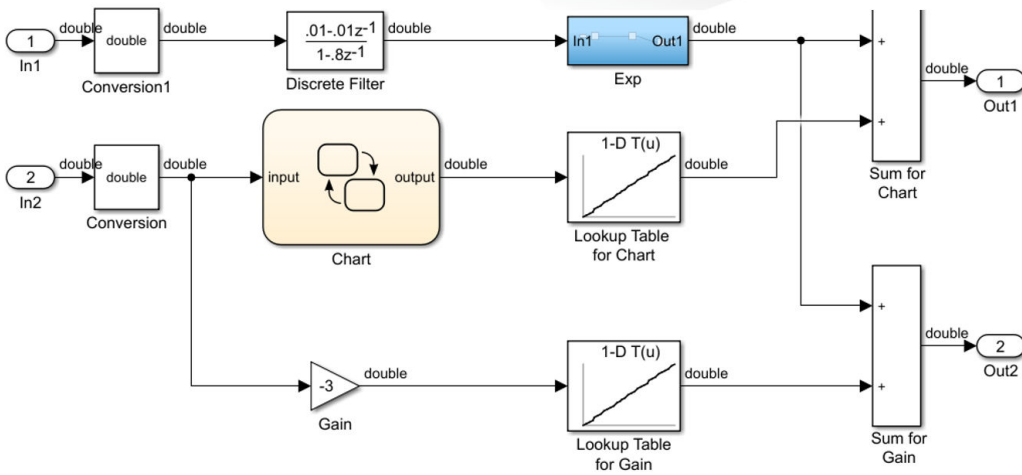
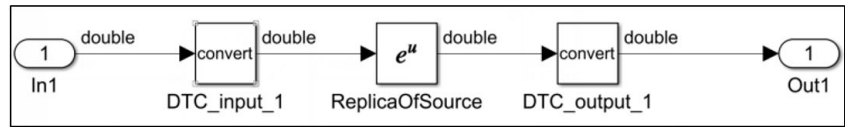
Open the model.

```
open_system('ex_fixed_point_workflow_lutapprox')
```





The Controller Subsystem in the model uses fixed-point data types, except in the Exp subsystem. This subsystem was created by the Fixed-Point Tool during the preparation stage of the conversion. In this example, you use the Lookup Table Optimizer to replace this subsystem with a lookup table approximation.



Identify the subsystems that you need to replace using the `DataTypeWorkflow.findDecoupledSubsystems` function.

```
decoupled = DataTypeWorkflow.findDecoupledSubsystems(gcs)
```

```
decoupled =
```

```
1x2 table
```

ID	BlockPath
1	{'ex_fixed_point_workflow_lutapprox/Controller Subsystem/Exp'}

To replace the functions, open the Lookup Table Optimizer. In the model menu, select **Analysis > Data Type Design > Lookup Table Optimizer**.

On the **Objective** page of the Lookup Table Optimizer, select **Simulink Block**. Click **Next**.

Under **Existing Block Information**, copy and paste the path to the decoupled subsystem created by the Fixed-Point Tool.

Objective > **Setup** > Create > Results

**Block Information**

Simulink Block Path

'ex\_fixed\_point\_workflow/Controller Subsystem/Exp'

Get Current Block

**Attributes of Memory Efficient LUT**

Collect Current Values from Model

Desired Output Data Type numeric(0,16,15)

Input	Desired Data Type	Minimum	Maximum
1	numeric(1,16,18)	-0.125	0.12499618530273438

automatically gather information needed for the optimization process including current output data type, number and data type of inputs, and ranges of input values. You can manually edit all of these fields to specify ranges and data types other than those currently specified on the block.

- Specify the **Desired Output Data Type** of the generated lookup table in the form `numericType(signedness, wordlength, fractionlength)`. For example, to specify a signed output data type of 16-bit word length and 8-bit fraction length, enter `numericType(1,16,8)`. See `numericType` for more information.
- Each input to the block being replaced represents a dimension of the replacement lookup table. Specify the minimum and maximum values of each dimension of the generated lookup table as scalars in the table.
- Specify the data type of each input to the block in the form `numericType(signedness, wordlength, fractionlength)`.

Back Next

Continue through the steps of the Lookup Table Optimizer to generate the lookup table approximation.



## Input Arguments

**system** — System containing the decoupled subsystems

character vector

System containing the decoupled subsystems, specified as a character vector.

## Output Arguments

**systemsToApproximate** — Subsystems to approximate with a lookup table

table

A list of the subsystems decoupled from the model by the Fixed-Point Tool to approximate, returned as a table.

## See Also

[DataTypeWorkflow.Converter](#) | [Lookup Table Optimizer](#)

## Topics

“Convert Floating-Point Model to Fixed Point”

“Use the Fixed-Point Tool to Prepare a System for Conversion”

**Introduced in R2019a**

## dec

Unsigned decimal representation of stored integer of `fi` object

## Syntax

`dec(a)`

## Description

`dec(a)` returns the stored integer of `fi` object `a` in unsigned decimal format as a character vector. `dec(a)` is equivalent to `a.dec`.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

## Examples

The code

```
a = fi([-1 1],1,8,7);
y = dec(a)
z = a.dec
```

returns

y =

128 127

z =

128 127

## **See Also**

bin | hex | oct | sdec | storedInteger

**Introduced before R2006a**

# denormalmax

Largest denormalized quantized number for quantizer object

## Syntax

```
x = denormalmax(q)
```

## Description

`x = denormalmax(q)` is the largest positive denormalized quantized number where `q` is a quantizer object. Anything larger than `x` is a normalized number. Denormalized numbers apply only to floating-point format. When `q` represents fixed-point numbers, this function returns `eps(q)`.

## Examples

```
q = quantizer('float',[6 3]);  
x = denormalmax(q)
```

```
x =
```

```
    0.1875
```

## Algorithms

When `q` is a floating-point quantizer object,

```
denormalmax(q) = realmin(q) - denormalmin(q)
```

When `q` is a fixed-point quantizer object,

```
denormalmax(q) = eps(q)
```

## **See Also**

denormalmin | eps | quantizer

**Introduced before R2006a**



# denormalmin

Smallest denormalized quantized number for quantizer object

## Syntax

```
x = denormalmin(q)
```

## Description

`x = denormalmin(q)` is the smallest positive denormalized quantized number where `q` is a quantizer object. Anything smaller than `x` underflows to zero with respect to the quantizer object `q`. Denormalized numbers apply only to floating-point format. When `q` represents a fixed-point number, `denormalmin` returns `eps(q)`.

## Examples

```
q = quantizer('float',[6 3]);  
x = denormalmin(q)
```

```
x =
```

```
    0.0625
```

## Algorithms

When `q` is a floating-point quantizer object,

$$x = 2^{E_{min} - f}$$

where  $E_{min}$  is equal to `exponentmin(q)`.

When `q` is a fixed-point quantizer object,

$$x = \text{eps}(q) = 2^{-f}$$

where  $f$  is equal to `fractionlength(q)`.

## **See Also**

`denormalmax` | `eps` | `quantizer`

**Introduced before R2006a**



Contrast the above to calculating 1/10 in fixed-point arithmetic with the quotient set to the same numeric type as before.

```
T = numerictype('Signed', false, ...
    'WordLength', 80, ...
    'FractionLength', 83);
a = fi(1);
b = fi(10);
c = divide(T, a, b);
c.bin

ans =
'1100110011001100110011001100110011001100110011001100110011001100110011001100110011001101'
```

Notice that when you use the `divide` function, the quotient is calculated to the full 80 bits, regardless of the precision of `a` and `b`. Thus, the `fi` object `c` represents 1/10 more precisely than a IEEE® standard double-precision floating-point number can.

## Input Arguments

### **T** — Numeric type of the output

`numerictype` object

Numeric type of the output, specified as a `numerictype` object.

### **a** — Numerator

scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array.

`a` and `b` must have the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Complex Number Support: Yes

### **b** — Denominator

scalar | vector | matrix | multidimensional array

Denominator, specified as a real scalar, vector, matrix, or multidimensional array.

**a** and **b** must have the same dimensions unless one is a scalar. If either **a** or **b** is scalar, then **c** has the dimensions of the nonscalar object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`  
 Complex Number Support: Yes

## Output Arguments

### **c** — Quotient

scalar | vector | matrix | multidimensional array

Solution, returned as a scalar, vector, matrix, or multidimensional array. When **a** and **b** are the same size, **c** is the same dimensions as **a** and **b**. If either **a** or **b** is scalar, then **c** has the dimensions of the nonscalar object.

## Algorithms

If **a** and **b** are both `fi` objects, **c** has the same `fimath` object as **a**. If **c** has a `fi` Fixed data type, and any one of the inputs have `fi` floating point data types, then the `fi` floating point is converted into a fixed-point value. Intermediate quantities are calculated using the `fimath` object of **a**.

If either **a** or **b** is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length. Intermediate quantities are calculated using the `fimath` object of the input `fi` object.

If **a** and **b** are both MATLAB built-in doubles, then **c** is the floating-point quotient `a./b`, and `numericType T` is ignored.

## Data Type Propagation Rules

For syntaxes for which Fixed-Point Designer software uses the `numericType T`, the `divide` function follows the data type propagation rules listed in the following table. In most cases, floating-point data types are propagated. This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input <code>fi</code> Objects <code>a</code> and <code>b</code>		Data Type of <code>numericType</code> Object <code>T</code>	Data Type of Output <code>c</code>
Built-in double	Built-in double	Any	Built-in double
<code>fi Fixed</code>	<code>fi Fixed</code>	<code>fi Fixed</code>	Data type of <code>numericType</code> object <code>T</code>
<code>fi Fixed</code>	<code>fi Fixed</code>	<code>fi double</code>	<code>fi double</code>
<code>fi Fixed</code>	<code>fi Fixed</code>	<code>fi single</code>	<code>fi single</code>
<code>fi Fixed</code>	<code>fi Fixed</code>	<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code> with properties of <code>numericType</code> object <code>T</code>
<code>fi double</code>	<code>fi double</code>	<code>fi Fixed</code>	<code>fi double</code>
<code>fi double</code>	<code>fi double</code>	<code>fi double</code>	<code>fi double</code>
<code>fi double</code>	<code>fi double</code>	<code>fi single</code>	<code>fi single</code>
<code>fi double</code>	<code>fi double</code>	<code>fi ScaledDouble</code>	<code>fi double</code>
<code>fi single</code>	<code>fi single</code>	<code>fi Fixed</code>	<code>fi single</code>
<code>fi single</code>	<code>fi single</code>	<code>fi double</code>	<code>fi double</code>
<code>fi single</code>	<code>fi single</code>	<code>fi single</code>	<code>fi single</code>
<code>fi single</code>	<code>fi single</code>	<code>fi ScaledDouble</code>	<code>fi single</code>
<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code>	<code>fi Fixed</code>	If either input <code>a</code> or <code>b</code> is of type <code>fi ScaledDouble</code> , then output <code>c</code> is of type <code>fi ScaledDouble</code> with properties of <code>numericType</code> object <code>T</code> .
<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code>	<code>fi double</code>	<code>fi double</code>
<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code>	<code>fi single</code>	<code>fi single</code>

Data Type of Input <code>fi</code> Objects <code>a</code> and <code>b</code>		Data Type of <code>numericType</code> Object <code>T</code>	Data Type of Output <code>c</code>
<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code>	<code>fi ScaledDouble</code>	If either input <code>a</code> or <code>b</code> is of type <code>fi ScaledDouble</code> , then output <code>c</code> is of type <code>fi ScaledDouble</code> with properties of <code>numericType</code> object <code>T</code> .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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 does not support the syntax `T.divide(a,b)`.

### See Also

`add` | `fi` | `fimath` | `mpy` | `mrdivide` | `numericType` | `rdivide` | `sub` | `sum`

Introduced before R2006a

## double

Double-precision floating-point real-world value of `fi` object

### Syntax

`double(a)`

### Description

`double(a)` returns the real-world value of a `fi` object in double-precision floating point. `double(a)` is equivalent to `a.double`.

Fixed-point numbers can be represented as

$$\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}$$

or, equivalently as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

### Examples

The code

```
a = fi([-1 1],1,8,7);  
y = double(a)  
z = a.double
```

returns

```
y =  
  
z =      -1      0.9922  
  
      -1      0.9922
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.

### See Also

`single`

**Introduced before R2006a**

## **embedded.fi class**

Fixed-point numeric object

### **Description**

Use the `fi` function to create an `embedded.fi` object.

### **See Also**

`embedded.fimath` | `embedded.numericitype` | `fi`

### **Topics**

Class Attributes (MATLAB)

Property Attributes (MATLAB)

# embedded.fimath class

fimath object

## Description

Use the `fimath` function to create an `embedded.fimath` object.

## See Also

`embedded.fi` | `embedded.numericType` | `fimath`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

## **embedded.numericity class**

numericity object

### **Description**

Use the `numericity` function to create an `embedded.numericity` object.

### **See Also**

`embedded.fi` | `embedded.fimath` | `numericity`

### **Topics**

Class Attributes (MATLAB)

Property Attributes (MATLAB)

## eps

Quantized relative accuracy for `fi` or `quantizer` objects

### Syntax

`eps(obj)`

### Description

`eps(obj)` returns the value of the least significant bit of the value of the `fi` object or `quantizer` object `obj`. The result of this function is equivalent to that given by the Fixed-Point Designer function `lsb`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports scalar fixed-point signals only.
- Code generation supports scalar, vector, and matrix, `fi` single and `fi` double signals.

### See Also

`intmax` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `upperbound`

**Introduced before R2006a**

## eq

Determine whether real-world values of two `fi` objects are equal

## Syntax

```
c = eq(a,b)
a == b
```

## Description

`c = eq(a,b)` is called for the syntax `a == b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a == b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

## Examples

### Compare Two `fi` Objects

Use the `isequal` function to determine if two `fi` objects have the same real-world value.

```
a = fi(pi);
b = fi(pi, 1, 32);
a == b
```

```
ans = logical
      0
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `eq` function returns `0` because the two `fi` objects do not have the same real-world value.

### Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the double is cast to the same word length and signedness of the `fi` object.

```
a = fi(pi);  
b = pi;  
a == b  
  
ans = logical  
     1
```

The `eq` function casts `b` to the same word length as `a`, and returns `1`. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

### See Also

`ge` | `gt` | `isequal` | `le` | `lt` | `ne`

Introduced before R2006a

## errmean

Mean of quantization error

### Syntax

```
m = errmean(q)
```

### Description

`m = errmean(q)` returns the mean of a uniformly distributed random quantization error that arises from quantizing a signal by quantizer object `q`.

---

**Note** The results are not exact when the signal precision is close to the precision of the quantizer.

---

### Examples

Find `m`, the mean of the quantization error for quantizer `q`:

```
q = quantizer;  
m = errmean(q)  
  
m =  
  
-1.525878906250000e-005
```

Now compare `m` to `m_est`, the sample mean from a Monte Carlo experiment:

```
r = realmax(q);  
u = 2*r*rand(1000,1)-r; % Original signal  
y = quantize(q,u); % Quantized signal  
e = y - u; % Error  
m_est = mean(e) % Estimate of the error mean
```



```
m_est =  
  -1.519507450175317e-005
```

## See Also

[errpdf](#) | [errvar](#) | [quantize](#)

**Introduced in R2008a**

## errpdf

Probability density function of quantization error

### Syntax

```
[f,x] = errpdf(q)  
f = errpdf(q,x)
```

### Description

`[f,x] = errpdf(q)` returns the probability density function `f` evaluated at the values in `x`. The vector `x` contains the uniformly distributed random quantization errors that arise from quantizing a signal by `quantizer` object `q`.

`f = errpdf(q,x)` returns the probability density function `f` evaluated at the values in vector `x`.

---

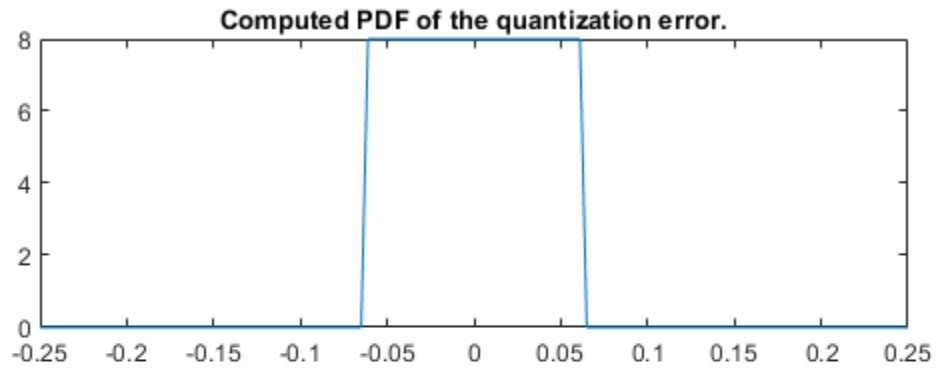
**Note** The results are not exact when the signal precision is close to the precision of the quantizer.

---

### Examples

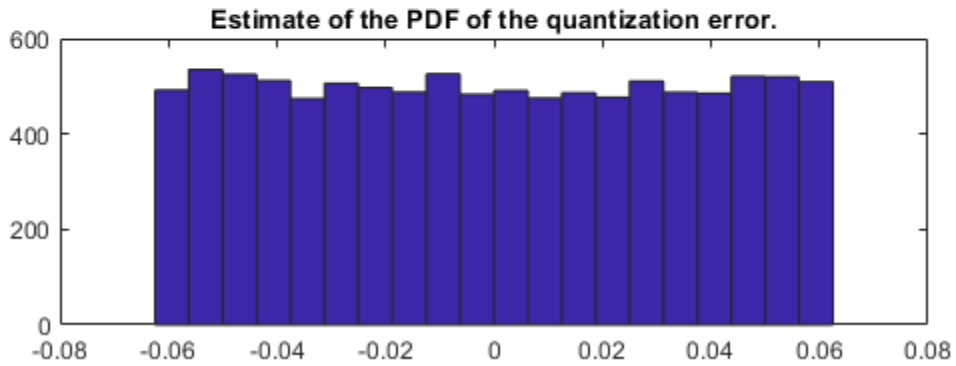
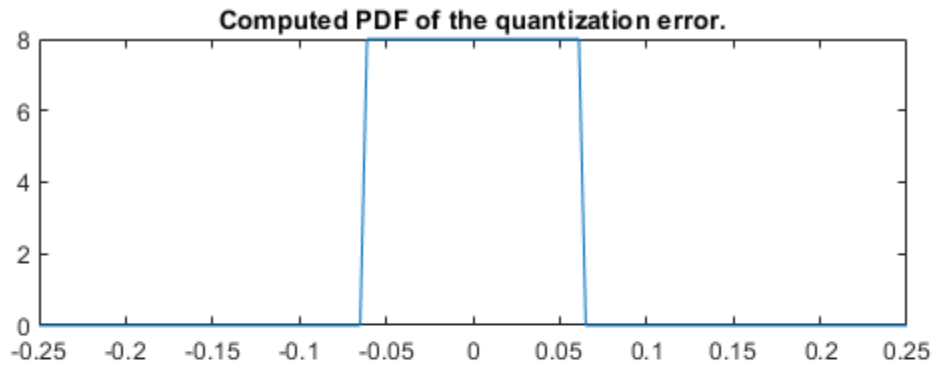
#### Compute the PDF of the quantization error

```
q = quantizer('nearest',[4 3]);  
[f,x] = errpdf(q);  
subplot(211)  
plot(x,f)  
title('Computed PDF of the quantization error.')
```



The output plot shows the probability density function of the quantization error. Compare this result to a plot of the sample probability density function from a Monte Carlo experiment:

```
r = realmax(q);  
u = 2*r*rand(10000,1)-r; % Original signal  
y = quantize(q,u);      % Quantized signal  
e = y - u;              % Error  
subplot(212)  
hist(e,20)  
gca.xlim = [min(x) max(x)];  
title('Estimate of the PDF of the quantization error.')
```



## See Also

`errmean` | `errvar` | `quantize`

Introduced in R2008a

## errvar

Variance of quantization error

### Syntax

```
v = errvar(q)
```

### Description

`v = errvar(q)` returns the variance of a uniformly distributed random quantization error that arises from quantizing a signal by quantizer object `q`.

---

**Note** The results are not exact when the signal precision is close to the precision of the quantizer.

---

### Examples

Find `v`, the variance of the quantization error for quantizer object `q`:

```
q = quantizer;
v = errvar(q)

v =

    7.761021455128987e-011
```

Now compare `v` to `v_est`, the sample variance from a Monte Carlo experiment:

```
r = realmax(q);
u = 2*r*rand(1000,1)-r; % Original signal
y = quantize(q,u);     % Quantized signal
e = y - u;             % Error
v_est = var(e)        % Estimate of the error variance

v_est =
```

7.520208858166330e-011

## **See Also**

errmean | errpdf | quantize

**Introduced in R2008a**

# exponentbias

Exponent bias for quantizer object

## Syntax

```
b = exponentbias(q)
```

## Description

`b = exponentbias(q)` returns the exponent bias of the quantizer object `q`. For fixed-point quantizer objects, `exponentbias(q)` returns 0.

## Examples

```
q = quantizer('double');  
b = exponentbias(q)
```

```
b =
```

```
1023
```

## Algorithms

For floating-point quantizer objects,

$$b = 2^{e-1} - 1$$

where  $e = \text{eps}(q)$ , and `exponentbias` is the same as the exponent maximum.

For fixed-point quantizer objects,  $b = 0$  by definition.

## **See Also**

eps | exponentlength | exponentmax | exponentmin

**Introduced before R2006a**



# exponentlength

Exponent length of quantizer object

## Syntax

```
e = exponentlength(q)
```

## Description

`e = exponentlength(q)` returns the exponent length of quantizer object `q`. When `q` is a fixed-point quantizer object, `exponentlength(q)` returns 0. This is useful because exponent length is valid whether the quantizer object mode is floating point or fixed point.

## Examples

```
q = quantizer('double');  
e = exponentlength(q)
```

```
e =
```

```
    11
```

## Algorithms

The exponent length is part of the format of a floating-point quantizer object `[w e]`. For fixed-point quantizer objects,  $e = 0$  by definition.

## See Also

`eps` | `exponentbias` | `exponentmax` | `exponentmin`

**Introduced before R2006a**

# exponentmax

Maximum exponent for quantizer object

## Syntax

```
exponentmax(q)
```

## Description

`exponentmax(q)` returns the maximum exponent for quantizer object `q`. When `q` is a fixed-point quantizer object, it returns 0.

## Examples

```
q = quantizer('double');  
emax = exponentmax(q)
```

```
emax =
```

```
1023
```

## Algorithms

For floating-point quantizer objects,

$$E_{max} = 2^{e-1} - 1$$

For fixed-point quantizer objects,  $E_{max} = 0$  by definition.

## See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmin`

**Introduced before R2006a**

# exponentmin

Minimum exponent for quantizer object

## Syntax

```
emin = exponentmin(q)
```

## Description

`emin = exponentmin(q)` returns the minimum exponent for quantizer object `q`. If `q` is a fixed-point quantizer object, `exponentmin` returns 0.

## Examples

```
q = quantizer('double');  
emin = exponentmin(q)
```

```
emin =
```

```
    -1022
```

## Algorithms

For floating-point quantizer objects,

$$E_{min} = -2^{e-1} + 2$$

For fixed-point quantizer objects,  $E_{min} = 0$ .

## See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmax`

**Introduced before R2006a**

# eye

Create identity matrix with fixed-point properties

## Syntax

```
I = eye('like',p)
I = eye(n,'like',p)
I = eye(n,m,'like',p)
I = eye(sz,'like',p)
```

## Description

`I = eye('like',p)` returns the scalar 1 with the same fixed-point properties and complexity (real or complex) as the prototype argument, `p`. The output, `I`, contains the same numeric type and `fimath` properties as `p`.

`I = eye(n,'like',p)` returns an `n`-by-`n` identity matrix like `p`, with ones on the main diagonal and zeros elsewhere.

`I = eye(n,m,'like',p)` returns an `n`-by-`m` identity matrix like `p`.

`I = eye(sz,'like',p)` returns an array like `p`, where the size vector, `sz`, defines `size(I)`.

## Examples

### Create Identity Matrix with Fixed-Point Properties

Create a prototype `fi` object, `p`.

```
p = fi([],1,16,14);
```

Create a 3-by-4 identity matrix with the same fixed-point properties as `p`.

```
I = eye(3,4,'like',p)
```

```
I=3×4 object
```

```
 1     0     0     0
 0     1     0     0
 0     0     1     0
```

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

### Create Identity Matrix with Attached `fimath`

Create a signed `fi` object with word length of 16, fraction length of 15 and `OverflowAction` set to `Wrap`.

```
format long
p = fi([],1,16,15,'OverflowAction','Wrap');
```

Create a 2-by-2 identity matrix with the same `numericType` properties as `p`.

```
X = eye(2,'like',p)
```

```
X=2×2 object
```

```
 0.999969482421875      0
      0  0.999969482421875
```

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

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

1 cannot be represented by the data type of `p`, so the value saturates. The output `fi` object `X` has the same `numericType` and `fimath` properties as `p`.



## Input Arguments

### **n** — Size of first dimension of **I**

integer value

Size of first dimension of **I**, specified as an integer value.

- If **n** is the only integer input argument, then **I** is a square **n**-by-**n** identity matrix.
- If **n** is 0, then **I** is an empty matrix.
- If **n** is negative, then it is treated as 0.

**Data Types:** single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **m** — Size of second dimension of **I**

integer value

Size of second dimension of **I**, specified as an integer value.

- If **m** is 0, then **I** is an empty matrix.
- If **m** is negative, then it is treated as 0.

**Data Types:** single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **sz** — Size of **I**

row vector of no more than two integer values

Size of **I**, specified as a row vector of no more than two integer values.

- If an element of **sz** is 0, then **I** is an empty matrix.
- If an element of **sz** is negative, then the element is treated as 0.

**Data Types:** single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **p** — Prototype

fi object | numeric variable

Prototype, specified as a **fi** object or numeric variable.

If the value 1 overflows the numeric type of `p`, the output saturates regardless of the specified `OverflowAction` property of the attached `fimath`. All subsequent operations performed on the output obey the rules of the attached `fimath`.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Tips

Using the `b = cast(a, 'like', p)` syntax to specify data types separately 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.

## See Also

`ones` | `zeros`

## Topics

“Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using `cast` and `zeros`”

“Manual Fixed-Point Conversion Best Practices”

**Introduced in R2015a**

## fi

Construct fixed-point numeric object

### Description

To assign a fixed-point data type to a number or variable, create a `fi` object using the `fi` constructor. You can specify numeric attributes and math rules in the constructor or using the `numericType` and `fiMath` objects.

### Creation

### Syntax

```
a = fi
a = fi(v)
a = fi(v, s)
a = fi(v, s, w)
a = fi(v, s, w, f)
a = fi(v, s, w, slope, bias)
a = fi(v, s, w, slopeadjustmentfactor, fixedexponent, bias)
a = fi(v, T)
a = fi( ____, F)
a = fi( ____, Name, Value)
```

### Description

`a = fi` returns a `fi` object with no value, 16-bit word length, and 15-bit fraction length.

`a = fi(v)` returns a fixed-point object with value `v` and default property values.

`a = fi(v, s)` returns a fixed-point object with signedness (signed or unsigned) `s`.

`a = fi(v, s, w)` creates a fixed-point object with word length specified by `w`.

`a = fi(v, s, w, f)` creates a fixed-point object with fraction length specified by `f`.

`a = fi(v, s, w, slope, bias)` creates a fixed-point object using slope and bias scaling.

`a = fi(v, s, w, slopeadjustmentfactor, fixedexponent, bias)` creates a fixed-point object using slope and bias scaling.

`a = fi(v, T)` creates a fixed-point object with value `v`, and numeric type properties, `T`.

`a = fi( ____, F)` creates a fixed-point object with math settings specified by `fimath` object `F`.

`a = fi( ____, Name, Value)` creates a fixed-point object with property values specified by one or more `Name, Value` pair arguments. `Name` must appear inside single quotes ( `' '` ). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **v — Value**

scalar | vector | matrix | multi-dimensional array

Value of the `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

The value of the output `fi` object is the value of the input quantized to the data type specified in the `fi` constructor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

### **s — Signedness**

1 (default) | 0

Signedness of the `fi` object, specified as a boolean. A value of `1`, or `true`, indicates a signed data type. A value of `0`, or `false`, indicates an unsigned data type.

Data Types: `logical`

### **w — Word length**

16 (default) | scalar integer

Word length, in bits, of the `fi` object, specified as a scalar integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **f** — Fraction length

15 (default) | scalar integer

Fraction length, in bits, of the `fi` object, specified as a scalar integer. If you do not specify a fraction length, the `fi` object automatically uses the fraction length that gives the best precision while avoiding overflow for the specified value, word length, and signedness.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **slope** — Slope

scalar integer

Slope of the scaling, specified as a scalar integer. The following equation represents the real-world value of a slope bias scaled number.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **bias** — Bias

scalar

Bias of the scaling, specified as a scalar. The following equation represents the real-world value of a slope bias scaled number.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **slopeadjustmentfactor** — Slope adjustment factor

scalar integer

The slope adjustment factor of a slope bias scaled number. The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

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

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **fixedexponent** — Fixed exponent

scalar integer

The fixed exponent of a slope bias scaled number. The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

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

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **T** — Numeric type properties

numeric type object

Numeric type properties of the `fi` object, specified as a numeric type object. For more information, see `numeric type`.

### **F** — Fixed-point math properties

`fimath` object

Fixed-point math properties of the `fi` object, specified as a `fimath` object. For more information, see `fimath`.

## **Properties**

“fi Object Properties”

## **Examples**

### Create a **fi** object

Create a signed **fi** object with a value of `pi`, a word length of eight bits, and a fraction length of 3 bits.

```
a = fi(pi, 1, 8, 3)
```

```
a =  
    3.1250
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 8  
    FractionLength: 3
```

### Create an Array of **fi** Objects

Create an array of **fi** objects with 16-bit word length and 12-bit fraction length.

```
a = fi((magic(3)/10), 1, 16, 12)
```

```
a=3×3 object  
    0.8000    0.1001    0.6001  
    0.3000    0.5000    0.7000  
    0.3999    0.8999    0.2000
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 12
```

### Create a **fi** object with Default Word Length and Fraction Length

When you specify only the value and the signedness of the **fi** object, the word length defaults to 16 bits, and the fraction length is set to achieve the best precision possible without overflow.

```
a = fi(pi, 1)
```

```
a =  
  3.1416  
  
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 13
```

### Create a `fi` Object with Default Precision

If you do not specify a fraction length, input argument `f`, the fraction length of the `fi` object defaults to the fraction length that offers the best precision.

```
a = fi(pi, 1, 8)  
  
a =  
  3.1563  
  
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 8  
      FractionLength: 5
```

The fraction length of `fi` object `a` is five because three bits are required to represent the integer portion of the value when the data type is signed. If the `fi` object uses an unsigned data type, only two bits are needed to represent the integer portion, leaving six fractional bits.

```
a = fi(pi, 0, 8)  
  
a =  
  3.1406  
  
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Unsigned  
      WordLength: 8  
      FractionLength: 6
```



## Create a fi Object with Slope and Bias Scaling

The real-world value of a slope bias scaled number is represented by:

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

To create a `fi` object that uses slope and bias scaling, include the `slope` and `bias` arguments after the word length in the constructor.

```
a = fi(pi, 1, 16, 3, 2)
```

```
a =  
    2
```

```
    DataTypeMode: Fixed-point: slope and bias scaling  
    Signedness: Signed  
    WordLength: 16  
    Slope: 3  
    Bias: 2
```

The `DataTypeMode` property of the `fi` object, `a`, is `slope and bias scaling`.

## Create a fi Object From a Non-Double Value

When the value input argument, `v`, of a `fi` object is a non-double, and you do not specify the word length or fraction length properties, the resulting `fi` object retains the numeric type of the input, `v`.

### Create a fi object from a built-in integer

When the input is a built-in integer, the fixed-point attributes match the attributes of the integer type.

```
v1 = uint32(5);  
a1 = fi(v1)
```

```
a1 =  
    5
```

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

```
        WordLength: 32
        FractionLength: 0

v2 = int8(5);
a2 = fi(v2)

a2 =
    5

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

### Create a **fi** object from a **fi** object

When the input value is a **fi** object, the output uses the same word length, fraction length, and signedness of the input **fi** object.

```
v = fi(pi, 1, 24, 12);
a = fi(v)

a =
    3.1416

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 24
        FractionLength: 12
```

### Create a **fi** object from a logical

When the input *v* is logical, the `DataTypeMode` property of the output **fi** object is `Boolean`.

```
v = true;
a = fi(v)

a =
    1

        DataTypeMode: Boolean
```

### Create a **fi** object from a single

When the input is single, the `DataTypeMode` property of the output is `Single`.

```

v = single(pi);
a = fi(pi)

a =
    3.1416

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

```

### Create a fi Object With an Associated fimath Object

The arithmetic attributes of a `fi` object are defined by a `fimath` object which is attached to that `fi` object.

Create a `fimath` object and specify the `OverflowAction`, `RoundingMethod`, and `ProductMode` properties.

```

F = fimath('OverflowAction', 'Wrap', 'RoundingMethod', 'Floor', 'ProductMode', 'KeepMSB')

F =
    RoundingMethod: Floor
    OverflowAction: Wrap
    ProductMode: KeepMSB
    ProductWordLength: 32
    SumMode: FullPrecision

```

Create a `fi` object and specify the `fimath` object, `F`, in the constructor.

```

a = fi(pi, F)

a =
    3.1415

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

    RoundingMethod: Floor
    OverflowAction: Wrap
    ProductMode: KeepMSB

```

```
ProductWordLength: 32
SumMode: FullPrecision
```

Use the `removefimath` function to remove the associated `fimath` object and restore the math settings to their default values.

```
a = removefimath(a)
```

```
a =
    3.1415
```

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

### Create a `fi` Object From a `numericType` Object

A `numericType` object contains all of the data type information of a `fi` object. `numericType` properties are, by transitivity, also properties of `fi` objects.

You can create a `fi` object that uses all of the properties of an existing `numericType` object by specifying the `numericType` object in the `fi` constructor.

```
T = numericType(0,24,16)
```

```
T =
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 24
FractionLength: 16
```

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

```
a =
    3.1416
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 24
FractionLength: 16
```

## Create a `fi` Object With Fraction Length Greater Than Word Length

When you use binary-point representation for a fixed-point number, the fraction length can be greater than the word length. In this case, there are implicit leading zeros (for positive numbers) or ones (for negative numbers) between the binary point and the first significant binary digit.

Consider a signed value with a word length of 8, fraction length of 10, and a stored integer value of 5. Calculate the real-world value using the following equation.

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

```
realWorldValue = 5*2^(-10)
```

```
realWorldValue = 0.0049
```

Create a signed `fi` object with value `realWorldValue`, a word length of 8 bits, and a fraction length of 10 bits.

```
a = fi(realWorldValue, 1, 8, 10)
```

```
a =  
    0.0049
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 8  
        FractionLength: 10
```

Get the stored integer value of `a` using the `int` function.

```
int(a)  
  
ans = int8  
     5
```

Use the `bin` function to view the stored integer value in binary.

```
bin(a)  
  
ans =  
'00000101'
```

Because the fraction length is two bits longer than the word length, the binary value of the stored integer is  $X.XX00000101$ , where  $X$  is a placeholder for implicit zeroes.  $0.0000000101$  (binary) is equivalent to  $0.0049$  (decimal).

### Create a `fi` Object With Negative Fraction Length

When you use binary-point representation for a fixed-point number, the fraction length can be negative. In this case, there are implicit trailing zeros (for positive numbers) or ones (for negative numbers) between the binary point and the first significant binary digit.

Consider a signed data type with a word length of 8, fraction length of -2 and a stored integer value of 5. Calculate the stored integer value using the following equation.

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

```
realWorldValue = 5*2^(2)
```

```
realWorldValue = 20
```

Create a signed `fi` object with value `realWorldValue`, a word length of 8 bits, and a fraction length of -2 bits.

```
a = fi(realWorldValue, 1, 8, -2)
```

```
a =  
    20
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 8  
    FractionLength: -2
```

Get the stored integer value of `a` using the `int` function.

```
int(a)
```

```
ans = int8  
     5
```

Get the binary value of `a` using the `bin` function.

```
bin(a)
ans =
'00000101'
```

Because the fraction length is negative, the binary value of the stored integer is 00000101XX, where X is a placeholder for implicit zeros. 0000010100 (binary) is equivalent to 20 (decimal).

### Create a fi Object Specifying Rounding and Overflow Modes

You can set math properties, such as rounding and overflow modes during the creation of 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
```

The `RoundingMethod` and `OverflowAction` properties are properties of the `fimath` object. Specifying these properties in the `fi` constructor associates a local `fimath` object with the `fi` object.

Use the `removefimath` function to remove the local `fimath` and set the math properties back to their default values.

```
a = removefimath(a)
a =
    3.1415
        DataTypeMode: Fixed-point: binary point scaling
```

```
Signedness: Signed
WordLength: 16
FractionLength: 13
```

### **Use `fi` as an Indexing Argument**

When using a `fi` object as an index, the value of the `fi` object must be an integer.

Set up an array to index into.

```
x = 10:-1:1;
```

Create an integer valued `fi` object and use it to index into `x`.

```
a = fi(3);
```

```
y = x(a)
```

```
y = 8
```

### **Use `fi` as the index in a for loop**

Create `fi` objects to use as the index of a for loop. The values of the indices must be integers.

```
a = fi(1, 0, 8, 0);
```

```
b = fi(2, 0, 8, 0);
```

```
c = fi(10, 0, 8, 0);
```

```
for x = a:b:c
    x
```

```
end
```

```
x =
    1
```

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

```
x =
    3
```



```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 8
        FractionLength: 0
x =
    5
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 8
        FractionLength: 0
x =
    7
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 8
        FractionLength: 0
x =
    9
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 8
        FractionLength: 0

```

### Set Data Type Override on a fi Object

The `fipref` object defines the display and logging attributes for all `fi` objects. Use the `DataTypeOverride` setting of the `fipref` object to override `fi` objects with doubles, singles, or scaled doubles.

Save the current `fipref` settings to restore later.

```
fp = fipref;
initialDT0 = fp.DataTypeOverride;
```

Create a `fi` object with the default settings and original `fipref` settings.

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

```
a =  
    3.1416  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 13
```

Turn on data type override to doubles and create a new `fi` object without specifying its `DataTypeOverride` property so that it uses the data type override settings specified using `fipref`.

```
fipref('DataTypeOverride', 'TrueDoubles')  
  
ans =  
    NumberDisplay: 'RealWorldValue'  
    NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'TrueDoubles'  
    DataTypeOverrideAppliesTo: 'AllNumericTypes'
```

```
a = fi(pi)  
  
a =  
    3.1416  
  
    DataTypeMode: Double
```

Now create a `fi` object and set its `DataTypeOverride` setting to `off` so that it ignores the data type override settings of the `fipref` object.

```
b = fi(pi, 'DataTypeOverride', 'Off')  
  
b =  
    3.1416  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 13
```

Restore the `fipref` settings saved at the start of the example.

```
fp.DataTypeOverride = initialDFO;
```

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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.
- All properties related to data type must be constant for code generation.
- `numericType` object information must be available for nonfixed-point Simulink inputs.

### See Also

`fimath` | `fipref` | `isfimathlocal` | `numericType` | `quantizer` | `sfi` | `ufi`

### Topics

“Create Fixed-Point Data”

“Perform Fixed-Point Arithmetic”

“Perform Binary-Point Scaling”

“fi Object Functions”

“Binary Point Interpretation”

**Introduced in R2006a**

## fiaccel

Accelerate fixed-point code and convert floating-point MATLAB code to fixed-point MATLAB code

### Syntax

```
fiaccel -options fcn  
fiaccel -float2fixed fcn
```

### Description

`fiaccel -options fcn` translates the MATLAB file `fcn.m` to a MEX function, which accelerates fixed-point code. To use `fiaccel`, your code must meet one of these requirements:

- The top-level function has no inputs or outputs, and the code uses `fi`
- The top-level function has an output or a non-constant input, and at least one output or input is a `fi`.
- The top-level function has at least one input or output containing a built-in integer class (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, or `uint64`), and the code uses `fi`.

---

**Note** If your top-level file is on a path that contains Unicode characters, code generation might not be able to find the file.

---

`fiaccel -float2fixed fcn` converts the floating-point MATLAB function, `fcn` to fixed-point MATLAB code.

## Input Arguments

### **fcn**

MATLAB function from which to generate a MEX function. *fcn* must be suitable for code generation. For information on code generation, see “Code Acceleration and Code Generation from MATLAB”

### **options**

Choice of compiler options. `fiaccl` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

`-args example_inputs`

Define the size, class, and complexity of MATLAB function inputs by providing a cell array of example input values. The position of the example input in the cell array must correspond to the position of the input argument in the MATLAB function definition. To generate a function that has fewer input arguments than the function definition has, omit the example values for the arguments that you do not want.

Specify the example inputs immediately after the function to which they apply.

Instead of an example value, you can provide a `coder.Type` object. To create a `coder.Type` object, use `coder.typeof`.

`-config config_object`

Specify MEX generation parameters, based on *config\_object*, defined as a MATLAB variable using `coder.mexconfig`. For example:

```
cfg = coder.mexconfig;
```

`-d out_folder`

Store generated files in the absolute or relative path specified by `out_folder`. If the folder specified by `out_folder` does not exist, `fiaccl` creates it for you.

If you do not specify the folder location, `fiaccl` generates files in the default folder:

`fiaccl/mex/fcn`.

`fcn` is the name of the MATLAB function specified at the command line.

The function does not support the following characters in folder names: asterisk (\*), question-mark (?), dollar (\$), and pound (#).

`-float2fixed float2fixed_cfg_name`

Generates fixed-point MATLAB code using the settings specified by the floating-point to fixed-point conversion configuration object named `float2fixed_cfg_name`.

For this option, `fiaccl` generates files in the folder `codegen/fcn_name/fixpt`.

You must set the `TestBenchName` property of `float2fixed_cfg_name`. For example:

```
fixptcfg.TestBenchName = 'myadd_test';
```

specifies that `myadd_test` is the test file for the floating-point to fixed-point configuration object `fixptcfg`.

You cannot use this option with the `-global` option.

`-g`

Compiles the MEX function in debug mode, with optimization turned off. If not specified, `fiaccl` generates the MEX function in optimized mode.

`-global global_values`

Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `fiaccel`. If you do not provide initial values for global variables using the `-global` option, `fiaccel` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `fiaccel` generates an error.

The generated MEX 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.

You cannot use this option with the `-float2fixed` option.

`-I include_path`

Add *include\_path* to the beginning of the code generation path.

`fiaccel` searches the code generation path *first* when converting MATLAB code to MEX code.

`-launchreport`

Generate and open a code generation report. If you do not specify this option, `fiaccel` generates a report only if error or warning messages occur or you specify the `-report` option.

<code>-nargout</code>	Specify the number of output arguments in the generated entry-point function. The code generator produces the specified number of output arguments in the order in which they occur in the MATLAB function definition.
<code>-o <i>output_file_name</i></code>	Generate the MEX function with the base name <i>output_file_name</i> plus a platform-specific extension.  <i>output_file_name</i> can be a file name or include an existing path.  If you do not specify an output file name, the base name is <i>fcn_mex</i> , which allows you to run the original MATLAB function and the MEX function and compare the results.
<code>-O <i>optimization_option</i></code>	Optimize generated MEX code, based on the value of <i>optimization_option</i> : <ul style="list-style-type: none"><li>• <code>enable:inline</code> — Enable function inlining</li><li>• <code>disable:inline</code> — Disable function inlining</li></ul> If not specified, <code>fiaccl</code> uses inlining for optimization.
<code>-report</code>	Generate a code generation report. If you do not specify this option, <code>fiaccl</code> generates a report only if error or warning messages occur or you specify the <code>-launchreport</code> option.
<code>-?</code>	Display help for <code>fiaccl</code> command.



## Examples

Create a test file and compute the moving average. Then, use `fiaccel` to accelerate the code and compare.

```
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

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

% Use fiaccel to create a MEX function and
% accelerate the code
x = fi(rand(100,1),1,16,15);
fiaccel test_moving_average -args {x} -report

% Compare the non-accelerated and accelerated code.
x = fi(rand(100,1),1,16,15);

% Non-compiled version
tic,avg = test_moving_average(x);toc
% Compiled version
tic,avg = test_moving_average_mex(x);toc
```

## Convert Floating-Point MATLAB Code to Fixed Point

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

### See Also

[coder.ArrayType](#) | [coder.Constant](#) | [coder.EnumType](#) | [coder.FiType](#) | [coder.FixptConfig](#) | [coder.MexConfig](#) | [coder.PrimitiveType](#) | [coder.StructType](#) | [coder.Type](#) | [coder.config](#) | [coder.mexconfig](#) | [coder.newtype](#) | [coder.resize](#) | [coder.typeof](#)

**Introduced in R2011a**

## filter

One-dimensional digital filter of `fi` objects

### Syntax

```
y = filter(b,1,x)
[y,zf] = filter(b,1,x,zi)
y = filter(b,1,x,zi,dim)
```

### Description

`y = filter(b,1,x)` filters the data in the fixed-point vector `x` using the filter described by the fixed-point vector `b`. The function returns the filtered data in the output `fi` object `y`. Inputs `b` and `x` must be `fi` objects. `filter` always operates along the first non-singleton dimension. Thus, the filter operates along the first dimension for column vectors and nontrivial matrices, and along the second dimension for row vectors.

`[y,zf] = filter(b,1,x,zi)` gives access to initial and final conditions of the delays, `zi`, and `zf`. `zi` is a vector of length `length(b) - 1`, or an array with the leading dimension of size `length(b) - 1` and with remaining dimensions matching those of `x`. `zi` must be a `fi` object with the same data type as `y` and `zf`. If you do not specify a value for `zi`, it defaults to a fixed-point array with a value of 0 and the appropriate `numericType` and size.

`y = filter(b,1,x,zi,dim)` performs the filtering operation along the specified dimension. If you do not want to specify the vector of initial conditions, use `[]` for the input argument `zi`.

### Input Arguments

#### **b**

Fixed-point vector of the filter coefficients.

**x**

Fixed-point vector containing the data for the function to filter.

**zi**

Fixed-point vector containing the initial conditions of the delays. If the initial conditions of the delays are zero, you can specify zero, or, if you do not know the appropriate size and numeric type for *zi*, use `[]`.

If you do not specify a value for *zi*, the parameter defaults to a fixed-point vector with a value of zero and the same numeric type and size as the output *zf* (default).

**dim**

Dimension along which to perform the filtering operation.

## Output Arguments

**y**

Output vector containing the filtered fixed-point data.

**zf**

Fixed-point output vector containing the final conditions of the delays.

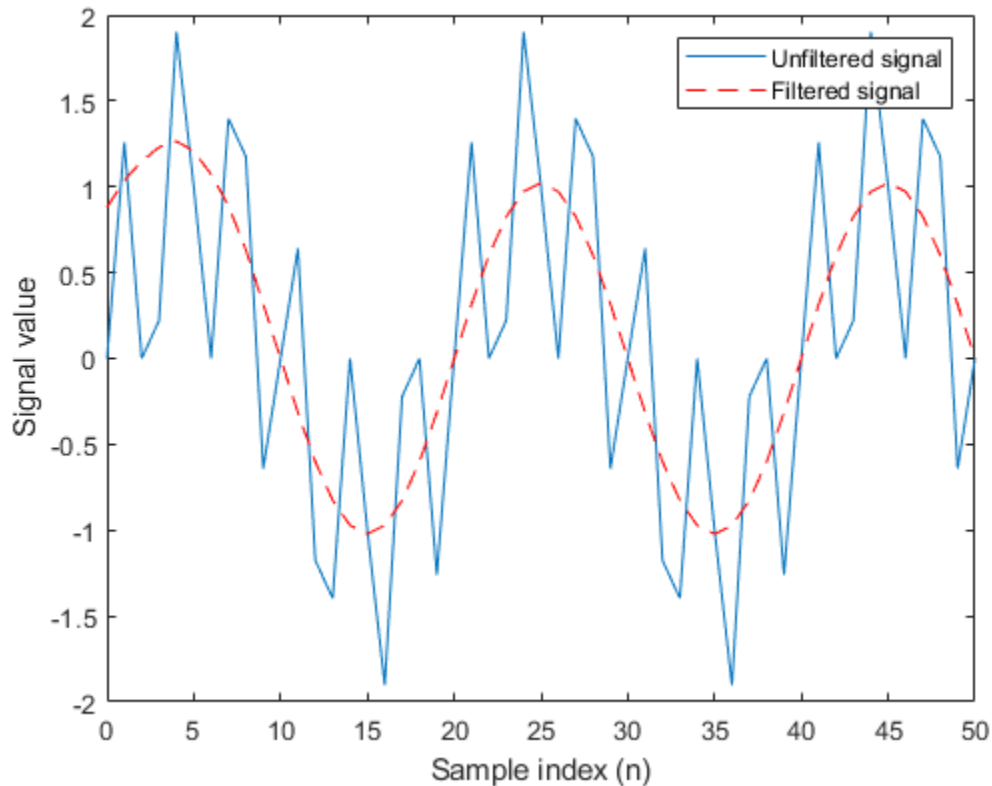
## Examples

### Filter a high-frequency fixed-point sinusoid from a signal

The following example filters a high-frequency fixed-point sinusoid from a signal that contains both a low- and high-frequency fixed-point sinusoid.

```
w1 = .1*pi;  
w2 = .6*pi;  
n = 0:999;  
xd = sin(w1*n) + sin(w2*n);  
x = sfi(xd,12);
```

```
b = ufi([.1:.1:1,1-.1:-.1:.1]/4,10);  
gd = (length(b)-1)/2;  
y = filter(b,1,x);  
  
% Plot results, accommodate for group-delay of filter  
plot(n(1:end-gd),x(1:end-gd))  
hold on  
plot(n(1:end-gd),y(gd+1:end),'r--')  
axis([0 50 -2 2])  
legend('Unfiltered signal','Filtered signal')  
xlabel('Sample index (n)')  
ylabel('Signal value')
```



The resulting plot shows both the unfiltered and filtered signals.

## More About

### ***Filter length (L)***

The filter length is `length(b)`, or the number of filter coefficients specified in the fixed-point vector *b*.

### **Filter order (N)**

The filter order is the number of states (delays) of the filter, and is equal to *L-1*.

## Tips

- The `filter` function only supports FIR filters. In the general filter representation, *b/a*, the denominator, *a*, of an FIR filter is the scalar 1, which is the second input of this function.
- The `numerictype` of *b* can be different than the `numerictype` of *x*.
- If you want to specify initial conditions, but do not know what `numerictype` to use, first try filtering your data without initial conditions. You can do so by specifying `[]` for the input *zi*. After performing the filtering operation, you have the `numerictype` of *y* and *zf* (if requested). Because the `numerictype` of *zi* must match that of *y* and *zf*, you now know the `numerictype` to use for the initial conditions.

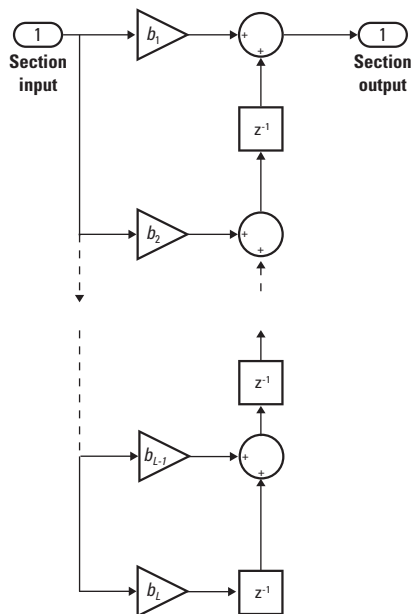
## Algorithms

The `filter` function uses a Direct-Form Transposed FIR implementation of the following difference equation:

$$y(n) = b_1 * x_n + b_2 * x_{n-1} + \dots + b_L * x_{n-N}$$

where *L* is the filter length on page 5-434 and *N* is the filter order on page 5-434.

The following diagram shows the direct-form transposed FIR filter structure used by the `filter` function:



## fimath Propagation Rules

The `filter` function uses the following rules regarding `fimath` behavior:

- `globalfimath` is obeyed.
- If any of the inputs has an attached `fimath`, then it is used for intermediate calculations.
- If more than one input has an attached `fimath`, then the `fimaths` must be equal.
- The output, `y`, is always associated with the default `fimath`.
- If the input vector, `zi`, has an attached `fimath`, then the output vector, `zf`, retains this `fimath`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`.

### **See Also**

`conv` | `filter`

**Introduced in R2010a**



# fimath

Set fixed-point math settings

## Syntax

```
F = fimath
F = fimath(...'PropertyName',PropertyValue...)
```

## Description

You can use the `fimath` constructor function in the following ways:

- `F = fimath` creates a `fimath` object with default `fimath` property settings:

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

- `F = fimath(...'PropertyName',PropertyValue...)` allows you to set the attributes of a `fimath` object using property name/property value pairs. All property names that you do not specify in the constructor use default values.

The properties of the `fimath` object are listed below. These properties are described in detail in “`fimath` Object Properties” in the Properties Reference.

- `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`.

---

- `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` — Word length, in bits, of the sum data type

## Examples

### Create a Default `fimath` Object

```
F = fimath
```

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

### Set Properties of a `fimath` Object

Set properties of a `fimath` object at the time of object creation by including properties after the arguments of the `fimath` constructor function. For example, 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
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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.
- If the `ProductMode` property of the `fimath` object is set to anything other than `FullPrecision`, the `ProductWordLength` and `ProductFractionLength` properties must be constant.
- If the `SumMode` property of the `fimath` object is set to anything other than `FullPrecision`, the `SumWordLength` and `SumFractionLength` properties must be constant.

### See Also

`fi` | `fipref` | `globalfimath` | `numericitype` | `quantizer` | `removefimath` | `setfimath`

### Topics

“`fimath` Object Construction”

“`fimath` Object Properties”

How Functions Use `fimath`

“`fimath` Properties Usage for Fixed-Point Arithmetic”

**Introduced before R2006a**

# fipref

Set fixed-point preferences

## Syntax

```
P = fipref  
P = fipref(...'PropertyName',PropertyValue...)
```

## Description

You can use the `fipref` constructor function in the following ways:

- `P = fipref` creates a default `fipref` object.
- `P = fipref(...'PropertyName',PropertyValue...)` allows you to set the attributes of a object using property name/property value pairs.

The properties of the `fipref` object are listed below. These properties are described in detail in “`fipref` Object Properties” on page 3-4.

- `FimathDisplay` — Display options for the local `fimath` attributes of `fi` objects. When `fi` objects do not have a local `fimath`, their `fimath` attributes are never displayed.
- `DataTypeOverride` — Data type override options.
- `DataTypeOverrideAppliesTo` — Data type override setting applicability.
- `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.

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.

See “View Fixed-Point Data” for more information on the display preferences used for most code examples in the documentation.

## Examples

### Example 1

Type

```
P = fipref
```

to create a default `fipref` object.

```
P =
```

```
    NumberDisplay: 'RealWorldValue'  
    NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'ForceOff'
```

### Example 2

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'
```

## See Also

`fi` | `fimath` | `numerictype` | `quantizer` | `savefipref`

## **Topics**

“fipref Object Construction”

“fipref Object Properties”

**Introduced before R2006a**

## **fix**

Round toward zero

### **Syntax**

```
y = fix(a)
```

### **Description**

`y = fix(a)` rounds `fi` object `a` to the nearest integer in the direction of zero and returns the result in `fi` object `y`.

`y` and `a` have the same `fi` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`fix` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.



## Examples

### Example 1

The following example demonstrates how the `fix` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 8  
      FractionLength: 3
```

```
y = fix(a)
```

```
y =
```

```
3
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 5  
      FractionLength: 0
```

### Example 2

The following example demonstrates how the `fix` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a =
```

```
0.0249
```

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

```

        WordLength: 8
        FractionLength: 12
    
```

```
y = fix(a)
```

```
y =
```

```
    0
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 2
        FractionLength: 0
    
```

### Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>ceil(a)</b>	<b>fix(a)</b>	<b>floor(a)</b>
- 2.5	-2	-2	-3
-1.75	-1	-1	-2
-1.25	-1	-1	-2
-0.5	0	0	-1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[ceil](#) | [convergent](#) | [floor](#) | [nearest](#) | [round](#)

**Introduced in R2008a**

## **fixed.aggregateType**

Compute aggregate numerictype

### **Syntax**

```
aggNT = fixed.aggregateType(A,B)
```

### **Description**

`aggNT = fixed.aggregateType(A,B)` computes the smallest binary point scaled numerictype that is able to represent both the full range and precision of inputs A and B.

### **Input Arguments**

**A**

An integer, binary point scaled fixed-point `fi` object, or numerictype object.

**B**

An integer, binary point scaled fixed-point `fi` object, or numerictype object.

### **Output Arguments**

**aggNT**

A numerictype object.

### **Examples**

Compute the aggregate numerictype of two numerictype objects.

```

% can represent range [-4,4) and precision 2^-13
a_nt = numerictype(1,16,13);
% can represent range [-2,2) and precision 2^-16
b_nt = numerictype(1,18,16);

% can represent range [-4,4) and precision 2^-16
aggNT = fixed.aggregateType(a_nt,b_nt)
aggNT =

```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 19
        FractionLength: 16

```

Compute the aggregate numerictype of two `fi` objects.

```

% Unsigned, WordLength: 16, FractionLength: 14
a_fi = ufi(pi,16);
% Signed, WordLength: 24, FractionLength: 21
b_fi = sfi(-pi,24);

% Signed, WordLength: 24, FractionLength: 21
aggNT = fixed.aggregateType(a_fi,b_fi)
aggNT =

```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 24
        FractionLength: 21

```

Compute the aggregate numerictype of a `fi` object and an integer.

```

% Unsigned, WordLength: 16, FractionLength: 14
% can represent range [0,3] and precision 2^-14
a_fi = ufi(pi,16);
% Unsigned, WordLength: 8, FractionLength: 0
% can represent range [0,255] and precision 2^0
cInt = uint8(0);

% Unsigned with WordLength: 14+8, FractionLength: 14
% can represent range [0,255] and precision 2^-14
aggNT = fixed.aggregateType(a_fi,cInt)
aggNT =

```

DataTypeMode: Fixed-point: binary point scaling  
Signedness: Unsigned  
WordLength: 22  
FractionLength: 14

## **See Also**

fi | numerictype

**Introduced in R2011b**

# fixed.Quantizer

Quantize fixed-point numbers

## Syntax

```
q = fixed.Quantizer
q = fixed.Quantizer(nt, rm, oa)
q = fixed.Quantizer(s, wl, fl, rm, oa)
q = fixed.Quantizer(Name, Value)
```

## Description

`q = fixed.Quantizer` creates a quantizer `q` that quantizes fixed-point (`fi`) numbers using default fixed-point settings.

`q = fixed.Quantizer(nt, rm, oa)` uses the `numericType` (`nt`) object information and the `RoundingMethod` (`rm`) and `OverflowAction` (`oa`) properties.

The `numericType`, `rounding method`, and `overflow action` apply only during the quantization. The resulting, quantized `q` does not have any `fi` attached to it.

`q = fixed.Quantizer(s, wl, fl, rm, oa)` uses the `Signed` (`s`), `WordLength` (`wl`), `FractionLength` (`fl`), `RoundingMethod` (`rm`), and `OverflowAction` (`oa`) properties.

`q = fixed.Quantizer(Name, Value)` creates a quantizer with the property options specified by one or more `Name, Value` pair arguments. You separate pairs of `Name, Value` arguments with commas. `Name` is the argument name, and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **nt**

Binary-point, scaled numerictype object or slope-bias scaled, fixed-point numerictype object. If your `fixed.Quantizer` uses a numerictype object that has either a Signedness of `Auto` or unspecified `Scaling`, an error occurs.

### **rm**

Rounding method to apply to the output data. Valid rounding methods are: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, and `Zero`. The associated property name is `RoundingMethod`.

**Default:** `Floor`

### **oa**

Overflow action to take in case of data overflow. Valid overflow actions are `Saturate` and `Wrap`. The associated property name is `OverflowAction`.

**Default:** `Wrap`

### **s**

Logical value, `true` or `false`, indicating whether the output is signed or unsigned, respectively. The associated property name is `Signed`.

**Default:** `true`

### **wl**

Word length (number of bits) of the output data. The associated property name is `WordLength`.

**Default:** `16`

### **fl**

Fraction length of the output data. The associated property name is `FractionLength`.

**Default:** `15`



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **Bias**

The bias is part of the numerical representation used to interpret a fixed-point number on page 5-456. Along with the slope, the bias forms the scaling of the number.

**Default:** 0

### **FixedExponent**

Fixed-point exponent associated with the object. The exponent is part of the numerical representation used to express a fixed-point number on page 5-456.

The exponent of a fixed-point number is equal to the negative of the fraction length. `FixedExponent` must be an integer.

**Default:** -15

### **FractionLength**

Fraction length of the stored integer value of the object, in bits. The fraction length can be any integer value.

This property automatically defaults to the best precision possible based on the value of the word length and the real-world value of the `fi` object.

**Default:** 15

### **OverflowAction**

Action to take in case of data overflow. Valid overflow actions are `Saturate` and `Wrap`. .

**Default:** `Wrap`

**RoundingMethod**

Rounding method to apply to the output data. Valid rounding methods are: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, and `Zero`.

**Default:** `Floor`

**Signed**

Whether the object is signed. The possible values of this property are:

- `1` — signed
- `0` — unsigned
- `true` — signed
- `false` — unsigned

---

**Note** Although the `Signed` property is still supported, the `Signedness` property always appears in the `numericType` object display. If you choose to change or set the signedness of your `numericType` object using the `Signed` property, MATLAB updates the corresponding value of the `Signedness` property.

---

**Default:** `true`

**Signedness**

Whether the object is signed, unsigned, or has an unspecified sign. The possible values of this property are:

- `Signed` — signed
- `Unsigned` — unsigned

**Default:** `Signed`

**Slope**

Slope associated with the object. The slope is part of the numerical representation used to express a fixed-point number on page 5-456. Along with the bias, the slope forms the scaling of a fixed-point number.

**Default:**  $2^{-15}$

### SlopeAdjustmentFactor

Slope adjustment associated with the object. The slope adjustment is equivalent to the fractional slope of a fixed-point number. The fractional slope is part of the numerical representation used to express a fixed-point number.

SlopeAdjustmentFactor must be greater than or equal to 1 and less than 2.

**Default:** 1

### WordLength

Word length of the stored integer value of the object, in bits. The word length can be any positive integer value.

**Default:** 16

## Output Arguments

**q**

Quantizer that quantizes *fi* input numbers

## Examples

Use `fixed.Quantizer` to reduce the word length that results from adding two fixed-point numbers.

```
q = fixed.Quantizer;
x1 = fi(0.1,1,16,15);
x2 = fi(0.8,1,16,15);
y = quantize(q,x1+x2);
```

Use `fixed.Quantizer` object to change a binary point scaled fixed-point *fi* to a slope-bias scaled fixed-point *fi*

```
qsb = fixed.Quantizer(numericType(1,7,1.6,0.2),...
    'Round', 'Saturate');
ysb = quantize(qsb,fi(pi,1,16,13));
```

## More About

### Fixed-point numbers

Fixed-point numbers can be represented as

$$\textit{real-worldvalue} = (\textit{slope} \times \textit{storedinteger}) + \textit{bias}$$

where the slope can be expressed as

$$\textit{slope} = \textit{fractionalslope} \times 2^{\textit{fixedexponent}}$$

## Tips

- Use `y = quantize(q, x)` to quantize input array `x` using the fixed-point settings of quantizer `q`. `x` can be any fixed-point number `fi`, except a Boolean value. If `x` is a scaled double, the `x` and `y` data will be the same, but `y` will have fixed-point settings. If `x` is a double or single then `y = x`. This functionality lets you share the same code for both floating-point data types and `fi` objects when quantizers are present.
- Use `n = numerictype(q)` to get a `numerictype` for the current settings of quantizer `q`.
- Use `clone(q)` to create a quantizer object with the same property values as `q`.
- If you use a `fixed.quantizer` in code generation, note that it is a handle object and must be declared as persistent.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fi` | `numerictype` | `quantizer`

## **Topics**

“Set numerictype Object Properties”

**Introduced in R2011b**

## fixpt\_instrument\_purge

Remove corrupt fixed-point instrumentation from model

### Compatibility

---

**Note** `fixpt_instrument_purge` will be removed in a future release.

---

### Syntax

```
fixpt_instrument_purge  
fixpt_instrument_purge(modelName, interactive)
```

### Description

The `fixpt_instrument_purge` script finds and removes fixed-point instrumentation from a model left by the Fixed-Point Tool and the fixed-point autoscaling script. The Fixed-Point Tool and the fixed-point autoscaling script each add callbacks to a model. For example, the Fixed-Point Tool appends commands to model-level callbacks. These callbacks make the Fixed-Point Tool respond to simulation events. Similarly, the autoscaling script adds instrumentation to some parameter values that gathers information required by the script.

Normally, these types of instrumentation are automatically removed from a model. The Fixed-Point Tool removes its instrumentation when the model is closed. The autoscaling script removes its instrumentation shortly after it is added. However, there are cases where abnormal termination of a model leaves fixed-point instrumentation behind. The purpose of `fixpt_instrument_purge` is to find and remove fixed-point instrumentation left over from abnormal termination.

`fixpt_instrument_purge(modelName, interactive)` removes instrumentation from model `modelName`. `interactive` is true by default, which prompts you to make each change. When `interactive` is set to false, all found instrumentation is automatically removed from the model.

## **See Also**

autofixexp | fxptdlg

**Introduced before R2006a**

## floor

Round toward negative infinity

### Syntax

```
y = floor(a)
```

### Description

`y = floor(a)` rounds `fi` object `a` to the nearest integer in the direction of negative infinity and returns the result in `fi` object `y`.

`y` and `a` have the same `fi` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`floor` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.



## Examples

### Example 1

The following example demonstrates how the `floor` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 8  
      FractionLength: 3
```

```
y = floor(a)
```

```
y =
```

```
3
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 5  
      FractionLength: 0
```

### Example 2

The following example demonstrates how the `floor` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a =
```

```
0.0249
```

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

```

        WordLength: 8
    FractionLength: 12

```

```
y = floor(a)
```

```
y =
```

```
    0
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 2
    FractionLength: 0

```

### Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>ceil(a)</b>	<b>fix(a)</b>	<b>floor(a)</b>
- 2.5	-2	-2	-3
-1.75	-1	-1	-2
-1.25	-1	-1	-2
-0.5	0	0	-1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[ceil](#) | [convergent](#) | [fix](#) | [nearest](#) | [round](#)

**Introduced in R2008a**

## fma

Multiply and add using fused multiply add approach

### Syntax

```
X = fma(A, B, C)
```

### Description

`X = fma(A, B, C)` computes  $A*B+C$  using a fused multiply add approach. Fused multiply add operations round only once, often making the result more accurate than performing a multiplication operation followed by an addition.

### Examples

#### Multiply and Add Three Inputs Using Fused Multiply Add

This example shows how to use the `fma` function to calculate  $A \times B + C$  using a fused multiply add approach.

Define the inputs and use the `fma` function to compute the multiply add operation.

```
a = half(10);  
b = half(10);  
c = half(2);  
x = fma(a, b, c)
```

```
x =
```

```
  half
```

```
  102
```

Compare the result of the `fma` function with the two-step approach of computing the product and then the sum.

```
temp = a * b;  
x = temp + c
```

```
x =
```

```
half
```

```
102
```

## Input Arguments

### A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `half`

### B — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `half`

### C — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `half`

## Output Arguments

### X — Result of multiply and add operation

scalar | vector | matrix | multidimensional array

Result of multiply and add operation,  $A*B+C$ , returned as a scalar, vector, matrix, or multidimensional array.

## **See Also**

half

**Introduced in R2019a**

## for

Execute statements specified number of times

## Syntax

```
for index = values
    statements
end
```

## Description

`for index = valuesstatements, end` executes a group of statements in a loop for a specified number of times.

If a colon operation with `fi` objects is used as the index, then the `fi` objects must be whole numbers.

Refer to the MATLAB `for` reference page for more information.

## Example

### Use `fi` in a For Loop

Use a `fi` object as the index of a for-loop.

```
a = fi(1,0,8,0);
b = fi(2,0,8,0);
c = fi(10,0,8,0);
```

```
for x = a:b:c
```

x  
end

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**Introduced in R2014b**



# fractionlength

Fraction length of quantizer object

## Syntax

`fractionlength(q)`

## Description

`fractionlength(q)` returns the fraction length of quantizer object `q`.

## Algorithms

For floating-point quantizer objects,  $f = w - e - 1$ , where  $w$  is the word length and  $e$  is the exponent length.

For fixed-point quantizer objects,  $f$  is part of the format `[w f]`.

## See Also

`fi` | `numericType` | `quantizer` | `wordlength`

**Introduced before R2006a**

## fxpopt

Optimize data types of a system

### Syntax

```
result = fxpopt(model, sud, options)
```

### Description

`result = fxpopt(model, sud, options)` optimizes the data types in the model or subsystem specified by `sud` in the model, `model`, with additional options specified in the `fxpOptimizationOptions` object, `options`.

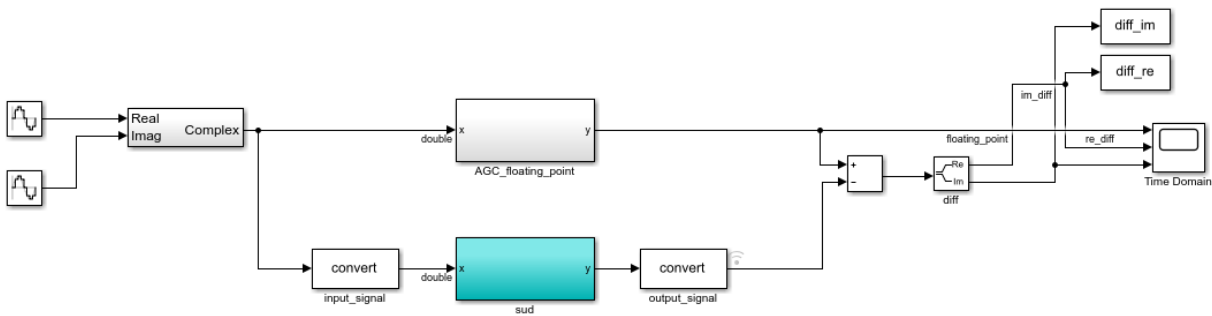
### Examples

#### Optimize Fixed-Point Data Types

This example shows how to optimize the data types used by a system based on specified tolerances.

To begin, open the system for which you want to optimize the data types.

```
model = 'ex_auto_gain_controller';  
sud = 'ex_auto_gain_controller/sud';  
open_system(model)
```



Copyright 2017 The MathWorks, Inc.

Create an `fxpOptimizationOptions` object to define constraints and tolerances to meet your design goals. Set the `UseParallel` property of the `fxpOptimizationOptions` object to `true` to run iterations of the optimization in parallel. You can also specify word lengths to allow in your design through the `AllowableWordLengths` property.

```
opt = fxpOptimizationOptions('AllowableWordLengths', 10:24, 'UseParallel', true)
```

```
opt =
```

```
fxpOptimizationOptions with properties:
```

```

    MaxIterations: 50
      MaxTime: 600
    Patience: 10
    Verbosity: High
AllowableWordLengths: [10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

```
Advanced Options
```

```
AdvancedOptions: [1x1 struct]
```

Use the `addTolerance` method to define tolerances for the differences between the original behavior of the system, and the behavior using the optimized fixed-point data types.

```
tol = 10e-2;
addTolerance(opt, [model '/output_signal'], 1, 'AbsTol', tol);
```

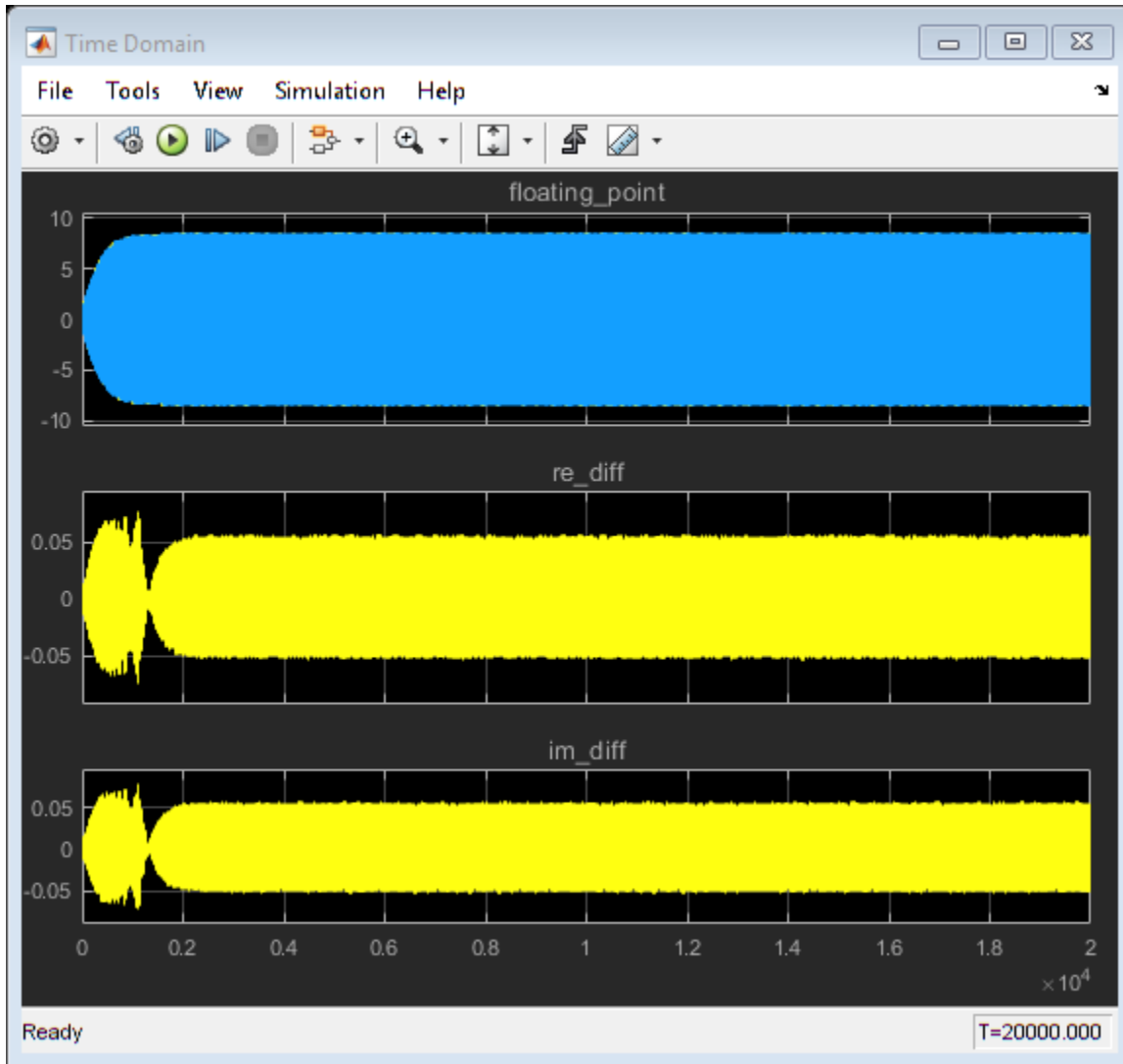
Use the `fxpopt` function to run the optimization. The software analyzes ranges of objects in your system under design and the constraints specified in the `fxpOptimizationOptions` object to apply heterogeneous data types to your system while minimizing total bit width.

```
result = fxpopt(model, sud, opt);

+ Preprocessing
+ Modeling the optimization problem
  - Constructing decision variables
  - Binding constraints
+ Running the optimization solver
  - Evaluating new solution: cost 180, does not meet the tolerances.
  - Evaluating new solution: cost 198, does not meet the tolerances.
  - Evaluating new solution: cost 216, does not meet the tolerances.
  - Evaluating new solution: cost 234, does not meet the tolerances.
  - Evaluating new solution: cost 252, does not meet the tolerances.
  - Evaluating new solution: cost 270, does not meet the tolerances.
  - Evaluating new solution: cost 288, does not meet the tolerances.
  - Evaluating new solution: cost 306, meets the tolerances.
  - Updated best found solution, cost: 306
  - Evaluating new solution: cost 304, meets the tolerances.
  - Updated best found solution, cost: 304
  - Evaluating new solution: cost 300, meets the tolerances.
  - Updated best found solution, cost: 300
  - Evaluating new solution: cost 299, meets the tolerances.
  - Updated best found solution, cost: 299
  - Evaluating new solution: cost 298, meets the tolerances.
  - Updated best found solution, cost: 298
  - Evaluating new solution: cost 297, meets the tolerances.
  - Updated best found solution, cost: 297
  - Evaluating new solution: cost 296, meets the tolerances.
  - Updated best found solution, cost: 296
  - Evaluating new solution: cost 295, meets the tolerances.
  - Updated best found solution, cost: 295
  - Evaluating new solution: cost 294, does not meet the tolerances.
  - Evaluating new solution: cost 293, meets the tolerances.
  - Updated best found solution, cost: 293
  - Evaluating new solution: cost 292, meets the tolerances.
  - Updated best found solution, cost: 292
  - Evaluating new solution: cost 291, meets the tolerances.
  - Updated best found solution, cost: 291
```

- Evaluating new solution: cost 290, meets the tolerances.
- Updated best found solution, cost: 290
- Evaluating new solution: cost 289, meets the tolerances.
- Updated best found solution, cost: 289
- Evaluating new solution: cost 287, meets the tolerances.
- Updated best found solution, cost: 287
- Evaluating new solution: cost 283, meets the tolerances.
- Updated best found solution, cost: 283
- Evaluating new solution: cost 282, meets the tolerances.
- Updated best found solution, cost: 282
- Evaluating new solution: cost 281, meets the tolerances.
- Updated best found solution, cost: 281
- Evaluating new solution: cost 280, meets the tolerances.
- Updated best found solution, cost: 280
- Evaluating new solution: cost 279, meets the tolerances.
- Updated best found solution, cost: 279
- Evaluating new solution: cost 278, meets the tolerances.
- Updated best found solution, cost: 278
- Evaluating new solution: cost 277, does not meet the tolerances.
- Evaluating new solution: cost 276, meets the tolerances.
- Updated best found solution, cost: 276
- Evaluating new solution: cost 275, meets the tolerances.
- Updated best found solution, cost: 275
- Evaluating new solution: cost 274, meets the tolerances.
- Updated best found solution, cost: 274
- Evaluating new solution: cost 273, meets the tolerances.
- Updated best found solution, cost: 273
- Evaluating new solution: cost 272, meets the tolerances.
- Updated best found solution, cost: 272
- Evaluating new solution: cost 270, meets the tolerances.
- Updated best found solution, cost: 270
- Evaluating new solution: cost 266, meets the tolerances.
- Updated best found solution, cost: 266
- Evaluating new solution: cost 265, meets the tolerances.
- Updated best found solution, cost: 265
- Evaluating new solution: cost 264, meets the tolerances.
- Updated best found solution, cost: 264
- Evaluating new solution: cost 263, meets the tolerances.
- Updated best found solution, cost: 263
- Evaluating new solution: cost 262, meets the tolerances.
- Updated best found solution, cost: 262
- Evaluating new solution: cost 261, meets the tolerances.
- Updated best found solution, cost: 261
- Evaluating new solution: cost 260, does not meet the tolerances.

- Evaluating new solution: cost 259, does not meet the tolerances.
- Evaluating new solution: cost 260, meets the tolerances.
- Updated best found solution, cost: 260
- Evaluating new solution: cost 259, meets the tolerances.
- Updated best found solution, cost: 259
- Evaluating new solution: cost 258, meets the tolerances.
- Updated best found solution, cost: 258
- Evaluating new solution: cost 257, meets the tolerances.
- Updated best found solution, cost: 257
- Evaluating new solution: cost 255, meets the tolerances.
- Updated best found solution, cost: 255
- Evaluating new solution: cost 251, meets the tolerances.
- Updated best found solution, cost: 251
- Evaluating new solution: cost 250, meets the tolerances.
- Updated best found solution, cost: 250
- Evaluating new solution: cost 249, meets the tolerances.
- Updated best found solution, cost: 249
- Evaluating new solution: cost 248, meets the tolerances.
- Updated best found solution, cost: 248
- Evaluating new solution: cost 247, meets the tolerances.
- Updated best found solution, cost: 247
- Evaluating new solution: cost 246, meets the tolerances.
- Updated best found solution, cost: 246
- Evaluating new solution: cost 245, does not meet the tolerances.
- Evaluating new solution: cost 244, does not meet the tolerances.
- Evaluating new solution: cost 245, meets the tolerances.
- Updated best found solution, cost: 245
- Evaluating new solution: cost 244, meets the tolerances.
- Updated best found solution, cost: 244
- + Optimization has finished.
  - Neighborhood search complete.
  - Maximum number of iterations completed.
- + Fixed-point implementation that met the tolerances found.
  - Total cost: 244
  - Maximum absolute difference: 0.077478
  - Use the explore method of the result to explore the implementation.



Use the `explore` method of the `OptimizationResult` object, `result`, to launch Simulation Data Inspector and explore the design containing the smallest total number of bits while maintaining the numeric tolerances specified in the `opt` object.

```
explore(result);
```

## Input Arguments

**model** — Model containing system under design, sud  
character vector

Name of the model containing the system that you want to optimize.

Data Types: char

**sud** — Model or subsystem whose data types you want to optimize  
character vector

Model or subsystem whose data types you want to optimize, specified as a character vector containing the path to the system.

Data Types: char

**options** — Additional optimization options  
fxpOptimizationOptions object

fxpOptimizationOptions object specifying additional options to use during the data type optimization process.

## Output Arguments

**result** — Object containing the optimized design  
OptimizationResult object

Result of the optimization, returned as an OptimizationResult object. Use the explore method of the object to open the Simulation Data Inspector and view the behavior of the optimized system. You can also explore other solutions found during the optimization that may or may not meet the constraints specified in the fxpOptimizationOptions object, options.



## See Also

### Classes

OptimizationResult | OptimizationSolution | fxpOptimizationOptions

### Functions

addTolerance | explore | showTolerances

### Topics

“Optimize Fixed-Point Data Types for a System”

**Introduced in R2018a**

## ge

Determine whether real-world value of one `fi` object is greater than or equal to another

## Syntax

```
c = ge(a,b)
a >= b
```

## Description

`c = ge(a,b)` is called for the syntax `a >= b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a >= b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

## Examples

### Compare Two `fi` Objects

Use the `ge` function to determine whether the real-world value of one `fi` object is greater than or equal to another.

```
a = fi(pi);
b = fi(pi, 1, 32);
b >= a
```

```
ans = logical
      0
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The **ge** function returns **0** because after quantization, the value of **a** is slightly greater than that of **b**.

### Compare a Double to a fi Object

When comparing a double to a **fi** object, the double is cast to the same word length and signedness of the **fi** object.

```
a = fi(pi);
b = pi;
a >= b
```

```
ans = logical
      1
```

The **ge** function casts **b** to the same word length as **a**, and returns **1** because the two inputs have the same real-world value. This behavior allows relational operations to work between **fi** objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

## **See Also**

eq | gt | le | lt | ne

**Introduced before R2006a**

## get

Property values of object

### Syntax

```
value = get(o, 'propertyname')  
structure = get(o)
```

### Description

`value = get(o, 'propertyname')` returns the property value of the property 'propertyname' for the object `o`. If you replace 'propertyname' by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(o)` returns a structure containing the properties and states of object `o`.

`o` can be a `fi`, `fimath`, `fipref`, `numericType`, or `quantizer` object.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The syntax `structure = get(o)` is not supported.

### See Also

`set`

**Introduced before R2006a**

## getlsb

Least significant bit

### Syntax

```
c = getlsb(a)
```

### Description

`c = getlsb(a)` returns the value of the least significant bit in `a` as a `u1, 0`.

`a` can be a scalar `fi` object or a vector `fi` object.

`getlsb` only supports `fi` objects with fixed-point data types.

### Examples

The following example uses `getlsb` to find the least significant bit in the `fi` object `a`.

```
a = fi(-26, 1, 6, 0);  
c = getlsb(a)
```

```
c =
```

```
0
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 1  
    FractionLength: 0
```

You can verify that the least significant bit in the `fi` object `a` is `0` by looking at the binary representation of `a`.

```
disp(bin(a))
```

```
100110
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` | `bitxor` | `bitxorreduce` | `getmsb`

**Introduced in R2007b**



# getmsb

Most significant bit

## Syntax

```
c = getmsb(a)
```

## Description

`c = getmsb(a)` returns the value of the most significant bit in `a` as a `u1,0`.

`a` can be a scalar `fi` object or a vector `fi` object.

`getmsb` only supports `fi` objects with fixed-point data types.

## Examples

The following example uses `getmsb` to find the most significant bit in the `fi` object `a`.

```
a = fi(-26, 1, 6, 0);  
c = getmsb(a)
```

```
c =
```

```
    1
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Unsigned  
        WordLength: 1  
        FractionLength: 0
```

```
>>
```

You can verify that the most significant bit in the `fi` object `a` is 1 by looking at the binary representation of `a`.

```
disp(bin(a))
```

```
100110
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` |  
`bitxor` | `bitxorreduce` | `getlsb`

**Introduced in R2007b**

# globalfimath

Configure global fimath and return handle object

## Syntax

```
G = globalfimath
G = globalfimath('PropertyName',PropertyValue1,...)
G = globalfimath(f)
```

## Description

`G = globalfimath` returns a handle object to the global fimath. The global fimath has identical properties to a `fimath` object but applies globally.

`G = globalfimath('PropertyName',PropertyValue1,...)` sets the global fimath using the named properties and their corresponding values. Properties that you do not specify in this syntax are automatically set to that of the current global fimath.

`G = globalfimath(f)` sets the properties of the global fimath to match those of the input `fimath` object `f`, and returns a handle object to it.

Unless, in a previous release, you used the `saveglobalfimathpref` function to save global fimath settings to your MATLAB preferences, the global fimath properties you set with the `globalfimath` function apply only to your current MATLAB session. It is best practice to remove global fimath from the MATLAB preferences so that you start each MATLAB session using the default `fimath` settings. To remove the global fimath, use the `removeglobalfimathpref` function.

## Examples

### Modifying globalfimath

Use the `globalfimath` function to set, change, and reset the global fimath.

Create a fimath object and use it as the global fimath.

```
G = globalfimath('RoundMode', 'Floor', 'OverflowMode', 'Wrap')
```

```
G =  
    RoundingMethod: Floor  
    OverflowAction: Wrap  
    ProductMode: FullPrecision  
    SumMode: FullPrecision
```

Create another fimath object using the new default.

```
F1 = fimath
```

```
F1 =  
    RoundingMethod: Floor  
    OverflowAction: Wrap  
    ProductMode: FullPrecision  
    SumMode: FullPrecision
```

Create a fi object, A, associated with the global fimath.

```
A = fi(pi)
```

```
A =  
    3.1416  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 13
```

Now set the "SumMode" property of the global fimath to "KeepMSB" and retain all the other property values of the current global fimath.

```
G = globalfimath('SumMode', 'KeepMSB')
```

```
G =  
    RoundingMethod: Floor  
    OverflowAction: Wrap  
    ProductMode: FullPrecision  
    SumMode: KeepMSB  
    SumWordLength: 32  
    CastBeforeSum: true
```

Change the global fimath by directly interacting with the handle object G.

```
G.ProductMode = 'SpecifyPrecision'

G =
    RoundingMethod: Floor
    OverflowAction: Wrap
    ProductMode: SpecifyPrecision
    ProductWordLength: 32
    ProductFractionLength: 30
    SumMode: KeepMSB
    SumWordLength: 32
    CastBeforeSum: true
```

Reset the global fimath to the factory default by calling the reset method on G. This is equivalent to using the resetglobalfimath function.

```
reset(G);
G

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

## Tips

If you always use the same `fimath` settings and you are not sharing code with other people, using the `globalfimath` function is a quick, convenient method to configure these settings. However, if you share the code with other people or if you use the `fiaccel` function to accelerate the algorithm or you generate C code for your algorithm, consider the following alternatives.

Goal	Issue Using <code>globalfimath</code>	Solution
Share code	If you share code with someone who is using different global <code>fimath</code> settings, they might see different results.	Separate the <code>fimath</code> properties from your algorithm by using types tables. For more information, see “Separate Data Type Definitions from Algorithm”.

Goal	Issue Using <code>globalfimath</code>	Solution
Accelerate your algorithm using <code>fiaccel</code> or generate C code from your MATLAB algorithm using <code>codegen</code>	You cannot use <code>globalfimath</code> within that algorithm. If you generate code with one <code>globalfimath</code> setting and run it with a different <code>globalfimath</code> setting, results might vary. For more information, see <a href="#">Specifying Default fimath Values for MEX Functions</a> .	Use types tables in the algorithm from which you want to generate code. This insulates you from the global settings and makes the code portable. For more information, see <a href="#">“Separate Data Type Definitions from Algorithm”</a> .

## See Also

[codegen](#) | [fiaccel](#) | [fimath](#) | [removeglobalfimathpref](#) | [resetglobalfimath](#)

**Introduced in R2010a**

## gt

Determine whether real-world value of one `fi` object is greater than another

### Syntax

```
c = gt(a,b)
a > b
```

### Description

`c = gt(a,b)` is called for the syntax `a > b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a > b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

### Examples

#### Compare Two `fi` Objects

Use the `gt` function to determine whether the real-world value of one `fi` object is greater than another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a > b
```

```
ans = logical
      1
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The **gt** function returns 1 because after quantization, the value of **a** is greater than that of **b**.

### Compare a Double to a fi Object

When comparing a double to a **fi** object, the double is cast to the same word length and signedness of the **fi** object.

```
a = fi(pi);
b = pi;
a > b
```

```
ans = logical
      0
```

The **gt** function casts **b** to the same word length as **a**, and returns 0 because the two inputs have the same real-world value. This behavior allows relational operations to work between **fi** objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.



## **See Also**

eq | ge | le | lt | ne

**Introduced before R2006a**

## half

Construct half-precision numeric object

### Description

Use the `half` constructor to assign a half-precision data type to a number or variable. Half-precision data types occupy only 16 bits of memory, but their floating-point representation enables them to handle wider dynamic ranges than integer or fixed-point data types of the same size.

For more information, see “Floating-Point Numbers”.

### Creation

### Syntax

```
a = half(v)
```

### Description

`a = half(v)` converts the values in `v` to half-precision.

### Input Arguments

**v** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## Object Functions

These functions are supported for use with half-precision inputs.

### Math and Arithmetic

abs	Absolute value and complex magnitude
ceil	Round toward positive infinity
cos	Cosine of argument in radians
cospi	Compute $\cos(X*\pi)$ accurately
dot	Dot product
exp	Exponential
expm1	Compute $\exp(x)-1$ accurately for small values of $x$
fix	Round toward zero
floor	Round toward negative infinity
fma	Multiply and add using fused multiply add approach
hypot	Square root of sum of squares (hypotenuse)
ldivide	Left array division
log	Natural logarithm
log1p	Compute $\log(1+x)$ accurately for small values of $x$
mean	Average or mean value of array
minus	Subtraction
mod	Remainder after division (modulo operation)
mtimes	Matrix multiplication
plus	Addition
pow10	Base 10 power and scale half-precision numbers
pow2	Base 2 power and scale floating-point numbers
prod	Product of array elements
rdivide	Right array division
rem	Remainder after division
round	Round to nearest decimal or integer
rsqrt	Reciprocal square root
sin	Sine of argument in radians
sinpi	Compute $\sin(X*\pi)$ accurately
sqrt	Square root
sum	Sum of array elements
tanh	Hyperbolic tangent
times	Multiplication
uminus	Unary minus
uplus	Unary plus

## Data Types

cast	Convert variable to different data type
double	Double-precision arrays
eps	Floating-point relative accuracy
Inf	Create array of all Inf values
int16	16-bit signed integer arrays
int32	32-bit signed integer arrays
int64	64-bit signed integer arrays
int8	8-bit signed integer arrays
isa	Determine if input has specified data type
isfloat	Determine whether input is floating-point data type
islogical	Determine if input is logical array
isnan	Determine which array elements are NaN
isnumeric	Determine whether input is numeric array
isreal	Determine whether array is real
logical	Convert numeric values to logicals
NaN	Create array of all NaN values
single	Single-precision arrays
uint16	16-bit unsigned integer arrays
uint32	32-bit unsigned integer arrays
uint64	64-bit unsigned integer arrays
uint8	8-bit unsigned integer arrays

## Relational and Logical Operators

all	Determine if all array elements are nonzero or true
any	Determine if any array elements are nonzero
eq	Determine equality
ge	Determine greater than or equal to
gt	Determine greater than
le	Determine less than or equal to
lt	Determine less than
ne	Determine inequality

## Array and Matrix Operations

cat	Concatenate arrays
colon	Vector creation, array subscripting, and for-loop iteration
eye	Identity matrix
full	Convert sparse matrix to full storage

horzcat	Horizontal concatenation for heterogeneous arrays
isempty	Determine whether array is empty
isfinite	Determine which array elements are finite
isinf	Determine which array elements are infinite
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
length	Length of largest array dimension
min	Minimum of timeseries data
max	Maximum of timeseries data
ndims	Number of array dimensions
numel	Number of array elements
ones	Create array of all ones
repmat	Repeat copies of array
reshape	Reshape array
size	Array size
subsasgn	Redefine subscripted assignment
subsref	Subscripted reference
transpose	Transpose vector or matrix
vertcat	Vertical concatenation for heterogeneous arrays
zeros	Create array of all zeros

## Language Fundamentals

display	Show information about variable or expression result
end	Terminate block of code or indicate last array index

## Graphics

bar	Bar graph
barh	Horizontal bar graph
fplot	Plot expression or function
line	Create primitive line
plot	2-D line plot
plot3	3-D point or line plot
plotmatrix	Scatter plot matrix
rgbplot	Plot colormap
scatter	Scatter plot
scatter3	3-D scatter plot
xlim	Set or query x-axis limits
ylim	Set or query y-axis limits
zlim	Set or query z-axis limits

## Examples

### Convert Value to Half Precision

To cast a double-precision number to half precision, use the `half` function.

```
a = half(pi)
```

```
a =
```

```
    half
```

```
    3.1406
```

You can also use the `half` function to cast an existing variable to half-precision.

```
v = single(magic(3))
```

```
v = 3x3 single matrix
```

```
     8     1     6
     3     5     7
     4     9     2
```

```
a = half(v)
```

```
a =
```

```
    3x3 half matrix
```

```
     8     1     6
     3     5     7
     4     9     2
```

## Limitations

The following functions which support half-precision inputs, do not support complex half-precision inputs.

- `rsqrt`
- `fma`

## See Also

`double` | `single`

## Topics

“Floating-Point Numbers”

**Introduced in R2018b**

## hex

Hexadecimal representation of stored integer of `fi` object

## Syntax

`hex(a)`

## Description

`hex(a)` returns the stored integer of `fi` object `a` in hexadecimal format as a character vector. `hex(a)` is equivalent to `a.hex`.

Fixed-point numbers can be represented as

$$\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}$$

or, equivalently as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

## Examples

### Example 5.1. Viewing `fi` Objects in Hexadecimal Format

The following code

```
a = fi([-1 1],1,8,7);  
y = hex(a)  
z = a.hex
```

returns

`y =`



```
      80    7f
z =
      80    7f
```

### Example 5.2. Writing Hex Data to a File

The following example shows how to write hex data from the MATLAB workspace into a text file.

First, define your data and create a writable text file called `hexdata.txt`:

```
x = (0:15)'/16;
a = fi(x,0,16,16);

h = fopen('hexdata.txt','w');
```

Use the `fprintf` function to write your data to the `hexdata.txt` file:

```
for k=1:length(a)
    fprintf(h,'%s\n',hex(a(k)));
end
fclose(h);
```

To see the contents of the file you created, use the `type` function:

```
type hexdata.txt
```

MATLAB returns:

```
0000
1000
2000
3000
4000
5000
6000
7000
8000
9000
a000
b000
c000
d000
```

```
e000  
f000
```

**Example 5.3. Reading Hex Data from a File**

The following example shows how to read hex data from a text file back into the MATLAB workspace.

Open `hexdata.txt` for reading and read its contents into a workspace variable:

```
h = fopen('hexdata.txt','r');  
  
nextline = '';  
str='';  
while ischar(nextline)  
    nextline = fgetl(h);  
    if ischar(nextline)  
        str = [str;nextline];  
    end  
end
```

Create a `fi` object with the correct scaling and assign it the hex values stored in the `str` variable:

```
b = fi([],0,16,16);  
b.hex = str
```

```
b =  
    0  
    0.0625  
    0.1250  
    0.1875  
    0.2500  
    0.3125  
    0.3750  
    0.4375  
    0.5000  
    0.5625  
    0.6250  
    0.6875  
    0.7500  
    0.8125  
    0.8750  
    0.9375
```

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

## **See Also**

bin | dec | oct | storedInteger

**Introduced before R2006a**

## hex2num

Convert hexadecimal string to number using `quantizer` object

### Syntax

```
x = hex2num(q,h)
[x1,x2,...] = hex2num(q,h1,h2,...)
```

### Description

`x = hex2num(q,h)` converts hexadecimal character vector `h` to numeric matrix `x`. The attributes of the numbers in `x` are specified by `quantizer` object `q`. When `h` is a cell array, `hex2num` returns `x` as a cell array of the same dimension containing numbers. For fixed-point hexadecimal representations, `hex2num` uses two's complement representation. For floating-point, the representation is IEEE Standard 754 style.

When there are fewer hexadecimal digits than needed to represent the number, the fixed-point conversion zero-fills on the left. Floating-point conversion zero-fills on the right.

`[x1,x2,...] = hex2num(q,h1,h2,...)` converts hexadecimal representations `h1`, `h2`,... to numeric matrices `x1`, `x2`,...

`hex2num` and `num2hex` are inverses of one another, with the distinction that `num2hex` returns the hexadecimal representations in a column.

### Examples

To create all the 4-bit fixed-point two's complement numbers in fractional form, use the following code.

```
q = quantizer([4 3]);
h = ['7 3 F B'; '6 2 E A'; '5 1 D 9'; '4 0 C 8'];
x = hex2num(q,h)

x =
```

0.8750	0.3750	-0.1250	-0.6250
0.7500	0.2500	-0.2500	-0.7500
0.6250	0.1250	-0.3750	-0.8750
0.5000	0	-0.5000	-1.0000

## See Also

[bin2num](#) | [num2bin](#) | [num2hex](#) | [num2int](#)

**Introduced before R2006a**

## horzcat

Horizontally concatenate multiple `fi` objects

### Syntax

```
c = horzcat(a,b,...)
[a, b, ...]
```

### Description

`c = horzcat(a,b,...)` is called for the syntax `[a, b, ...]` when any of `a, b, ...`, is a `fi` object.

`[a b, ...]` or `[a,b, ...]` is the horizontal concatenation of matrices `a` and `b`. `a` and `b` must have the same number of rows. Any number of matrices can be concatenated within one pair of brackets. N-D arrays are horizontally concatenated along the second dimension. The first and remaining dimensions must match.

Horizontal and vertical concatenation can be combined together as in `[1 2;3 4]`.

`[a b; c]` is allowed if the number of rows of `a` equals the number of rows of `b`, and if the number of columns of `a` plus the number of columns of `b` equals the number of columns of `c`.

The matrices in a concatenation expression can themselves be formed via a concatenation as in `[a b;[c d]]`.

---

**Note** The `fimath` and `numericType` properties of a concatenated matrix of `fi` objects `c` are taken from the leftmost `fi` object in the list `(a,b,...)`.

---

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

vertcat

**Introduced before R2006a**

## innerprodintbits

Number of integer bits needed for fixed-point inner product

### Syntax

```
innerprodintbits(a,b)
```

### Description

`innerprodintbits(a,b)` computes the minimum number of integer bits necessary in the inner product of  $a' * b$  to guarantee that no overflows occur and to preserve best precision.

- `a` and `b` are `fi` vectors.
- The values of `a` are known.
- Only the numeric type of `b` is relevant. The values of `b` are ignored.

### Examples

The primary use of this function is to determine the number of integer bits necessary in the output `Y` of an FIR filter that computes the inner product between constant coefficient row vector `B` and state column vector `Z`. For example,

```
for k=1:length(X);  
    Z = [X(k);Z(1:end-1)];  
    Y(k) = B * Z;  
end
```

### Algorithms

In general, an inner product grows  $\log_2(n)$  bits for vectors of length  $n$ . However, in the case of this function the vector `a` is known and its values do not change. This knowledge is used to compute the smallest number of integer bits that are necessary in the output to guarantee that no overflow will occur.



The largest gain occurs when the vector  $\mathbf{b}$  has the same sign as the constant vector  $\mathbf{a}$ . Therefore, the largest gain due to the vector  $\mathbf{a}$  is  $\mathbf{a} * \text{sign}(\mathbf{a}')$ , which is equal to  $\text{sum}(\text{abs}(\mathbf{a}))$ .

The overall number of integer bits necessary to guarantee that no overflow occurs in the inner product is computed by:

$$n = \text{ceil}(\log_2(\text{sum}(\text{abs}(\mathbf{a})))) + \text{number of integer bits in } \mathbf{b} + 1 \text{ sign bit}$$

The extra sign bit is only added if both  $\mathbf{a}$  and  $\mathbf{b}$  are signed and  $\mathbf{b}$  attains its minimum. This prevents overflow in the event of  $(-1)*(-1)$ .

**Introduced before R2006a**

## int

Get stored integer value of a `fi` object

### Syntax

```
i = int(a)
```

### Description

`i = int(a)` returns the integer value of a `fi` object, stored in one of the built-in integer data types.

### Examples

#### Get the Stored Integer Value of a `fi` Object

Create a `fi` object with default settings. Use the `int` function to get its stored integer value. The output is an `int16` because the input used the default word length of 16-bits.

```
a = fi(pi);  
b = int(a)
```

```
b = int16  
    25736
```

Create a `fi` object that uses a 20-bit word length and get the stored integer value of the `fi` object.

```
a = fi(pi,1,20);  
b = int(a)
```

```
b = int32  
    411775
```

The output is an `int32` to accommodate the larger input word length.

---

## Input Arguments

### **a** — Fixed-point numeric object

scalar | vector | matrix | multidimensional array

Fixed-point numeric object from which you want to get the stored integer value. The word length of the input determines the data type of the output.

Data Types: `fi`

Complex Number Support: Yes

## Output Arguments

### **i** — Stored integer value

scalar | vector | matrix | multidimensional array

Stored integer value of the input `fi` object, returned as one of the built-in integer data types. The word length of the input determines the data type of the output. The output has the same dimensions as the input.

## See Also

### Functions

`bin` | `hex` | `oct` | `sdec` | `storedInteger`

**Introduced in R2006a**

## int8

Convert `fi` object to signed 8-bit integer

### Syntax

```
c = int8(a)
```

### Description

`c = int8(a)` returns the built-in `int8` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int8`.

### Examples

This example shows the `int8` values of a `fi` object.

```
a = fi([-pi 0.1 pi],1,8);  
c = int8(a)
```

```
c =
```

```
    -3     0     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int64](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint64](#) | [uint8](#)

**Introduced before R2006a**

## int16

Convert `fi` object to signed 16-bit integer

### Syntax

```
c = int16(a)
```

### Description

`c = int16(a)` returns the built-in `int16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int16`.

### Examples

This example shows the `int16` values of a `fi` object.

```
a = fi([-pi 0.1 pi],1,16);  
c = int16(a)
```

```
c =
```

```
    -3     0     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int32](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint64](#) | [uint8](#)

**Introduced before R2006a**

## int32

Convert `fi` object to signed 32-bit integer

### Syntax

```
c = int32(a)
```

### Description

`c = int32(a)` returns the built-in `int32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int32`.

### Examples

This example shows the `int32` values of a `fi` object.

```
a = fi([-pi 0.1 pi],1,32);  
c = int32(a)
```

```
c =
```

```
    -3     0     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint64](#) | [uint8](#)



**Introduced before R2006a**

## int64

Convert `fi` object to signed 64-bit integer

### Syntax

```
c = int64(a)
```

### Description

`c = int64(a)` returns the built-in `int64` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int64`.

### Examples

This example shows the `int64` values of a `fi` object.

```
a = fi([-pi 0.1 pi],1,64);  
c = int64(a)
```

```
c =
```

```
    -3     0     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint64](#) | [uint8](#)

**Introduced in R2008b**

## intmax

Largest positive stored integer value representable by `numericType` of `fi` object

### Syntax

```
x = intmax(a)
```

### Description

`x = intmax(a)` returns the largest positive stored integer value representable by the `numericType` of `a`.

### Examples

```
a = fi(pi, true, 16, 12);  
x = intmax(a)
```

```
x =
```

```
    32767
```

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

### See Also

`eps` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

**Introduced before R2006a**

## intmin

Smallest stored integer value representable by numeric type of `fi` object

### Syntax

```
x = intmin(a)
```

### Description

`x = intmin(a)` returns the smallest stored integer value representable by the numeric type of `a`.

### Examples

```
a = fi(pi, true, 16, 12);  
x = intmin(a)
```

```
x =
```

```
-32768
```

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

### See Also

`eps` | `intmax` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

**Introduced before R2006a**

## isboolean

Determine whether input is Boolean

### Syntax

```
y = isboolean(a)  
y = isboolean(T)
```

### Description

`y = isboolean(a)` returns 1 when the `DataType` property of `fi` object `a` is `boolean`, and 0 otherwise.

`y = isboolean(T)` returns 1 when the `DataType` property of `numericType` object `T` is `boolean`, and 0 otherwise.

### See Also

`isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

**Introduced in R2008a**

# isdouble

Determine whether input is double-precision data type

## Syntax

```
y = isdouble(a)  
y = isdouble(T)
```

## Description

`y = isdouble(a)` returns 1 when the `DataType` property of `fi` object `a` is `double`, and 0 otherwise.

`y = isdouble(T)` returns 1 when the `DataType` property of `numericType` object `T` is `double`, and 0 otherwise.

## See Also

`isboolean` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` |  
`isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` |  
`issingle`

**Introduced in R2008a**

## isequal

Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal

### Syntax

```
y = isequal(a,b,...)
y = isequal(F,G,...)
y = isequal(T,U,...)
y = isequal(q,r,...)
```

### Description

`y = isequal(a,b,...)` returns 1 if all the `fi` object inputs have the same real-world value. Otherwise, the function returns 0.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

`y = isequal(F,G,...)` returns 1 if all the `fimath` object inputs have the same properties. Otherwise, the function returns 0.

`y = isequal(T,U,...)` returns 1 if all the `numerictype` object inputs have the same properties. Otherwise, the function returns 0.

`y = isequal(q,r,...)` returns 1 if all the `quantizer` object inputs have the same properties. Otherwise, the function returns 0.

## Examples

### Compare Two `fi` Objects

Use the `isequal` function to determine if two `fi` objects have the same real-world value.



```
a = fi(pi);  
b = fi(pi, 1, 32);  
y = isequal(a, b)  
  
y = logical  
    0
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The `isequal` function returns `0` because the two `fi` objects do not have the same real-world value.

### Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the double is cast to the same word length and signedness of the `fi` object.

```
a = fi(pi);  
b = pi;  
y = isequal(a, b)  
  
y = logical  
    1
```

The `isequal` function casts **b** to the same word length as **a**, and returns `1`. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`eq` | `fi` | `fimath` | `ispropequal` | `numerictype` | `quantizer`

**Introduced before R2006a**

# isequivalent

Determine if two `numerictype` objects have equivalent properties

## Syntax

```
y = isequivalent (T1, T2)
```

## Description

`y = isequivalent (T1, T2)` determines whether the `numerictype` object inputs have equivalent properties and returns a logical 1 (`true`) or 0 (`false`). Two `numerictype` objects are equivalent if they describe the same data type.

## Examples

### Compare two `numerictype` objects

Use `isequivalent` to determine if two `numerictype` objects have the same data type.

```
T1 = numerictype(1, 16, 2^-12, 0)
```

```
T1 =
```

```
      DataTypeMode: Fixed-point: slope and bias scaling
      Signedness: Signed
      WordLength: 16
      Slope: 2^-12
      Bias: 0
```

```
T2 = numerictype(1, 16, 12)
```

```
T2 =
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 12
```

```
isequivalent(T1,T2)
```

```
ans = logical
      1
```

Although the Data Type Mode is different for T1 and T2, the function returns 1 (true) because the two objects have the same data type.

## Input Arguments

**T1, T2 — Inputs to be compared**

numeric type objects

Inputs to be compared, specified as numeric type objects.

## See Also

`eq` | `isequal` | `ispropequal`

**Introduced in R2014a**

## isfi

Determine whether variable is `fi` object

### Syntax

```
y = isfi(a)
```

### Description

`y = isfi(a)` returns 1 if `a` is a `fi` object, and 0 otherwise.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Avoid using the `isfi` function in code that you intend to convert using the automated workflow. The value returned by `isfi` in the fixed-point code might differ from the value returned in the original MATLAB algorithm. The behavior of the fixed-point code might differ from the behavior of the original algorithm.

### See Also

`fi` | `isfimath` | `isfipref` | `isnumericitype` | `isquantizer`

Introduced before R2006a

## **isfimath**

Determine whether variable is `fimath` object

### **Syntax**

```
y = isfimath(F)
```

### **Description**

`y = isfimath(F)` returns 1 if `F` is a `fimath` object, and 0 otherwise.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`fimath` | `isfi` | `isfipref` | `isnumericitype` | `isquantizer`

**Introduced before R2006a**

# isfimathlocal

Determine whether `fi` object has local `fimath`

## Syntax

```
y = isfimathlocal(a)
```

## Description

`y = isfimathlocal(a)` returns 1 if the `fi` object `a` has a local `fimath` object, and 0 if `a` does not have a local `fimath`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fimath` | `isfi` | `isfimathlocal` | `isfipref` | `isnumericitype` | `isquantizer` | `removefimath` | `sfi` | `ufi`

**Introduced in R2009b**

## **isfipref**

Determine whether input is `fipref` object

### **Syntax**

```
y = isfipref(P)
```

### **Description**

`y = isfipref(P)` returns 1 if `P` is a `fipref` object, and 0 otherwise.

### **See Also**

`fipref` | `isfi` | `isfimath` | `isnumericitype` | `isquantizer`

**Introduced in R2008a**



## isfixed

Determine whether input is fixed-point data type

### Syntax

```
y = isfixed(a)
y = isfixed(T)
y = isfixed(q)
```

### Description

`y = isfixed(a)` returns 1 when the `DataType` property of `fi` object `a` is `Fixed`, and 0 otherwise.

`y = isfixed(T)` returns 1 when the `DataType` property of `numericType` object `T` is `Fixed`, and 0 otherwise.

`y = isfixed(q)` returns 1 when `q` is a fixed-point quantizer, and 0 otherwise.

### See Also

`isboolean` | `isdouble` | `isfloat` | `isscaleddouble` | `isscaledtype` |  
`isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` |  
`issingle`

**Introduced in R2008a**

## isfloat

Determine whether input is floating-point data type

### Syntax

```
y = isfloat(a)  
y = isfloat(T)  
y = isfloat(q)
```

### Description

`y = isfloat(a)` returns 1 when the `DataType` property of `fi` object `a` is `single`, `double`, or `half`, and 0 otherwise.

`y = isfloat(T)` returns 1 when the `DataType` property of `numericType` object `T` is `single`, `double`, or `half`, and 0 otherwise.

`y = isfloat(q)` returns 1 when `q` is a floating-point quantizer, and 0 otherwise.

### See Also

`isboolean` | `isdouble` | `isfixed` | `isscaleddouble` | `isscaledtype` |  
`isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` |  
`issingle`

**Introduced in R2008a**

# isnumerictype

Determine whether input is numerictype object

## Syntax

```
y = isnumerictype(T)
```

## Description

`y = isnumerictype(T)` returns 1 if `T` is a numerictype object, and 0 otherwise.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`isfi` | `isfimath` | `isfipref` | `isquantizer` | `numerictype`

**Introduced before R2006a**

## **ispropequal**

Determine whether properties of two `fi` objects are equal

### **Syntax**

```
y = ispropequal(a,b,...)
```

### **Description**

`y = ispropequal(a,b,...)` returns 1 if all the inputs are `fi` objects and all the inputs have the same properties. Otherwise, the function returns 0.

To compare the real-world values of two `fi` objects `a` and `b`, use `a == b` or `isequal(a,b)`.

### **See Also**

`fi` | `isequal`

**Introduced before R2006a**

## isquantizer

Determine whether input is quantizer object

### Syntax

```
y = isquantizer(q)
```

### Description

`y = isquantizer(q)` returns 1 when `q` is a quantizer object, and 0 otherwise.

### See Also

`isfi` | `isfimath` | `isfipref` | `isnumericitype` | `quantizer`

**Introduced in R2008a**

## isscaleddouble

Determine whether input is scaled double data type

### Syntax

```
y = isscaleddouble(a)  
y = isscaleddouble(T)
```

### Description

`y = isscaleddouble(a)` returns 1 when the `DataType` property of `fi` object `a` is `ScaledDouble`, and 0 otherwise.

`y = isscaleddouble(T)` returns 1 when the `DataType` property of `numericType` object `T` is `ScaledDouble`, and 0 otherwise.

### See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaledtype` |  
`isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` |  
`issingle`

**Introduced in R2008a**

# isscaledtype

Determine whether input is fixed-point or scaled double data type

## Syntax

```
y = isscaledtype(a)  
y = isscaledtype(T)
```

## Description

`y = isscaledtype(a)` returns 1 when the `DataType` property of `fi` object `a` is `Fixed` or `ScaledDouble`, and 0 otherwise.

`y = isscaledtype(T)` returns 1 when the `DataType` property of `numericType` object `T` is `Fixed` or `ScaledDouble`, and 0 otherwise.

## See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` |  
`isscalingbinarypoint` | `isscaling_slopebias` | `isscaling_unspecified` |  
`issingle` | `numericType`

**Introduced in R2008a**

## isscalingbinarypoint

Determine whether input has binary point scaling

### Syntax

```
y = isscalingbinarypoint(a)  
y = isscalingbinarypoint(T)
```

### Description

`y = isscalingbinarypoint(a)` returns 1 when the `fi` object `a` has binary point scaling or trivial slope and bias scaling. Otherwise, the function returns 0. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`y = isscalingbinarypoint(T)` returns 1 when the `numericType` object `T` has binary point scaling or trivial slope and bias scaling. Otherwise, the function returns 0. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

### See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

**Introduced in R2010b**



# isscalingslopebias

Determine whether input has nontrivial slope and bias scaling

## Syntax

```
y = isscalingslopebias(a)  
y = isscalingslopebias(T)
```

## Description

`y = isscalingslopebias(a)` returns 1 when the `fi` object `a` has nontrivial slope and bias scaling, and 0 otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`y = isscalingslopebias(T)` returns 1 when the `numeric_type` object `T` has nontrivial slope and bias scaling, and 0 otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

## See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingunspecified` | `issingle`

**Introduced in R2010b**

## isscalingunspecified

Determine whether input has unspecified scaling

### Syntax

```
y = isscalingunspecified(a)  
y = isscalingunspecified(T)
```

### Description

`y = isscalingunspecified(a)` returns 1 if fi object `a` has a fixed-point or scaled double data type and its scaling has not been specified.

`y = isscalingunspecified(T)` returns 1 if `numericType` object `T` has a fixed-point or scaled double data type and its scaling has not been specified.

### See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `issingle`

**Introduced in R2010b**

## issigned

Determine whether `fi` object is signed

### Syntax

```
y = issigned(a)
```

### Description

`y = issigned(a)` returns 1 if the `fi` object `a` is signed, and 0 if it is unsigned.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**Introduced before R2006a**

## issingle

Determine whether input is single-precision data type

### Syntax

```
y = issingle(a)  
y = issingle(T)
```

### Description

`y = issingle(a)` returns 1 when the `DataType` property of `fi` object `a` is `single`, and 0 otherwise.

`y = issingle(T)` returns 1 when the `DataType` property of `numericType` object `T` is `single`, and 0 otherwise.

### See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified`

**Introduced in R2008a**

# isslopebiasscaled

Determine whether numeric type object has nontrivial slope and bias

## Syntax

```
y = isslopebiasscaled(T)
```

## Description

`y = isslopebiasscaled(T)` returns 1 when numeric type object `T` has nontrivial slope and bias scaling, and 0 otherwise. Slope and bias scaling is trivial when the slope is an integer power of 2, and the bias is 0.

## See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `issingle` | `numerictype`

**Introduced in R2008a**

## le

Determine whether real-world value of `fi` object is less than or equal to another

## Syntax

```
c = le(a,b)
a <= b
```

## Description

`c = le(a,b)` is called for the syntax `a <= b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a <= b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

## Examples

### Compare Two `fi` Objects

Use the `le` function to determine whether the real-world value of one `fi` object is less than or equal to another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a <= b
```

```
ans = logical
      0
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The `le` function returns `0` because after quantization, the value of **a** is greater than that of **b**.

### Compare a Double to a fi Object

When comparing a double to a `fi` object, the double is cast to the same word length and signedness of the `fi` object.

```
a = fi(pi);
b = pi;
a <= b
```

```
ans = logical
      1
```

The `le` function casts **b** to the same word length as **a**, and returns `1` because the two inputs have the same real-world value. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

## **See Also**

eq | ge | gt | lt | ne

**Introduced before R2006a**



# logreport

Quantization report

## Syntax

```
logreport(a)
logreport(a, b, ...)
```

## Description

`logreport(a)` displays the `minlog`, `maxlog`, `lowerbound`, `upperbound`, `noverflows`, and `nunderflows` for the `fi` object `a`.

`logreport(a, b, ...)` displays the report for each `fi` object `a`, `b`, ...

## Examples

The following example produces a `logreport` for `fi` objects `a` and `b`:

```
fipref('LoggingMode','On');
a = fi(pi);
b = fi(randn(10),1,8,7);
```

Warning: 27 overflows occurred in the `fi` assignment operation.

Warning: 1 underflow occurred in the `fi` assignment operation.

```
logreport(a,b)
      minlog      maxlog  lowerbound  upperbound  noverflows  nunderflows
a   3.141602   3.141602    -4         3.999878         0           0
b        -1    0.9921875    -1         0.9921875        27           1
```

## See Also

`fipref` | `quantize` | `quantizer`

**Introduced in R2008a**

## lowerbound

Lower bound of range of `fi` object

### Syntax

`lowerbound(a)`

### Description

`lowerbound(a)` returns the lower bound of the range of `fi` object `a`. If `L=lowerbound(a)` and `U=upperbound(a)`, then `[L,U]=range(a)`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`eps` | `intmax` | `intmin` | `lsb` | `range` | `realmax` | `realmin` | `upperbound`

**Introduced before R2006a**

## lsb

Scaling of least significant bit of `fi` object, or value of least significant bit of quantizer object

## Syntax

```
b = lsb(a)
p = lsb(q)
```

## Description

`b = lsb(a)` returns the scaling of the least significant bit of `fi` object `a`. The result is equivalent to the result given by the `eps` function.

`p = lsb(q)` returns the quantization level of quantizer object `q`, or the distance from `1.0` to the next largest floating-point number if `q` is a floating-point quantizer object.

## Examples

This example uses the `lsb` function to find the value of the least significant bit of the quantizer object `q`.

```
q = quantizer('fixed',[8 7]);
p = lsb(q)

p =

    0.0078
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports scalar fixed-point signals only.
- Code generation supports scalar, vector, and matrix, `fi` single and double signals.

### See Also

`eps` | `intmax` | `intmin` | `lowerbound` | `quantize` | `range` | `realmax` | `realmin` | `upperbound`

**Introduced before R2006a**

## lt

Determine whether real-world value of one `fi` object is less than another

## Syntax

```
c = lt(a,b)
a < b
```

## Description

`c = lt(a,b)` is called for the syntax `a < b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a < b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

## Examples

### Compare Two `fi` Objects

Use the `lt` function to determine whether the real-world value of one `fi` object is less than another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a < b
```

```
ans = logical
      0
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The `lt` function returns `0` because after quantization, the value of **a** is greater than that of **b**.

### Compare a Double to a fi Object

When comparing a double to a `fi` object, the double is cast to the same word length and signedness of the `fi` object.

```
a = fi(pi);
b = pi;
a < b
```

```
ans = logical
      0
```

The `lt` function casts **b** to the same word length as **a**, and returns `0` because the two inputs have the same real-world value. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

## **See Also**

eq | ge | gt | le | ne

**Introduced before R2006a**

## mat2str

Convert matrix to string

### Syntax

```
str = mat2str(A)
str = mat2str(A, n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

### Description

`str = mat2str(A)` converts `fi` object `A` to a string representation. The output is suitable for input to the `eval` function such that `eval(str)` produces the original `fi` object exactly.

`str = mat2str(A, n)` converts `fi` object `A` to a string representation using `n` bits of precision.

`str = mat2str(A, 'class')` creates a string representation with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` bits of precision and includes the class of `A`.

### Examples

#### Convert `fi` Object to a String

Convert the `fi` object `a` to a string.

```
a = fi(pi);
str = mat2str(a)
```



```
str =  
'3.1416015625'
```

### Convert fi Object to a String with Specified Precision

Convert the `fi` object `a` to a string using eight bits of precision.

```
a = fi(pi);  
str = mat2str(a, 8)
```

```
str =  
'3.1416016'
```

## Input Arguments

### A — Input array

scalar | vector | matrix

Input array, specified as a scalar, vector, or matrix. A cannot be a multidimensional array.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### n — Number of bits of precision

positive integer

Number of bits of precision in the output, specified as a positive integer.

**Data Types:** `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### str — String representation of input array

character array

String representation of input array, returned as a character array.

## **See Also**

`mat2str` | `toString`

**Introduced in R2015b**

## max

Largest element in array of `fi` objects

### Syntax

```
x = max(a)
x = max(a, [], dim)
[x, y] = max( ___ )
```

```
m = max(a, b)
```

### Description

`x = max(a)` returns the largest elements along different dimensions of `fi` array `a`.

If `a` is a vector, `max(a)` returns the largest element in `a`.

If `a` is a matrix, `max(a)` treats the columns of `a` as vectors, returning a row vector containing the maximum element from each column.

If `a` is a multidimensional array, `max` operates along the first nonsingleton dimension and returns an array of maximum values.

`x = max(a, [], dim)` returns the largest elements along dimension `dim`.

`[x, y] = max( ___ )` finds the indices of the maximum values and returns them in array `y`, using any of the input arguments in the previous syntaxes. If the largest value occurs multiple times, the index of the first occurrence is returned.

`m = max(a, b)` returns an array the same size as `a` and `b` with the largest elements taken from `a` or `b`.

### Examples

**Largest Element in a Vector**

Create a fixed-point vector, and return the maximum value from the vector.

```
a = fi([1,5,4,9,2],1,16);  
x = max(a)
```

```
x =  
    9
```

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

**Largest Element of Each Matrix Row**

Create a fixed-point matrix.

```
a = fi(magic(4),1,16)
```

```
a=4x4 object  
    16     2     3    13  
     5    11    10     8  
     9     7     6    12  
     4    14    15     1
```

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

Find the largest element of each row by finding the maximum values along the second dimension.

```
x = max(a,[],2)
```

```
x=4x1 object  
    16  
    11  
    12  
    15
```

```

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

```

The output vector,  $x$ , is a column vector that contains the largest element of each row.

### Largest Element of Each Matrix Column

Create a fixed-point matrix.

```
a = fi(magic(4),1,16)
```

```
a=4x4 object
```

```

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

```

```

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

```

Find the largest element of each column.

```
x = max(a)
```

```
x=1x4 object
```

```

    16    14    15    13

```

```

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

```

The output,  $x$ , is a row vector that contains the largest elements from each column of  $a$ .

Find the index of each of the maximum elements.

```
[x,y] = max(a)
```

```
x=1×4 object
```

```
16 14 15 13
```

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

```
y = 1×4
```

```
1 4 4 1
```

Vector *y* contains the indices to the minimum elements in *x*.

### Maximum Elements from Two Arrays

Create two fixed-point arrays of the same size.

```
a = fi([2.3,4.7,6;0,7,9.23],1,16);  
b = fi([9.8,3.21,1.6;pi,2.3,1],1,16);
```

Find the largest elements from *a* or *b*.

```
m = max(a,b)
```

```
m=2×3 object
```

```
9.7998 4.7002 6.0000  
3.1416 7.0000 9.2300
```

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

*m* contains the largest elements from each pair of corresponding elements in *a* and *b*.

## Largest Element of a Complex Vector

Create a complex fixed-point vector, **a**.

```
a = fi([1+2i,3+6i,6+3i,2-4i],1,16)
```

```
a=1x4 object
  1.0000 + 2.0000i   3.0000 + 6.0000i   6.0000 + 3.0000i   2.0000 - 4.0000i

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 12
```

The function finds the largest element of a complex vector by taking the element with the largest magnitude.

```
abs(a)
```

```
ans=1x4 object
  2.2361   6.7083   6.7083   4.4722

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 12
```

In vector **a**, the largest elements, at position 2 and 3, have a magnitude of 6.7083. The **max** function returns the largest element in output **x** and the index of that element in output **y**.

```
[x,y] = max(a)
```

```
x =
  3.0000 + 6.0000i

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 12
```

```
y = 2
```

Although the elements at index 2 and 3 have the same magnitude, the index of the first occurrence of that value is always returned.

## Input Arguments

### **a** — Input **fi** array

*fi* object | numeric variable

*fi* input array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of *a* and *b* must match unless one is a scalar.

The `max` function ignores NaNs.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Complex Number Support:** Yes

### **b** — Second input **fi** array

*fi* object | numeric variable

Second *fi* input array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of *a* and *b* must match unless one is a scalar.

The `max` function ignores NaNs.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Complex Number Support:** Yes

### **dim** — dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. *dim* can also be a *fi* object. If you do not specify a value, the default value is the first array dimension whose size does not equal 1.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



## Output Arguments

### **x — Maximum values**

scalar | vector | matrix | multidimensional array

Maximum values, returned as a scalar, vector, matrix, or multidimensional array. **x** always has the same data type as the input.

### **y — Index of maximum values**

scalar | vector | matrix | multidimensional array

Indices of the maximum values in array **x**, returned as a scalar, vector, matrix, or multidimensional array. If the largest value occurs more than once, then **y** contains the index to the first occurrence of the value. **y** is always of data type `double`.

### **m — Array of maximum values**

scalar | vector | matrix | multidimensional array

Array of maximum values of **a** and **b**, returned as a scalar, vector, matrix, or multidimensional array.

## Algorithms

When **a** or **b** is complex, the `max` function returns the elements with the largest magnitude. If two magnitudes are equal, then `max` returns the first value. This behavior differs from how the builtin `max` function resolves ties between complex numbers.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`mean` | `median` | `min` | `sort`

**Introduced before R2006a**

# maxlog

Log maximums

## Syntax

```
y = maxlog(a)
y = maxlog(q)
```

## Description

`y = maxlog(a)` returns the largest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = maxlog(q)` is the maximum value after quantization during a call to `quantize(q, ...)` for quantizer object `q`. This value is the maximum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `maxlog(q)` is equivalent to `get(q, 'maxlog')` and `q.maxlog`.

## Examples

### Example 1: Using maxlog with fi objects

```
P = fipref('LoggingMode','on');
format long g
a = fi([-1.5 eps 0.5], true, 16, 15);
a(1) = 3.0;
maxlog(a)
```

```
Warning: 1 overflow occurred in the fi
assignment operation.
> In embedded.fi.fi at 510
```

```
In fi at 220
```

```
Warning: 1 underflow occurred in the fi
assignment operation.
```

```
> In embedded.fi.fi at 510
  In fi at 220
```

```
Warning: 1 overflow occurred in the fi
assignment operation.
```

```
ans =
```

```
0.999969482421875
```

The largest value `maxlog` can return is the maximum representable value of its input. In this example, `a` is a signed `fi` object with word length 16, fraction length 15 and range:

$$-1 \leq x \leq 1 - 2^{-15}$$

You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)
```

```
r =
```

```
-1 0.999969482421875
```

## Example 2: Using `maxlog` with quantizer objects

```
q = quantizer;
warning on
format long g
x = [-20:10];
y = quantize(q,x);
maxlog(q)
```

```
Warning: 29 overflows.
```

```
> In embedded.quantizer.quantize at 74
```

```
ans =
```

```
.999969482421875
```

The largest value `maxlog` can return is the maximum representable value of its input. You can obtain the range of `x` after quantization using the `range` function:

```
format long g  
r = range(q)
```

```
r =
```

```
          -1          0.999969482421875
```

## See Also

`fipref` | `minlog` | `noverflows` | `nunderflows` | `reset` | `resetlog`

**Introduced before R2006a**

## mean

Average or mean value of fixed-point array

### Syntax

```
c = mean(a)  
c = mean(a,dim)
```

### Description

*c* = mean(*a*) computes the mean value of the fixed-point array *a* along its first nonsingleton dimension.

*c* = mean(*a*,*dim*) computes the mean value of the fixed-point array *a* along dimension *dim*. *dim* must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The input to the mean function must be a real-valued fixed-point array.

The fixed-point output array *c* has the same `numericType` properties as the fixed-point input array *a*. If the input, *a*, has a local `fimath`, then it is used for intermediate calculations. The output, *c*, is always associated with the default `fimath`.

When *a* is an empty fixed-point array (value = []), the value of the output array is zero.

### Examples

Compute the mean value along the first dimension (rows) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);  
% x is a signed FI object with a 32-bit word length  
% and a best-precision fraction length of 28-bits  
mx1 = mean(x,1)
```

Compute the mean value along the second dimension (columns) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);  
% x is a signed fi object with a 32-bit word length  
% and a best-precision fraction length of 28 bits  
mx2 = mean(x,2)
```

## Algorithms

The general equation for computing the mean of an array  $a$ , across dimension  $dim$  is:

```
sum(a,dim)/size(a,dim)
```

Because `size(a,dim)` is always a positive integer, the algorithm casts `size(a,dim)` to an unsigned 32-bit `fi` object with a fraction length of zero (`SizeA`). The algorithm then computes the mean of  $a$  according to the following equation, where  $T_x$  represents the numeric type properties of the fixed-point input array  $a$ :

```
c = Tx.divide(sum(a,dim), SizeA)
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`max` | `median` | `min`

**Introduced in R2010a**

## median

Median value of fixed-point array

### Syntax

```
c = median(a)
c = median(a,dim)
```

### Description

`c = median(a)` computes the median value of the fixed-point array *a* along its first nonsingleton dimension.

`c = median(a,dim)` computes the median value of the fixed-point array *a* along dimension *dim*. *dim* must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The input to the `median` function must be a real-valued fixed-point array.

The fixed-point output array *c* has the same `numericType` properties as the fixed-point input array *a*. If the input, *a*, has a local `fimath`, then it is used for intermediate calculations. The output, *c*, is always associated with the default `fimath`.

When *a* is an empty fixed-point array (value = []), the value of the output array is zero.

### Examples

Compute the median value along the first dimension of a fixed-point array.

```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx1 = median(x,1)
```

Compute the median value along the second dimension (columns) of a fixed-point array.



```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx2 = median(x, 2)
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[max](#) | [mean](#) | [min](#)

**Introduced in R2010a**

## min

Smallest element in array of `fi` objects

### Syntax

```
x = min(a)
x = min(a,[],dim)
[x,y] = min( ___ )
m = min(a,b)
```

### Description

`x = min(a)` returns the smallest elements along different dimensions of `fi` array `a`.

If `a` is a vector, `min(a)` returns the smallest element in `a`.

If `a` is a matrix, `min(a)` treats the columns of `a` as vectors, returning a row vector containing the minimum element from each column.

If `a` is a multidimensional array, `min` operates along the first nonsingleton dimension and returns an array of minimum values.

`x = min(a,[],dim)` returns the smallest elements along dimension `dim`.

`[x,y] = min( ___ )` finds the indices of the minimum values and returns them in array `y`, using any of the input arguments in the previous syntaxes. If the smallest value occurs multiple times, the index of the first occurrence is returned.

`m = min(a,b)` returns an array the same size as `a` and `b` with the smallest elements taken from `a` or `b`.

### Examples

### Smallest Element in a Vector

Create a fixed-point vector, and return the minimum value from the vector.

```
a = fi([1,5,4,9,2],1,16);
x = min(a)
```

```
x =
```

```
1
```

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

### Minimum Element of Each Matrix Row

Create a matrix of fixed-point values.

```
a = fi(magic(4),1,16)
```

```
a=4x4 object
```

```
16     2     3    13
 5    11    10     8
 9     7     6    12
 4    14    15     1
```

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

Find the smallest element of each row by finding the minimum values along the second dimension.

```
x = min(a,[],2)
```

```
x=4x1 object
```

```
2
5
6
1
```

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

The output, `x`, is a column vector that contains the smallest element of each row of `a`.

### Minimum Element of Each Matrix Column

Create a fixed-point matrix.

```
a = fi(magic(4),1,16)
```

```
a=4x4 object
```

```
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

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

Find the smallest element of each column.

```
x = min(a)
```

```
x=1x4 object
```

```
     4     2     3     1
```

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

The output, `x`, is a row vector that contains the smallest element of each column of `a`.

Find the index of each of the minimum elements.

```
[x,y] = min(a)
```

```
x=1×4 object
```

```
4      2      3      1
```

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

```
y = 1×4
```

```
4      1      1      4
```

### Minimum Elements from Two Arrays

Create two fixed-point arrays of the same size.

```
a = fi([2.3,4.7,6;0,7,9.23],1,16);
b = fi([9.8,3.21,1.6;pi,2.3,1],1,16);
```

Find the minimum elements from a or b.

```
m = min(a,b)
```

```
m=2×3 object
```

```
2.2998    3.2100    1.6001
0         2.2998    1.0000
```

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

m contains the smallest elements from each pair of corresponding elements in a and b.

### Minimum Element of a Complex Vector

Create a complex fixed-point vector, **a**.

```
a = fi([1+2i,2+i,3+8i,9+i],1,8)
```

```
a=1x4 object
  1.0000 + 2.0000i   2.0000 + 1.0000i   3.0000 + 8.0000i   9.0000 + 1.0000i

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

The function finds the smallest element of a complex vector by taking the element with the smallest magnitude.

```
abs(a)
```

```
ans=1x4 object
  2.2500   2.2500   8.5000   9.0000

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

In vector **a**, the smallest elements, at position 1 and 2, have a magnitude of 2.25. The `min` function returns the smallest element in output **x**, and the index of that element in output, **y**.

```
[x,y] = min(a)
```

```
x =
  1.0000 + 2.0000i

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

```
y = 1
```

Although the elements at index 1 and 2 have the same magnitude, the index of the first occurrence of that value is always returned.

## Input Arguments

### **a** – Input **fi** array

*fi* object | numeric variable

*fi* input array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of *a* and *b* must match unless one is a scalar.

The `min` function ignores NaNs.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Complex Number Support:** Yes

### **b** – Second input **fi** array

*fi* object | numeric variable

Second *fi* input array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of *a* and *b* must match unless one is a scalar.

The `min` function ignores NaNs.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Complex Number Support:** Yes

### **dim** – dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. *dim* can also be a *fi* object. If you do not specify a value, the default value is the first array dimension whose size does not equal 1.

**Data Types:** *fi*|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

### **x — Minimum values**

scalar | vector | matrix | multidimensional array

Minimum values, returned as a scalar, vector, matrix, or multidimensional array. **x** always has the same data type as the input.

### **y — Index of minimum values**

scalar | vector | matrix | multidimensional array

Indices of the minimum values in array **x**, returned as a scalar, vector, matrix, or multidimensional array. If the smallest value occurs more than once, then **y** contains the index to the first occurrence of the value. **y** is always of data type `double`.

### **m — Array of minimum values**

scalar | vector | matrix | multidimensional array

Array of minimum values of **a** and **b**, returned as a scalar, vector, matrix, or multidimensional array.

## Algorithms

When **a** or **b** is complex, the `min` function returns the element with the smallest magnitude. If two magnitudes are equal, then `min` returns the first value. This behavior differs from how the builtin `min` function resolves ties between complex numbers.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`max` | `mean` | `median` | `sort`



**Introduced before R2006a**

## minlog

Log minimums

### Syntax

```
y = minlog(a)
y = minlog(q)
```

### Description

`y = minlog(a)` returns the smallest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = minlog(q)` is the minimum value after quantization during a call to `quantize(q, ...)` for quantizer object `q`. This value is the minimum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `minlog(q)` is equivalent to `get(q, 'minlog')` and `q.minlog`.

### Examples

#### Example 1: Using minlog with fi objects

```
P = fipref('LoggingMode','on');
a = fi([-1.5 eps 0.5], true, 16, 15);
a(1) = 3.0;
minlog(a)
```

```
ans =
```

```
-1
```

The smallest value `minlog` can return is the minimum representable value of its input. In this example, `a` is a signed `fi` object with word length 16, fraction length 15 and range:

$$-1 \leq x \leq 1 - 2^{-15}$$

You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)

r =

           -1           0.999969482421875
```

## Example 2: Using `minlog` with quantizer objects

```
q = quantizer;
warning on
x = [-20:10];
y = quantize(q,x);
minlog(q)

Warning: 29 overflows.
> In embedded.quantizer.quantize at 74

ans =

           -1
```

The smallest value `minlog` can return is the minimum representable value of its input. You can obtain the range of `x` after quantization using the `range` function:

```
format long g
r = range(q)

r =

           -1           0.999969482421875
```

## See Also

`fipref` | `maxlog` | `noverflows` | `nunderflows` | `reset` | `resetlog`

**Introduced before R2006a**

# minus

Matrix difference between `fi` objects

## Syntax

`minus(a,b)`

## Description

`minus(a,b)` is called for the syntax `a - b` when `a` or `b` is an object.

`a - b` subtracts matrix `b` from matrix `a`. `a` and `b` must have the same dimensions unless one is a scalar value (a 1-by-1 matrix). A scalar value can be subtracted from any other value.

`minus` does not support `fi` objects of data type `Boolean`.

---

**Note** For information about the `fimath` properties involved in Fixed-Point Designer calculations, see “`fimath` Properties Usage for Fixed-Point Arithmetic” and “`fimath` ProductMode and SumMode” in the Fixed-Point Designer User's Guide.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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.

### **See Also**

`mtimes` | `plus` | `times` | `uminus`

**Introduced before R2006a**

## mod

Modulus after division for `fi` objects

### Syntax

$M = \text{mod}(X, Y)$

### Description

$M = \text{mod}(X, Y)$  if  $Y \neq 0$ , returns  $X - n * Y$ , where  $n = \text{floor}(X./Y)$ . The inputs  $X$  and  $Y$  must be real arrays of the same size, or either can be a real scalar. For fixed-point or integer input arguments, the output data type is the aggregate type of both input signedness, word lengths, and fraction lengths. For floating-point input arguments, the output data type is the same as the inputs.

The `mod` function ignores and discards any `fimath` attached to the inputs. The output is always associated with the default `fimath`.

---

**Note** The combination of fixed-point and floating-point inputs is not currently supported.

---

### Input Arguments

**X**

Integer, fixed-point, or floating-point array, or real scalar.

**Y**

Array of the same size as  $X$ , or real scalar.

## Output Arguments

### M

Result of modulus operation. If both inputs *X* and *Y* are floating-point, then the data type of *M* is the same as the inputs. If either input *X* or *Y* is fixed-point, then the data type of *M* is the aggregate numerictype. This value equals that of `fixed.aggregateType(X,Y)`.

## Examples

Calculate the mod of two `fi` objects.

```
% 7-bit signed fixed-point object
x = fi(-3,1,7,0);
% 15-bit signed fixed-point object
y = fi(2,1,15,0);
M1 = mod(x,y)
M1 =

     1

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 15
      FractionLength: 0
M2 = mod(y,x)
M2 =

    -1

      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 15
      FractionLength: 0
```

Convert the `fi` inputs in the previous example to double type, and calculate the mod.

```
Mf1 = mod(double(x),double(y))
Mf1 =

     1
Mf2 = mod(double(y),double(x))
```



Mf2 =

-1

## **See Also**

`fixed.aggregateType | mod`

**Introduced in R2011b**

## mpower

Fixed-point matrix power (^)

### Syntax

```
c = mpower(a,k)  
c = a^k
```

### Description

$c = \text{mpower}(a, k)$  and  $c = a^k$  compute matrix power. The exponent  $k$  requires a positive, real-valued integer value.

The fixed-point output array  $c$  has the same local fimath as the input  $a$ . If  $a$  has no local fimath, the output  $c$  also has no local fimath. The matrix power operation is performed using default fimath settings.

### Examples

Compute the power of a 2-dimensional square matrix for exponent values 0, 1, 2, and 3.

```
x = fi([0 1; 2 4], 1, 32);
```

```
px0 = x^0  
px1 = x^1  
px2 = x^2  
px3 = x^3
```

### Tips

For more information about the `mpower` function, see the MATLAB `mpower` reference page.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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, MATLAB computes the output using the `ProductMode` of the governing `fimath`.

### See Also

mpower | power

**Introduced in R2010a**

## mpy

Multiply two objects using `fimath` object

### Syntax

```
c = mpy(F,a,b)
```

### Description

`c = mpy(F,a,b)` performs elementwise multiplication on `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` has no local `fimath`.

`a` and `b` can both be `fi` objects with the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object. `a` and `b` can also be doubles, singles, or integers.

### Examples

In this example, `c` is the 40-bit product of `a` and `b` with fraction length 30.

```
a = fi(pi);  
b = fi(exp(1));  
F = fimath('ProductMode','SpecifyPrecision',...  
          'ProductWordLength',40,'ProductFractionLength',30);  
c = mpy(F, a, b)
```

```
c =
```

```
8.5397
```

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

```

    WordLength: 40
    FractionLength: 30

```

## Algorithms

`c = mpy(F, a, b)` is similar to

```

a.fimath = F;
b.fimath = F;
c = a .* b

```

```

c =
    8.5397

```

```

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

```

```

    RoundingMethod: nearest
    OverflowAction: saturate
    ProductMode: SpecifyPrecision
    ProductWordLength: 40
    ProductFractionLength: 30
    SumMode: FullPrecision

```

but not identical. When you use `mpy`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` has no local `fimath`. When you use the syntax `c = a .* b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See “`fimath` Rules for Fixed-Point Arithmetic” in the Fixed-Point Designer User's Guide for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax `F.mpy(a,b)`. You must use the syntax `mpy(F,a,b)`.
- 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 `0n`.

### See Also

`add` | `divide` | `fi` | `fimath` | `mrdivide` | `numerictype` | `rdivide` | `sub` | `sum`

**Introduced before R2006a**

# mrdivide, /

**Package:** embedded

Right-matrix division

## Syntax

```
X = A/b
X = mrdivide(A, b)
```

## Description

$X = A/b$  performs right-matrix division.

$X = \text{mrdivide}(A, b)$  is an alternative way to execute  $X = A/b$ .

## Examples

### Divide fi Matrix by a Constant

In this example, you use the forward slash (/) operator to perform right matrix division on a 3-by-3 magic square of `fi` objects. Because the numerator input is a `fi` object, the denominator input `b` must be a scalar.

```
A = fi(magic(3))
```

```
A=3x3 object
```

```
  8     1     6
  3     5     7
  4     9     2
```

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

FractionLength: 11

**b = fi(3,1,12,8)**

b =  
3

DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 12  
FractionLength: 8

**X = A/b**

X=3×3 *object*

2.6250	0.3750	2.0000
1.0000	1.6250	2.3750
1.3750	3.0000	0.6250

DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 3

### Perform Matrix Division

You can perform right-matrix division when neither input is a `fi` object. The matrix dimensions must be compatible for matrix division.

**A = [2, 3, 1; 0, 8, 4; 1, 1, 0]**

A = 3×3

2	3	1
0	8	4
1	1	0

**B = [7, 6, 6; 1, 0, 5; 9, 0, 4]**

B = 3×3



```

7     6     6
1     0     5
9     0     4

```

```
X = mrdivide(A,B)
```

```
X = 3×3
```

```

0.5000    -0.2927    -0.1341
1.3333     0.0325    -1.0407
0.1667    -0.2033     0.0041

```

## Input Arguments

### A — Numerator

scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array. If one or both of the inputs is a `fi` object, then `b` must be a scalar. When `b` is a scalar, `mrdivide` is equivalent to `rdivide`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`  
 Complex Number Support: Yes

### b — Denominator

scalar | vector | matrix | multidimensional array

Denominator, specified as a real scalar, vector, matrix, or multidimensional array. If one or both of the inputs is a `fi` object, then `b` must be a scalar. When `b` is a scalar, `mrdivide` is equivalent to `rdivide`.

If neither input is a `fi` object, then the sizes of the input matrices must be compatible for matrix division.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

## Output Arguments

### **X — Quotient**

scalar | vector | matrix | multidimensional array

Solution, returned as an array with the same dimensions as the numerator input **A**. When **A** is complex, the real and imaginary parts of **A** are independently divided by **b**.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

[add](#) | [divide](#) | [fi](#) | [fimath](#) | [numerictype](#) | [rdivide](#) | [sub](#) | [sum](#)

**Introduced in R2009a**

## mtimes

Matrix product of `fi` objects

### Syntax

`mtimes(a,b)`

### Description

`mtimes(a,b)` is called for the syntax `a * b` when `a` or `b` is an object.

`a * b` is the matrix product of `a` and `b`. A scalar value (a 1-by-1 matrix) can multiply any other value. Otherwise, the number of columns of `a` must equal the number of rows of `b`.

`mtimes` does not support `fi` objects of data type `Boolean`.

---

**Note** For information about the `fimath` properties involved in Fixed-Point Designer calculations, see “`fimath` Properties Usage for Fixed-Point Arithmetic” and “`fimath` ProductMode and SumMode” in the Fixed-Point Designer documentation.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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`.

### See Also

`minus` | `plus` | `times` | `uminus`

**Introduced before R2006a**

## ne

Determine whether real-world values of two `fi` objects are not equal

## Syntax

```
c = ne(a,b)
a ~= b
```

## Description

`c = ne(a,b)` is called for the syntax `a ~= b` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a ~= b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to 1 where the relation is true, and 0 where the relation is false.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

## Examples

### Compare Two `fi` Objects

Use the `ne` function to determine whether the real-world values of two `fi` objects are not equal.

```
a = fi(pi);
b = fi(pi, 1, 32);
a ~= b
```

```
ans = logical
      1
```

Input **a** has a 16-bit word length, while input **b** has a 32-bit word length. The `ne` function returns 1 because after quantization, the value of **a** is greater than that of **b**.

### Compare a Double to a fi Object

When comparing a double to a `fi` object, the double is cast to the same word length and signedness of the `fi` object.

```
a = fi(pi);
b = pi;
a ~= b
```

```
ans = logical
      0
```

The `ne` function casts **b** to the same word length as **a**, and returns 0 because the two inputs have the same real-world value. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

## **See Also**

eq | ge | gt | le | lt

**Introduced before R2006a**

## nearest

Round toward nearest integer with ties rounding toward positive infinity

### Syntax

```
y = nearest(a)
```

### Description

`y = nearest(a)` rounds `fi` object `a` to the nearest integer or, in case of a tie, to the nearest integer in the direction of positive infinity, and returns the result in `fi` object `y`.

`y` and `a` have the same `fi` object and `DataType` property.

When the `DataType` property of `a` is `Single`, `Double`, or `Boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`nearest` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.



## Examples

### Example 1

The following example demonstrates how the `nearest` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

```
y = nearest(a)
```

```
y =
```

```
3
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 6
      FractionLength: 0
```

### Example 2

The following example demonstrates how the `nearest` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a =
```

```
0.0249
```

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

```

        FractionLength: 12
y = nearest(a)
y =
    0
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 2
        FractionLength: 0
    
```

### Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>convergent(a)</b>	<b>nearest(a)</b>	<b>round(a)</b>
-3.5	-4	-3	-4
-2.5	-2	-2	-3
-1.5	-2	-1	-2
-0.5	0	0	-1
0.5	0	1	1
1.5	2	2	2
2.5	2	3	3
3.5	4	4	4

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`ceil` | `convergent` | `fix` | `floor` | `round`

**Introduced in R2008a**

## noperations

Number of operations

### Syntax

noperations(q)

### Description

noperations(q) is the number of quantization operations during a call to `quantize(q, ...)` for quantizer object `q`. This value accumulates over successive calls to `quantize`. You reset the value of `noperations` to zero by issuing the command `resetlog(q)`.

Each time any data element is quantized, `noperations` is incremented by one. The real and complex parts are counted separately. For example, `(complex * complex)` counts four quantization operations for products and two for sum, because  $(a+bi)(c+di) = (a*c - b*d) + (a*d + b*c)$ . In contrast, `(real*real)` counts one quantization operation.

In addition, the real and complex parts of the inputs are quantized individually. As a result, for a complex input of length 204 elements, `noperations` counts 408 quantizations: 204 for the real part of the input and 204 for the complex part.

If any inputs, states, or coefficients are complex-valued, they are all expanded from real values to complex values, with a corresponding increase in the number of quantization operations recorded by `noperations`. In concrete terms, `(real*real)` requires fewer quantizations than `(real*complex)` and `(complex*complex)`. Changing all the values to complex because one is complex, such as the coefficient, makes the `(real*real)` into `(real*complex)`, raising `noperations` count.

### See Also

`maxlog` | `minlog`

**Introduced before R2006a**

## noverflows

Number of overflows

### Syntax

```
y = noverflows(a)  
y = noverflows(q)
```

### Description

`y = noverflows(a)` returns the number of overflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fi` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = noverflows(q)` returns the accumulated number of overflows resulting from quantization operations performed by a `quantizer` object `q`.

### See Also

`maxlog` | `minlog` | `nunderflows` | `resetlog`

**Introduced before R2006a**

## nts

Determine fixed-point data type

### Syntax

```
nts  
nts({'block',PORT})  
nts({line-handle})  
nts({gsl})
```

### Description

`nts` opens the `NumericTypeScope` window. To connect to a signal in a Simulink model, select the signal and then, in the `NumericTypeScope` window, select **File > Connect to Simulink Signal**.

The `NumericTypeScope` suggests a fixed-point data type in the form of a `numericType` object based on the dynamic range of the input data and the criteria that you specify in the “Bit Allocation Panel” on page 5-618. The scope allows you to visualize the dynamic range of data in the form of a  $\log_2$  histogram. It displays the data values on the X-axis and the number or percentage of occurrences on the Y-axis. Each bin in the histogram corresponds to a bit in a word. For example,  $2^0$  corresponds to the first integer bit in the binary word,  $2^{-1}$  corresponds to the first fractional bit in the binary word.

`nts({'block',PORT})` opens the `NumericTypeScope` window and connects the scope to the signal output from `block` on output port with index `PORT`. If the block has more than one output port, you must specify the port index. The scope cannot connect to more than one output port.

`nts({line-handle})` opens the `NumericTypeScope` window and connects the scope to the Simulink signal which has the line handle specified in `line-handle`.

`nts({gsl})` opens the `NumericTypeScope` window and connects the scope to the currently selected Simulink signal. You must select a signal in a Simulink model first, otherwise the scope opens with no signal selected.

## Input Arguments

### **block**

Full path to the specified block.

### **line-handle**

Handle of the Simulink signal that you want to view with the scope. To get the handle of the currently selected signal, at the MATLAB command line, enter `gsl`.

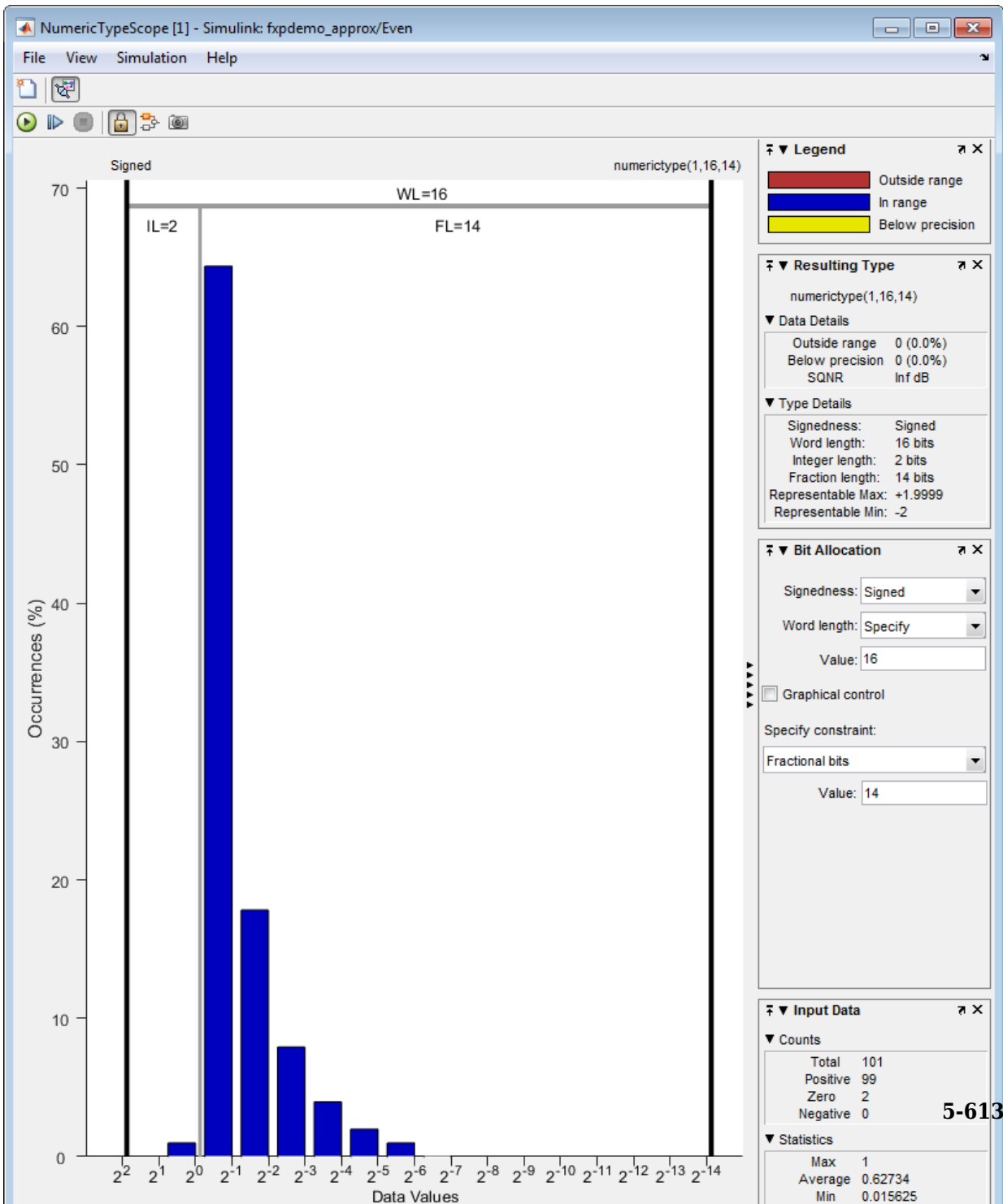
### **PORT**

Index of the output port that you want to view with the scope. If the block has more than one output port, you must specify the index. The scope cannot connect to more than one output port.

## The NumericTypeScope Window



The `NumericTypeScope` opens with the default toolbars on page 5-614 displayed at the top of the window and the dialog panels on page 5-618 to the right.











## Toolbars

By default the scope displays a toolbar that provides these options:

Button	Action
	New NumericTypeScope.
	Connect to Simulink signal. The scope connects to the currently selected signal. If a block with only one output port is selected and the <b>Connect scope on selection of</b> is set to <b>Signal lines or blocks</b> , connects to the output port of the selected block. For more information, see “Sources Pane” on page 5-617.

After connecting the scope to a signal in a Simulink model, the scope displays an additional toolbar with the following options:

Button	Action
	Stop simulation
	Start simulation
	Simulate one step
	Snapshot. Freezes the display so that you can examine the results. To reenale display refreshing, click the button again.
	Highlight Simulink signal.
	Persistent. By default, the scope makes a persistent connection to the selected signal. If you want to view different signals during the simulation, click this button to make a floating connection. You can then select any signal in the model and the scope displays it.

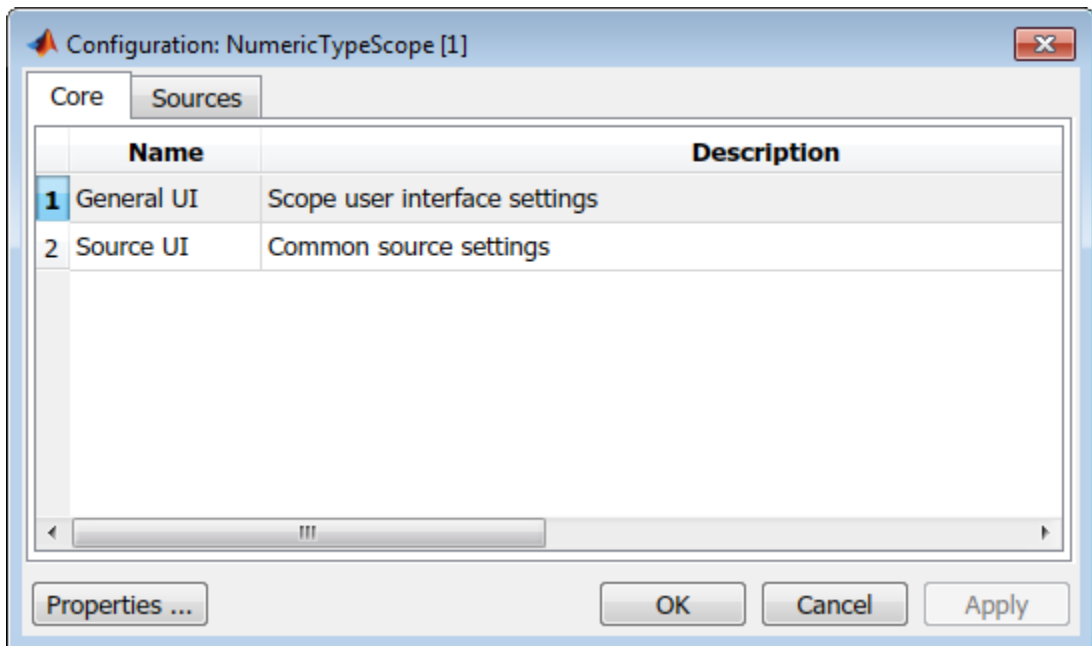
## Dialog Boxes and Panels

- “Configuration Dialog Box” on page 5-615
- “Dialog Panels” on page 5-618

## Configuration Dialog Box

Use the NumericTypeScope configuration dialog box to control the behavior and appearance of the scope window.

To open the **Configuration** dialog box, from the scope main menu, select **File > Configuration > Edit**, or, with the scope as your active window, press the **N** key.



For information about each pane, see “Core Pane” on page 5-616 and “Sources Pane” on page 5-617.

To save configuration settings for future use, select **File > Configuration > Save as**. The configuration settings you save become the default configuration settings for the NumericTypeScope.

---

**Caution** Before saving your own set of configuration settings in the matlab/toolbox/fixpoint folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

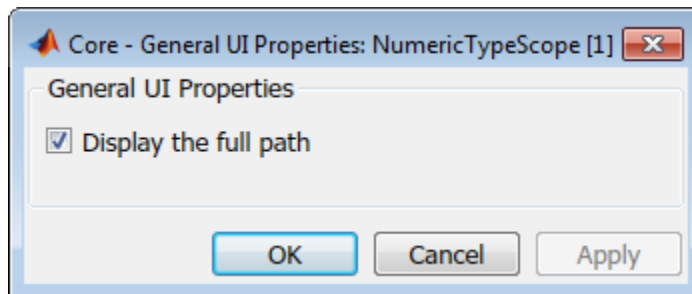
---

To save your configuration settings for future use, save them in the `matlab/toolbox/fixpoint` folder with the file name `NumericTypeScopeSL.cfg`. You can re-save your configuration settings at anytime, but you must save them in this folder with this filename.

### Core Pane

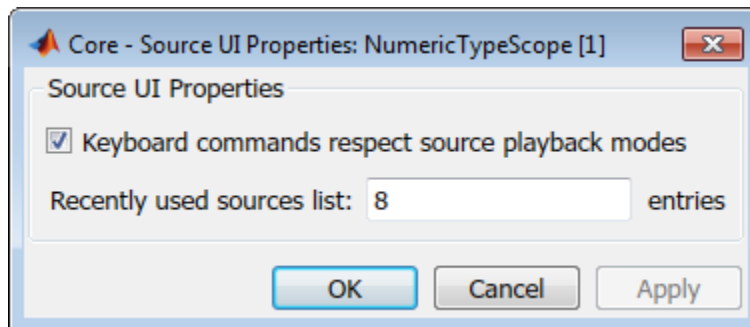
The **Core** pane controls the general settings of the scope.

To open the **Core - General UI Properties** dialog box, select **General UI** and then click **Properties**.



- **Display the full source path in the title bar**—Select this check box to display the full path to the selected block in the model. Otherwise, the scope displays only the block name.

To open the **Core - Source UI Properties** dialog box, select **Source UI** and then click **Properties**.



- **Keyboard commands respect source playback modes**—Has no effect. The following table shows the keyboard shortcut mapping. You cannot disable this mapping.

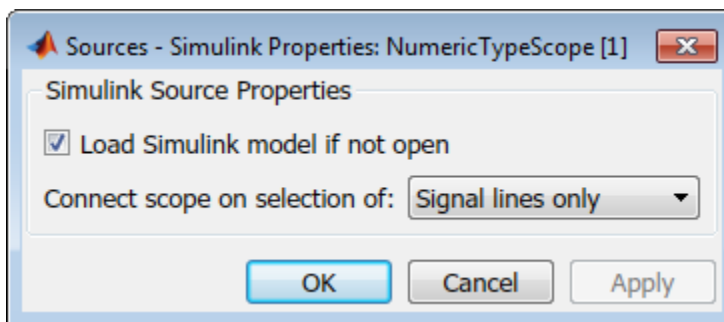
Action	Keyboard Shortcut
Open new NumericTypeScope	Insert
Change configuration	N
Display keyboard help	K
Play simulation	P
Pause simulation	Space
Stop simulation	S
Step forward	Right arrow, Page down

- **Recently used sources list**—Sets the maximum number of recently used sources displayed under the **Files** menu option.

### Sources Pane

The **Sources** pane controls how the scope connects to Simulink. You cannot disable the Simulink source.

To open the **Sources - Simulink Properties** dialog box, select the **Sources** tab and then click **Properties**.



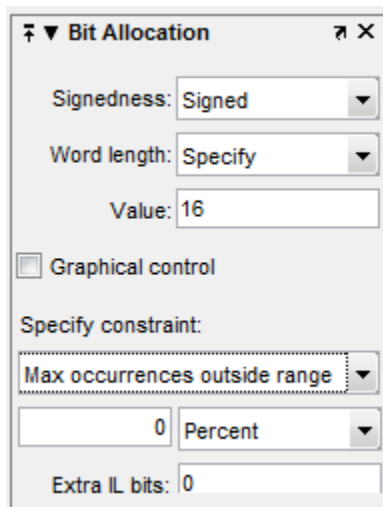
- **Load Simulink model if not open**—When selected, if you specify a signal in a Simulink model that is not currently open, the scope opens the model.

- **Connect scope on selection of**—Connects the scope only when you select signal lines or when you select signal lines or blocks. If you select **Signal lines or blocks**, the scope cannot connect to blocks that have more than one output port.

## Dialog Panels

### Bit Allocation Panel

The scope **Bit Allocation** panel provides options for specifying data type criteria. Adjust these criteria to observe the effect on suggested numerictype. For streaming data, the suggested numerictype adjusts over time in order to continue to satisfy the specified criteria.



The screenshot shows the Bit Allocation dialog panel with the following settings:

- Signedness:** Signed
- Word length:** Specify
- Value:** 16
- Graphical control
- Specify constraint:** Max occurrences outside range
- Value:** 0
- Unit:** Percent
- Extra IL bits:** 0

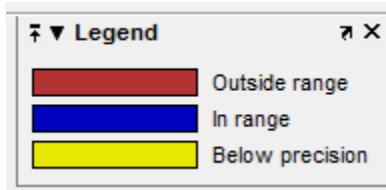
You can:

- Specify a known word length and signedness and, using **Specify constraint**, add additional constraints such as the maximum number of occurrences outside range or the smallest value that the suggested data type must represent.
- Specify **Integer length** and **Fraction length** constraints so that the scope suggests an appropriate word length.
- Set the **Signedness** and **Word length** to Auto so that the scope suggests values for these parameters.

- Enable **Graphical control** and use the cursors on either side of the binary point to adjust the fraction length and observe the effect on the suggested numerictype on the input data. For example, you can see the number of values that are outside range, below precision, or both. You can also view representable minimum and maximum values of the changed suggested data type.
- Specify extra bits for either the fraction length or the integer length. The extra bits act as a safety margin to minimize the risk of overflow and precision loss.

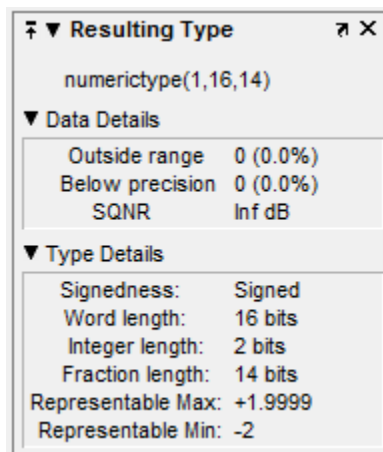
## Legend

The scope **Legend** panel informs you which colors the scope uses to indicate values. These colors represent values that are outside range, in range, or below precision when displayed in the scope.



## Resulting Type

The **Resulting Type** panel describes the fixed-point data type as defined by scope settings. By manipulating the visual display (via the **Bit Allocation** panel or with the cursors), you can change the data type specification.

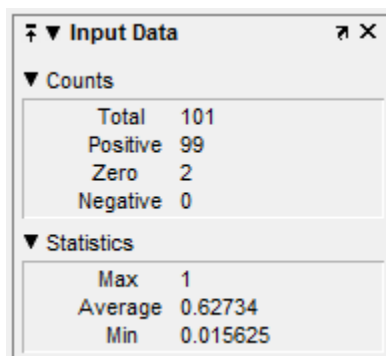


The **Data Details** section displays the percentage of values that fall outside range or below precision with the `numericType` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data to calculate the SQNR. When scope information or the `numericType` changes, the SQNR resets.

The **Type Details** section provides details about the fixed-point data type. You can copy the `numericType` specification by right-clicking the **Resulting Type** pane and then selecting `Copy numericType`.

### Input Data

The **Input Data** panel provides statistical information about the values currently displayed in the `NumericScopeType`.



The screenshot shows a window titled "Input Data" with a close button. It contains two sections: "Counts" and "Statistics".

Counts	
Total	101
Positive	99
Zero	2
Negative	0

Statistics	
Max	1
Average	0.62734
Min	0.015625

## Examples

### Connect a `NumericTypeScope` to a signal in a Simulink model

Open a `NumericTypeScope` window and connect to a signal.

Open the model.

```
fxpdemo_approx
```

Open a `NumericTypeScope`.

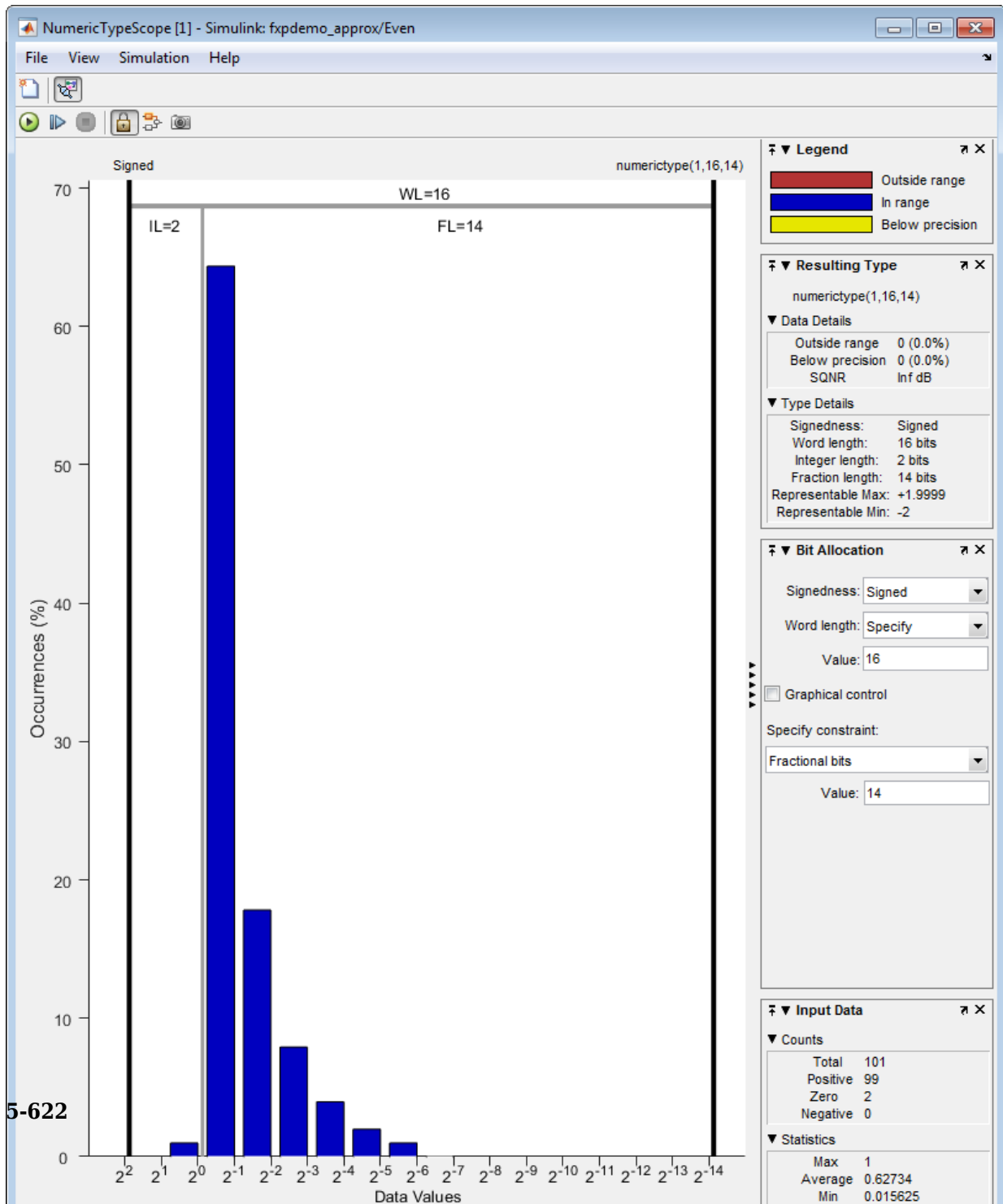
```
nts
```



In the fxdemo\_approx model, select the yEven signal.

In the NumericTypeScope window, select **File > Connect to Simulink Signal**.

Run the simulation to view the dynamic range of the output. The NumericTypeScope suggests a data type for the output.



## Connect a NumericTypeScope to a block output port

Connect a NumericTypeScope to a block output port and view the dynamic range of block output.

Specify the block path and name and the output port number.

```
blk='fxpdemo_approx/Even';  
nts({blk,1})
```

Run the simulation to view the dynamic range of the output. The NumericTypeScope suggests a data type for the output.

## Specify a Simulink signal to display

Connect a NumericTypeScope to a signal selected in a model.

Open the model.

```
fxpdemo_approx
```

In the fxpdemo\_approx model, select the yEven signal.

Open a NumericTypeScope, specifying the line handle of the selected signal.

```
nts({gs1})
```

## Tips

- Use the NumericTypeScope to help you identify any values that are outside range or below precision based on the current data type.

When the information is available, the scope indicates values that are outside range, below precision, and in range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are in range of the current data type.
- Red — Histogram bin contains values that are outside range in the current data type.

- Yellow — Histogram bin contains values that are below precision in the current data type.
- Select **View > Vertical Units** to select whether to display values as a percentage or as an actual count.
- Use the **View > Bring All NumericTypeScope Windows Forward** menu option to manage your NumericTypeScope windows. Selecting this option or pressing **Ctrl+F** brings all NumericTypeScope windows into view.

### See Also

hist | log2 | numerictypescope

**Introduced in R2012a**

## num2bin

Convert number to binary representation using `quantizer` object

### Syntax

```
y = num2bin(q,x)
```

### Description

`y = num2bin(q,x)` converts numeric array `x` into binary character vectors returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to binary. If `x` is a structure, each numeric field of `x` is converted to binary.

`num2bin` and `bin2num` are inverses of one another, differing in that `num2bin` returns the binary representations in a column.

### Examples

```
x = magic(3)/9;  
q = quantizer([4,3]);  
y = num2bin(q,x)
```

Warning: 1 overflow.

y =

```
0111  
0010  
0011  
0000  
0100  
0111  
0101  
0110  
0001
```

## **See Also**

`bin2num` | `hex2num` | `num2hex` | `num2int`

**Introduced before R2006a**

# num2hex

Convert number to hexadecimal equivalent using `quantizer` object

## Syntax

```
y = num2hex(q,x)
```

## Description

`y = num2hex(q,x)` converts numeric array `x` into hexadecimal returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to hexadecimal. If `x` is a structure, each numeric field of `x` is converted to hexadecimal.

For fixed-point `quantizer` objects, the representation is two's complement. For floating-point `quantizer` objects, the representation is IEEE Standard 754 style.

For example, for `q = quantizer('double')`

```
num2hex(q,nan)
```

```
ans =
```

```
fff8000000000000
```

The leading fraction bit is 1, all other fraction bits are 0. Sign bit is 1, exponent bits are all 1.

```
num2hex(q,inf)
```

```
ans =
```

```
7ff0000000000000
```

Sign bit is 0, exponent bits are all 1, all fraction bits are 0.

```
num2hex(q,-inf)
```

```
ans =
```

```
fff0000000000000
```

Sign bit is 1, exponent bits are all 1, all fraction bits are 0.

`num2hex` and `hex2num` are inverses of each other, except that `num2hex` returns the hexadecimal values in a column.

## Examples

This is a floating-point example using a quantizer object `q` that has 6-bit word length and 3-bit exponent length.

```
x = magic(3);  
q = quantizer('float',[6 3]);  
y = num2hex(q,x)
```

```
y =
```

```
18  
12  
14  
0c  
15  
18  
16  
17  
10
```

## See Also

`bin2num` | `hex2num` | `num2bin` | `num2int`

**Introduced before R2006a**



# num2int

Convert number to signed integer

## Syntax

```
y = num2int(q,x)
[y1,y,...] = num2int(q,x1,x,...)
```

## Description

`y = num2int(q,x)` uses `q.format` to convert numeric `x` to an integer.

`[y1,y,...] = num2int(q,x1,x,...)` uses `q.format` to convert numeric values `x1`, `x2`, ... to integers `y1`, `y2`, ...

## Examples

All the two's complement 4-bit numbers in fractional form are given by

```
x = [0.875 0.375 -0.125 -0.625
      0.750 0.250 -0.250 -0.750
      0.625 0.125 -0.375 -0.875
      0.500 0.000 -0.500 -1.000];
```

```
q=quantizer([4 3]);
```

```
y = num2int(q,x)
```

```
y =
```

```
 7     3    -1    -5
 6     2    -2    -6
 5     1    -3    -7
 4     0    -4    -8
```

## Algorithms

When `q` is a fixed-point quantizer object, `f` is equal to `fractionlength(q)`, and `x` is numeric

$$y = x \times 2^f$$

When `q` is a floating-point quantizer object, `y = x.num2int` is meaningful only for fixed-point quantizer objects.

## See Also

`bin2num` | `hex2num` | `num2bin` | `num2hex`

**Introduced before R2006a**

# num2str

Convert numbers to character array

## Syntax

```
s = num2str(A)
s = num2str(A,precision)
s = num2str(A,formatSpec)
```

## Description

`s = num2str(A)` converts `fi` object `A` into a character array representation. The output is suitable for input to the `eval` function such that `eval(s)` produces the original `fi` object exactly.

`s = num2str(A,precision)` converts `fi` object `A` to a character array representation using the number of digits of precision specified by `precision`.

`s = num2str(A,formatSpec)` applies a format specified by `formatSpec` to all elements of `A`.

## Examples

### Convert a `fi` Object to a Character Vector

Create a `fi` object, `A`, and convert it to a character vector.

```
A = fi(pi)
```

```
A =
```

```
3.1416
```

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

```
        Signedness: Signed
        WordLength: 16
        FractionLength: 13

S = num2str(A)

S =

    '3.1416'
```

### Convert a fi Object to a Character with Specified Precision

Create a `fi` object and convert it to a character vector with 8 digits of precision.

```
A = fi(pi)

A =

    3.1416

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

S = num2str(A,8)

S =

    '3.1416016'
```

## Input Arguments

### A — Input array

numeric array

Input array, specified as a numeric array.

Data Types: `fi` | `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`  
Complex Number Support: Yes

**precision — Number of digits of precision**

positive integer

Maximum number of significant digits in the output string, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**formatSpec — Format of output fields**

formatting operators

Format of the output fields, specified using formatting operators. `formatSpec` also can include ordinary text and special characters.

For more information on formatting operators, see the `num2str` reference page in the MATLAB documentation.

## Output Arguments

**s — Text representation of input array**

character array

Text representation of the input array, returned as a character array.

## See Also

`mat2str` | `num2str` | `tostring`**Introduced in R2016a**

## numel

Number of data elements in `fi` array

### Syntax

```
n = numel(A)
```

### Description

`n = numel(A)` returns the number of elements, `n`, in `fi` array `A`.

Using `numel` in your MATLAB code returns the same result for built-in types and `fi` objects. Use `numel` to write data-type independent MATLAB code for array handling.

### Examples

#### Number of Elements in 2-D `fi` Array

Create a 2-by-3- array of `fi` objects.

```
X = fi(ones(2,3),1,24,12)
```

```
X=2×3 object
```

```
    1    1    1
    1    1    1
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 24
      FractionLength: 12
```

`numel` counts 6 elements in the matrix.

```
n = numel(X)
```

```
n = 6
```

## Number of Elements in Multidimensional fi Array

Create a 2-by-3-by-4 array of `fi` objects.

```
X = fi(ones(2,3,4),1,24,12)
```

```
X =
```

```
(:,:,1) =
```

```
    1     1     1
    1     1     1
```

```
(:,:,2) =
```

```
    1     1     1
    1     1     1
```

```
(:,:,3) =
```

```
    1     1     1
    1     1     1
```

```
(:,:,4) =
```

```
    1     1     1
    1     1     1
```

```
        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 24
        FractionLength: 12
```

`numel` counts 24 elements in the matrix.

```
n = numel(X)
```

```
n = 24
```

## Input Arguments

### A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

Complex Number Support: Yes

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`numel`

**Introduced in R2013b**



# numerictype

Construct numerictype object

## Syntax

```
T = numerictype
T = numerictype(s)
T = numerictype(s,w)
T = numerictype(s,w,f)
T = numerictype(s,w,slope,bias)
T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)
T = numerictype(property1,value1, ...)
T = numerictype(T1, property1, value1, ...)
T = numerictype('double')
T = numerictype('single')
T = numerictype('boolean')
```

## Description

You can use the numerictype constructor function in the following ways:

- `T = numerictype` creates a default numerictype object.
- `T = numerictype(s)` creates a numerictype object with Fixed-point: unspecified scaling, Signed property value `s`, and 16-bit word length.
- `T = numerictype(s,w)` creates a numerictype object with Fixed-point: unspecified scaling, Signed property value `s`, and word length `w`.
- `T = numerictype(s,w,f)` creates a numerictype object with Fixed-point: binary point scaling, Signed property value `s`, word length `w` and fraction length `f`.
- `T = numerictype(s,w,slope,bias)` creates a numerictype object with Fixed-point: slope and bias scaling, Signed property value `s`, word length `w`, slope, and bias.
- `T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)` creates a numerictype object with Fixed-point: slope and bias scaling,

Signed property value *s*, word length *w*, *slopeadjustmentfactor*, *fixedexponent*, and *bias*.

- `T = numerictype(property1,value1, ...)` allows you to set properties for a `numerictype` object using property name/property value pairs. All properties for which you do not specify a value get assigned their default value.
- `T = numerictype(T1, property1, value1, ...)` allows you to make a copy of an existing `numerictype` object, while modifying any or all of the property values.
- `T = numerictype('double')` creates a double `numerictype`.
- `T = numerictype('single')` creates a single `numerictype`.
- `T = numerictype('boolean')` creates a Boolean `numerictype`.

The properties of the `numerictype` object are listed below. These properties are described in detail in “`numerictype` Object Properties”.

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeOverride` — Data type override settings. Note that this property is not visible when its value is the default, `Inherit`.
- `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

## Examples

### Create a default `numerictype` object

Type

```
T = numerictype
```

to create a default numerictype object.

```
T =
```

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

## Create a numerictype object with specified word and fraction lengths

The following code creates a signed numerictype object with a 32-bit word length and 30-bit fraction length.

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

```
T =
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 30
```

## Create a numerictype object with unspecified scaling

If you omit the argument *f*, the scaling is unspecified.

```
T = numerictype(1, 32)
```

```
T =
```

```
    DataTypeMode: Fixed-point: unspecified scaling
    Signedness: Signed
    WordLength: 32
```

## Create a numeric type object with default word length and scaling

If you omit the arguments `w` and `f`, the word length is automatically set to 16 bits and the scaling is unspecified.

```
T = numericType(1)
```

```
T =
```

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

## Create a numeric type object with specified property values

You can use property name/property value pairs to set `numericType` properties when you create the object.

```
T = numericType('Signed', true, 'DataTypeMode', ...
    'Fixed-point: slope and bias scaling', ...
    'WordLength', 32, 'Slope', 2^-2, 'Bias', 4)
```

```
T =
```

```
DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 32
Slope: 0.25
Bias: 4
```

---

**Note** When you create a `numericType` object using property name/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)`. See “Example: Construct a `numericType` Object with Property Name and Property Value Pairs” in the Fixed-Point Designer User's Guide for more information.

---

## Create a numerictype object with unspecified sign

You can create a numerictype object with an unspecified sign by using property name/property values pairs to set the Signedness property to Auto.

```
T = numerictype('Signedness', 'Auto')
```

```
T =
```

```

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

```

---

**Note** Although you can create numerictype objects with an unspecified sign (Signedness: Auto), all fi objects must have a Signedness of Signed or Unsigned. If you use a numerictype object with Signedness: Auto to construct a fi object, the Signedness property of the fi object automatically defaults to Signed.

---

## Create a numerictype object with specified data type

You can create a numerictype object with a specific data type by including the property name/property value pair in the numerictype constructor.

```
T = numerictype(0, 24, 12, 'DataType', 'ScaledDouble')
```

```
T =
```

```

    DataTypeMode: Scaled double: binary point scaling
    Signedness: Unsigned
    WordLength: 24
    FractionLength: 12

```

MATLAB returns an unsigned numerictype object, T, with the specified WordLength of 24, FractionLength of 12, and with DataType set to ScaledDouble.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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.
- All `numericType` object properties related to the data type must be constant.

### See Also

`fi` | `fimath` | `fipref` | `quantizer`

### Topics

“`numericType` Objects Usage to Share Data Type and Scaling Settings of `fi` objects”

“`numericType` Object Properties”

**Introduced before R2006a**

# NumericTypeScope

Determine fixed-point data type

## Syntax

```
H = NumericTypeScope  
show(H)  
step(H, data)  
release(H)  
reset(H)
```

## Description

The `NumericTypeScope` is an object that provides information about the dynamic range of your data. The scope provides a visual representation of the dynamic range of your data in the form of a `log2` histogram. In this histogram, the bit weights appear along the X-axis, and the percentage of occurrences along the Y-axis. Each bin of the histogram corresponds to a bit in the binary word. For example,  $2^0$  corresponds to the first integer bit in the binary word,  $2^{-1}$  corresponds to the first fractional bit in the binary word.

The scope suggests a data type in the form of a `numericType` object that satisfies the specified criteria. See the section on Bit Allocation in “Dialog Panels” on page 5-649.

`H = NumericTypeScope` returns a `NumericTypeScope` object that you can use to view the dynamic range of data in MATLAB. To view the `NumericTypeScope` window after creating *H*, use the `show` method.

`show(H)` opens the `NumericTypeScope` object *H* and brings it into view. Closing the scope window does not delete the object from your workspace. If the scope object still exists in your workspace, you can open it and bring it back into view using the `show` method.

`step(H, data)` processes your data and allows you to visualize the dynamic range. The object *H* retains previously collected information about the variable between each call to `step`.

`release(H)` releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

`reset(H)` clears all stored information from the `NumericTypeScope` object *H*. Resetting the object clears the information displayed in the scope window.

## Identifying Values Outside Range and Below Precision

The `NumericTypeScope` can also help you identify any values that are outside range or below precision based on the current data type. To prepare the `NumericTypeScope` to identify them, provide an input variable that is a `fi` object and verify that one of the following conditions is true:

- The `DataTypeMode` of the `fi` object is set to `Scaled doubles: binary point scaling`.
- The `DataTypeOverride` on page 3-4 property of the Fixed-Point Designer `fipref` object is set to `ScaledDoubles`.

When the information is available, the scope indicates values that are outside range, below precision, and in range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are in range of the current data type.
- Red — Histogram bin contains values that are outside range in the current data type.
- Yellow — Histogram bin contains values that are below precision in the current data type.

For an example of the scope color coding, see the figures in “Vertical Units” on page 5-652.

See also Legend in “Dialog Panels” on page 5-649.

See the “Examples” on page 5-0 section to learn more about using the `NumericTypeScope` to select data types.

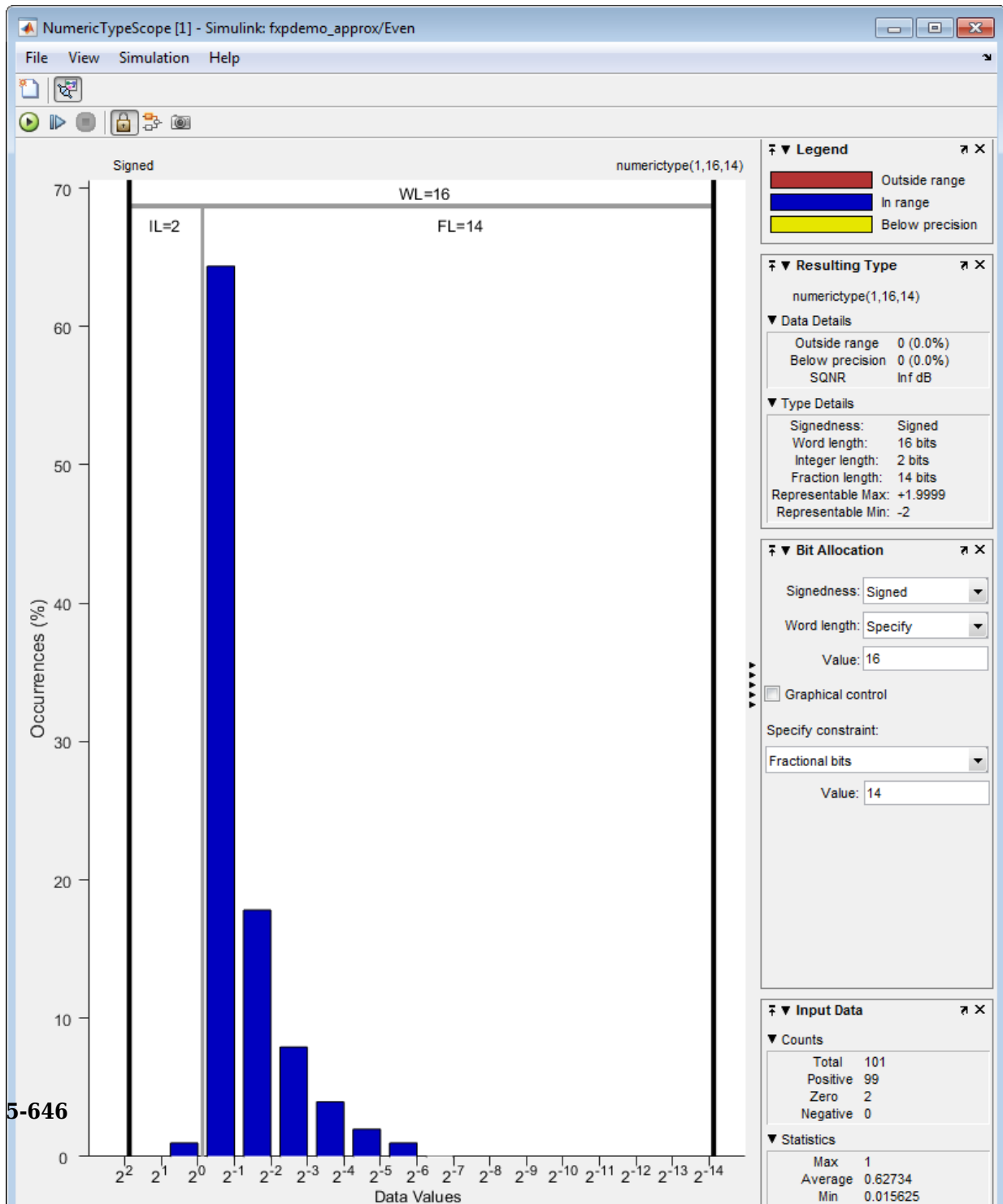


## **Dialog Boxes and Toolbar**

- “The NumericTypeScope Window” on page 5-645
- “Configuration Dialog Box” on page 5-647
- “Dialog Panels” on page 5-649
- “Vertical Units” on page 5-652
- “Bring All NumericType Scope Windows Forward” on page 5-654
- “Toolbar (Mac Only)” on page 5-655

## **The NumericTypeScope Window**

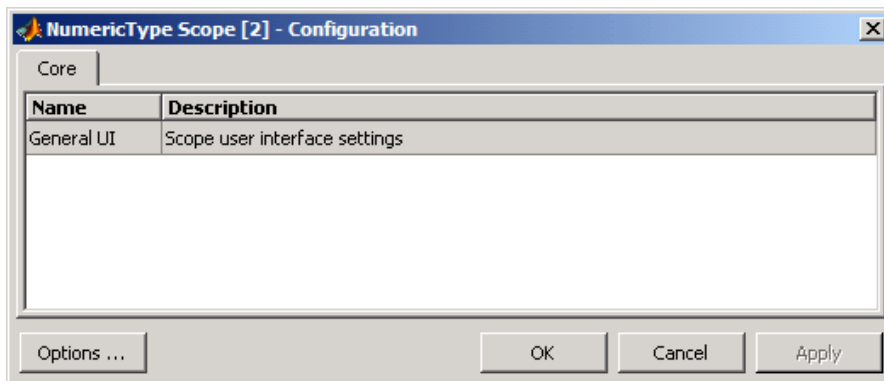
The NumericTypeScope opens with the default toolbars displayed at the top of the window and the dialog panels to the right.



## Configuration Dialog Box

The NumericTypeScope configuration allows you to control the behavior and appearance of the scope window.

To open the Configuration dialog box, select **File > Configuration > Edit**, or, with the scope as your active window, press the **N** key.



The Configuration Dialog box contains a series of panes each containing a table of configuration options. See the reference section for each pane for instructions on setting the options on each one. This dialog box has one pane, the Core pane, with only one option, for General UI settings for the scope user interface.

To save configuration settings for future use, select **File > Configuration > Save as**. The configuration settings you save become the default configuration settings for the NumericTypeScope object.

---

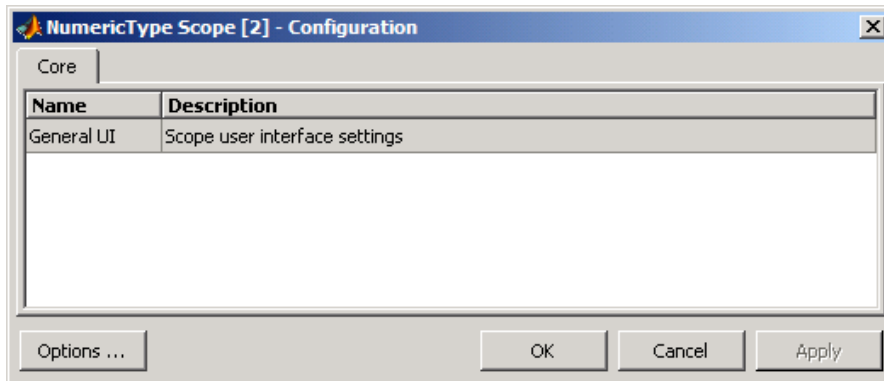
**Caution** Before saving your own set of configuration settings in the matlab/toolbox/fixedpoint/fixedpoint folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

---

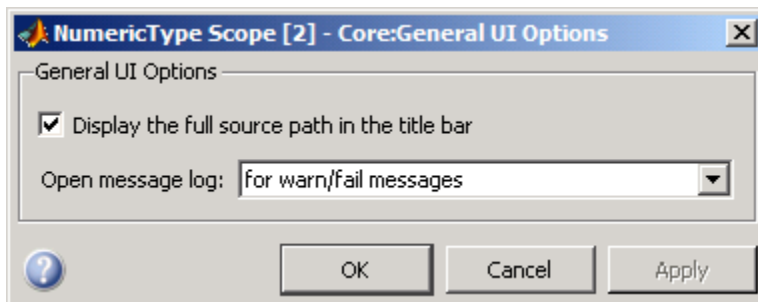
To save your configuration settings for future use, save them in the matlab/toolbox/fixedpoint/fixedpoint folder with the file name `NumericTypeScopeComponent.cfg`. You can re-save your configuration settings at anytime, but remember to do so in the specified folder using the specified file name.

## Core Pane

The Core pane in the Configuration dialog box controls the general settings of the scope.



Click General UI and then click **Options** to open the Core:General UI Options dialog box.



- **Display the full source path in the title bar**—Select this check box to display the file name and variable name in the scope title bar. If the scope is not from a file, or if you clear this check box, the scope displays only the variable name in the title bar.
- **Open message log**—Control when the Message Log window opens. The Message log window helps you debug issues with the scope. Choose to open the Message Log window for any of these conditions:
  - for any new messages
  - for warn/fail messages
  - only for fail messages

- manually

The option defaults to for warn/fail messages.

You can open the Message Log at any time by selecting **Help > Message Log** or by pressing **Ctrl+M**. The Message Log dialog box provides a system level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by Type and Category.

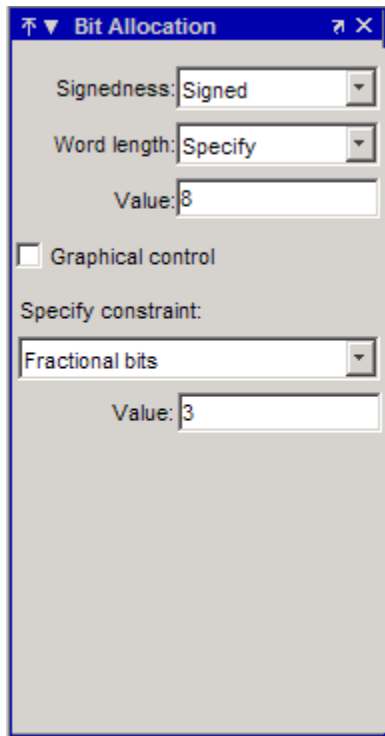
- **Type**—Select the type of messages to display in the Message Log. You can select All, Info, Warn, or Fail. Type defaults to All.
- **Category**—Select the category of messages to display in the Message Log. You can select All, Configuration, or Extension. The scope uses Configuration messages to indicate when new configuration files are loaded, and Extension messages to indicate when components are registered. Category defaults to All.

## Dialog Panels

- “Bit Allocation” on page 5-649
- “Legend” on page 5-651
- “Resulting Type” on page 5-651
- “Input Data” on page 5-652

### Bit Allocation

The scope Bit Allocation dialog panel, as shown in the following figure, offers you several options for specifying data type criteria.

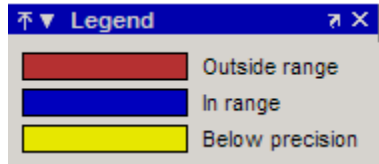


You can use this panel to specify a known word length and the desired maximum occurrences outside range. You can also use the panel to specify the desired number of occurrences outside range and the smallest value to be represented by the suggested data type. For streaming data, the suggested numeric type object adjusts over time in order to continue to satisfy the specified criteria.

The scope also allows you to interact with the histogram plot. When you select **Graphical control** on the Bit Allocation dialog panel, you enable cursors on either side of the binary point. You can interact with these cursors and observe the effect of the suggested numeric type on the input data. For example, you can see the number of values that are outside range, below precision, or both. You can also view representable minimum and maximum values of the data type.

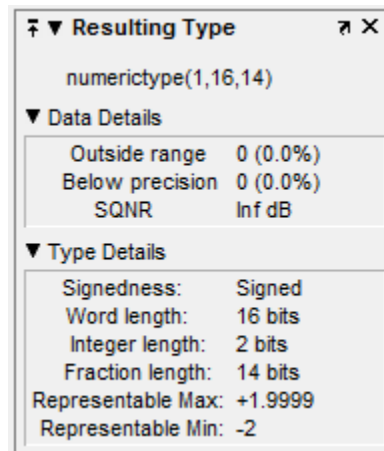
## Legend

The scope Legend panel informs you which colors the scope uses to indicate values. These colors represent values that are outside range, in range, or below precision when displayed in the scope.



## Resulting Type

The Resulting Type panel describes the fixed-point data type as defined by scope settings. By manipulating the visual display (via the Bit Allocation panel or with the cursors) you can change the value of the data type.

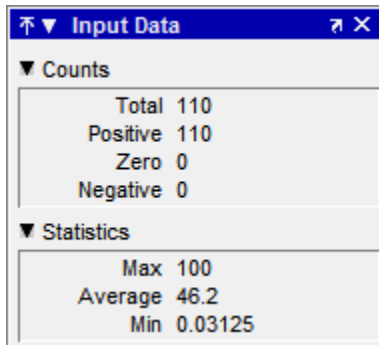


The Data Details section displays the percentage of values that fall outside range or below precision with the `numerictype` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data to calculate the SQNR. When scope information or the `numerictype` changes, the SQNR resets.

Type Details section provides details about the fixed-point data type.

### Input Data

The Input Data panel provides statistical information about the values currently displayed in the `NumericScopeType` object.



The screenshot shows a window titled "Input Data" with a blue header bar. Below the header, there are two sections: "Counts" and "Statistics".

Counts	
Total	110
Positive	110
Zero	0
Negative	0

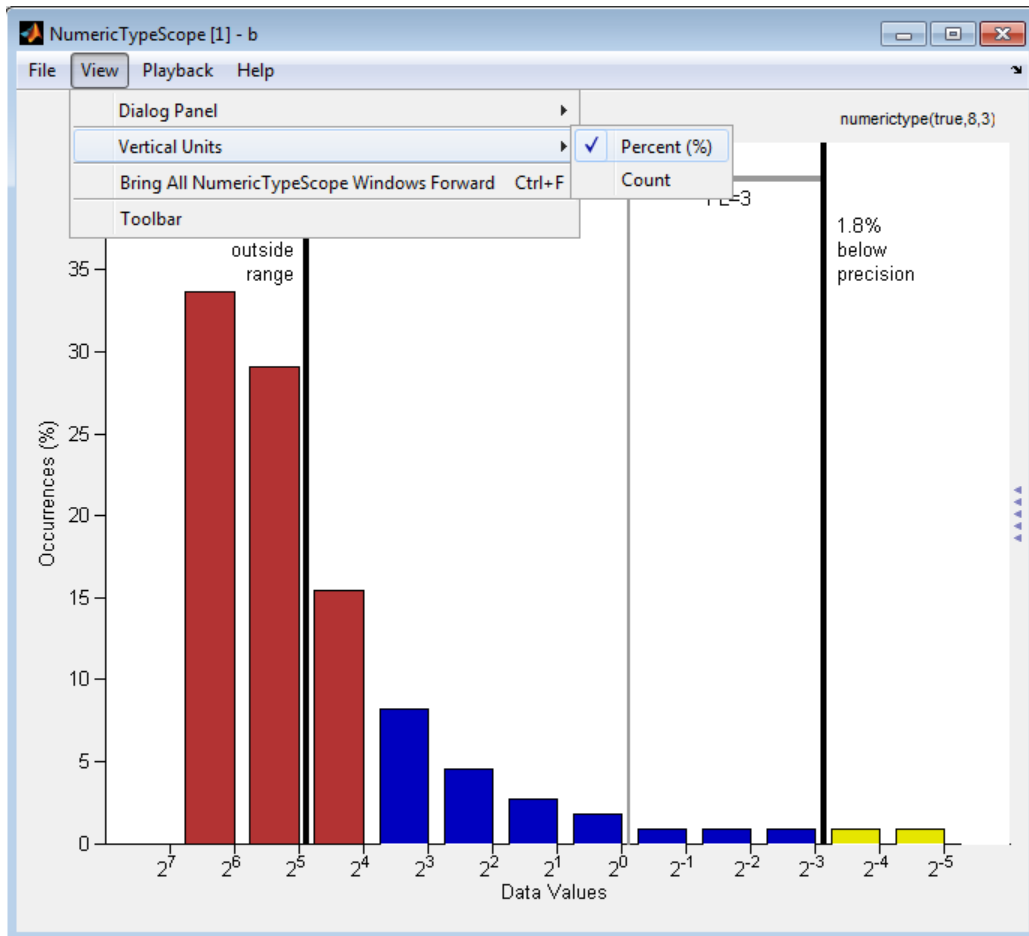
  

Statistics	
Max	100
Average	46.2
Min	0.03125

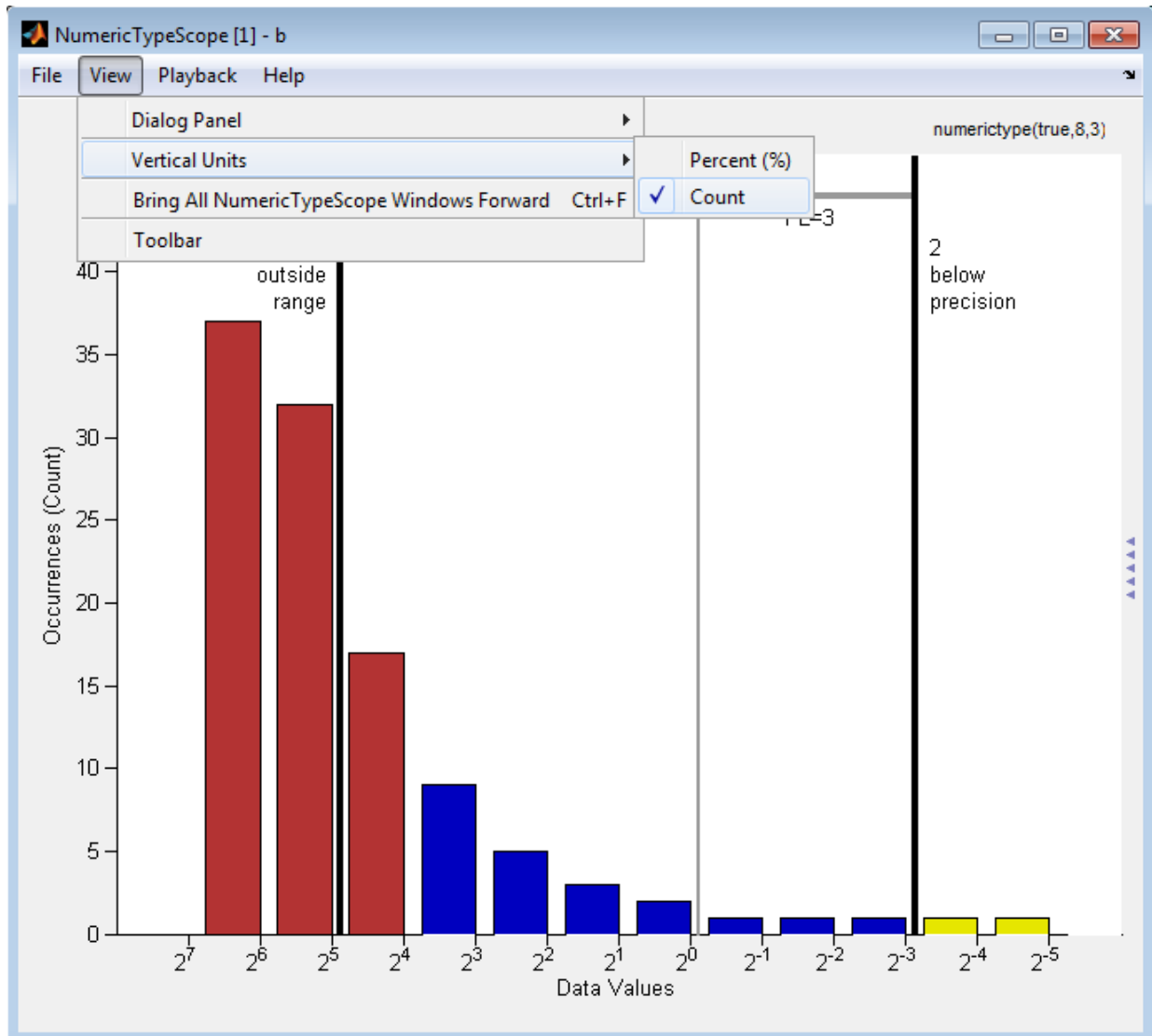
### Vertical Units

Use the Vertical Units selection to display values that are outside range or below precision as a percentage or as an actual count. For example, the following image shows the values that are outside range or below precision as a percentage of the total values.





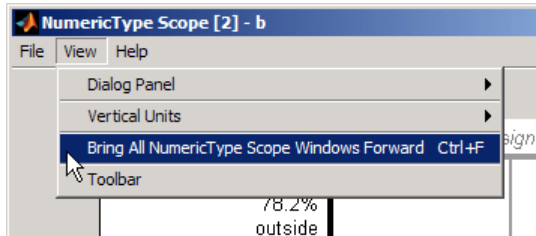
This next example shows the values that are outside range or below precision as an actual count.



## Bring All NumericType Scope Windows Forward

The NumericScopeType GUI offers a **View > Bring All NumericType Scopes Forward** menu option to help you manage your NumericTypeScope windows. Selecting this

option or pressing **Ctrl+F** brings all NumericTypeScope windows into view. If a NumericTypeScope window is not currently open, this menu option opens the window and brings it into view.



## Toolbar (Mac Only)

Activate the Toolbar by selecting **View > Toolbar**. When this tool is active, you can dock or undock the scope from the GUI.

The toolbar feature is for the Mac only. Selecting **Toolbar** on Windows® and UNIX® versions displays only an empty toolbar. The docking icon always appears in the GUI in the upper-right corner for these versions.

## Methods

### release

Use this method to release system resources (such as memory, file handles or hardware connections) and allow all properties and input characteristics to be changed.

Example:

```
>>release(H)
```

### reset

Use this method to clear the information stored in the object *H*. Doing so allows you to reuse *H* to process data from a different variable.

Example:

```
>>reset(H)
```

## **show**

Use this method to open the scope window and bring it into view.

Example:

```
>>show(H)
```

## **step**

Use this method to process your data and visualize the dynamic range in the scope window.

Example:

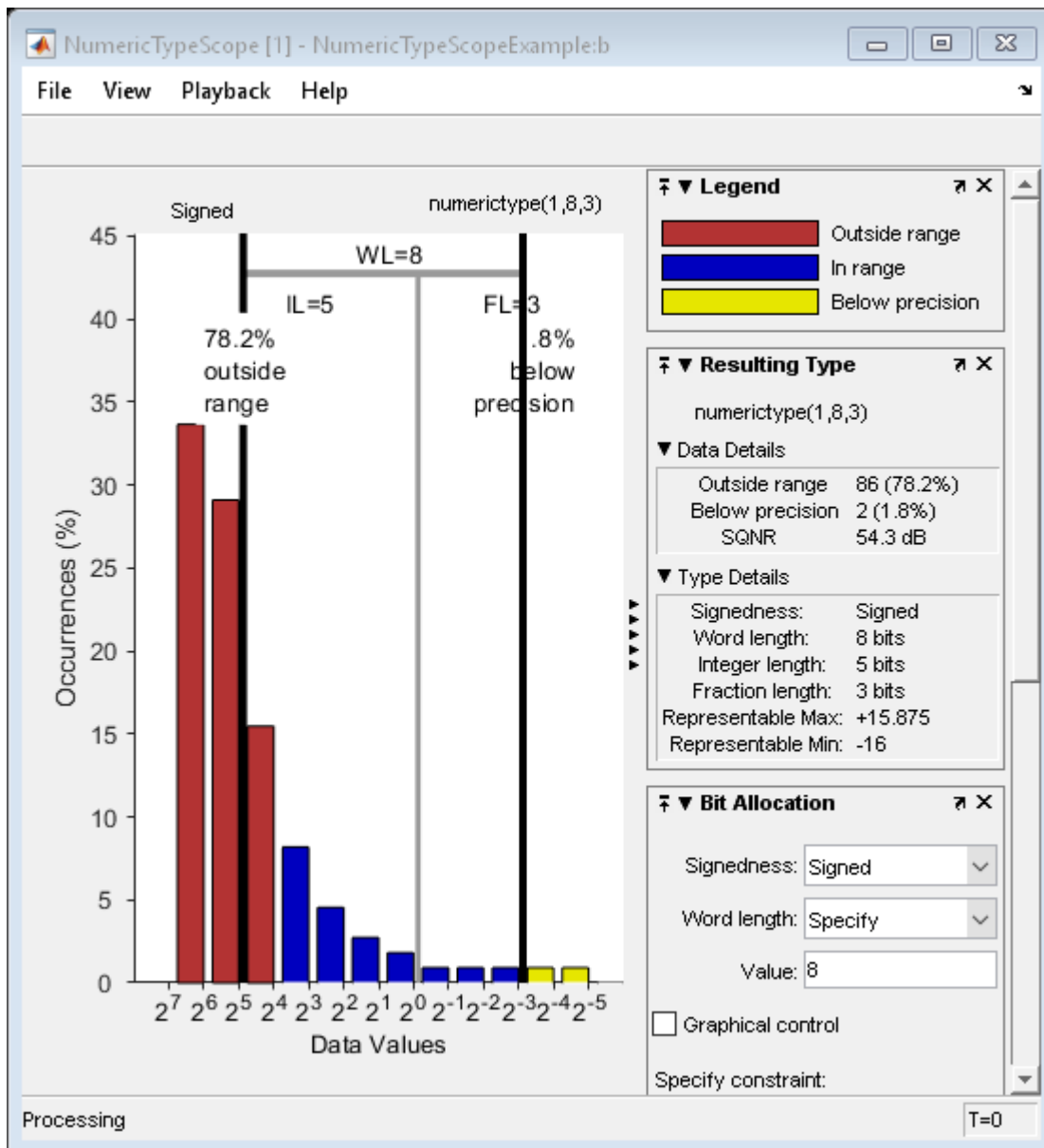
```
>>step(H, data)
```

# **Examples**

## **View the Dynamic Range of a fi Object**

Set the fi object `DataTypeOverride` to `Scaled Doubles`, and then view its dynamic range.

```
fp = fipref;  
initialDT0Setting = fp.DataTypeOverride;  
fp.DataTypeOverride = 'ScaledDoubles';  
a = fi(magic(10),1,8,2);  
b = fi([a; 2.^(-5:4)],1,8,3);  
h = NumericTypeScope;  
step(h,b);  
fp.DataTypeOverride = initialDT0Setting;
```



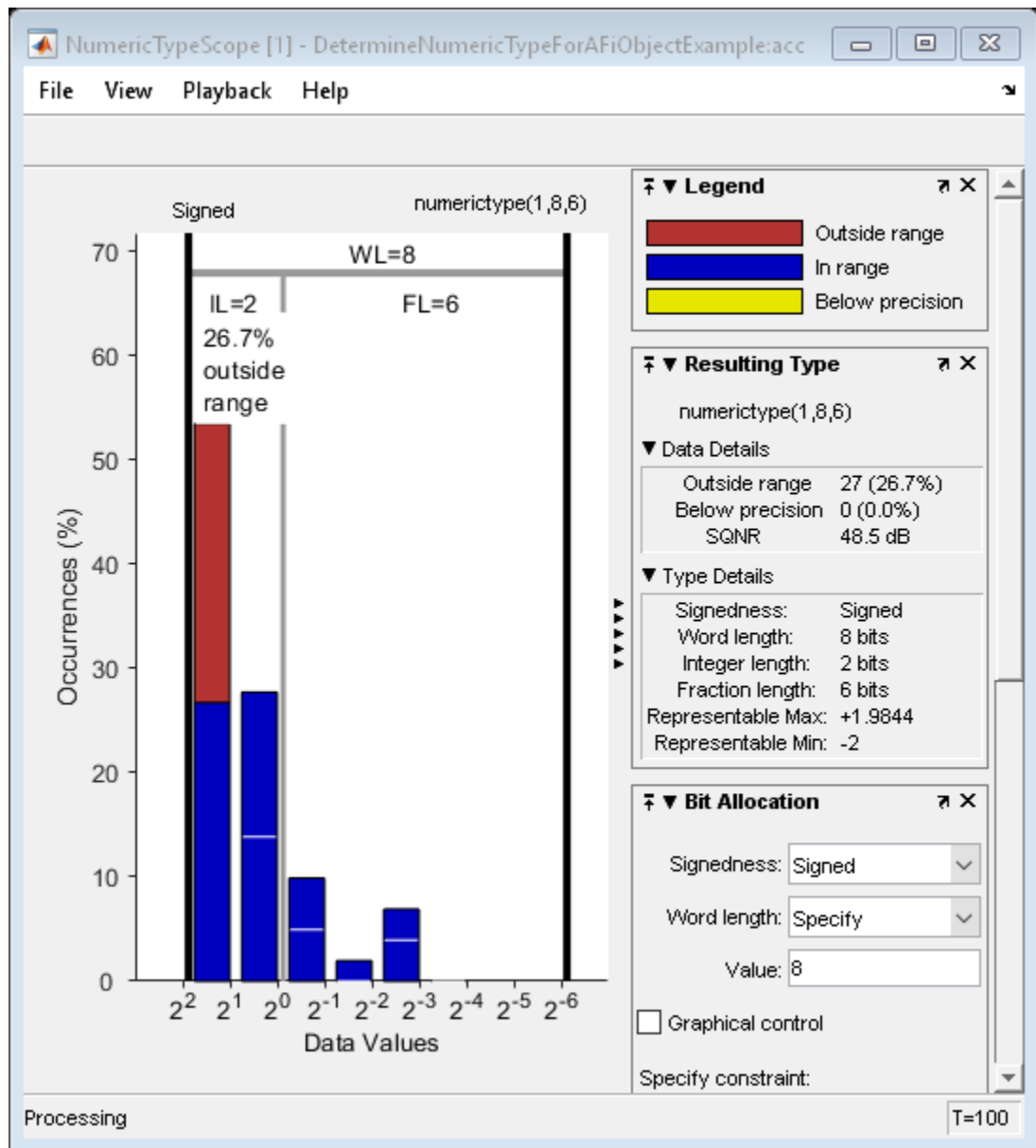
The log2 histogram display shows that the values appear both outside range and below precision in the variable. In this case, `b` has a data type of `numeric(1,8,3)`. The `numeric(1,8,3)` data type provides 5 integer bits (including the signed bit), and 3 fractional bits. Thus, this data type can represent only values between  $-2^4$  and  $2^4-2^{-3}$  (from -16 to 15.8750). Given the range and precision of this data type, values greater than  $2^4$  fall outside the range and values less than  $2^{-3}$  fall below the precision of the data type. When you examine the `NumericTypeScope` display, you can see that values requiring bits 5, 6, and 7 are outside range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numeric(0,13,5)`.

### Determine Numeric Type For a fi Object

View the dynamic range, and determine an appropriate numeric type for a `fi` object with a `DataTypeMode` of Scaled double: binary point scaling.

Create a `numeric` object with a `DataTypeMode` of Scaled double: binary point scaling. You can then use that `numeric` object to construct your `fi` objects. Because you set the `DataTypeMode` to Scaled double: binary point scaling, the `NumericTypeScope` can now identify overflows in your data.

```
T = numericType;
T.DataTypeMode = 'Scaled double: binary point scaling';
T.WordLength = 8;
T.FractionLength = 6;
a = fi(sin(0:100)*3.5, T);
b = fi(cos(0:100)*1.75,T);
acc = fi(0,T);
h = NumericTypeScope;
for i = 1:length(a)
    acc(:) = a(i)*0.7+b(i);
    step(h,acc)
end
```

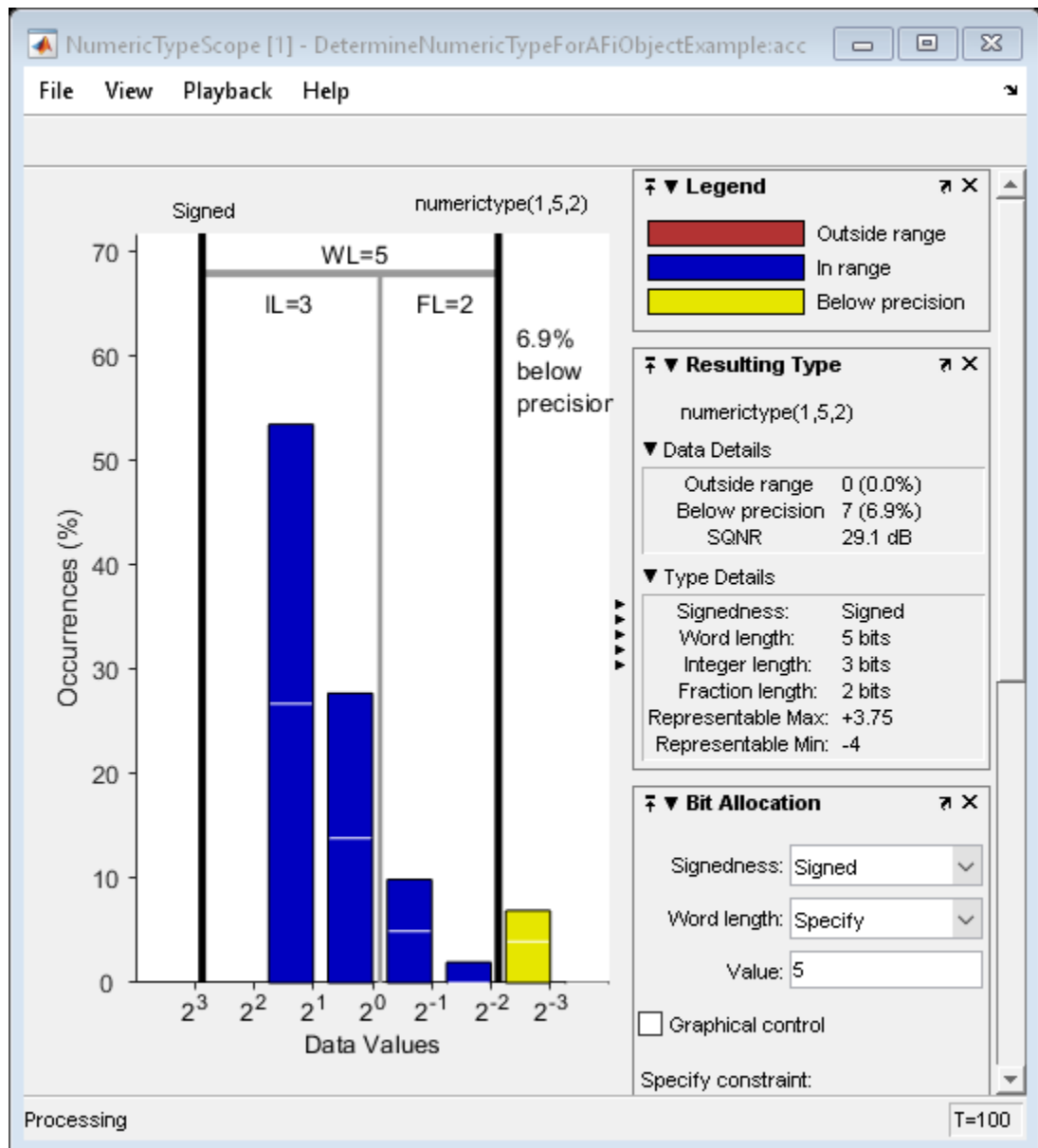


This dynamic range analysis shows that you can represent the entire range of data in the accumulator with 5 bits; two to the left of the binary point (integer bits) and three to the right of it (fractional bits). You can verify that this data type is able to represent all the values by changing the `WordLength` and `FractionLength` properties of the `numericType` object `T`. Then, use `T` to redefine the accumulator.

To view the dynamic range analysis based on this new data type, reset the `NumericTypeScope` object `h`, and rerun the loop.

```
T.WordLength = 5;
T.FractionLength = 2;
acc = fi(0,T);
release(h)
reset(h)
for i = 1:length(a)
    acc(:) = a(i)*0.7 + b(i);
    step(h,acc)
end
```





## See Also

hist | log2

**Introduced in R2010a**

# nunderflows

Number of underflows

## Syntax

```
y = nunderflows(a)  
y = nunderflows(q)
```

## Description

`y = nunderflows(a)` returns the number of underflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fi` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = nunderflows(q)` returns the accumulated number of underflows resulting from quantization operations performed by a `quantizer` object `q`.

## See Also

`maxlog` | `minlog` | `noverflows` | `resetlog`

**Introduced before R2006a**

## oct

Octal representation of stored integer of `fi` object

## Syntax

`oct(a)`

## Description

`oct(a)` returns the stored integer of `fi` object `a` in octal format. `oct(a)` is equivalent to `a.oct`.

Fixed-point numbers can be represented as

$$\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}$$

or, equivalently as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

## Examples

The following code

```
a = fi([-1 1],1,8,7);  
y = oct(a)  
z = a.oct
```

returns

y =

200 177

z =

200 177

## See Also

bin | dec | hex | storedInteger

**Introduced before R2006a**

## ones

Create array of all ones with fixed-point properties

### Syntax

```
X = ones('like',p)
X = ones(n,'like',p)
X = ones(sz1,...,szN,'like',p)
X = ones(sz,'like',p)
```

### Description

`X = ones('like',p)` returns a scalar 1 with the same `numericType`, complexity (real or complex), and `fimath` as `p`.

`X = ones(n,'like',p)` returns an `n`-by-`n` array of ones like `p`.

`X = ones(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of ones like `p`.

`X = ones(sz,'like',p)` returns an array of ones like `p`. The size vector, `sz`, defines `size(X)`.

### Examples

#### 2-D Array of Ones With Fixed-Point Attributes

Create a 2-by-3 array of ones with specified `numericType` and `fimath` properties.

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create a 2-by-3- array of ones that has the same `numericType` properties as `p`.

```
X = ones(2,3,'like',p)
```

```

X=2x3 object
    1     1     1
    1     1     1

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 24
    FractionLength: 12

```

### Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz = 1x2
```

```
    3     2
```

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create an array of ones that is the same size as A and has the same numeric type properties as p.

```
X = ones(sz, 'like', p)
```

```

X=3x2 object
    1     1
    1     1
    1     1

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 24
    FractionLength: 12

```

### Square Array of Ones With Fixed-Point Attributes

Create a 4-by-4 array of ones with specified numeric type and `fimath` properties.

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of ones that has the same numeric type properties as `p`.

```
X = ones(4, 'like', p)
```

```
X=4x4 object
```

```
  1     1     1     1
  1     1     1     1
  1     1     1     1
  1     1     1     1
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 24
      FractionLength: 12
```

### Create Array of Ones with Attached `fimath`

Create a signed `fi` object with word length of 16, fraction length of 15 and `OverflowAction` set to `Wrap`.

```
format long
p = fi([],1,16,15,'OverflowAction','Wrap');
```

Create a 2-by-2 array of ones with the same numeric type properties as `p`.

```
X = ones(2, 'like', p)
```

```
X=2x2 object
```

```
  0.999969482421875    0.999969482421875
  0.999969482421875    0.999969482421875
```



```

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

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

```

1 cannot be represented by the data type of `p`, so the value saturates. The output `fi` object `X` has the same `numericType` and `fimath` properties as `p`.

### Complex Fixed-Point One

Create a scalar fixed-point `1` that is not real valued, but instead is complex like an existing array.

Define a complex `fi` object.

```
p = fi( [1+2i 3i],1,24,12);
```

Create a scalar `1` that is complex like `p`.

```
X = ones('like',p)
```

```
X =
    1.0000 + 0.0000i
```

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
            WordLength: 24
      FractionLength: 12

```

### Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    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
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818    0.195262145875635    0.097631072937818];
    a = [1.000000000000000    -0.942809041582063    0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);

    % Create a step input using ones with the
    % floating-point data type
    t = 0:20;
    x_float = ones(size(t),'like',T_float.data);

    % Initialize the states using zeros with the
    % floating-point data type
    z_float = zeros(1,2,'like',T_float.data);
```

```

% Run the floating-point algorithm
y_float = my_filter(b,a,x_float,z_float,T_float);

% Define fixed-point types
T_fixed.coeffs = fi([],true,8,6);
T_fixed.data   = fi([],true,8,6);

% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t), 'like', T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2, 'like', T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf', 'subplot', 'plot', 'legend')
clf
subplot(211)
plot(t,y_float, 'co-', t,y_fixed, 'kx-')
legend('Floating-point output', 'Fixed-point output')
title('Step response')
subplot(212)
plot(t,y_float - double(y_fixed), 'rs-')
legend('Error')
figure(gcf)
end

```

## Input Arguments

### **n** — Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If n is zero, X is an empty matrix.
- If n is negative, it is treated as zero.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **sz1, . . . , szN — Size of each dimension**

two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **sz — Output size**

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Example: `sz = [2,3,4]` defines X as a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **p — Prototype**

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

If the value 1 overflows the numeric type of `p`, the output saturates regardless of the specified `OverflowAction` property of the attached `fimath`. All subsequent operations performed on the output obey the rules of the attached `fimath`.

Complex Number Support: Yes

## Tips

Using the `b = cast(a, 'like', p)` syntax to specify data types separately 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.

## See Also

`cast` | `ones` | `zeros`

## Topics

“Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using `cast` and `zeros`”

“Manual Fixed-Point Conversion Workflow”

“Manual Fixed-Point Conversion Best Practices”

**Introduced in R2013a**

# plus

Matrix sum of `fi` objects

## Syntax

`plus(a,b)`

## Description

`plus(a,b)` is called for the syntax `a + b` when `a` or `b` is an object.

`a + b` adds matrices `a` and `b`. `a` and `b` must have the same dimensions unless one is a scalar value (a 1-by-1 matrix). A scalar value can be added to any other value.

`plus` does not support `fi` objects of data type `Boolean`.

---

**Note** For information about the `fimath` properties involved in Fixed-Point Designer calculations, see “`fimath` Properties Usage for Fixed-Point Arithmetic” and “`fimath` ProductMode and SumMode” in the Fixed-Point Designer documentation.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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.

## See Also

`minus` | `mtimes` | `times` | `uminus`

**Introduced before R2006a**

## pow10

Base 10 power and scale half-precision numbers

### Syntax

```
Y = pow10(X)
```

### Description

$Y = \text{pow10}(X)$  returns an array,  $Y$ , whose elements are 10 raised to the power  $X$ .

---

**Note** This function supports only half-precision inputs.

---

### Examples

#### Base 10 Power

Create a half-precision vector,  $X$ .

```
X = half([1;2;3;4])
```

```
X =
```

```
4x1 half column vector
```

```
1  
2  
3  
4
```

Compute an array,  $Y$ , whose elements are 10 raised to the power  $X$ .

```
Y = pow10(X)
```



Y =

4x1 half column vector

```
    10
   100
  1000
 10000
```

## Input Arguments

### X — Power

scalar | vector | matrix | multidimensional array

Power, specified as a half-precision numeric scalar, vector, matrix, or multidimensional array

Data Types: Half

## Output Arguments

### Y — Output array

scalar | vector | matrix | multidimensional array

Array whose elements are 10 raised to the power X, returned as a half-precision scalar, vector, matrix, or multidimensional array.

## See Also

half

**Introduced in R2018b**

## pow2

Efficient fixed-point multiplication by  $2^K$

### Syntax

$b = \text{pow2}(a, K)$

### Description

$b = \text{pow2}(a, K)$  returns the value of  $a$  shifted by  $K$  bits where  $K$  is an integer and  $a$  and  $b$  are `fi` objects. The output  $b$  always has the same word length and fraction length as the input  $a$ .

---

**Note** In fixed-point arithmetic, shifting by  $K$  bits is equivalent to, and more efficient than, computing  $b = a * 2^k$ .

---

If  $K$  is a non-integer, the `pow2` function will round it to `floor` before performing the calculation.

The scaling of  $a$  must be equivalent to binary point-only scaling; in other words, it must have a power of 2 slope and a bias of 0.

$a$  can be real or complex. If  $a$  is complex, `pow2` operates on both the real and complex portions of  $a$ .

The `pow2` function obeys the `OverflowAction` and `RoundingMethod` properties associated with  $a$ . If obeying the `RoundingMethod` property associated with  $a$  is not important, try using the `bitshift` function.

The `pow2` function does not support `fi` objects of data type `Boolean`.

The function also does not support the syntax  $b = \text{pow2}(a)$  when  $a$  is a `fi` object.

## Examples

### Example 5.4. Example 1

In the following example, `a` is a real-valued `fi` object, and `K` is a positive integer.

The `pow2` function shifts the bits of `a` 3 places to the left, effectively multiplying `a` by  $2^3$ .

```
a = fi(pi,1,16,8)
b = pow2(a,3)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

`a =`

3.1406

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 8
```

`b =`

25.1250

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 8
```

`binary_a =`

0000001100100100

`binary_b =`

0001100100100000

### Example 5.5. Example 2

In the following example, `a` is a real-valued `fi` object, and `K` is a negative integer.

The `pow2` function shifts the bits of `a` 4 places to the right, effectively multiplying `a` by  $2^{-4}$ .

```
a = fi(pi,1,16,8)
b = pow2(a,-4)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

```
a =
```

```
3.1406
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 8
```

```
b =
```

```
0.1953
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 8
```

```
binary_a =
```

```
0000001100100100
```

```
binary_b =
```

```
0000000000110010
```

### **Example 5.6. Example 3**

The following example shows the use of `pow2` with a complex `fi` object:

```
format long g
P = fipref('NumericTypeDisplay', 'short');
a = fi(57 - 2i, 1, 16, 8)
```

```
a =
```

```
                    57 -                2i
                s16,8
pow2(a, 2)
ans =
                127.99609375 -            8i
                s16,8
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

### See Also

`bitshift` | `bitsll` | `bitsra` | `bitsrl`

**Introduced before R2006a**

## power, .^

**Package:** embedded

Fixed-point element-wise power

### Syntax

```
C = A.^B
C = power(A, B)
```

### Description

$C = A.^B$  raises each element of  $A$  to the corresponding power in  $B$ .

$C = \text{power}(A, B)$  is an alternative way to compute  $A.^B$ .

### Examples

#### Raise Each Element of a Matrix to a Scalar Power

Create a fixed-point matrix and raise it to a scalar power.

```
A = fi([1, 3; 4, 2])
```

```
A=2x2 object
```

```
  1   3
  4   2
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 12
```

```
C = A.^3
```

```
C=2x2 object
    1    27
   64     8
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 48
    FractionLength: 36
```

## Input Arguments

### A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array. Inputs **A** and **B** must either be the same size or have sizes that are compatible (for example, **A** is an  $M$ -by- $N$  matrix and **B** is a scalar or 1-by- $N$  row vector).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | logical | fi  
Complex Number Support: Yes

### B — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a non-negative, real, integer-valued scalar, vector, matrix, or multidimensional array. Inputs **A** and **B** must either be the same size or have sizes that are compatible (for example, **A** is an  $M$ -by- $N$  matrix and **B** is a scalar or 1-by- $N$  row vector).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | logical | fi

## Output Arguments

### C — Power

scalar | vector | matrix | multidimensional array

Power, returned as an array with the same dimensions as the input `A`. When `A` has a local `fimath` object, the output `C` also has the same local `fimath` object. The array power operation is always performed using the default `fimath` settings.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the exponent `k` is a variable, the `ProductMode` property of the governing `fimath` must be `SpecifyPrecision`.

### See Also

`mpower` | `power`

**Introduced in R2010a**



# qr

Orthogonal-triangular decomposition

## Description

The Fixed-Point Designer `qr` function differs from the MATLAB `qr` function as follows:

- The input `A` in `qr(A)` must be a real, signed `fi` object.
- The `qr` function ignores and discards any `fimath` attached to the input. The output is always associated with the default `fimath`.
- Pivoting is not supported for fixed-point inputs. You cannot use the following syntaxes:
  - `[~,~,E] = qr(...)`
  - `qr(A, 'vector')`
  - `qr(A,B, 'vector')`
- Economy size decomposition is not supported for fixed-point inputs. You cannot use the following syntax: `[Q,R] = qr(A,0)`.
- The least-squares-solution form is not supported for fixed-point inputs. You cannot use the following syntax: `qr(A,B)`.

Refer to the MATLAB `qr` reference page for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**Introduced in R2014a**

## quantize

Quantize fixed-point numbers

### Syntax

```
y = quantize(x)
y = quantize(x,nt)
y = quantize(x,nt,rm)
y = quantize(x,nt,rm,oa)

yBP = quantize(x,s)
yBP = quantize(x,s,wl)
yBP = quantize(x,s,wl,fl)
yBP = quantize(x,s,wl,fl,rm)
yBP = quantize(x,s,wl,fl,rm,oa)
```

### Description

`y = quantize(x)` quantizes `x` using these default values:

- `numerictype (true,16,15)`
- Floor rounding method
- Wrap overflow action

The `numerictype`, rounding method, and overflow action apply only during the quantization. The resulting value, quantized `y`, does not have any `fimath` attached to it.

`y = quantize(x,nt)` quantizes `x` to the specified `numerictype` `nt`. The rounding method and overflow action use default values.

`y = quantize(x,nt,rm)` quantizes `x` to the specified `numerictype`, `nt` and rounding method, `rm`. The overflow action uses the default value.

`y = quantize(x,nt,rm,oa)` quantizes `x` to the specified `numerictype`, `nt`, rounding method, `rm`, and overflow action, `oa`.

`yBP = quantize(x,s)` quantizes `x` to a binary-point, scaled fixed-point number. The `s` input specifies the sign to be used in `numericType` (`s,16,15`). Unspecified properties use these default values:

- `WordLength` 16
- `FractionLength` 15
- `RoundingMethod` Floor
- `OverflowAction` Wrap

`yBP = quantize(x,s,wl)` uses the specified word length, `wl`. The fraction length defaults to `wl-1`. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl)` uses the specified fraction length, `fl`. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl,rm)` uses the specified rounding method, `rm`. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl,rm,oa)` uses the specified overflow action, `oa`.

## Examples

### Quantize Binary-Point Scaled to Binary-Point Scaled Data

Create `numericType` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numericType(1,8,4);
```

Define the input.

```
x_BP = fi(pi)
```

```
x_BP =  
    3.1416
```

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

Use the defined `numericType`, `ntBP`, to quantize the input, `x_BP`, to a binary-point scaled data type.

```
yBP1 = quantize(x_BP, ntBP)
```

```
yBP1 =  
    3.1250
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 8  
    FractionLength: 4
```

### **Quantize Binary-Point Scaled to Slope-Bias Data**

Create a `numericType` object, `ntSB`, which specifies a slope-bias data type.

```
ntSB = numericType('Scaling', 'SlopeBias', ...  
    'SlopeAdjustmentFactor', 1.8, 'Bias', ...  
    1, 'FixedExponent', -12);
```

Define the input.

```
x_BP = fi(pi)
```

```
x_BP =  
    3.1416
```

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

Use the defined `numericType`, `ntSB`, to quantize the input, `x_BP`, to a slope-bias data type.

```
ySB1 = quantize(x_BP, ntSB)
```

```
ySB1 =  
    3.1415
```

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

```

WordLength: 16
Slope: 0.000439453125
Bias: 1

```

### Quantize Slope-Bias Scaled to Binary-Point Scaled Data

Create a `numerictype` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numerictype(1,8,4);
```

Define the input.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```

x_SB=5x3 object
    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003

```

```

DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 3.0517578125e-5
Bias: -0.125

```

Use the defined `numerictype`, `ntBP`, to quantize the input, `x_SB`, to a binary point scaled data type.

```
yBP2 = quantize(x_SB,ntBP,'Nearest','Saturate')
```

```

yBP2=5x3 object
    0.8125    0.1250    0.1875
    0.8750    0.2500    0.8750
    0.1250    0.5625    0.8750
    0.8750    0.8750    0.5000
    0.6250    0.8750    0.8125

```

```

DataTypeMode: Fixed-point: binary point scaling

```

```
Signedness: Signed
WordLength: 8
FractionLength: 4
```

### Quantize Slope-Bias Scaled to Slope-Bias Scaled Data

Create a `numerictype` object, `ntSB`, which specifies a slope-bias data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
    'SlopeAdjustmentFactor',1.8,'Bias',...
    1,'FixedExponent',-12);
```

Define the input.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```
x_SB=5x3 object
    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003
```

```
DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
    Slope: 3.0517578125e-5
    Bias: -0.125
```

Use the defined `numerictype`, `ntSB`, to quantize the input, `x_SB`, to a slope-bias data type.

```
ySB2 = quantize(x_SB,ntSB,'Ceiling','Wrap')
```

```
ySB2=5x3 object
    0.8150    0.0978    0.1580
    0.8752    0.2789    0.8752
    0.1272    0.5469    0.8752
    0.8752    0.8752    0.4854
    0.6326    0.8752    0.8005
```

```

DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 0.000439453125
Bias: 1

```

### Quantize Built-in Integer to Binary-Point Scaled Data

Create a `numericType` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numericType(1,8,4);
```

Define the input.

```
xInt = int8(-16:4:16)
```

```
xInt = 1x9 int8 row vector
```

```
-16  -12  -8   -4   0   4   8  12  16
```

Use the defined `numericType`, `ntBP`, to quantize the input `|xInt|` to a binary point scaled data type.

```
yBP3 = quantize(xInt,ntBP,'Zero')
```

```
yBP3=1x9 object
```

```
0   4  -8  -4   0   4  -8  -4   0
```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 8
FractionLength: 4

```

Show the range of the quantized output.

```
range(yBP3)
```

```
ans=1x2 object
```

```
-8.0000  7.9375
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 8
FractionLength: 4
```

The first two and last three values are wrapped because they are outside the representable range of the output type.

### Quantize Built-in Integer to Slope-Bias Data

Create a `numericType` object `ntSB`, which specifies a slope-bias data type.

```
ntSB = numericType('Scaling', 'SlopeBias', ...
    'SlopeAdjustmentFactor', 1.8, 'Bias', ...
    1, 'FixedExponent', -12);
```

Define the input.

```
xInt = int8(-16:4:16)
xInt = 1x9 int8 row vector
    -16    -12     -8     -4      0       4       8      12      16
```

Use the defined `numericType`, `ntSB`, to quantize the input, `xInt`, to a slope-bias data type.

```
ySB3 = quantize(xInt, ntSB, 'Round', 'Saturate')
ySB3=1x9 object
Columns 1 through 7
-13.4000 -11.9814 -7.9877 -3.9939 -0.0002  3.9936  7.9873
Columns 8 through 9
 11.9811  15.3996

DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 0.000439453125
```



```
Bias: 1
```

Show the range of the quantized output.

```
range(ySB3)
```

```
ans=1×2 object
   -13.4000    15.3996
```

```
DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 0.000439453125
Bias: 1
```

The first and last values saturate because they are at the limits of the representable range of the output type.

## Input Arguments

### **x** — Input data

fi objects or built-in integers

Input data to quantize. Valid inputs are:

- Built-in signed or unsigned integers (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`)
- Binary point scaled fixed-point fi
- Slope-bias scaled fixed-point fi

Although `fi` doubles and `fi` singles are allowed as inputs, they pass through the `quantize` function without being quantized.

### **nt** — Numeric type

(`true`, 16, 15) (default)

Numeric type object that defines the sign, word length, and fraction length of a fixed-point number.

**rm — Rounding method**

Floor (default) | Ceiling | Convergent | Nearest | Round | Zero

Rounding method to use

**oa — Overflow action**

Wrap (default) | Saturate

Action to take when a data overflow occurs

**s — Signedness**

true (default) | false

Whether the fixed-point number is signed (`true`) or unsigned (`false`)

**wl — Word length**

16 (default)

Word length of the fixed-point number

**fl — Fraction length**

15 (default)

Fraction length of the fixed-point number

## Output Arguments

**y — Quantized output**

fi object

Quantized value of the input

**yBP — Quantized output**

fi object

Input quantized to binary-point scaled value

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fi` | `fimath` | `fixed.Quantizer` | `numericType`

### Topics

“Compute Quantization Error”

**Introduced before R2006a**

## quantizenumeric

Quantize numeric data

### Syntax

```
y = quantizenumeric(x, s, w, f, r, o)
y = quantizenumeric(x, s, w, f, r)
```

### Description

`y = quantizenumeric(x, s, w, f, r, o)` quantizes the value specified in `x` based on the numeric type information specified in `s`, `w`, `f`, `r`, and `o`.

`y = quantizenumeric(x, s, w, f, r)` quantizes the value specified in `x` using the numeric type information specified in `s`, `w`, `f`, and `r`.

### Examples

#### Quantize value of pi

Quantize the value of pi using specified numeric type properties.

```
x = pi;
y = quantizenumeric(x,1,16,13,'ceil')

y = 3.1416
```

Specify a different rounding method. Observe how it affects the quantized value.

```
x = pi;
y = quantizenumeric(x,1,16,13,'fix')

y = 3.1415
```

## Input Arguments

### **x — Value to quantize**

scalar | vector | array

The value to quantize, specified as a scalar, vector, matrix or multidimensional array.

Data Types: double

### **s — signedness**

1 | 0

The signedness of the quantized value, specified as either 0 (unsigned) or 1 (signed).

Data Types: double

### **w — word length**

scalar integer

The word length of the quantized value, specified as a scalar integer.

Data Types: double

### **f — fraction length**

scalar integer

The fraction length of the quantized value, specified as a scalar integer.

Data Types: double

### **r — Rounding method**

character vector

Rounding method to use for quantization, specified as one of the following:

- `ceil`— Round towards positive infinity (same as 'ceiling')
- `ceiling`— Round towards positive infinity (same as 'ceil')
- `convergent`— Convergent rounding
- `fix`— Round towards zero (same as 'zero')
- `floor`— Round towards negative infinity
- `nearest`— Round towards nearest with ties rounding towards positive infinity

- `round`— Round towards nearest with ties rounding up in absolute value
- `zero`— Round towards zero (same as 'fix')

Data Types: char

### **o — Overflow action**

`saturate` | `wrap`

Overflow action to use for quantization, specified as either 'saturate' or 'wrap'. When no overflow action is specified, `quantize` numeric uses `saturate`.

Example:

Data Types: char

## **Output Arguments**

### **y — quantized output value**

scalar | vector | matrix | multidimensional array

The quantized output value. `y` always has the same dimensions as `x`, and is always a double.

## **See Also**

`cast` | `fi` | `fimath` | `fixed.Quantizer` | `numericType` | `quantize` | `quantizer`

**Introduced in R2016a**

# quantize method

Apply quantizer object to data

## Syntax

```
y = quantize(q, x)
[y1,y2,...] = quantize(q,x1,x2,...)
```

## Description

`y = quantize(q, x)` uses the quantizer object `q` to quantize `x`. When `x` is a numeric array, each element of `x` is quantized. When `x` is a cell array, each numeric element of the cell array is quantized. When `x` is a structure, each numeric field of `x` is quantized. Quantize does not change nonnumeric elements or fields of `x`, nor does it issue warnings for nonnumeric values. The output `y` is a built-in double. When the input `x` is a structure or cell array, the fields of `y` are built-in doubles.

`[y1,y2,...] = quantize(q,x1,x2,...)` is equivalent to `y1 = quantize(q,x1), y2 = quantize(q,x2),...`

The quantizer object states

- `max` — Maximum value before quantizing
- `min` — Minimum value before quantizing
- `noverflows` — Number of overflows
- `nunderflows` — Number of underflows
- `noperations` — Number of quantization operations

are updated during the call to `quantize`, and running totals are kept until a call to `resetlog` is made.

## Examples

### Custom Precision Floating-Point

The following example demonstrates using `quantize` to quantize data.

```
u=linspace(-15, 15, 1000);  
q=quantizer([6 3], 'float');  
range(q)
```

```
ans = 1×2
```

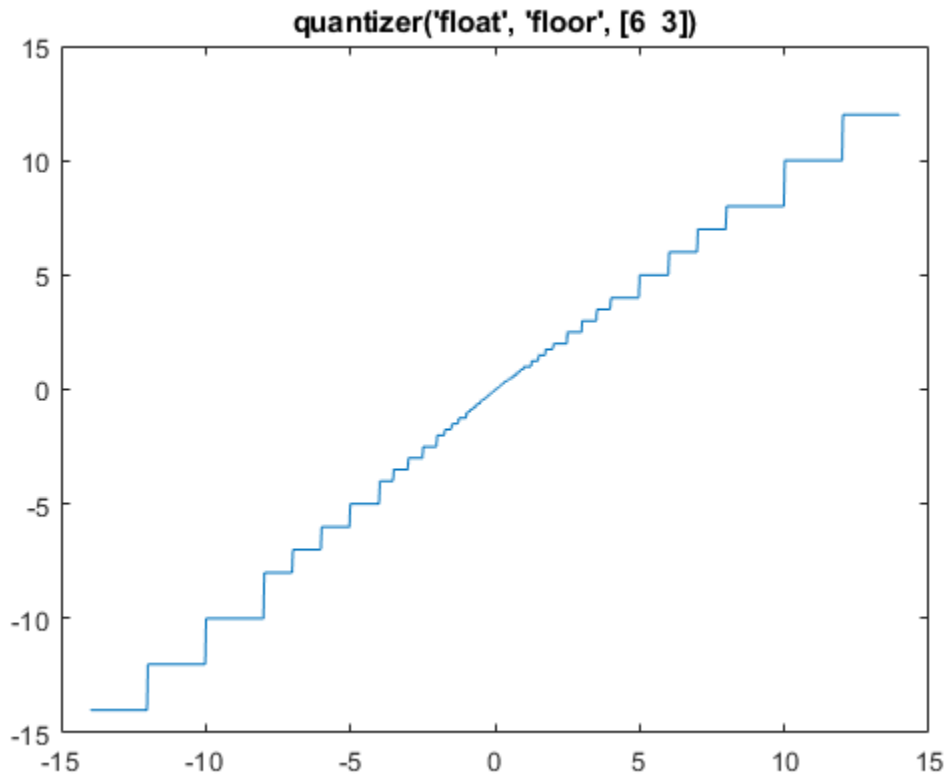
```
    -14     14
```

```
y=quantize(q, u);
```

```
Warning: 68 overflow(s) occurred in the fi quantize operation.
```

```
plot(u, y); title(tostring(q))
```





### Fixed-Point

The following example demonstrates using `quantize` to quantize data.

```
u=linspace(-15, 15, 1000);  
q=quantizer([6 2], 'wrap');  
range(q)
```

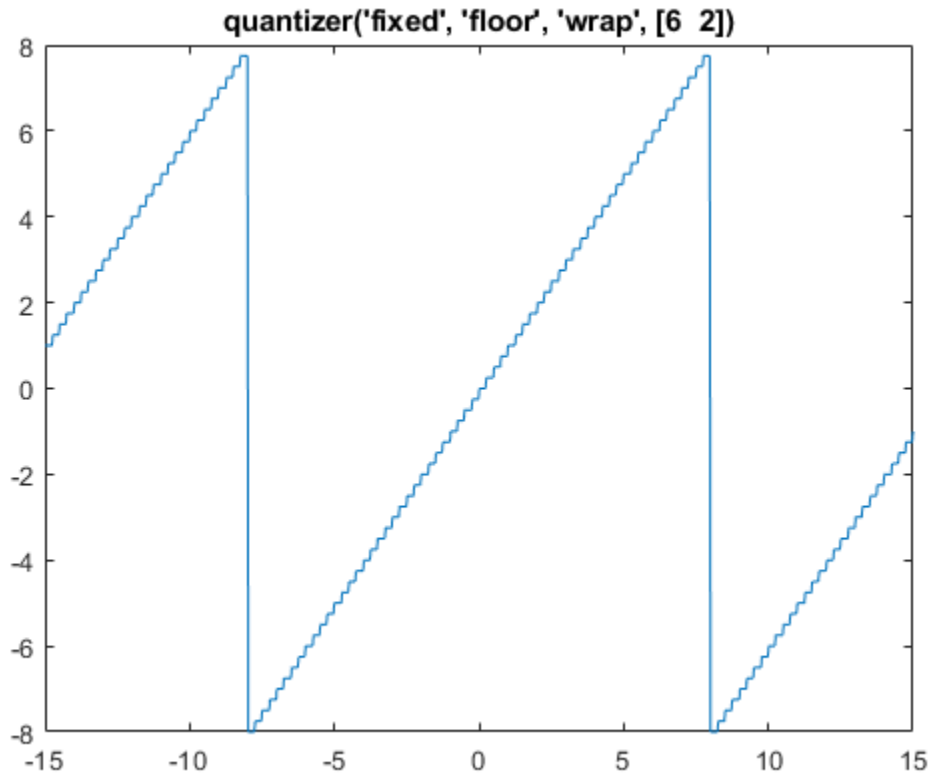
```
ans = 1x2
```

```
-8.0000    7.7500
```

```
y=quantize(q, u);
```

```
Warning: 468 overflow(s) occurred in the fi quantize operation.
```

```
plot(u, y); title(tostring(q))
```



## See Also

[assignmentquantizer](#) | [quantizer](#) | [set](#) | [unitquantize](#) | [unitquantizer](#)

**Introduced in R2012b**

# quantizer

Construct quantizer object

## Syntax

```
q = quantizer
q = quantizer('PropertyName',PropertyValue1,...)
q = quantizer(PropertyValue1,PropertyValue2,...)
q = quantizer(struct)
q = quantizer(pn,pv)
```

## Description

`q = quantizer` creates a `quantizer` object with properties set to their default values. To use this object to quantize values, use the `quantize` method.

`q = quantizer('PropertyName',PropertyValue1,...)` uses property name/property value pairs.

`q = quantizer(PropertyValue1,PropertyValue2,...)` creates a `quantizer` object with the listed property values. When two values conflict, `quantizer` sets the last property value in the list. Property values are unique; you can set the property names by specifying just the property values in the command.

`q = quantizer(struct)`, where `struct` is a structure whose field names are property names, sets the properties named in each field name with the values contained in the structure.

`q = quantizer(pn,pv)` sets the named properties specified in the cell array of character vectors `pn` to the corresponding values in the cell array `pv`.

The `quantizer` object property values are listed below. These properties are described in detail in “`quantizer` Object Properties” on page 3-7.

<b>Property Name</b>	<b>Property Value</b>	<b>Description</b>
mode	'double'	Double-precision mode. Override all other parameters.
	'float'	Custom-precision floating-point mode.
	'fixed'	Signed fixed-point mode.
	'single'	Single-precision mode. Override all other parameters.
	'ufixed'	Unsigned fixed-point mode.
roundmode	'ceil'	Round toward positive infinity.
	'convergent'	Round to nearest integer with ties rounding to nearest even integer.
	'fix'	Round toward zero.
	'floor'	Round toward negative infinity.
	'Nearest'	Round to nearest integer with ties rounding toward positive infinity.
	'Round'	Round to nearest integer with ties rounding to nearest integer with greater absolute value.
overflowmode (fixed-point only)	'saturate'	Saturate on overflow.
	'wrap'	Wrap on overflow.

Property Name	Property Value	Description
format	[wordlength fractionlength]	Format for fixed or ufixed mode.
	[wordlength exponentlength]	Format for float mode.

The default property values for a quantizer object are

```
DataMode = fixed
RoundMode = floor
OverflowMode = saturate
Format = [16 15]
```

Along with the preceding properties, quantizer objects have read-only states: max, min, noverflows, nunderflows, and noperations. They can be accessed through quantizer/get or q.maxlog, q.minlog, q.noverflows, q.nunderflows, and q.noperations, but they cannot be set. They are updated during the quantizer/quantize method, and are reset by the resetlog function.

The following table lists the read-only quantizer object states:

Property Name	Description
max	Maximum value before quantizing
min	Minimum value before quantizing
noverflows	Number of overflows
nunderflows	Number of underflows
noperations	Number of data points quantized

## Examples

The following example operations are equivalent.

Setting quantizer object properties by listing property values only in the command,

```
q = quantizer('fixed', 'Ceiling', 'Saturate', [5 4])
```

Using a structure struct to set quantizer object properties,

```
struct.mode = 'fixed';
struct.roundmode = 'ceil';
struct.overflowmode = 'saturate';
struct.format = [5 4];
q = quantizer(struct);
```

Using property name and property value cell arrays `pn` and `pv` to set quantizer object properties,

```
pn = {'mode', 'roundmode', 'overflowmode', 'format'};
pv = {'fixed', 'ceil', 'saturate', [5 4]};
q = quantizer(pn, pv)
```

Using property name/property value pairs to configure a quantizer object,

```
q = quantizer('mode', 'fixed', 'roundmode', 'ceil', ...
'overflowmode', 'saturate', 'format', [5 4]);
```

### See Also

[assignmentquantizer](#) | [fi](#) | [fimath](#) | [fipref](#) | [numerictype](#) | [quantize](#) | [quantizenumeric](#) | [set](#) | [unitquantize](#) | [unitquantizer](#)

**Introduced before R2006a**

# randquant

Generate uniformly distributed, quantized random number using quantizer object

## Syntax

```
randquant(q,n)
randquant(q,m,n)
randquant(q,m,n,p,...)
randquant(q,[m,n])
randquant(q,[m,n,p,...])
```

## Description

`randquant(q,n)` uses quantizer object `q` to generate an `n`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer object. When `q` is a floating-point quantizer object, `randquant` populates the `n`-by-`n` array with values covering the range

-[square root of `realmax(q)`] to [square root of `realmax(q)`]

`randquant(q,m,n)` uses quantizer object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer object. When `q` is a floating-point quantizer object, `randquant` populates the `m`-by-`n` array with values covering the range

-[square root of `realmax(q)`] to [square root of `realmax(q)`]

`randquant(q,m,n,p,...)` uses quantizer object `q` to generate an `m`-by-`n`-by-`p`-by ... matrix with random entries whose values cover the range of `q` when `q` is fixed-point quantizer object. When `q` is a floating-point quantizer object, `randquant` populates the matrix with values covering the range

-[square root of `realmax(q)`] to [square root of `realmax(q)`]

`randquant(q,[m,n])` uses quantizer object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer object. When `q` is a floating-point quantizer object, `randquant` populates the `m`-by-`n` array with values covering the range

-[square root of `realmax(q)`] to [square root of `realmax(q)`]

`randquant(q, [m, n, p, ...])` uses quantizer object `q` to generate `p` `m`-by-`n` matrices containing random entries whose values cover the range of `q` when `q` is a fixed-point quantizer object. When `q` is a floating-point quantizer object, `randquant` populates the `m`-by-`n` arrays with values covering the range

-[square root of `realmax(q)`] to [square root of `realmax(q)`]

`randquant` produces pseudorandom numbers. The number sequence `randquant` generates during each call is determined by the state of the generator. Because MATLAB resets the random number generator state at startup, the sequence of random numbers generated by the function remains the same unless you change the state.

`randquant` works like `rng` in most respects.

## Examples

```
q=quantizer([4 3]);  
rng('default')  
randquant(q,3)
```

```
ans =
```

```
    0.5000    0.6250   -0.5000  
    0.6250    0.1250         0  
   -0.8750   -0.8750    0.7500
```

## See Also

[quantizer](#) | [rand](#) | [range](#) | [realmax](#)

**Introduced before R2006a**



## range

Numerical range of `fi` or quantizer object

### Syntax

```
range(a)
[min_val, max_val] = range(a)
r = range(q)
[min_val, max_val] = range(q)
```

### Description

`range(a)` returns a `fi` object with the minimum and maximum possible values of `fi` object `a`. All possible quantized real-world values of `a` are in the range returned. If `a` is a complex number, then all possible values of `real(a)` and `imag(a)` are in the range returned.

`[min_val, max_val] = range(a)` returns the minimum and maximum values of `fi` object `a` in separate output variables.

`r = range(q)` returns the two-element row vector  $r = [a \ b]$  such that for all real  $x$ ,  $y = \text{quantize}(q, x)$  returns  $y$  in the range  $a \leq y \leq b$ .

`[min_val, max_val] = range(q)` returns the minimum and maximum values of the range in separate output variables.

### Examples

```
q = quantizer('float', [6 3]);
r = range(q)

r =

    -14     14
q = quantizer('fixed', [4 2], 'floor');
```

```
[min_val,max_val] = range(q)
```

```
min_val =
```

```
-2
```

```
max_val =
```

```
1.7500
```

## Algorithms

If  $q$  is a floating-point quantizer object,  $a = -\text{realmax}(q)$ ,  $b = \text{realmax}(q)$ .

If  $q$  is a signed fixed-point quantizer object (`datamode = 'fixed'`),

$$a = -\text{realmax}(q) - \text{eps}(q) = \frac{-2^{w-1}}{2^f}$$

$$b = \text{realmax}(q) = \frac{2^{w-1} - 1}{2^f}$$

If  $q$  is an unsigned fixed-point quantizer object (`datamode = 'ufixed'`),

$$a = 0$$

$$b = \text{realmax}(q) = \frac{2^w - 1}{2^f}$$

See `realmax` for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

eps | exponentmax | exponentmin | fractionlength | intmax | intmin |  
lowerbound | lsb | max | min | realmax | realmin | upperbound

**Introduced before R2006a**

## **rdivide, ./**

**Package:** embedded

Right-array division

### **Syntax**

```
X = A./B
X = rdivide(A, B)
```

### **Description**

$X = A ./ B$  performs right-array division by dividing each element of  $A$  by the corresponding element of  $B$ .

$X = \text{rdivide}(A, B)$  is an alternative way to execute  $X = A ./ B$ .

### **Examples**

#### **Perform Right-Array Division of Two Matrices**

In this example, you perform right-array division on a 3-by-3 magic square of `fi` objects. Each element of the 3-by-3 magic square is divided by the corresponding element in the 3-by-3 input array `b`.

The `mrdivide` function outputs a 3-by-3 array of signed `fi` objects, each of which has a word length of 16 bits and fraction length of 11 bits.

```
a = fi(magic(3))
```

```
a=3×3 object
    8     1     6
    3     5     7
    4     9     2
```

```

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

```

```
b = int8([3 3 4; 1 2 4 ; 3 1 2 ])
```

```
b = 3x3 int8 matrix
```

```

    3    3    4
    1    2    4
    3    1    2

```

```
c = a./b
```

```
c=3x3 object
```

```

    2.6665    0.3335    1.5000
    3.0000    2.5000    1.7500
    1.3335    9.0000    1.0000

```

```

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

```

## Input Arguments

### A — Numerator

scalar | vector | matrix | multidimensional array

If inputs A and B are not the same size, one of them must be a scalar value.

If A is complex, the real and imaginary parts of A are independently divided by B.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
 uint32 | uint64 | logical | fi  
 Complex Number Support: Yes

### B — Denominator

scalar | vector | matrix | multidimensional array

If inputs A and B are not the same size, one of them must be a scalar value.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

## Output Arguments

### X — Quotient

scalar | vector | matrix | multidimensional array

Quotient, returned as a scalar, vector, matrix, or multidimensional array. The following table shows the rules used to assign property values to the output of the `rdivide` function.

Output Property	Rule
Signedness	If either input is Signed, the output is Signed. If both inputs are Unsigned, the output is Unsigned.
WordLength	The output word length equals the maximum of the input word lengths.
FractionLength	For $c = a./b$ , the fraction length of output $c$ equals the fraction length of $a$ minus the fraction length of $b$ .

## Algorithms

The following table shows the rules the `rdivide` function uses to handle inputs with different data types.

Case	Rule
Interoperation of <code>fi</code> objects and built-in integers	Built-in integers are treated as fixed-point objects. For example, $B = \text{int8}(2)$ is treated as an <code>s8,0 fi</code> object.

Case	Rule
Interoperation of <code>fi</code> objects and constants	MATLAB for code generation treats constant integers as fixed-point objects with the same word length as the <code>fi</code> object and a fraction length of 0.
Interoperation of mixed data types	<p>Similar to all other <code>fi</code> object functions, when inputs <code>a</code> and <code>b</code> have different data types, the data type with the higher precedence determines the output data type. The order of precedence is as follows:</p> <ol style="list-style-type: none"> <li>1 ScaledDouble</li> <li>2 Fixed-point</li> <li>3 Built-in double</li> <li>4 Built-in single</li> </ol> <p>When both inputs are <code>fi</code> objects, the only data types that are allowed to mix are <code>ScaledDouble</code> and <code>Fixed-point</code>.</p>

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`add` | `divide` | `fi` | `fimath` | `mrdivide` | `numericType` | `sub` | `sum`

**Introduced in R2009a**

## realmax

Largest positive fixed-point value or quantized number

### Syntax

```
realmax(a)  
realmax(q)
```

### Description

`realmax(a)` is the largest real-world value that can be represented in the data type of `fi` object `a`. Anything larger overflows.

`realmax(q)` is the largest quantized number that can be represented where `q` is a quantizer object. Anything larger overflows.

### Examples

```
q = quantizer('float',[6 3]);  
x = realmax(q)
```

```
x =
```

```
14
```

### Algorithms

If `q` is a floating-point quantizer object, the largest positive number, `x`, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point quantizer object, the largest positive number, `x`, is



$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point quantizer object (`datamode = 'ufixed'`), the largest positive number, `x`, is

$$x = \frac{2^w - 1}{2^f}$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`eps` | `exponentmax` | `exponentmin` | `fractionlength` | `intmax` | `intmin` | `lowerbound` | `lsb` | `quantizer` | `range` | `realmin` | `upperbound`

**Introduced before R2006a**

## realmin

Smallest positive normalized fixed-point value or quantized number

### Syntax

```
x=realmin(a)
x=realmin(q)
```

### Description

`x=realmin(a)` is the smallest positive real-world value that can be represented in the data type of `fi` object `a`. Anything smaller than `x` underflows or is an IEEE “denormal” number.

`x=realmin(q)` is the smallest positive normal quantized number where `q` is a quantizer object. Anything smaller than `x` underflows or is an IEEE “denormal” number.

### Examples

```
q = quantizer('float',[6 3]);
x = realmin(q)
```

```
x =
```

```
0.2500
```

### Algorithms

If `q` is a floating-point quantizer object,  $x = 2^{E_{min}}$  where  $E_{min} = \text{exponentmin}(q)$  is the minimum exponent.

If `q` is a signed or unsigned fixed-point quantizer object,  $x = 2^{-f} = \varepsilon$  where  $f$  is the fraction length.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[eps](#) | [exponentmax](#) | [exponentmin](#) | [fractionlength](#) | [intmax](#) | [intmin](#) | [lowerbound](#) | [lsb](#) | [range](#) | [realmax](#) | [upperbound](#)

**Introduced before R2006a**

## reinterpretpcast

Convert fixed-point data types without changing underlying data

### Syntax

```
c = reinterpretpcast(a, T)
```

### Description

`c = reinterpretpcast(a, T)` converts the input `a` to the data type specified by `numericType` object `T` without changing the underlying data. The result is returned in `fi` object `c`.

The input `a` must be a built-in integer or a `fi` object with a fixed-point data type. `T` must be a `numericType` object with a fully specified fixed-point data type. The word length of inputs `a` and `T` must be the same.

The `reinterpretpcast` function differs from the MATLAB `typecast` and `cast` functions in that it only operates on `fi` objects and built-in integers, and it does not allow the word length of the input to change.

### Examples

In the following example, `a` is a signed `fi` object with a word length of 8 bits and a fraction length of 7 bits. The `reinterpretpcast` function converts `a` into an unsigned `fi` object `c` with a word length of 8 bits and a fraction length of 0 bits. The real-world values of `a` and `c` are different, but their binary representations are the same.

```
a = fi([-1 pi/4], 1, 8, 7)
T = numericType(0, 8, 0);
c = reinterpretpcast(a, T)
a =
```

```
-1.0000    0.7891
```

```

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

```

c =

```

    128    101

```

```

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

```

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

```

binary_a = bin(a)
binary_c = bin(c)
binary_a =

10000000    01100101

binary_c =

10000000    01100101

```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

cast | fi | numerictype | typecast

**Introduced in R2008b**

## removefimath

Remove fimath object from fi object

### Syntax

```
y = removefimath(x)
```

### Description

`y = removefimath(x)` returns a `fi` object `y` with `x`'s `numericType` and value, and no `fimath` object attached. You can use this function as `y = removefimath(y)`, which gives you localized control over the `fimath` settings. This function also is useful for preventing errors about embedded `.fimath` of both operands needing to be equal.

### Examples

#### Remove fimath Object from fi Object

This example shows how to define a `fi` object, define a `fimath` object, attach the `fimath` object to the `fi` object and then, remove the attached `fimath` object.

```
a = fi(pi)

a =
    3.1416

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

f = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
a = setfimath(a,f)

a =
    3.1416
```

```

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

        RoundingMethod: Floor
        OverflowAction: Wrap
          ProductMode: FullPrecision
          SumMode: FullPrecision

b = removefimath(a)

b =
    3.1416

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

```

### Set and Remove fimath for Code Generation

Use the pattern `x = setfimath(x,f)` and `y = removefimath(y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```

function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
    f = fimath('OverflowAction','Wrap',...
              'RoundingMethod','Floor',...
              'SumMode','KeepLSB',...
              'SumWordLength',32);
    a = setfimath(a,f);
    b = setfimath(b,f);
    y = a + b;
    y = removefimath(y);
end

```

If you have the MATLAB Coder product, you can generate C code. This example generates C code on a computer with 32-bit, native integer type.

```

a = fi(0,1,16,15);
b = fi(0,1,16,15);

```

```
codegen -config:lib fixed_point_32bit_KeepLSB_plus_example...  
        -args {a,b} -launchreport
```

```
int fixed_point_32bit_KeepLSB_plus_example(short a, short b)  
{  
    return a + b;  
}
```

## Input Arguments

### **x** — Input data

fi object | built-in integer | double | single

Input data, specified as a **fi** object or built-in integer, from which to copy the data type and value to the output. **x** must be a **fi** object or an integer data type (**int8**, **int16**, **int32**, **int64**, **uint8**, **uint16**, **uint32**, or **uint64**). If **x** is not a **fi** object or integer data type, then  $y = x$ .

## Output Arguments

### **y** — Output fi object

fi object | built-in integer | double | single

Output **fi** object, returned as a **fi** object with no **fimath** object attached. The data type and value of the output match the input. If the input, **x**, is not a **fi** object  $y = x$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

**fi** | **fimath** | **setfimath**



**Introduced in R2012b**

## rescale

Change scaling of `fi` object

### Syntax

```
b = rescale(a, fractionlength)
```

```
b = rescale(a, slope, bias)
```

```
b = rescale(a, slopeadjustmentfactor, fixedexponent, bias)
```

```
b = rescale(a, ..., PropertyName, PropertyValue, ...)
```

### Description

The `rescale` function acts similarly to the `fi` copy function with the following exceptions:

- The `fi` copy constructor preserves the real-world value, while `rescale` preserves the stored integer value.
- `rescale` does not allow the `Signed` and `WordLength` properties to be changed.

### Examples

In the following example, `fi` object `a` is rescaled to create `fi` object `b`. The real-world values of `a` and `b` are different, while their stored integer values are the same:

```
p = fipref('FimathDisplay','none',...  
          'NumericTypeDisplay','short');  
a = fi(10,1,8,3)
```

```
a =
```

```
    10  
    s8,3
```

```
b = rescale(a,1)

b =
    40
    s8,1

stored_integer_a = storedInteger(a);
stored_integer_b = storedInteger(b);
isequal(stored_integer_a, stored_integer_b)

ans =

1
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

fi

**Introduced before R2006a**

## reset

Reset objects to initial conditions

### Syntax

```
reset(P)  
reset(q)
```

### Description

reset(P) resets the `fipref` object P to its initial conditions.

reset(q) resets the following quantizer object properties to their initial conditions:

- `minlog`
- `maxlog`
- `noverflows`
- `nunderflows`
- `noperations`

### See Also

`resetlog`

**Introduced before R2006a**

# resetglobalfimath

Set global fimath to MATLAB factory default

## Syntax

```
resetglobalfimath
```

## Description

`resetglobalfimath` sets the global fimath to the MATLAB factory default in your current MATLAB session. The MATLAB factory default has the following properties:

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

## Examples

In this example, you create your own `fimath` object `F` and set it as the global fimath. Then, using the `resetglobalfimath` command, reset the global fimath to the MATLAB factory default setting.

```
F = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
globalfimath(F);
F1 = fimath
a = fi(pi)
```

```
F1 =
```

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

```
a =  
  
3.1416  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 13
```

Now, set the global `fimath` back to the factory default setting using `resetglobalfimath`:

```
resetglobalfimath;  
F2 = fimath  
a = fi(pi)  
  
F2 =  
  
    RoundingMethod: Nearest  
    OverflowAction: Saturate  
    ProductMode: FullPrecision  
    SumMode: FullPrecision  
  
a =  
  
3.1416  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: 13
```

You've now set the global `fimath` in your current MATLAB session back to the factory default setting. To use the factory default setting of the global `fimath` in future MATLAB sessions, you must use the `removeglobalfimathpref` command.

## Alternatives

`reset(G)` — If  $G$  is a handle to the global `fimath`, `reset(G)` is equivalent to using the `resetglobalfimath` command.

## See Also

`fimath` | `globalfimath` | `removeglobalfimathpref`

**Introduced in R2010a**

## removeglobalfimathpref

Remove global fimath preference

### Syntax

```
removeglobalfimathpref
```

### Description

`removeglobalfimathpref` removes your global fimath from the MATLAB preferences. Once you remove the global fimath from your preferences, you cannot save it to them again. It is best practice to remove global fimath from the MATLAB preferences so that you start each MATLAB session using the default `fimath` settings.

The `removeglobalfimathpref` function does not change the global fimath for your current MATLAB session. To revert back to the factory default setting of the global fimath in your current MATLAB session, use the `resetglobalfimath` command.

### Examples

#### Example 5.7. Removing Your Global fimath from the MATLAB Preferences

Typing

```
removeglobalfimathpref;
```

at the MATLAB command line removes your global fimath from the MATLAB preferences. Using the `removeglobalfimathpref` function allows you to:

- Continue using your global fimath in the current MATLAB session
- Use the MATLAB factory default setting of the global fimath in all future MATLAB sessions



To revert back to the MATLAB factory default setting of the global fimath in both your current and future MATLAB sessions, use both the `resetglobalfimath` and the `removeglobalfimathpref` commands:

```
resetglobalfimath;  
removeglobalfimath;
```

## See Also

`fimath` | `globalfimath` | `resetglobalfimath`

**Introduced in R2010a**

## **resetlog**

Clear log for `fi` or quantizer object

### **Syntax**

```
resetlog(a)  
resetlog(q)
```

### **Description**

`resetlog(a)` clears the log for `fi` object `a`.

`resetlog(q)` clears the log for quantizer object `q`.

Turn logging on or off by setting the `fipref` property `LoggingMode`.

### **See Also**

`fipref` | `maxlog` | `minlog` | `noperations` | `noverflows` | `nunderflows` | `reset`

**Introduced before R2006a**

## round

Round `fi` object toward nearest integer or round input data using `quantizer` object

### Syntax

```
y = round(a)
y = round(q,x)
```

### Description

`y = round(a)` rounds `fi` object `a` to the nearest integer. In the case of a tie, `round` rounds values to the nearest integer with greater absolute value. The rounded value is returned in `fi` object `y`.

`y` and `a` have the same `fi`math object and `DataType` property.

When the `DataType` of `a` is `single`, `double`, or `boolean`, the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numericType` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is 0, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is 2. If `a` is unsigned, then the minimum word length of `y` is 1.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`round` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

`y = round(q,x)` uses the `RoundingMethod` and `FractionLength` settings of `q` to round the numeric data `x`, but does not check for overflows during the operation. Input `x` must be a builtin numeric variable. Use the `cast` function to work with `fi` objects.

## Examples

### Example 1

The following example demonstrates how the `round` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 8  
      FractionLength: 3
```

```
y = round(a)
```

```
y =
```

```
3
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 6  
      FractionLength: 0
```

### Example 2

The following example demonstrates how the `round` function affects the `numericType` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025, 1, 8, 12)
```

```
a =
```

```
0.0249
```

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

```

Signedness: Signed
WordLength: 8
FractionLength: 12

```

```
y = round(a)
```

```
y =
```

```
0
```

```

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 2
FractionLength: 0

```

### Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

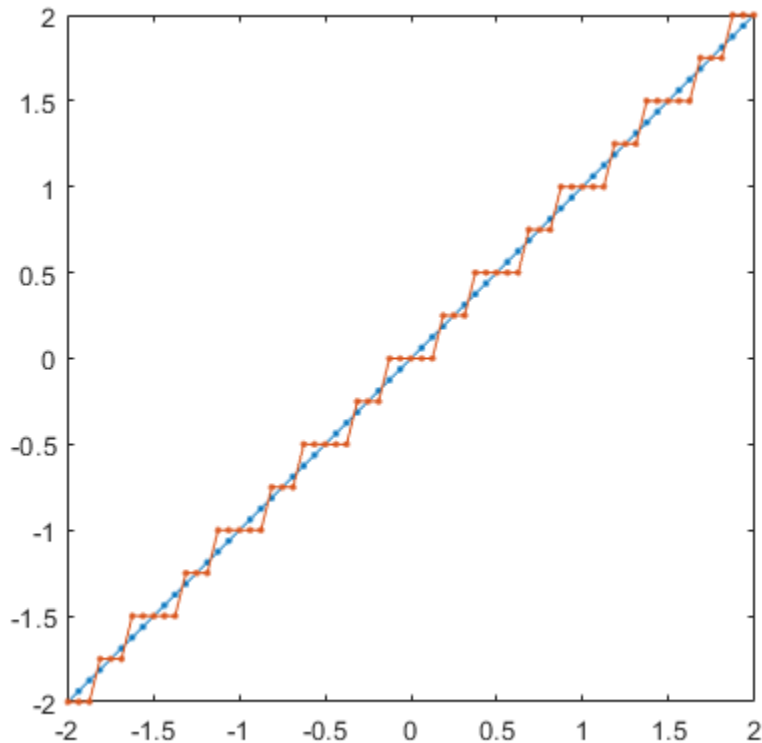
The following table illustrates these differences for a given `fi` object `a`.

<b>a</b>	<b>convergent(a)</b>	<b>nearest(a)</b>	<b>round(a)</b>
-3.5	-4	-3	-4
-2.5	-2	-2	-3
-1.5	-2	-1	-2
-0.5	0	0	-1
0.5	0	1	1
1.5	2	2	2
2.5	2	3	3
3.5	4	4	4

## Quantize an input

Create a quantizer object, and use it to quantize input data. The quantizer object applies its properties to the input data to return quantized output.

```
q = quantizer('fixed', 'convergent', 'wrap', [3 2]);  
x = (-2:eps(q)/4:2)';  
y = round(q,x);  
plot(x,[x,y],'.-');  
axis square;
```



Applying quantizer object `q` to the data resulted in a staircase-shape output plot. Linear data input results in output where `y` shows distinct quantization levels.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[ceil](#) | [convergent](#) | [fix](#) | [floor](#) | [nearest](#) | [quantize](#) | [quantizer](#)

**Introduced before R2006a**

## rsqrt

Reciprocal square root

### Syntax

```
Y = rsqrt(X)
```

### Description

`Y = rsqrt(X)` returns the reciprocal square root of each element of the half-precision input array, `X`.

---

**Note** This function supports only half-precision inputs.

---

### Examples

#### Reciprocal Square Root of Matrix Elements

Create a matrix of half-precision values.

```
X = half(magic(3))
```

```
X =
```

```
3x3 half matrix
```

```
    8    1    6
    3    5    7
    4    9    2
```

Compute the reciprocal square root of each element of `X`.

```
y = rsqrt(X)
```



y =

3x3 half matrix

0.3535	1.0000	0.4082
0.5771	0.4473	0.3779
0.5000	0.3333	0.7070

## Input Arguments

### **X** — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a half-precision numeric scalar, vector, matrix, or multidimensional array

Data Types: Half

## See Also

half

**Introduced in R2018b**

## **savefipref**

Save `fi` preferences for next MATLAB session

### **Syntax**

```
savefipref
```

### **Description**

`savefipref` saves the settings of the current `fipref` object for the next MATLAB session.

### **See Also**

`fipref`

**Introduced before R2006a**

## sdec

Signed decimal representation of stored integer of `fi` object

## Syntax

`sdec(a)`

## Description

Fixed-point numbers can be represented as

$$\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}$$

or, equivalently as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

`sdec(a)` returns the stored integer of `fi` object `a` in signed decimal format.

## Examples

The code

```
a = fi([-1 1],1,8,7);  
sdec(a)
```

returns

```
-128 127
```

## **See Also**

`bin` | `dec` | `hex` | `oct` | `storedInteger`

**Introduced before R2006a**

---

## set

Set or display property values for quantizer objects

### Syntax

```
set(q, PropertyValue1, PropertyValue2, ...)
```

```
set(q, s)
```

```
set(q, pn, pv)
```

```
set(q, 'PropertyName1', PropertyValue1, 'PropertyName2',  
    PropertyValue2, ...)
```

```
q.PropertyName = Value
```

```
s = set(q)
```

### Description

`set(q, PropertyValue1, PropertyValue2, ...)` sets the properties of quantizer object `q`. If two property values conflict, the last value in the list is the one that is set.

`set(q, s)`, where `s` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(q, pn, pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

`set(q, 'PropertyName1', PropertyValue1, 'PropertyName2',  
 PropertyValue2, ...)` sets multiple property values with a single statement.

---

**Note** You can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

---

`q.PropertyName = Value` uses dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of quantizer object `q`.

`s = set(q)` returns a structure containing the possible values for the properties of quantizer object `q`.

---

**Note** The `set` function operates on quantizer objects. To learn about setting the properties of other objects, see properties of `fi`, `fimath`, `fipref`, and `numericType` objects.

---

## See Also

`get`

**Introduced before R2006a**

## setfimath

Attach fimath object to fi object

### Syntax

```
y = setfimath(x,f)
```

### Description

`y = setfimath(x,f)` returns a `fi` object, `y`, with `x`'s `numericType` and value, and attached `fimath` object, `f`. This function and the related `removefimath` function are useful for preventing errors about `embedded.fimath` of both operands needing to be equal.

The `y = setfimath(x,f)` syntax does not modify the input, `x`. To modify `x`, use `x = setfimath(x,f)`. If you use `setfimath` in an expression, such as `a*setfimath(b,f)`, the `fimath` object is used in the temporary variable, but `b` is not modified.

### Examples

#### Add fimath object to fi Object

Define a `fi` object, define a `fimath` object, and use `setfimath` to attach the `fimath` object to the `fi` object.

Create a `fi` object without a `fimath` object.

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

```
a =  
    3.1416
```

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

```
WordLength: 16
FractionLength: 13
```

Create a `fimath` object and attach it to the `fi` object.

```
f = fimath('OverflowAction','Wrap','RoundingMethod','Floor');
b = setfimath(a,f)
```

```
b =
    3.1416
```

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

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

### Set and Remove `fimath` for Code Generation

Use the pattern `x = setfimath(x,f)` and `y = removefimath(y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```
function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
    f = fimath('OverflowAction','Wrap',...
        'RoundingMethod','Floor',...
        'SumMode','KeepLSB',...
        'SumWordLength',32);
    a = setfimath(a,f);
    b = setfimath(b,f);
    y = a + b;
    y = removefimath(y);
end
```

If you have the MATLAB Coder product, you can generate C code. This example generates C code on a computer with 32-bit, native integer type.



```

a = fi(0,1,16,15);
b = fi(0,1,16,15);
codegen -config:lib fixed_point_32bit_KeepLSB_plus_example...
        -args {a,b} -launchreport

```

```

int fixed_point_32bit_KeepLSB_plus_example(short a, short b)
{
    return a + b;
}

```

## Input Arguments

### **x** — Input data

fi object | built-in integer | double | single

Input data, specified as a `fi` object or built-in integer value, from which to copy the data type and value to the output. `x` must be a `fi` object or an integer data type (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`). Otherwise, the `fimath` object is not applied. If `x` is not a `fi` object or integer data type,  $y = x$ .

### **f** — Input `fimath` object

`fimath` object

Input `fimath` object, specified as an existing `fimath` object to attach to the output. An error occurs if `f` is not a `fimath` object.

## Output Arguments

### **y** — Output `fi` object

`fi` object

Output `fi` object, returned as a `fi` object with the same data type and value as the `x` input. `y` also has attached `fimath` object, `f`. If the input, `x`, is not a `fi` object or integer data type, then  $y = x$ .

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`fi` | `fimath` | `removefimath`

**Introduced in R2012b**

# sfi

Construct signed fixed-point numeric object

## Syntax

```
a = sfi
a = sfi(v)
a = sfi(v,w)
a = sfi(v,w,f)
a = sfi(v,w,slope,bias)
a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

## Description

You can use the `sfi` constructor function in the following ways:

- `a = sfi` is the default constructor and returns a signed `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = sfi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = sfi(v,w)` returns a signed fixed-point object with value `v`, word length `w`, and best-precision fraction length.
- `a = sfi(v,w,f)` returns a signed fixed-point object with value `v`, word length `w`, and fraction length `f`.
- `a = sfi(v,w,slope,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slope`, and `bias`.
- `a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`fi` objects created by the `sfi` constructor function have the following general types of properties:

- “Data Properties” on page 5-814
- “fimath Properties” on page 5-752
- “numerictype Properties” on page 5-815

These properties are described in detail in “fi Object Properties” on page 3-2 in the Properties Reference.

---

**Note** `fi` objects created by the `sfi` constructor function have no local `fimath`.

---

### 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
- `hex` — Stored integer value of a `fi` object in hexadecimal
- `int` — Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type. You can also use `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64` to get the stored integer value of a `fi` object in these formats
- `oct` — Stored integer value of a `fi` object in octal

These properties are described in detail in “fi Object Properties” on page 3-2.

### fimath Properties

When you create a `fi` object with the `sfi` constructor function, that `fi` object does not have a local `fimath` object. You can attach a `fimath` object to that `fi` object if you do not want to use the default `fimath` settings. For more information, see “`fimath` Object Construction” in the Fixed-Point Designer documentation.

- `fimath` — fixed-point math object

The following `fimath` properties are always writable and, by transitivity, are also properties of a `fi` object.

- `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`.

---

- `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

These properties are described in detail in “`fimath` Object Properties”.

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

For further details on these properties, see “`numericType` Object Properties”.

## Examples

---

**Note** For information about the display format of `fi` objects, refer to `Display Settings`.

For examples of casting, see “`Cast fi Objects`”.

---

### Example 1

For example, the following creates a signed `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = sfi(pi,8,3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 3
```

Default `fimath` properties are associated with `a`. When a `fi` object does not have a local `fimath` object, no `fimath` object properties are displayed in its output. To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)
```

```
ans =
```

```
0
```

A returned value of `0` means the `fi` object does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

## Example 2

The value `v` can also be an array:

```
a = sfi((magic(3)/10),16,12)
```

```
a =
```

```
0.8000    0.1001    0.6001
0.3000    0.5000    0.7000
0.3999    0.8999    0.2000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 12
```

## Example 3

If you omit the argument `f`, it is set automatically to the best precision possible:

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

a =

3.1563

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

### Example 4

If you omit *w* and *f*, they are set automatically to 16 bits and the best precision possible, respectively:

a = sfi(pi)

a =

3.1416

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

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All properties related to data type must be constant for code generation.

### See Also

[fi](#) | [fimath](#) | [fipref](#) | [isfimathlocal](#) | [numerictype](#) | [quantizer](#) | [ufi](#)



**Introduced in R2009b**

## shiftdata

Shift data to operate on specified dimension

### Syntax

```
[x,perm,nshifts] = shiftdata(x,dim)
```

### Description

`[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first non-singleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

## Examples

### Example 1

This example shifts `x`, a 3-x-3 magic square, permuting dimension 2 to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a 3-x-3 magic square:

```
x = fi(magic(3))
```

```
x =
```

```
     8     1     6
```

```
3 5 7
4 9 2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =
```

```
8 3 4
1 5 9
6 7 2
```

```
perm =
```

```
2 1
```

```
nshifts =
```

```
[]
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =
```

```
8 1 6
3 5 7
4 9 2
```

## Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5
```

```
x =
```

```
    1    2    3    4    5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

`x` is returned as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =
```

```
    1  
    2  
    3  
    4  
    5
```

```
perm =
```

```
    []
```

```
nshifts =
```

```
    1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =
```

```
    1    2    3    4    5
```

## **See Also**

unshiftdata

**Introduced in R2008a**

## showfixptsimerrors

Show overflows from most recent fixed-point simulation

### Compatibility

---

**Note** `showfixptsimerrors` will be removed in a future release. Use `fxptdlg` instead.

---

### Syntax

```
showfixptsimerrors
```

### Description

The `showfixptsimerrors` script displays any overflows from the most recent fixed-point simulation. This information is also visible in the Fixed-Point Tool.

### See Also

`autofixexp` | `fxptdlg`

**Introduced before R2006a**

# showfixptsimranges

Show logged maximum values, minimum values, and overflow data from fixed-point simulation

## Compatibility

---

**Note** showfixptsimranges will be removed in a future release. Use fxptdlg instead.

---

## Syntax

```
showfixptsimranges  
showfixptsimranges(action)
```

## Description

showfixptsimranges displays the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the MATLAB Command Window.

showfixptsimranges(action) stores the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the workspace variable FixPtSimRanges. If action is 'verbose', the logged data also appears in the MATLAB Command Window. If action is 'quiet', no data appears.

## See Also

autofixexp | fxptdlg

**Introduced before R2006a**

## showInstrumentationResults

Results logged by instrumented, compiled C code function

### Syntax

```
showInstrumentationResults('mex_fcn')  
showInstrumentationResults ('mex_fcn' '-options')  
showInstrumentationResults mex_fcn  
showInstrumentationResults mex_fcn -options
```

### Description

`showInstrumentationResults('mex_fcn')` opens the Code Generation Report, showing results from calling the instrumented MEX function `mex_fcn`. Hovering over variables and expressions in the report displays the logged information. The logged information includes minimum and maximum values, proposed fraction or word lengths, percent of current range, and whether the value is always a whole number, depending on which options you specify. If you specify to include them in the `buildInstrumentedMex` function, histograms are also included. The same information is displayed in a summary table in the Variables tab.

`showInstrumentationResults ('mex_fcn' '-options')` specifies options for the instrumentation results section of the Code Generation Report.

`showInstrumentationResults mex_fcn` and `showInstrumentationResults mex_fcn -options` are alternative syntaxes for opening the Code Generation Report.

When you call `showInstrumentationResults`, a file named `instrumentation/mex_fcn/html/index.html` is created. `mex_fcn` is the name of the corresponding instrumented MEX function. Selecting this file opens a web-based version of the Code Generation Report. To open this file from within MATLAB, right-click on the file and select **Open Outside MATLAB**. `showInstrumentationResults` returns an error if the instrumented `mex_fcn` has not yet been called.



## Input Arguments

### `mex_fcn`

Instrumented MEX function created using `buildInstrumentedMex`.

### `options`

Instrumentation results options.

<code>-defaultDT <i>T</i></code>	Default data type to propose for double or single data type inputs, where <i>T</i> is either a numeric type object or one of the following: 'remainFloat', 'double', 'single', 'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', or 'uint64'. If you specify an int or uint, the signedness and word length are that int or uint value and a fraction length is proposed. The default is remainFloat, which does not propose any data types.
<code>-nocode</code>	Do not display MATLAB code in the printable report. Display only the tables of logged variables. This option only has effect in combination with the <code>-printable</code> option.
<code>-optimizeWholeNumbers</code>	Optimize the word length of variables whose simulation min/max logs indicate that they are always whole numbers.
<code>-percentSafetyMargin <i>N</i></code>	Safety margin for simulation min/max, where <i>N</i> is a percent value.
<code>-printable</code>	Create and open a printable HTML report. The report opens in the system browser.
<code>-proposeFL</code>	Propose fraction lengths for specified word lengths.
<code>-proposeWL</code>	Propose word lengths for specified fraction lengths.

## Examples

Generate an instrumented MEX function, then run a test bench. Call `showInstrumentationResults` to open the Code Generation Report.

---

**Note** The logged results from `showInstrumentationResults` are an accumulation of all previous calls to the instrumented MEX function. To clear the log, see `clearInstrumentationResults`.

---

- 1 Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('showInstrumentationResults')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
    'fidemos','fi_m_radix2fft_withscaling.m'),...
    'testfft.m','f')
```

- 2 Define prototype input arguments.

```
T = numericType('DataType','ScaledDouble','Scaling',...
    'Unspecified');
```

```
n = 128;
x = complex(fi(zeros(n,1),T));
W = coder.Constant(fi(fidemo.fi_radix2twiddles(n),T));
```

- 3 Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name.

```
buildInstrumentedMex testfft -o testfft_instrumented...
    -args {x,W} -histogram
```

- 4 Run a test bench to record instrumentation results. Call `showInstrumentationResults` to open a report. View the simulation minimum and maximum values, proposed fraction length, percent of current range, and whole number status by pausing over a variable in the report.


```
for i=1:20
    x(:) = 2*rand(size(x))-1;
    y = testfft_instrumented(x);
end
```

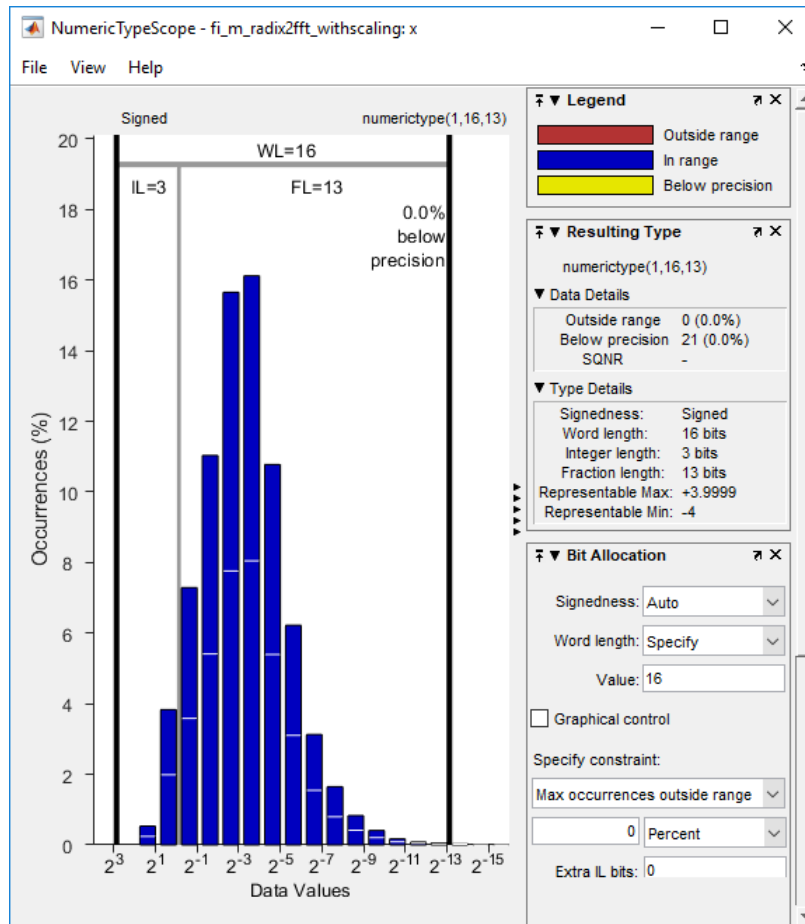
```
showInstrumentationResults testfft_instrumented...
-proposeFL -percentSafetyMargin 10
```

The screenshot displays the MATLAB environment with the 'testfft.m' source code open. The code defines a function `fi_m_radix2fft_withscaling` that performs a radix-2 FFT with scaling. A 'VARIABLE INFO' popup is shown over the code, providing details for the variable `x`, including its size (128 x 1), class (embedded.fi), and instrumentation results such as a proposed fraction length of 14 and a percent of current range of 100. Below the code, the 'Variables' tab is active, showing a table of variables with their types, sizes, and classes. The variable `x` is highlighted, and a histogram icon is visible in the 'Always Whole Number' column for that row.

Name	Type	Size	Class	DT Mode	Signedness	WL	FL	Proposed Signedness	Proposed FL	Proposed FL	Percent of Current Range	Always Whole Number	Sim Min	Sim Max
x	I/O	128 × 1	complex embedded.fi	ScaledDouble	Signed	16	15	-	-	14	100	No	-0.999852132	0.998897942
w	Input	127 × 1	complex embedded.fi	ScaledDouble	Signed	16	14	-	-	14	51	No	-1	1
n	Local	1 × 1	double	-	-	-	-	-	-	-	-	Yes	128	128
t	Local	1 × 1	double	-	-	-	-	-	-	-	-	Yes	7	7
LL	Local	1 × 7	int32	-	-	-	-	-	-	-	-	Yes	2	128
rr	Local	1 × 7	int32	-	-	-	-	-	-	-	-	Yes	1	64
LL2	Local	1 × 7	int32	-	-	-	-	-	-	-	-	Yes	1	64

1

View the histogram for a variable by clicking  in the **Variables** tab.



For information on the figure, refer to the NumericTypeScope reference page.

- 2 Close the histogram display and then, clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

- 3 Clear the MEX function, then delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanUp;
```

## See Also

`NumericTypeScope` | `buildInstrumentedMex` | `clearInstrumentationResults` | `codegen` | `fiaccel` | `mex`

**Introduced in R2011b**

## **sin**

Sine of fixed-point values

### **Syntax**

```
y = sin(theta)
```

### **Description**

`y = sin(theta)` returns the sine on page 5-771 of `fi` input `theta` using a table-lookup algorithm.

### **Input Arguments**

#### **theta**

`theta` can be a real-valued, signed or unsigned scalar, vector, matrix, or N-dimensional array containing the fixed-point angle values in radians. Valid data types of `theta` are:

- `fi` single
- `fi` double
- `fi` fixed-point with binary point scaling
- `fi` scaled double with binary point scaling

### **Output Arguments**

#### **y**

`y` is the sine of `theta`. `y` is a signed, fixed-point number in the range `[-1,1]`. It has a 16-bit word length and 15-bit fraction length (`numericType(1,16,15)`).

## Examples

Calculate the sine of fixed-point input values.

```
theta = fi([-pi/2,-pi/3,-pi/4 0, pi/4,pi/3,pi/2])
```

```
theta =
```

```
theta =
```

```
-1.5708 -1.0472 -0.7854 0 0.7854 1.0472 1.5708
```

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

```
y = sin(theta)
```

```
y =
```

```
-1.0000 -0.8661 -0.7072 0 0.7070 0.8659 0.9999
```

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

## More About

### Sine

The sine of angle  $\theta$  is defined as

$$\sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

## Algorithms

The `sin` function computes the sine of fixed-point input using an 8-bit lookup table as follows:

- 1 Perform a modulo  $2\pi$ , so the input is in the range  $[0, 2\pi)$  radians.
- 2 Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
- 3 Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
- 4 Use the 8 most-significant bits to obtain the first value from the table.
- 5 Use the next-greater table value as the second value.
- 6 Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

## `fimath` Propagation Rules

The `sin` function ignores and discards any `fimath` attached to the input, `theta`. The output, `y`, is always associated with the default `fimath`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`angle` | `atan2` | `cordiccos` | `cordicsin` | `cos` | `sin`

### Topics

Demo: Fixed-Point Sine and Cosine Calculation



**Introduced in R2012a**

## sign

Perform signum function on array

### Syntax

```
c = sign(a)
```

### Description

`c = sign(a)` returns an array `c` the same size as `a`, where each element of `c` is

- 1 if the corresponding element of `a` is greater than zero
- 0 if the corresponding element of `a` is zero
- -1 if the corresponding element of `a` is less than zero

The elements of `c` are of data type `int8`.

`sign` does not support complex `fi` inputs.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**Introduced before R2006a**

## single

Single-precision floating-point real-world value of `fi` object

### Syntax

`single(a)`

### Description

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

`single(a)` returns the real-world value of a `fi` object in single-precision floating point.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.

## **See Also**

double

**Introduced before R2006a**

## sort

Sort elements of real-valued `fi` object in ascending or descending order

### Description

This function accepts `fi` objects as inputs.

`sort` does not support complex fixed-point inputs, or pairs of `Name`, `Value` arguments. Refer to the MATLAB `sort` reference page for more information.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `dimensions` argument must be a built-in type; it cannot be a `fi` object.

**Introduced in R2008b**

## sqrt

Square root of `fi` object

### Syntax

```
c = sqrt(a)
c = sqrt(a,T)
c = sqrt(a,F)
c = sqrt(a,T,F)
```

### Description

This function computes the square root of a `fi` object using a bisection algorithm.

`c = sqrt(a)` returns the square root of `fi` object `a`. Intermediate quantities are calculated using the `fimath` associated with `a`. The `numericType` object of `c` is determined automatically for you using an internal rule on page 5-779.

`c = sqrt(a,T)` returns the square root of `fi` object `a` with `numericType` object `T`. Intermediate quantities are calculated using the `fimath` associated with `a`. See “Data Type Propagation Rules” on page 5-779.

`c = sqrt(a,F)` returns the square root of `fi` object `a`. Intermediate quantities are calculated using the `fimath` object `F`. The `numericType` object of `c` is determined automatically for you using an internal rule on page 5-779. When `a` is a built-in `double` or `single` data type, this syntax is equivalent to `c = sqrt(a)` and the `fimath` object `F` is ignored.

`c = sqrt(a,T,F)` returns the square root `fi` object `a` with `numericType` object `T`. Intermediate quantities are also calculated using the `fimath` object `F`. See “Data Type Propagation Rules” on page 5-779.

`sqrt` does not support complex, negative-valued, or [Slope Bias] inputs.

## Internal Rule

For syntaxes where the `numerictype` object of the output is not specified as an input to the `sqrt` function, it is automatically calculated according to the following internal rule:

$$sign_c = sign_a$$

$$WL_c = \text{ceil}\left(\frac{WL_a}{2}\right)$$

$$FL_c = WL_c - \text{ceil}\left(\frac{WL_a - FL_a}{2}\right)$$

## Data Type Propagation Rules

For syntaxes for which you specify a `numerictype` object T, the `sqrt` function follows the data type propagation rules listed in the following table. In general, these rules can be summarized as “floating-point data types are propagated.” This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input fi Object a	Data Type of numerictype object T	Data Type of Output c
Built-in double	Any	Built-in double
Built-in single	Any	Built-in single
fi Fixed	fi Fixed	Data type of numerictype object T
fi ScaledDouble	fi Fixed	ScaledDouble with properties of numerictype object T
fi double	fi Fixed	fi double
fi single	fi Fixed	fi single
Any fi data type	fi double	fi double
Any fi data type	fi single	fi single

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Complex and [Slope Bias] inputs error out.
- Negative inputs yield a 0 result.

**Introduced in R2006b**



# storedInteger

Stored integer value of `fi` object

## Syntax

```
st_int = storedInteger(f)
```

## Description

`st_int = storedInteger(f)` returns the stored integer value of `fi` object `f`.

Fixed-point numbers can be represented as

$$\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}$$

or, equivalently as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

## Input Arguments

**f** — Fixed-point numeric object

`fi` object

Fixed-point numeric object from which you want to get the stored integer value.

## Output Arguments

**st\_int** — Stored integer value of `fi` object

integer

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

The returned stored integer value is the smallest built-in integer data type in which the stored integer value `f` fits. Signed `fi` values return stored integers of type `int8`, `int16`, `int32`, or `int64`. Unsigned `fi` values return stored integers of type `uint8`, `uint16`, `uint32`, or `uint64`. The return type is determined based on the stored integer word length (WL):

- $WL \leq 8$  bits, the return type is `int8` or `uint8`.
- $8 \text{ bits} < WL \leq 16$  bits, the return type is `int16` or `uint16`.
- $16 \text{ bits} < WL \leq 32$  bits, the return type is `int32` or `uint32`.
- $32 \text{ bits} < WL \leq 64$  bits, the return type is `int64` or `uint64`.

---

**Note** When the word length is greater than 64 bits, the `storedInteger` function errors. For bit-true integer representation of very large word lengths, use `bin`, `oct`, `dec`, `hex`, or `sdec`.

---

## Examples

### Stored Integer Value of `fi` Objects

Find the stored integer values for two `fi` objects. Use the `class` function to display the stored integer data types.

```
x = fi([0.2 0.3 0.5 0.3 0.2]);
in_x = storedInteger(x);
c1 = class(in_x)

numtp = numericType('WordLength',17);
x_n = fi([0.2 0.3 0.5 0.3 0.2], 'numericType', numtp);
in_xn = storedInteger(x_n);
c2 = class(in_xn)
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`int16` | `int32` | `int64` | `int8` | `storedIntegerToDouble` | `uint16` | `uint32` | `uint64` | `uint8`

**Introduced in R2012a**

## storedIntegerToDouble

Convert stored integer value of `fi` object to built-in double value

### Syntax

```
d = storedIntegerToDouble(f)
```

### Description

`d = storedIntegerToDouble(f)` converts the stored integer value of `fi` object, `f`, to a double-precision floating-point value, `d`.

If the input word length is greater than 52 bits, a quantization error may occur. `INF` is returned if the stored integer value of the input `fi` object is outside the representable range of built-in double values.

### Input Arguments

**f**

`fi` object

### Examples

#### Convert Stored Integer Value of `fi` Object to Double-Precision Value

Convert the stored integer of a `fi` value to a double-precision value. Use the `class` function to verify that the stored integer is a double-precision value.

```
f = fi(pi,1,16,12);  
d = storedIntegerToDouble(f);  
dtype = class(d)
```

```
dtype =  
'double'
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[class](#) | [fi](#) | [storedInteger](#)

**Introduced in R2012a**

## stripscaling

Stored integer of `fi` object

### Syntax

```
I = stripscaling(a)
```

### Description

`I = stripscaling(a)` returns the stored integer of `a` as a `fi` object with binary-point scaling, zero fraction length and the same word length and sign as `a`.

### Examples

Stripscaling is useful for converting the value of a `fi` object to its stored integer value.

```
fipref('NumericTypeDisplay','short', ...
       'FimathDisplay','none');
format long g
a = fi(0.1,true,48,47)

a =

           0.10000000000000001
      s48,47
b = stripscaling(a)

b =

           14073748835533
      s48,0
bin(a)

ans =

00001100110011001100110011001100110011001100110011001101
```

```
bin(b)
```

```
ans =
```

```
000011001100110011001100110011001100110011001101
```

Notice that the stored integer values of `a` and `b` are identical, while their real-world values are different.

**Introduced before R2006a**

## sub

Subtract two objects using `fimath` object

## Syntax

```
c = sub(F,a,b)
```

## Description

`c = sub(F,a,b)` subtracts objects `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` has no local `fimath`.

`a` and `b` must both be `fi` objects and must have the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object.

## Examples

In this example, `c` is the 32-bit difference of `a` and `b` with fraction length 16.

```
a = fi(pi);  
b = fi(exp(1));  
F = fimath('SumMode','SpecifyPrecision',...  
          'SumWordLength',32,'SumFractionLength',16);  
c = sub(F, a, b)  
  
c =  
  
    0.4233
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 32  
        FractionLength: 16
```



## Algorithms

`c = sub(F, a, b)` is similar to

```
a.fimath = F;
b.fimath = F;
c = a - b
```

```
c =
    0.4233
```

```
    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
```

but not identical. When you use `sub`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` has no local `fimath`. When you use the syntax `c = a - b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See “`fimath` Rules for Fixed-Point Arithmetic” in the Fixed-Point Designer User's Guide for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax `F.sub(a, b)`. You must use the syntax `sub(F, a, b)`.

## **See Also**

`add` | `divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide`

**Introduced before R2006a**

# subsasgn

Subscripted assignment

## Syntax

```
a(I) = b
a(I,J) = b
a(I,:) = b
a(:,I) = b
a(I,J,K,...) = b
a = subsasgn(a,S,b)
```

## Description

`a(I) = b` assigns the values of `b` into the elements of `a` specified by the subscript vector `I`. `b` must have the same number of elements as `I` or be a scalar value.

`a(I,J) = b` assigns the values of `b` into the elements of the rectangular submatrix of `a` specified by the subscript vectors `I` and `J`. `b` must have `LENGTH(I)` rows and `LENGTH(J)` columns.

A colon used as a subscript, as in `a(I,:) = b` or `a(:,I) = b` indicates the entire column or row.

For multidimensional arrays, `a(I,J,K,...) = b` assigns `b` to the specified elements of `a`. `b` must be `length(I)-by-length(J)-by-length(K)-...` or be shiftable to that size by adding or removing singleton dimensions.

`a = subsasgn(a,S,b)` is called for the syntax `a(i)=b`, `a{i}=b`, or `a.i=b` when `a` is an object. `S` is a structure array with the following fields:

- `type` — One of the following: `'()'` , `'{}'` , or `'.'` specifying the subscript type
- `subs` — Cell array or character vector containing the actual subscripts

For instance, the syntax `a(1:2,:) = b` calls `a=subsasgn(a,S,b)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2, ':'}`. A colon used as a subscript is passed as `':'`.

You can use fixed-point assignment, for example `a(:) = b`, to cast a value with one `numericType` object into another `numericType` object. This subscripted assignment statement assigns the value of `b` into `a` while keeping the `numericType` object of `a`. Subscripted assignment works the same way for integer data types.

## Examples

### Cast a 16-bit Number into an 8-bit Number

For `fi` objects `a` and `b`, there is a difference between

```
a = b
```

and

```
a(:) = b
```

In the first case, `a = b` replaces `a` with `b` while `a` assumes the value, `numericType` object and `fi` object associated with `b`. In the second case, `a(:) = b` assigns the value of `b` into `a` while keeping the `numericType` object of `a`. You can use this to cast a value with one `numericType` object into another `numericType` object.

For example, cast a 16-bit number into an 8-bit number.

```
a = fi(0, 1, 8, 7)
```

```
a =
```

```
0
```

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

```
b = fi(pi/4, 1, 16, 15)
```

```
b =
```

```
0.7854
```

```

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

```

```
a(:) = b
```

```
a =
    0.7891
```

```

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

```

### Emulate a 40-bit Accumulator of a DSP

This example defines a variable `acc` to emulate a 40-bit accumulator of a DSP. The products and sums in this example are assigned into the accumulator using the syntax `acc(1)=...`. Assigning values into the accumulator is like storing a value in a register. To begin, turn the logging mode on and define the variables. In this example, `n` is the number of points in the input data `x` and output data `y`, and `t` represents time. The remaining variables are all defined as `fi` objects. The input data `x` is a high-frequency sinusoid added to a low-frequency sinusoid.

```

fipref('LoggingMode', 'on');
n = 100;
t = (0:n-1)/n;
x = fi(sin(2*pi*t) + 0.2*cos(2*pi*50*t));
b = fi([.5 .5]);
y = fi(zeros(size(x)), numericType(x));
acc = fi(0.0, true, 40, 30);

```

The following loop takes a running average of the input `x` using the coefficients in `b`. Notice that `acc` is assigned into `acc(1)=...` versus using `acc=...`, which would overwrite and change the data type of `acc`.

```

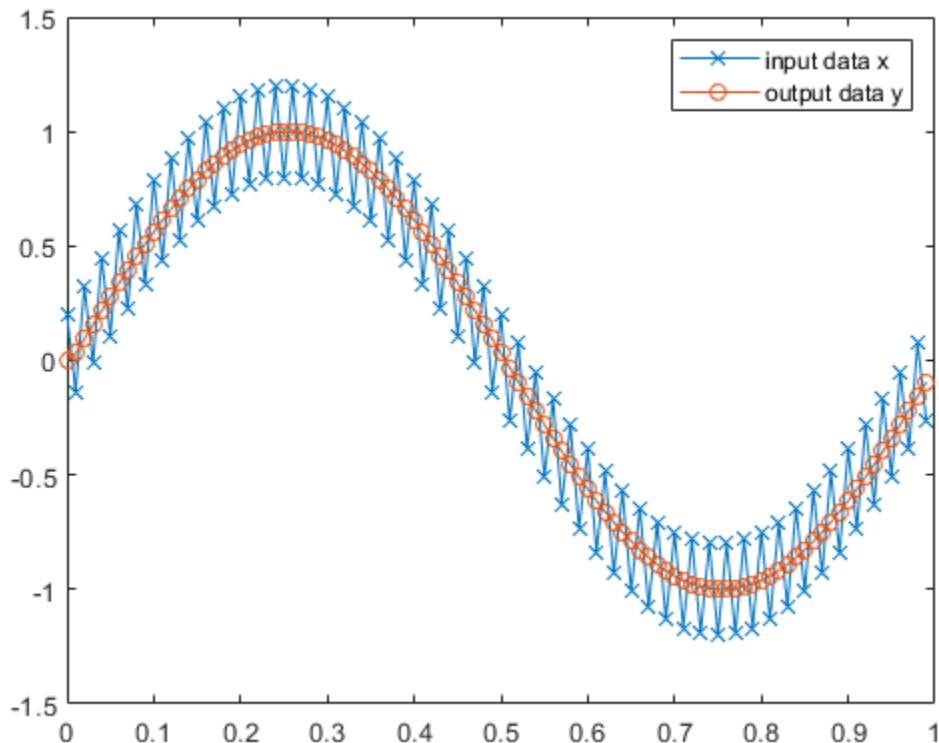
for k = 2:n
    acc(1) = b(1)*x(k);
    acc(1) = acc + b(2)*x(k-1);
end

```

```
    y(k) = acc;  
end
```

By averaging every other sample, the loop shown above passes the low-frequency sinusoid through and attenuates the high-frequency sinusoid.

```
plot(t,x,'x-',t,y,'o-')  
legend('input data x','output data y')
```



The log report shows the minimum and maximum logged values and ranges of the variables used. Because `acc` is assigned into, rather than overwritten, these logs reflect the accumulated minimum and maximum values.

```
logreport(x, y, b, acc)
```

	minlog	maxlog	lowerbound	upperbound	overflow
x	-1.200012	1.197998	-2	1.999939	(
y	-0.9990234	0.9990234	-2	1.999939	(
b	0.5	0.5	-1	0.9999695	(
acc	-0.9990234	0.9989929	-512	512	(

Display acc to verify that its data type did not change.

```
acc
```

```
acc =  
-0.0941
```

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

Reset the fipref object to restore its default values.

```
reset(fipref)
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

## See Also

subsref

## Topics

“Cast fi Objects”

**Introduced before R2006a**



# suboref

Subscripted reference

## Description

This function accepts `fi` objects as inputs.

Refer to the MATLAB `suboref` reference page for more information.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

**Introduced before R2006a**

## sum

Sum of array elements

### Syntax

```
S= sum (A)
S= sum ( A, dim)
S = sum ( ___ , type )
```

### Description

`S= sum (A)` returns the sum along different dimensions of the `fi` array `A`.

If `A` is a vector, `sum(A)` returns the sum of the elements.

If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.

If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`S= sum ( A, dim)` sums along the dimension `dim` of `A`.

`S = sum ( ___ , type )` returns an array in the class specified by `type`, using any of the input arguments in the previous syntaxes. `type` can be `'double'` or `'native'`.

- If `type` is `'double'`, then `sum` returns a double-precision array, regardless of the input data type.
- If `type` is `'native'`, then `sum` returns an array with the same class of input array `A`.

The `fimath` object is used in the calculation of the sum. If `SumMode` is `FullPrecision`, `KeepLSB`, or `KeepMSB`, then the number of integer bits of growth for `sum(A)` is `ceil(log2(size(A,dim)))`.

`sum` does not support `fi` objects of data type `Boolean`.

## Examples

### Sum of Vector Elements

Create a `fi` vector, and specify `fimath` properties in the constructor.

```
A=fi([1 2 5 8 5], 'SumMode', 'KeepLSB', 'SumWordLength', 32)
```

```
A=1x5 object
    1     2     5     8     5

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

    RoundingMethod: Nearest
    OverflowAction: Saturate
    ProductMode: FullPrecision
    SumMode: KeepLSB
    SumWordLength: 32
    CastBeforeSum: true
```

Compute the sum of the elements of A.

```
S=sum(A)
```

```
S =
    21

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 11

    RoundingMethod: Nearest
    OverflowAction: Saturate
    ProductMode: FullPrecision
    SumMode: KeepLSB
    SumWordLength: 32
    CastBeforeSum: true
```

The output *S* is a scalar with the specified `SumWordLength` of 32. The `FractionLength` of *S* is 11 because `SumMode` was set to `KeepLSB`.

### Sum of Elements in Each Column

Create a `fi` array, and compute the sum of the elements in each column.

```
A=fi([1 2 8;3 7 0;1 2 2])
```

```
A=3×3 object
```

```
 1     2     8
 3     7     0
 1     2     2
```

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

```
S=sum(A)
```

```
S=1×3 object
```

```
 5     11     10
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 18
      FractionLength: 11
```

MATLAB® returns a row vector with the sums of each column of *A*. The `WordLength` of *S* has increased by two bits because `ceil(log2(size(A,1)))=2`. The `FractionLength` remains the same because the default setting of `SumMode` is `FullPrecision`.

### Sum of Elements in Each Row

Compute the sum along the second dimension (`dim=2`) of 3-by-3 matrix *A*.

```
A=fi([1 2 8;3 7 0;1 2 2])
```

```
A=3×3 object
```

```
1    2    8
3    7    0
1    2    2
```

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

```
S=sum(A, 2)
```

```
S=3×1 object
```

```
11
10
5
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 18
    FractionLength: 11
```

MATLAB® returns a column vector of the sums of the elements in each row. The WordLength of S is 18 because  $\text{ceil}(\log_2(\text{size}(A,2)))=2$ .

### Sum of Elements Preserving Data Type

Compute the sums of the columns of A so that the output array, S, has the same data type.

```
A=fi([1 2 8;3 7 0;1 2 2]), class(A)
```

```
A=3×3 object
```

```
1    2    8
3    7    0
1    2    2
```

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

```
ans =  
'embedded.fi'  
  
S=sum(A, 'native'), class(S)  
  
S=1×3 object  
    5    11    10  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 18  
    FractionLength: 11  
  
ans =  
'embedded.fi'
```

MATLAB® preserves the data type of *A* and returns a row vector *S* of type `embedded.fi`.

## Input Arguments

### **A** — Input `fi` array

`fi` object | numeric variable

`fi` input array, specified as a scalar, vector, matrix, or multidimensional array.

**Data Types:** `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

### **dim** — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. `dim` can also be a `fi` object. If no value is specified, the default is the first array dimension whose size does not equal 1.

**Data Types:** `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**type — Output class**`'double' | 'native'`

Output class, specified as `'double'` or `'native'`, defines the data type that the operation is performed in and returned in.

Data Types: `char`

## Output Arguments

**S — Sum array**`scalar | vector | matrix | multidimensional array`

Sum array, returned as a scalar, vector, matrix, or multidimensional array.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`.

### See Also

`add | divide | fi | fimath | mpy | mrdivide | numerictype | rdivide | sub | sum`

**Introduced before R2006a**

## times

Element-by-element multiplication of `fi` objects

### Syntax

```
C = A.*B
C = times(A, B)
```

### Description

`C = A.*B` performs element-by-element multiplication of `A` and `B`, and returns the result in `C`.

`C = times(A, B)` is an alternate way to execute `A.*B`.

### Examples

#### Multiply a `fi` Object by a Scalar

Use the `times` function to perform element-by-element multiplication of a `fi` object and a scalar.

```
a=4;
b=fi([2 4 7; 9 0 2])
```

```
b=2×3 object
```

```
  2   4   7
  9   0   2
```

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



`a` is a scalar double, and `b` is a matrix of `fi` objects. When doing arithmetic between a `fi` and a double, the double is cast to a `fi` with the same word length and signedness of the `fi`, and best-precision fraction length. The result of the operation is a `fi`.

```
c=a.*b
```

```
c=2x3 object
```

```
 8    16    28
36     0     8
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 32
      FractionLength: 23
```

During the operation, `a` was cast to a `fi` object with wordlength 16. The output, `c`, is a `fi` object with word length 32, the sum of the word lengths of the two multiplicands, `a` and `b`. This is because the default setting of `ProductMode` in `fimath` is `FullPrecision`.

## Multiply Two fi Objects

Use the `times` function to perform element-by-element multiplication of two `fi` objects.

```
a=fi([5 9 9; 1 2 -3], 1, 16, 3)
```

```
a=2x3 object
```

```
 5     9     9
 1     2    -3
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 3
```

```
b=fi([2 4 7; 9 0 2], 1, 16, 3)
```

```
b=2x3 object
```

```
 2     4     7
 9     0     2
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 3
```

```
c=a.*b
```

```
c=2x3 object
```

```
10 36 63
 9  0 -6
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 6
```

The word length and fraction length of `c` are equal to the sums of the word lengths and fraction lengths of `a` and `b`. This is because the default setting of `ProductMode` in `fimath` is `FullPrecision`.

## Input Arguments

### A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in types. `A` and `B` must have the same dimensions unless one is a scalar value.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

### B — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in types. `A` and `B` must have the same dimensions unless one is a scalar value.

**Data Types:** `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Complex Number Support:** Yes

## Output Arguments

### **C** — Output array

`scalar` | `vector` | `matrix` | `multidimensional array`

Output array, specified as a scalar, vector, matrix or multidimensional array.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- 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 `0n`.

### **See Also**

`minus` | `mtimes` | `plus` | `uminus`

**Introduced before R2006a**

## toeplitz

Create Toeplitz matrix

### Syntax

```
t = toeplitz(a,b)
t = toeplitz(b)
```

### Description

`t = toeplitz(a,b)` returns a nonsymmetric Toeplitz matrix having `a` as its first column and `b` as its first row. `b` is cast to the `numericType` of `a`.

`t = toeplitz(b)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `b`, where `b` is the first row of the matrix.

The output `fi` object `t` has the same `numericType` properties as the leftmost `fi` object input. If the leftmost `fi` object input has a local `fi`math, the output `fi` object `t` is assigned the same local `fi`math. Otherwise, the output `fi` object `t` has no local `fi`math.

### Examples

`toeplitz(a,b)` casts `b` into the data type of `a`. In this example, overflow occurs:

```
fipref('NumericTypeDisplay','short');
format short g
a = fi([1 2 3],true,8,5)

a =

     1     2     3
    s8,5
b = fi([1 4 8],true,16,10)

b =
```

```

      1      4      8
      s16,10
toeplitz(a,b)

ans =

      1      3.9688      3.9688
      2      1      3.9688
      3      2      1
      s8,5

```

`toeplitz(b,a)` casts `a` into the data type of `b`. In this example, overflow does not occur:

```

toeplitz(b,a)

ans =

      1      2      3
      4      1      2
      8      4      1
      s16,10

```

If one of the arguments of `toeplitz` is a built-in data type, it is cast to the data type of the `fi` object.

```

x = [1 exp(1) pi]

x =

      1      2.7183      3.1416

toeplitz(a,x)

ans =

      1      2.7188      3.1563
      2      1      2.7188
      3      2      1
      s8,5
toeplitz(x,a)

ans =

      1      2      3

```

2.7188	1	2
3.1563	2.7188	1
s8,5		

**Introduced before R2006a**

# tostring

Convert numeric type or quantizer object to string

## Syntax

```
s = tostring(f)
s = tostring(F)
s = tostring(T)
s = tostring(q)
```

## Description

`s = tostring(f)` converts `fi` object `f` to a character vector `s` such that `eval(s)` would create a `fi` object with the same properties as `f`.

`s = tostring(F)` converts `fi` object `F` to a character vector `s` such that `eval(s)` would create a `fi` object with the same properties as `F`.

`s = tostring(T)` converts numeric type object `T` to a character vector `s` such that `eval(s)` would create a numeric type object with the same properties as `T`.

`s = tostring(q)` converts quantizer object `q` to a character vector `s`. After converting `q`, the function `eval(s)` can use `s` to create a quantizer object with the same properties as `q`.

## Examples

### Convert a numeric type Object to a String

This example uses the `tostring` function to convert a numeric type object `T` to a string `s`.

```
T = numericType(1,16,15);
s = tostring(T);
```

```
T1 = eval(s);  
isequal(T,T1)
```

```
ans =
```

```
1
```

### Convert a `fi` Object to a character vector

This example uses the `tostring` function to convert a `fi` object `f` to a character vector `s`.

```
f = fi(pi,1,16,10);  
s = tostring(f);  
f1 = eval(s);  
isequal(f,f1)
```

```
ans =
```

```
1
```

### See Also

`eval` | `fi` | `fimath` | `numericType` | `quantizer`

**Introduced before R2006a**



## ufi

Construct unsigned fixed-point numeric object

### Syntax

```
a = ufi
a = ufi(v)
a = ufi(v,w)
a = ufi(v,w,f)
a = ufi(v,w,slope,bias)
a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

### Description

You can use the `ufi` constructor function in the following ways:

- `a = ufi` is the default constructor and returns an unsigned `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = ufi(v)` returns an unsigned fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = ufi(v,w)` returns an unsigned fixed-point object with value `v`, word length `w`, and best-precision fraction length.
- `a = ufi(v,w,f)` returns an unsigned fixed-point object with value `v`, word length `w`, and fraction length `f`.
- `a = ufi(v,w,slope,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slope`, and `bias`.
- `a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`fi` objects created by the `ufi` constructor function have the following general types of properties:

- “Data Properties” on page 5-814
- “fimath Properties” on page 5-814
- “numerictype Properties” on page 5-815

These properties are described in detail in “fi Object Properties” on page 3-2 in the Properties Reference.

---

**Note** `fi` objects created by the `ufi` constructor function have no local `fimath`.

---

### 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
- `hex` — Stored integer value of a `fi` object in hexadecimal
- `int` — Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type. You can also use `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64` to get the stored integer value of a `fi` object in these formats
- `oct` — Stored integer value of a `fi` object in octal

These properties are described in detail in “fi Object Properties” on page 3-2.

### fimath Properties

When you create a `fi` object with the `ufi` constructor function, that `fi` object does not have a local `fimath` object. You can attach a `fimath` object to that `fi` object if you do not want to use the default `fimath` settings. For more information, see “fimath Object Construction” in the Fixed-Point Designer documentation.

- `fimath` — fixed-point math object

The following `fimath` properties are always writable and, by transitivity, are also properties of a `fi` object.

- `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`.

---

- `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

These properties are described in detail in “`fimath` Object Properties”.

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

For further details on these properties, see “`numerictype` Object Properties”.

## Examples

---

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

For examples of casting, see “`Cast fi Objects`”.

---

### Example 1

For example, the following creates an unsigned `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = ufi(pi,8,3)
```

```
a =
```

```
3.1250
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness:   Unsigned
      WordLength:   8
      FractionLength: 3
```

Default `fimath` properties are associated with `a`. When a `fi` object does not have a local `fimath` object, no `fimath` object properties are displayed in its output. To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)
```

```
ans =
```

```
0
```

A returned value of `0` means the `fi` object does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

## Example 2

The value `v` can also be an array:

```
a = ufi((magic(3)/10),16,12)
```

```
a =
```

```
0.8000    0.1001    0.6001
0.3000    0.5000    0.7000
0.3999    0.8999    0.2000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness:   Unsigned
      WordLength:   16
      FractionLength: 12
```

```
>>
```

### Example 3

If you omit the argument `f`, it is set automatically to the best precision possible:

```
a = ufi(pi,8)
```

```
a =
```

```
3.1406
```

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

### Example 4

If you omit `w` and `f`, they are set automatically to 16 bits and the best precision possible, respectively:

```
a = ufi(pi)
```

```
a =
```

```
3.1416
```

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

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All properties related to data type must be constant for code generation.

## **See Also**

`fi` | `fimath` | `fipref` | `isfimathlocal` | `numerictype` | `quantizer` | `sfi`

**Introduced in R2009b**

## uint8

Convert `fi` object to unsigned 8-bit integer

### Syntax

```
c = uint8(a)
```

### Description

`c = uint8(a)` returns the built-in `uint8` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint8`.

### Examples

This example shows the `uint8` values of a `fi` object.

```
a = fi([-pi 0.5 pi],0,8);  
c = uint8(a)
```

```
c =
```

```
    0     1     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint64](#)



**Introduced before R2006a**

## uint16

Convert `fi` object to unsigned 16-bit integer

### Syntax

```
c = uint16(a)
```

### Description

`c = uint16(a)` returns the built-in `uint16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint16`.

### Examples

This example shows the `uint16` values of a `fi` object.

```
a = fi([-pi 0.5 pi],0,16);  
c = uint16(a)
```

```
c =
```

```
    0     1     3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint32](#) | [uint64](#) | [uint8](#)

**Introduced before R2006a**

## uint32

Stored integer value of `fi` object as built-in `uint32`

### Syntax

```
c = uint32(a)
```

### Description

`c = uint32(a)` returns the built-in `uint32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint32`.

### Examples

This example shows the `uint32` values of a `fi` object.

```
a = fi([-pi 0.5 pi],0,32);  
c = uint32(a)
```

```
c =
```

```
0    1    3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint64](#) | [uint8](#)

**Introduced before R2006a**

## uint64

Convert `fi` object to unsigned 64-bit integer

### Syntax

```
c = uint64(a)
```

### Description

`c = uint64(a)` returns the built-in `uint64` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint64`.

### Examples

This example shows the `uint64` values of a `fi` object.

```
a = fi([-pi 0.5 pi],0,64);  
c = uint64(a)
```

```
c =
```

```
0    1    3
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[int16](#) | [int32](#) | [int64](#) | [int8](#) | [storedInteger](#) | [uint16](#) | [uint32](#) | [uint8](#)

**Introduced in R2008b**

## uminus

Negate elements of `fi` object array

## Syntax

```
uminus(a)
```

## Description

`uminus(a)` is called for the syntax `-a` when `a` is an object. `-a` negates the elements of `a`.

`uminus` does not support `fi` objects of data type `Boolean`.

## Examples

When wrap occurs,  $-(-1) = -1$  :

```
fipref('NumericTypeDisplay','short', ...
      'fimathDisplay','none');
format short g
a = fi(-1,true,8,7,'OverflowAction','Wrap')

a =

    -1
    s8,7
-a

ans =

    -1
    s8,7
b = fi([-1-i -1-i],true,8,7,'OverflowAction','Wrap')

b =
```



```

      -1 -      1i      -1 -      1i
      s8,7
-b

```

ans =

```

      -1 -      1i      -1 -      1i
      s8,7
b'

```

ans =

```

      -1 -      1i
      -1 -      1i
      s8,7

```

When saturation occurs,  $-(-1) = 0.99\dots$  :

```
c = fi(-1,true,8,7,'OverflowAction','Saturate')
```

c =

```

      -1
      s8,7
-c

```

ans =

```

      0.99219
      s8,7
d = fi([-1-i -1-i],true,8,7,'OverflowAction','Saturate')
```

d =

```

      -1 -      1i      -1 -      1i
      s8,7
-d

```

ans =

```

      0.99219 +      0.99219i      0.99219 +      0.99219i
      s8,7
d'

```

ans =

-1 + 0.99219i  
-1 + 0.99219i  
s8,7

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

minus | mtimes | plus | times

**Introduced before R2006a**

# unitquantize

Quantize except numbers within eps of +1

## Syntax

```
y = unitquantize(q, x)
[y1,y2,...] = unitquantize(q,x1,x2,...)
```

## Description

`y = unitquantize(q, x)` works the same as `quantize` except that numbers within `eps(q)` of +1 are made exactly equal to +1 .

`[y1,y2,...] = unitquantize(q,x1,x2,...)` is equivalent to `y1 = unitquantize(q,x1)`, `y2 = unitquantize(q,x2)`,...

## Examples

This example demonstrates the use of `unitquantize` with a quantizer object `q` and a vector `x`.

```
q = quantizer('fixed','floor','saturate',[4 3]);
x = (0.8:.1:1.2)';
y = unitquantize(q,x);
z = [x y]
e = eps(q)
```

This quantization outputs an array containing the original values of `x` and the quantized values of `x`, followed by the value of `eps(q)`:

`z =`

```
0.8000    0.7500
0.9000    1.0000
1.0000    1.0000
1.1000    1.0000
```

1.2000    1.0000

e =

0.1250

## **See Also**

eps | quantize | quantizer | unitquantizer

**Introduced in R2008a**

# unitquantizer

Constructor for unitquantizer object

## Syntax

```
q = unitquantizer(...)
```

## Description

`q = unitquantizer(...)` constructs a unitquantizer object, which is the same as a quantizer object in all respects except that its `quantize` method quantizes numbers within `eps(q)` of +1 to exactly +1.

See `quantizer` for parameters.

## Examples

In this example, a vector `x` is quantized by a unitquantizer object `u`.

```
u = unitquantizer([4 3]);  
x = (0.8:.1:1.2)';  
y = quantize(u,x);  
z = [x y]  
e = eps(u)
```

This quantization outputs an array containing the original values of `x` and the values of `x` that were quantized by the unitquantizer object `u`. The output also includes `e`, the value of `eps(u)`.

`z =`

```
0.8000    0.7500  
0.9000    1.0000  
1.0000    1.0000  
1.1000    1.0000  
1.2000    1.0000
```

e =

0.1250

## **See Also**

quantize | quantizer | unitquantize

**Introduced in R2008a**

# unshiftdata

Inverse of shiftdata

## Syntax

```
y = unshiftdata(x,perm,nshifts)
```

## Description

`y = unshiftdata(x,perm,nshifts)` restores the orientation of the data that was shifted with `shiftdata`. The permutation vector is given by `perm`, and `nshifts` is the number of shifts that was returned from `shiftdata`.

`unshiftdata` is meant to be used in tandem with `shiftdata`. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

## Examples

### Example 1

This example shifts `x`, a 3-by-3 magic square, permuting dimension 2 to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a 3-by-3 magic square:

```
x = fi(magic(3))
```

```
x =
```

```
     8     1     6
     3     5     7
     4     9     2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

This command returns the permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =
```

```
     8     3     4
     1     5     9
     6     7     2
```

```
perm =
```

```
     2     1
```

```
nshifts =
```

```
     []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =
```

```
     8     1     6
     3     5     7
     4     9     2
```

## Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5
```



```
x =  
    1    2    3    4    5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

This command returns `x` as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =  
    1  
    2  
    3  
    4  
    5
```

```
perm =  
    []
```

```
nshifts =  
    1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =  
    1    2    3    4    5
```

## See Also

`shiftdata`

**Introduced in R2008a**

# upperbound

Upper bound of range of `fi` object

## Syntax

`upperbound(a)`

## Description

`upperbound(a)` returns the upper bound of the range of `fi` object `a`. If `L = lowerbound(a)` and `U = upperbound(a)`, then `[L,U] = range(a)`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`eps` | `intmax` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin`

**Introduced before R2006a**

## vertcat

Vertically concatenate multiple `fi` objects

### Syntax

```
c = vertcat(a,b,...)
[a; b; ...]
[a;b]
```

### Description

`c = vertcat(a,b,...)` is called for the syntax `[a; b; ...]` when any of `a, b, ...`, is a `fi` object.

`[a;b]` is the vertical concatenation of matrices `a` and `b`. `a` and `b` must have the same number of columns. Any number of matrices can be concatenated within one pair of brackets. N-D arrays are vertically concatenated along the first dimension. The remaining dimensions must match.

Horizontal and vertical concatenation can be combined, as in `[1 2;3 4]`.

`[a b; c]` is allowed if the number of rows of `a` equals the number of rows of `b`, and if the number of columns of `a` plus the number of columns of `b` equals the number of columns of `c`.

The matrices in a concatenation expression can themselves be formed via a concatenation, as in `[a b;[c d]]`.

---

**Note** The `fimath` and `numericType` objects of a concatenated matrix of `fi` objects `c` are taken from the leftmost `fi` object in the list `(a,b,...)`.

---

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

### **See Also**

horzcat

**Introduced before R2006a**

## **wordlength**

Word length of quantizer object

### **Syntax**

```
wordlength(q)
```

### **Description**

`wordlength(q)` returns the word length of the quantizer object `q`.

### **Examples**

```
q = quantizer([16 15]);  
wordlength(q)
```

```
ans =
```

```
    16
```

### **See Also**

`exponentlength` | `fi` | `fractionlength` | `numerictype` | `quantizer`

**Introduced before R2006a**

## zeros

Create array of all zeros with fixed-point properties

### Syntax

```
X = zeros('like',p)
X = zeros(n,'like',p)
X = zeros(sz1,...,szN,'like',p)
X = zeros(sz,'like',p)
```

### Description

`X = zeros('like',p)` returns a scalar  $\theta$  with the same `numericType`, complexity (real or complex), and `fimath` as `p`.

`X = zeros(n,'like',p)` returns an `n`-by-`n` array of zeros like `p`.

`X = zeros(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of zeros like `p`.

`X = zeros(sz,'like',p)` returns an array of zeros like `p`. The size vector, `sz`, defines `size(X)`.

### Examples

#### 2-D Array of Zeros With Fixed-Point Attributes

Create a 2-by-3 array of zeros with specified `numericType` and `fimath` properties.

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create a 2-by-3 array of zeros that has the same `numericType` properties as `p`.

```
X = zeros(2,3,'like',p)
```

```
X=2×3 object
```

```
  0    0    0
  0    0    0
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 24
      FractionLength: 12
```

### Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz = 1×2
```

```
  3    2
```

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create an array of zeros that is the same size as A and has the same numerictype properties as p.

```
X = zeros(sz,'like',p)
```

```
X=3×2 object
```

```
  0    0
  0    0
  0    0
```

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



FractionLength: 12

### Square Array of Zeros With Fixed-Point Attributes

Create a 4-by-4 array of zeros with specified numeric type and `fi` math properties.

Create a signed `fi` object with word length of 24 and fraction length of 12.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of zeros that has the same numeric type properties as `p`.

```
X = zeros(4, 'like', p)
```

*X=4x4 object*

```
0     0     0     0
0     0     0     0
0     0     0     0
0     0     0     0
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 24
FractionLength: 12
```

### Complex Fixed-Point Zero

Create a scalar fixed-point `0` that is not real valued, but instead is complex like an existing array.

Define a complex `fi` object.

```
p = fi([1+2i 3i],1,24,12);
```

Create a scalar `1` that is complex like `p`.

```
X = zeros('like',p)
```

```
X =  
    0.0000 + 0.0000i  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 24  
    FractionLength: 12
```

### Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)  
    % Cast the coefficients to the coefficient type  
    b = cast(b,'like',T.coeffs);  
    a = cast(a,'like',T.coeffs);  
    % Create the output using zeros with the data type  
    y = zeros(size(x),'like',T.data);  
    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
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example  
  
    % Define coefficients for a filter with specification
```

```
% [b,a] = butter(2,0.25)
b = [0.097631072937818    0.195262145875635    0.097631072937818];
a = [1.00000000000000000    -0.942809041582063    0.333333333333333];

% Define floating-point types
T_float.coeffs = double([]);
T_float.data   = double([]);

% Create a step input using ones with the
% floating-point data type
t = 0:20;
x_float = ones(size(t), 'like', T_float.data);

% Initialize the states using zeros with the
% floating-point data type
z_float = zeros(1,2, 'like', T_float.data);

% Run the floating-point algorithm
y_float = my_filter(b,a,x_float,z_float,T_float);

% Define fixed-point types
T_fixed.coeffs = fi([],true,8,6);
T_fixed.data   = fi([],true,8,6);

% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t), 'like', T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2, 'like', T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf', 'subplot', 'plot', 'legend')
clf
subplot(211)
plot(t,y_float,'co-',t,y_fixed,'kx-')
legend('Floating-point output','Fixed-point output')
title('Step response')
subplot(212)
plot(t,y_float - double(y_fixed),'rs-')
```

```
    legend('Error')
    figure(gcf)
end
```

## Input Arguments

### **n** — Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If n is zero, X is an empty matrix.
- If n is negative, it is treated as zero.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **sz1, ..., szN** — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **sz** — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, X is an empty array.

- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output,  $X$ , does not include those dimensions.

Example: `sz = [2,3,4]` defines  $X$  as a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **p** — **Prototype**

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

Complex Number Support: Yes

## Tips

Using the `b = cast(a, 'like', p)` syntax to specify data types separately 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.

## See Also

`cast` | `ones` | `zeros`

## Topics

“Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using `cast` and `zeros`”

“Manual Fixed-Point Conversion Workflow”

“Manual Fixed-Point Conversion Best Practices”

**Introduced in R2013a**

# Classes — Alphabetical List

---

## **coder.CellType class**

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of MATLAB cell arrays

### **Description**

Specifies the set of cell arrays that the generated code accepts. Use only with the `fiaccl -args` option. Do not pass as an input to a generated MEX function.

### **Construction**

`t = coder.typeof(cells)` creates a `coder.CellType` object for a cell array that has the same cells and cell types as `cells`. The cells in `cells` are type objects or example values.

`t = coder.typeof(cells,sz,variable_dims)` creates a `coder.CellType` object that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds do not change. If you do not specify the `variable_dims` input parameter, except for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not 1 or 0.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.typeof` to create a `coder.CellType` object for a variable-size cell array.

`t = coder.newtype('cell',cells)` creates a `coder.CellType` object for a cell array that has the cells and cell types specified by `cells`. The cells in `cells` must be type objects.

`t = coder.newtype('cell',cells,sz,variable_dims)` creates a `coder.CellType` that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper



bounds do not change. If you do not specify the `variable_dims` input parameter, except for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not 1 or 0.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.newtype` to create a `coder.CellType` object for a variable-size cell array.

## Input Arguments

### **cells** — Specification of cell types

cell array

Cell array that specifies the cells and cell types for the output `coder.CellType` object. For `coder.typeof`, `cells` can contain type objects or example values. For `coder.newtype`, `cells` must contain type objects.

### **sz** — Size of cell array

row vector of integer values

Specifies the upper bound for each dimension of the cell array type object. For `coder.newtype`, `sz` cannot change the number of cells for a heterogeneous cell array.

For `coder.newtype`, the default is `[1 1]`.

### **variable\_dims** — Dimensions that are variable size

row vector of logical values

Specifies whether each dimension is variable size (`true`) or fixed size (`false`).

For `coder.newtype`, the default is `true` for dimensions for which `sz` specifies an upper bound of `inf` and `false` for all other dimensions.

When `cells` specifies a cell array whose elements have different classes, you cannot create a `coder.CellType` object for a variable-size cell array.

## Properties

### **Cells** — Types of cells

cell array

A cell array that specifies the `coder.Type` of each cell.

**ClassName — Name of class**

character vector or string scalar

Class of values in this set.

**SizeVector — Size of cell array**

row vector of integer values

The upper bounds of dimensions of the cell array.

**VariableDims — Dimensions that are variable size**

row vector of logical values

A vector that specifies whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

## Methods

- `isHeterogeneous` Determine whether cell array type represents a heterogeneous cell array
- `isHomogeneous` Determine whether cell array type represents a homogeneous cell array
- `makeHeterogeneous` Make a heterogeneous copy of a cell array type
- `makeHomogeneous` Create a homogeneous copy of a cell array type

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Create a Type for a Cell Array Whose Elements Have the Same Class

Create a type for a cell array whose first element has class `char` and whose second element has class `double`.

```
t = coder.typeof({1 2 3})  
t =  
coder.CellType  
  1x3 homogeneous cell  
  base: 1x1 double
```

The type is homogeneous.

### Create a Heterogeneous Type for a Cell Array Whose Elements Have the Same Class

To create a heterogeneous type when the elements of the example cell array type have the same class, use the `makeHeterogeneous` method.

```
t = makeHeterogeneous(coder.typeof({1 2 3}))  
t =  
coder.CellType  
  1x3 locked heterogeneous cell  
  f1: 1x1 double  
  f2: 1x1 double  
  f3: 1x1 double
```

The cell array type is heterogeneous. It is represented as a structure in the generated code.

### Create a Cell Array Type for a Cell Array Whose Elements Have Different Classes

Define variables that are example cell values.

```
a = 'a';  
b = 1;
```

Pass the example cell values to `coder.typeof`.

```
t = coder.typeof({a, b})  
t =
```

```
coder.CellType
  1x2 heterogeneous cell
    f0: 1x1 char
    f1: 1x1 double
```

### **Create a Type for a Variable-Size Homogeneous Cell Array from an Example Cell Array Whose Elements Have Different Classes**

Create a type for a cell array that contains two character vectors that have different sizes.

```
t = coder.typeof({'aa', 'bbb'})
```

```
t =
```

```
coder.CellType
  1x2 heterogeneous cell
    f0: 1x2 char
    f1: 1x3 char
```

The cell array type is heterogeneous.

Create a type using the same cell array input. This time, specify that the cell array type has variable-size dimensions.

```
t = coder.typeof({'aa', 'bbb'},[1,10],[0,1])
```

```
t =
```

```
coder.CellType
  1x:10 locked homogeneous cell
    base: 1x:3 char
```

The cell array type is homogeneous. `coder.typeof` determined that the base type `1x:3 char` can represent `'aa'`, and `'bbb'`.

### **Create a New Cell Array Type from a Cell Array of Types**

Create a type for a scalar `int8`.

```
ta = coder.newtype('int8',[1 1]);
```

Create a type for a `:1x:2` double row vector.

```
tb = coder.newtype('double',[1 2],[1 1]);
```

Create a cell array type whose cells have the types specified by ta and tb.

```
t = coder.newtype('cell',{ta,tb})
```

```
t =
```

```
coder.CellType
  1x2 heterogeneous cell
    f0: 1x1 int8
    f1: 1x2 double
```

## Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a `1x2 heterogeneous` type. The first element is `double` and the second element is `1x2 double`. The type can also be a `1x3 homogeneous` type in which the elements have class `double` and size `1x2`. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

## See Also

`coder.ArrayType` | `coder.ClassType` | `coder.Constant` | `coder.EnumType` |  
`coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` |  
`coder.newtype` | `coder.resize` | `coder.typeof` | `fiaccel`

## Topics

“Code Generation for Cell Arrays”

**Introduced in R2015b**

# coder.ClassType class

**Package:** coder

**Superclasses:** coder.ArrayType

Represent set of MATLAB classes

## Description

Specifies the set of value class objects that the generated code can accept. Use only with the `fiaccl -args` option. Do not pass as an input to a generated MEX function.

## Construction

`t = coder.typeof(value_class_object)` creates a `coder.ClassType` object for the object `value_class_object`.

`t = coder.newtype(value_class_name)` creates a `coder.ClassType` object for an object of the class `value_class_name`.

## Input Arguments

### **value\_class\_object**

Value class object from which to create the `coder.ClassType` object.

`value_class_object` is an expression that evaluates to an object of a value class. For example:

```
v = myValueClass;  
t = coder.typeof(v);
```

```
t = coder.typeof(myValueClass(2,3));
```

### **value\_class\_name**

Name of a value class definition file on the MATLAB path. Specify as a character vector or string scalar. For example:

```
t = coder.newtype('myValueClass');
```

## Properties

When you create a `coder.ClassType` object `t` from a value class object `v` by using `coder.typeof`, the properties of `t` are the same as the properties of `v` with the attribute `Constant` set to `false`.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Create Type Based on Example Object

Create a type based on an example object in the workspace.

Create a value class `myRectangle`.

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

Create a function that takes an object of `myRectangle` as an input.



```
function z = getarea(r)
    %#codegen
    z = calcarea(r);
end
```

Create an object of myRectangle.

```
v = myRectangle(1,2)
v =
    myRectangle with properties:
        length: 1
        width: 2
```

Create a coder.ClassType object based on v.

```
t = coder.typeof(v)
t =
    coder.ClassType
    1x1 myRectangle
        length: 1x1 double
        width : 1x1 double
```

coder.typeof creates a coder.ClassType object that has the same properties names and types as v has.

Generate code for getarea. Specify the input type by passing the coder.ClassType object, t, to the -args option.

```
codegen getarea -args {t} -report
```

## Create Type by Using coder.newtype

Create a coder.ClassType object for an object of the value class mySquare by using coder.newtype.

Create value class mySquare that has one property, side.

```
classdef mySquare
    properties
```

```
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

Create a `coder.ClassType` type for `mySquare`.

```
t = coder.newtype('mySquare')
```

Specify the type of `side`.

```
t.Properties.side = coder.typeof(2)
```

## Tips

- After you create a `coder.ClassType`, you can modify the types of the properties. For example:

```
t = coder.typeof(myClass)
t.Properties.prop1 = coder.typeof(int16(2));
t.Properties.prop2 = coder.typeof([1 2 3]);
```

- After you create a `coder.ClassType`, you can add properties. For example:

```
t = coder.typeof(myClass)
t.Properties.newprop1 = coder.typeof(int8(2));
t.Properties.newprop2 = coder.typeof([1 2 3]);
```

- When you generate code, the properties of the `coder.ClassType` object that you pass to `codegen` must be consistent with the properties in the class definition file. However, if the class definition file has properties that your code does not use, the `coder.ClassType` object does not have to include those properties. The code generator removes properties that you do not use.

## See Also

`coder.ArrayType` | `coder.CellType` | `coder.Constant` | `coder.EnumType` |  
`coder.FiType` | `coder.PrimitiveType` | `coder.Type` | `coder.newtype` |  
`coder.resize` | `coder.typeof` | `fiaccel`

**Introduced in R2017a**

## **coder.MexConfig**

**Package:** coder

Code acceleration configuration object for use with `fiaccl`

### **Description**

A `coder.MexConfig` object contains all the configuration parameters that the `fiaccl` function uses when accelerating fixed-point code via a generated MEX function. To use this object, first create it using the lowercase `coder.mexconfig` function and then, pass it to the `fiaccl` function using the `-config` option.

### **Construction**

`cfg = coder.mexconfig` creates a `coder.MexConfig` object, `cfg`, for `fiaccl` MEX function generation.

### **Properties**

#### **CompileTimeRecursionLimit**

For compile-time recursion, control the number of copies of a function that are allowed in the generated code. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to 0. The default compile-time recursion limit is high enough for most recursive functions that require compile-time recursion. If code generation fails because of the compile-time recursion limit, and you want compile-time recursion, try to increase the limit. Alternatively, change your MATLAB code so that the code generator uses run-time recursion

**Default:** *integer*, 50

#### **ConstantFoldingTimeout**

Maximum number of constant folder instructions

Specify, as a positive integer, the maximum number of instructions to be executed by the constant folder.

**Default:** 10000

### **DynamicMemoryAllocation**

Dynamic memory allocation for variable-size data

By default, when this property is set to 'Threshold' , dynamic memory allocation is enabled for all variable-size arrays whose size is greater than `DynamicMemoryAllocationThreshold` and `fiaccl` allocates memory for this variable-size data dynamically on the heap. Set this property to 'Off' to allocate memory statically on the stack. Set it to 'AllVariableSizeArrays' to allocate memory for all variable-size arrays dynamically on the heap . You must use dynamic memory allocation for all unbounded variable-size data.

This property, `DynamicMemoryAllocation`, is enabled only when `EnableVariableSizing` is true. When you set `DynamicMemoryAllocation` to 'Threshold', it enables the `DynamicMemoryAllocationThreshold` property.

**Default:** Threshold

### **DynamicMemoryAllocationThreshold**

Memory allocation threshold

Specify the integer size of the threshold for variable-size arrays above which `fiaccl` allocates memory on the heap.

**Default:** 65536

### **EnableAutoExtrinsicCalls**

Specify whether `fiaccl` treats common visualization functions as extrinsic functions. When this option is enabled, `fiaccl` detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. It calls out to MATLAB for these functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

**Default:** true

**EchoExpressions**

Show results of code not terminated with semicolons

Set this property to `true` to have the results of code instructions that do not terminate with a semicolon appear in the MATLAB Command Window. If you set this property to `false`, code results do not appear in the MATLAB Command Window.

**Default:** `true`

**EnableRuntimeRecursion**

Allow recursive functions in the generated code. If your MATLAB code requires run-time recursion and this parameter is `false`, code generation fails.

**Default:** `true`

**EnableDebugging**

Compile generated code in debug mode

Set this property to `true` to compile the generated code in debug mode. Set this property to `false` to compile the code in normal mode.

**Default:** `false`

**EnableVariableSizing**

Variable-sized arrays support

Set this property to `true` to enable support for variable-sized arrays and to enable the `DynamicMemoryAllocation` property. If you set this property to `false`, variable-sized arrays are not supported.

**Default:** `true`

**ExtrinsicCalls**

Extrinsic function calls

An extrinsic function is a function on the MATLAB path that the generated code dispatches to MATLAB software for execution. `fiaccl` does not compile or generate code for extrinsic functions. Set this property to `true` to have `fiaccl` generate code for

the call to a MATLAB function, but not generate the function's internal code. Set this property to `false` to have `fiaccel` ignore the extrinsic function and not generate code for the call to the MATLAB function. If the extrinsic function affects the output of `fiaccel`, a compiler error occurs.

`ExtrinsicCalls` affects how MEX functions built by `fiaccel` generate random numbers when using the MATLAB `rand`, `randi`, and `randn` functions. If extrinsic calls are enabled, the generated mex function uses the MATLAB global random number stream to generate random numbers. If extrinsic calls are not enabled, the MEX function built with `fiaccel` uses a self-contained random number generator.

If you disable extrinsic calls, the generated MEX function cannot display run-time messages from `error` or `assert` statements in your MATLAB code. The MEX function reports that it cannot display the error message. To see the error message, enable extrinsic function calls and generate the MEX function again.

**Default:** `true`

### **GenerateReport**

Code generation report

Set this property to `true` to create an HTML code generation report. Set this property to `false` to not create the report.

**Default:** `false`

### **GlobalDataSyncMethod**

MEX function global data synchronization with MATLAB global workspace

Set this property to `SyncAlways` so synchronize global data at MEX function entry and exit and for all extrinsic calls to ensure maximum consistency between MATLAB and the generated MEX function. If the extrinsic calls do not affect global data, use this option in conjunction with the `coder.extrinsic -sync:off` option to turn off synchronization for these calls to maximize performance.

If you set this property to `SyncAtEntryAndExits`, global data is synchronized only at MEX function entry and exit. If your code contains extrinsic calls, but only a few affect global data, use this option in conjunction with the `coder.extrinsic -sync:on` option to turn on synchronization for these calls to maximize performance.

If you set this property to `NoSync`, no synchronization occurs. Ensure that your MEX function does not interact with MATLAB globals before disabling synchronization otherwise inconsistencies between MATLAB and the MEX function might occur.

**Default:** `SyncAlways`

### **InlineStackLimit**

Stack size for inlined functions

Specify, as a positive integer, the stack size limit on inlined functions.

**Default:** `4000`

### **InlineThreshold**

Maximum size of functions to be inlined

Specify, as a positive integer, the maximum size of functions to be inlined.

**Default:** `10`

### **InlineThresholdMax**

Maximum size of functions after inlining

Specify, as a positive integer, the maximum size of functions after inlining.

**Default:** `200`

### **IntegrityChecks**

Memory integrity

Set this property to `true` to detect any violations of memory integrity in code generated for MATLAB. When a violation is detected, execution stops and a diagnostic message displays. Set this property to `false` to disable both memory integrity checks and the runtime stack.

**Default:** `true`

### **LaunchReport**

Code generation report display



Set this property to `true` to open the HTML code generation report automatically when code generation completes. Set this property to `false` to disable displaying the report automatically. This property applies only if you set the `GenerateReport` property to `true`.

**Default:** `true`

### **ReportPotentialDifferences**

Specify whether to report potential behavior differences between generated code and MATLAB code. If `ReportPotentialDifferences` is `true`, the code generation report has a tab that lists the potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

**Default:** `true`

### **ResponsivenessChecks**

Responsiveness checks

Set this property to `true` to turn on responsiveness checks. Set this property to `false` to disable responsiveness checks.

**Default:** `true`

### **SaturateOnIntegerOverflow**

Integer overflow action

Overflows saturate to either the minimum or maximum value that the data type can represent. Set this property to `true` to have overflows saturate. Set this property to `false` to have overflows wrap to the appropriate value representable by the data type.

**Default:** `true`

### **StackUsageMax**

Maximum stack usage per application

Specify, as a positive integer, 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`.

**Default:** 200000

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

Use the lowercase `coder.mexconfig` function to create a `coder.MexConfig` configuration object. Set this object to disable run-time checks.

```
cfg = coder.mexconfig
% Turn off Integrity Checks, Extrinsic Calls,
% and Responsiveness Checks
cfg.IntegrityChecks = false;
cfg.ExtrinsicCalls = false;
cfg.ResponsivenessChecks = false;
% Use fiaccel to generate a MEX function for file foo.m
fiaccel -config cfg foo
```

## See Also

[coder.ArrayType](#) | [coder.Constant](#) | [coder.EnumType](#) | [coder.FiType](#) | [coder.PrimitiveType](#) | [coder.StructType](#) | [coder.Type](#) | [coder.mexconfig](#) | [coder.newtype](#) | [coder.resize](#) | [coder.typeof](#) | [fiaccel](#)

# coder.SingleConfig class

**Package:** coder

Double-precision to single-precision conversion configuration object

## Description

A `coder.SingleConfig` object contains the configuration parameters that the `convertToSingle` function requires to convert double-precision MATLAB code to single-precision MATLAB code. To pass this object to the `convertToSingle` function, use the `-config` option.

## Construction

`scfg = coder.config('single')` creates a `coder.SingleConfig` object for double-precision to single-precision conversion.

## Properties

### **OutputFileNameSuffix** — Suffix for single-precision file name

'\_single' (default) | character vector

Suffix that the single-conversion process uses for generated single-precision files.

### **LogI0ForComparisonPlotting** — Enable simulation data logging for comparison plotting of input and output variables

false (default) | true

Enable simulation data logging to plot the data differences introduced by single-precision conversion.

### **PlotFunction** — Name of function for comparison plots

' ' (default) | character vector

Name of function to use for comparison plots.

To enable comparison plotting, set `LogIOForComparisonPlotting` to true. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function must accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

### **PlotWithSimulationDataInspector — Specify use of Simulation Data Inspector for comparison plots**

false (default) | true

Use Simulation Data Inspector for comparison plots.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

### **TestBenchName — Name of test file**

' ' (default) | character vector | cell array of character vectors

Test file name or names, specified as a character vector or cell array of character vectors. Specify at least one test file.

If you do not explicitly specify input parameter data types, the conversion uses the first file to infer these data types.

### **TestNumerics — Enable numerics testing**

false (default) | true

Enable numerics testing to verify the generated single-precision code. The test file runs the single-precision code.

## **Methods**

`addFunctionReplacement` Replace double-precision function with single-precision function during single-precision conversion

## Examples

### Generate Single-Precision MATLAB Code

Create a `coder.SingleConfig` object.

```
scfg= coder.config('single');
```

Set the properties of the doubles-to-singles configuration object. Specify the test file. In this example, the name of the test file is `myfunction_test`. The conversion process uses the test file to infer input data types and collect simulation range data. Enable numerics testing and generation of comparison plots.

```
scfg.TestBenchName = 'myfunction_test';  
scfg.TestNumerics = true;  
scfg.LogIOForComparisonPlotting = true;
```

Run `convertToSingle`. Use the `-config` option to specify the `coder.SingleConfig` object that you want to use. In this example, the MATLAB function name is `myfunction`.

```
convertToSingle -config scfg myfunction
```

## See Also

`coder.config` | `convertToSingle`

## Topics

“Generate Single-Precision MATLAB Code”

**Introduced in R2015b**

## **DataTypeWorkflow.Converter class**

**Package:** DataTypeWorkflow

Create fixed-point converter object

### **Description**

A `DataTypeWorkflow.Converter` object contains the methods and parameters needed to collect simulation and derived data, propose and apply data types to the model, and analyze results. This class performs the same fixed-point conversion tasks as the Fixed-Point Tool.

### **Construction**

`Converter = DataTypeWorkflow.Converter(systemToScale)` creates a converter object for the `systemToScale`. The converter object contains the methods and parameters needed to collect simulation and derived data, propose and apply data types to the model, and analyze results.

`Converter = DataTypeWorkflow.Converter(referencedModelSystem, 'TopModel', topModel)` creates a converter object with referenced model `referencedModel` specified as the system to scale. The top model `topModel` is used during the range collection phase of conversion.

### **Input Arguments**

**systemToScale** — Name of system to scale

character vector

The name of the model or subsystem to scale, specified as a character vector.

Example: `converter =  
DataTypeWorkflow.Converter('ex_fixed_point_workflow');`

**referencedModelSystem — Name of referenced model or system inside a referenced model**

character vector

The name of the referenced model or the subsystem within a referenced model to convert to fixed point, specified as a character vector.

**topModel — Name of top level model**

character vector

The name of the top-level model which references `referencedModel`, specified as a character vector. `topModel` is used during the range collection phase of conversion.

## Properties

**CurrentRunName — Current run in the converter object**

character vector

Name of the current run stored in the converter object, specified as a character vector.

Example: `converter.CurrentRunName = 'FixedPointRun'`

Data Types: `char`

**RunNames — Names of all runs**

cell array of character vectors

Names of runs stored in the converter object, specified as a cell array of character vectors.

Data Types: `cell`

**SelectedSystemToScale — Name of model or subsystem**

character vector

Name of the model or subsystem to scale, specified as a character vector.

Data Types: `char`

**ShortcutsForSelectedSystem — Available system shortcuts**

cell array of character vectors

Names of the system settings shortcuts available for the selected system, specified as a cell array of character vectors. You can create additional configurations from within the Fixed-Point Tool. For more information, see “Use Shortcuts to Manage Runs”.

Data Types: `cell`

### **TopModel** — Name of top level model

character vector

The name of the top-level model which references `referencedModel`, specified as a character vector. `topModel` is used during the range collection phase of conversion.

Data Types: `char`

## **Methods**

<code>applyDataTypes</code>	Apply proposed data types to model
<code>applySettingsFromRun</code>	Apply system settings used in previous run to model
<code>applySettingsFromShortcut</code>	Apply settings from shortcut to model
<code>deriveMinMax</code>	Derive range information for model
<code>proposeDataTypes</code>	Propose data types for system
<code>results</code>	Find results for selected system in converter object
<code>proposalIssues</code>	Get results which have comments associated with them
<code>saturationOverflows</code>	Get results where saturation occurred
<code>simulateSystem</code>	Simulate converter’s system
<code>verify</code>	Compare behavior of baseline and autoscaled systems
<code>wrapOverflows</code>	Get results where wrapping occurred

## **Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).



## Alternatives

The `DataTypeWorkflow.Converter` class offers a command-line approach to using the Fixed-Point Tool. See `fxptdlg` for more information.

## See Also

`DataTypeWorkflow.ProposalSettings`

## Topics

[“Convert a Model to Fixed Point Using the Command Line”](#)

[“The Command-Line Interface for the Fixed-Point Tool”](#)

## DataTypeWorkflow.DiffRunResult class

**Package:** DataTypeWorkflow

Results from comparing two simulation runs

### Description

The `DataTypeWorkflow.DiffRunResult` class manages the results from comparing two simulation runs. A `DataTypeWorkflow.DiffRunResult` object contains a `DataTypeWorkflow.DiffSignalResult` object for each signal compared.

### Construction

The `DataTypeWorkflow.Converter.compareRuns` method returns a handle to a `DataTypeWorkflow.DiffRunResult` object.

### Properties

**count** — Number of compared signal results

scalar

Number of compared signal results, stored as an `int32`.

Data Types: `int32`

**dateCreated** — Date of object creation

serial date number

Date of object creation, stored in serial date number format. For more information, see now in the MATLAB documentation.

Data Types: `double`

**matlabVersion** — Version of MATLAB used

character vector

Version of MATLAB used to create instance of `DataTypeWorkflow.DiffRunResult`, stored as a character vector.

Data Types: `char`

**runName1 — Name of first run**

character vector

Name of first run compared, specified as a character vector.

Data Types: `char`

**runName2 — Name of second run**

character vector

Name of second run compared, specified as a character vector.

Data Types: `char`

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

`DataTypeWorkflow.DiffSignalResult` | `Simulink.sdi.DiffRunResult`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

## DataTypeWorkflow.DiffSignalResult class

**Package:** DataTypeWorkflow

Results from comparing two signals

### Description

The `DataTypeWorkflow.DiffSignalResult` object manages the results from comparing two signals. A `DataTypeWorkflow.DiffSignalResult` object contains the value differences of the signals, the tolerance data, and the data after any specified synchronization methods are performed.

### Construction

The `DataTypeWorkflow.Converter.compareRuns` method returns a handle to a `DataTypeWorkflow.DiffSignalResult` object, which contains the comparison results.

### Properties

**diff** — Value differences after synchronizing data

`timeseries` object

A MATLAB `timeseries` object specifying the value differences after synchronizing the two time series data.

**match** — Whether the two `timeseries` objects match

0 | 1

A boolean indicating if the two `timeseries` objects match according to the specified tolerance and time synchronization options.

Data Types: `logical`

**result1** — Result object to compare

`DataTypeWorkflow.Result` object

DataTypeWorkflow.Result object that is being compared.

**result2 — Result object to compare**

DataTypeWorkflow.Result object

DataTypeWorkflow.Result object that is being compared.

**sync1 — Time series 1 after synchronization has been applied**

timeseries object

A MATLAB `timeseries` object specifying time series 1 after synchronization has been applied.

**sync2 — Time series 2 after synchronization has been applied**

timeseries object

A MATLAB `timeseries` object specifying time series 2 after synchronization has been applied.

**tol — Absolute tolerance value at each synchronized time point**

timeseries object

A MATLAB `timeseries` object specifying the actual absolute tolerance value at each synchronized time point.

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

DataTypeWorkflow.Result | Simulink.sdi.DiffSignalResult

## Topics

“Convert a Model to Fixed Point Using the Command Line”

## DataTypeWorkflow.ProposalSettings class

**Package:** DataTypeWorkflow

Proposal settings object for data type proposals

### Description

The `DataTypeWorkflow.ProposalSettings` class manages the properties related to how data types are proposed for a model.

### Construction

`propSettings = DataTypeWorkflow.ProposalSettings` creates a proposal settings object. A proposal settings object manages properties related to how data types are proposed for a model, including default floating point data type, and safety margins for the proposed data types.

### Properties

#### **DefaultWordLength — Default word length for floating point signals**

16 | scalar

Default word length for floating-point signals, specified as a double. Use this setting when the `ProposeFractionLength` property is set to `true`.

Data Types: double

#### **DefaultFractionLength — Default fraction length for floating-point signals**

4 | scalar

Default fraction length for floating-point signals, specified as a double. Use this setting when the `ProposeWordLength` property is set to `true`.

Data Types: double

**ProposeFractionLength — Propose fraction lengths for specified word length**`true (default) | false`

Set to `true` to propose fraction lengths for the default word length specified in the `DefaultWordLength` property. Setting this property to `true` automatically sets the `ProposeWordLength` property to `false`.

Data Types: `logical`

**ProposeForInherited — Propose fixed-point data types for objects with an inherited output data type**`true (default) | false`

Specify whether to propose fixed-point data types for objects in the system with inherited output data types.

Data Types: `logical`

**ProposeForFloatingPoint — Propose fixed-point data types for objects with a floating-point output data type**`true (default) | false`

Specify whether to propose fixed-point data types for objects in the system with floating-point output data types.

Data Types: `logical`

**ProposeSignedness — Propose signedness for objects in the system**`true (default) | false`

Specify whether to propose signedness for objects in the system.

The software bases the signedness proposal on collected range information and block constraints. Signals that are always strictly positive get an unsigned data type proposal, gaining an additional bit of precision. If you set this property to `false`, the software proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the software will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

Data Types: `logical`

**ProposeWordLength — Propose word lengths for specified default fraction lengths**

false (default) | true

Set to true to propose word lengths for the default fraction length specified in the `DefaultFractionLength` property. Setting this property to true automatically sets the `ProposeFractionLength` property to false.

Data Types: logical

**SafetyMargin — Safety margin for simulation minimum and maximum values**

0 (default) | scalar

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter. This allows you to specify a range different from that obtained from the simulation run.

Example: A value of 55 specifies that a range at least 55 percent larger is desired. A value of -15 specifies that a range of up to 15 percent smaller is acceptable.

Data Types: double

**UseDerivedMinMax — Whether to use derived ranges to propose data types**

true (default) | false

Specify whether to use derived ranges for data type proposals.

Data Types: logical

**UseSimMinMax — Whether to use simulation ranges to propose data types**

true (default) | false

Specify whether to use simulation ranges for data type proposals.

Data Types: logical

## Methods

<code>addTolerance</code>	Specify numeric tolerance for converted system
<code>clearTolerances</code>	Clear all tolerances specified on a system
<code>showTolerances</code>	Show tolerances specified for a system



## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Alternatives

The properties of the `DataTypeWorkflow.ProposalSettings` class can also be controlled from the **Settings** menu in the Fixed-Point Tool. See `fxptdlg` for more information.

## See Also

`DataTypeWorkflow.Converter`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

## DataTypeWorkflow.Result class

**Package:** DataTypeWorkflow

Object containing run result information

### Description

The `DataTypeWorkflow.Result` class manages the results of simulation, derivation, and data type proposals.

### Construction

The `DataTypeWorkflow.Converter.results` method returns a handle to a `DataTypeWorkflow.Result` object.

### Properties

#### **Comments — Comments associated with the signal**

cell array of character vectors

Any comments associated with the signal, stored as a cell array of character vectors.

Data Types: `cell`

#### **CompiledDataType — Data type used during simulation**

character vector

Character vector containing the data type used during simulation.

Data Types: `char`

#### **DerivedMax — Derived maximum value**

scalar

The derived maximum value for the signal or internal data based on specified design maximums.

Data Types: double

**DerivedMin — Derived minimum value**

scalar

The derived minimum value for the signal or internal data based on specified design minimums.

Data Types: double

**ProposedDataType — Proposed data type**

character vector

Character vector containing the data type proposed for the signal or internal data type associated with this result.

Data Types: char

**ResultName — Name of signal**

character vector

The name of the signal or internal data associated with this result, stored as a character vector.

Data Types: char

**RunName — Name of run associated with result**

character vector

Name of run associated with result, specified as a character vector.

Data Types: char

**Saturations — Number of saturations that occurred**

scalar

The number of occurrences where the signal or internal data associated with this result saturated at the maximum or minimum of its specified data type. This field is cumulative of all the executions of the run the result is associated with.

Data Types: double

**SimMax — Simulation maximum**

scalar

The maximum values obtained for the signal or internal data during all of the saved executions of the run this result is associated with.

Data Types: `double`

### **SimMin — Simulation minimum**

scalar

The minimum value obtained for the signal or internal data during all of the saved executions of the run this result is associated with.

Data Types: `double`

### **SpecifiedDataType — Specified data type of signal**

character vector

The data type currently specified for a signal, which will take effect the next time the system is run.

Data Types: `char`

### **Wraps — Number of wraps that occurred**

scalar

The number of occurrences where the signal or internal data associated with this result wrapped around the maximum or minimum of its specified data type. This field is cumulative of all the executions of the run the result is associated with.

Data Types: `double`

## **Copy Semantics**

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## **See Also**

`DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings`

## **Topics**

“Convert a Model to Fixed Point Using the Command Line”

## DataTypeWorkflow.VerificationResult class

**Package:** DataTypeWorkflow

Verification results after converting a system to fixed point

### Description

A `VerificationResult` object contains the results after converting a system to fixed point. The `VerificationResult` indicates whether a conversion was successful based on tolerances specified on the `DataTypeWorkflow.ProposalSettings` object used during the conversion.

The `DataTypeWorkflow.VerificationResult` class is a handle class.

### Creation

`verificationResult = verify(converter, BaselineRunName, RunName)` simulates the system under design specified by the `DataTypeWorkflow.Converter` object, `converter`, and stores the run information in a new run, `RunName`. It returns a `DataTypeWorkflow.VerificationResult` object which compares the baseline and verification runs.

The `DataTypeWorkflow.Converter` object contains instrumentation data from the run specified by `BaselineRunName`, as well as the tolerances specified on the associated `DataTypeWorkflow.ProposalSettings` object. The software determines if the behavior of the verification run is acceptable using the tolerances specified on the `ProposalSettings` object.

### Properties

#### **RunName — Name of the verification run to create**

character vector

Name of the run to create during the embedded simulation, specified as a character vector.

Data Types: char

**BaselineRunName — Run to compare against**

character vector

Baseline run to compare against, specified as a character vector

Data Types: char

**Status — Whether the verification run meets the specified tolerances**

Pass | Warn | Fail

Whether the verification run meets the specified tolerances, returned as either Pass, Warn, or Fail. For additional details, use the explore method and open the Simulation Data Inspector.

Status	Description
Pass	All signals with a specified tolerance on the associated ProposalSettings object are within the specified tolerances in the verification run.
Fail	One or more signals with a specified tolerance on the associated ProposalSettings object are not within the specified tolerances in the verification run.

Data Types: char

## Methods

### Public Methods

explore Explore comparison of baseline and fixed-point implementations

### See Also

DataTypeWorkflow.Converter | DataTypeWorkflow.ProposalSettings

## **Topics**

“Convert a Model to Fixed Point Using the Command Line”

**Introduced in R2019a**



# fixed.DataGenerator

Creates value set and generates data

## Description

Use the `fixed.DataSpecification` and `fixed.DataGenerator` objects to generate simulation inputs to test the full operating range of your designs.

## Creation

## Syntax

```
data = fixed.DataGenerator(Name, Value)
```

## Description

`data = fixed.DataGenerator(Name, Value)` creates a `DataGenerator` object with additional properties specified as `Name`, `Value` pair arguments.

## Properties

### **DataSpecifications** — Properties of generated data

`fixed.DataSpecification` object | cell array of `fixed.DataSpecification` objects

Properties of the data to generate, specified as a `fixed.DataSpecification` object.

Specifying a cell array of `DataSpecification` objects produces a single `DataGenerator` object for input to a system with the same number of inputs and in the same order as elements in the cell array.

### **NumDataPointsLimit** — Maximum number of data points in generated data

100000 (default) | integer-valued scalar

Maximum number of data points in generated data, specified as an integer-valued scalar. For more information, see `getNumDataPointsInfo`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

<code>getUniqueValues</code>	Get unique values from <code>fixed.DataGenerator</code> object
<code>getNumDataPointsInfo</code>	Get information about number of data points in generated data
<code>outputAllData</code>	Get data from <code>fixed.DataGenerator</code> object

## Examples

### Create a `fixed.DataGenerator` object

Create a `DataGenerator` object by specifying a `DataSpecification` object in the constructor.

Create the `DataSpecification` object with an interval from  $-2\pi$  to  $2\pi$  with a data type of `single`.

```
dataspec = fixed.DataSpecification('single', 'Intervals', {-2*pi, 2*pi})
```

```
dataspec =  
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'single'  
        Intervals: [-6.2832,6.2832]  
    ExcludeDenormals: false  
    ExcludeNegativeZero: false  
    MandatoryValues: <empty>  
        Complexity: 'real'  
        Dimensions: 1
```

Use the `DataSpecification` object to create a `DataGenerator` object. Limit the number of data points in the generated data to 5000 points. You can specify these properties as name-value pairs in the constructor of the `DataGenerator` object.

```
datagen = fixed.DataGenerator('DataSpecifications', dataspec, 'NumDataPointsLimit', 5000)
```

```

datagen =
  fixed.DataGenerator with properties:

    DataSpecifications: {[1x1 fixed.DataSpecification]}
    NumDataPointsLimit: 5000

```

Use the `outputAllData` function to see the generated data.

```
myData = outputAllData(datagen)
```

```
myData = 1x262 single row vector
```

```

-6.2832   -6.2832   -4.0000   -4.0000   -4.0000   -2.0000   -2.0000   -2.0000   -1.0000

```

## Algorithms

### Data Generation for One-Dimensional, Two-Dimensional, and Complex Data

When you use a `DataGenerator` object to generate data for a `DataSpecification` object with the `Dimensions` property set to 1, the output data always contains the minimum and maximum values of the specified intervals, and any values specified by the `MandatoryValues` property.

When you generate data for a `DataSpecification` object with the `Dimensions` property set to a value greater than 1, the output is generated by taking a cartesian product of the one-dimensional output.

For example, consider the following two `DataSpecification` objects. The two objects are identical except that one is one-dimensional, and the other is two-dimensional.

```

dataspec_1d = fixed.DataSpecification('single',...
    'Intervals', {-1,1}, 'Dimensions',1);
dataspec_2d = fixed.DataSpecification('single',...
    'Intervals', {-1,1}, 'Dimensions',2);

```

Create two `DataGenerator` objects based on these specifications. Set the maximum number of data points in the generated data to `inf`.

```
datagen_1d = fixed.DataGenerator('DataSpecifications', ...
    dataspec_1d, 'NumDataPointsLimit', inf);
datagen_2d = fixed.DataGenerator('DataSpecifications', ...
    dataspec_2d, 'NumDataPointsLimit', inf);
```

Get the size of the generated data for each of the configurations.

```
size_1d_data = size(outputAllData(datagen_1d))
size_2d_data = size(outputAllData(datagen_2d))
```

```
size_1d_data =
    1    244
size_2d_data =
    2    59536
```

The length of the two-dimensional data is exactly the squared length of the one-dimensional data.

The `DataGenerator` generates complex data in a similar way to the two-dimensional data. Create a `DataSpecification` object with `Dimensions` set to 1 and the `Complexity` set to `complex`. Create a `DataGenerator` object using this specification.

```
dataspec_complex = fixed.DataSpecification('single', ...
    'Intervals', {-1,1}, 'Dimensions', 1, 'Complexity', 'complex');

datagen_complex = fixed.DataGenerator('DataSpecifications', ...
    dataspec_complex, 'NumDataPointsLimit', inf);
```

Get the size of the generated data from this configuration.

```
size_complex_data = size(outputAllData(datagen_complex))
size_complex_data =
    1    59536
```

The length of the output data for the one-dimensional complex data is the same as the length of the two-dimensional real data.

## See Also

### Objects

`fixed.DataSpecification` | `fixed.Interval`

**Introduced in R2019b**

# fixed.DataSpecification

Specify properties of data to generate

## Description

Use the `fixed.DataSpecification` and `fixed.DataGenerator` objects to generate simulation inputs to test the full operating range of your designs.

## Creation

## Syntax

```
dataspec = fixed.DataSpecification(numerictype)
dataspec = fixed.DataSpecification(numerictype, Name, Value)
```

## Description

`dataspec = fixed.DataSpecification(numerictype)` creates a `DataSpecification` object with default property values and data type specified by `numerictype`.

`dataspec = fixed.DataSpecification(numerictype, Name, Value)` creates a `DataSpecification` object with data type specified by `numerictype`, and additional properties specified as `Name, Value` pair arguments.

## Input Arguments

### **numerictype** — Data type of generated data

character vector | `Simulink.NumericType` object | `embedded.numerictype` object

Data type of the generated data, specified as a string or character vector that evaluates to a numeric data type, or as a `Simulink.NumericType` or `embedded.numerictype` object.

Example: `dataspec = fixed.DataSpecification('double')`

Example: `dataspec = fixed.DataSpecification('fixdt(1,16,4)')`

Example: `dataspec = fixed.DataSpecification(Simulink.NumericType);`

## Properties

### **DataTypeStr — Data type of generated data**

character vector | `Simulink.NumericType` object | `embedded.numericType` object

Data type of the generated data, specified as a string or character vector that evaluates to a numeric data type, or as a `Simulink.NumericType` or `embedded.numericType` object.

This property cannot be edited after construction.

### **Intervals — Intervals within which to generate numeric data**

`fixed.Interval` object | array of `fixed.Interval` objects | cell array containing inputs to `fixed.Interval` constructor

Numeric intervals in which to generate numeric data, specified as a `fixed.Interval` object, an array of `fixed.Interval` objects, or a cell array containing inputs to the `fixed.Interval` constructor.

If you do not specify an interval, the default interval uses end points equal to the minimum and maximum representable values of the specified numeric type.

Example: `dataspec.Intervals = {-1,1};`

Example: `dataspec.Intervals = fixed.Interval(-1,1);`

### **ExcludeDenormals — Whether to exclude denormal numbers from generated data**

false (default) | true

Whether to exclude denormal numbers from generated data, specified as a logical.

This property is only applicable when the `DataTypeStr` property is a floating-point type.

Data Types: `logical`

**ExcludeNegativeZero — Whether to exclude negative zero from generated data**

false (default) | true

Whether to exclude negative zero from generated data, specified as a logical.

This property is only applicable when the `DataTypeStr` property is a floating-point type.

Data Types: logical

**MandatoryValues — Values to include in the generated data**

<empty> (default) | scalar | vector | matrix | multidimensional array

Values to include in the generated data, specified as a scalar, vector, matrix, or multidimensional array. If the values specified in `MandatoryValues` are outside the range of the data type specified in `DataTypeStr`, the values are saturated to the nearest representable value.

Example: `dataspec.MandatoryValues = [-215, 216];`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

**Complexity — Complexity of generated data**

'real' (default) | 'complex'

Complexity of the generated data, specified as either 'real' or 'complex'.

Example: `dataspec.Complexity = 'complex';`

Data Types: char | string

**Dimensions — Dimension of the generated data**

1 (default) | positive scalar integer | row vector of positive integers

Dimension of the generated data, specified as a positive scalar integer or row vector of positive integers.

Example: `dataspec.Dimensions = 3;`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



## Object Functions

contains	Determine whether value domain of a DataSpecification object contains a specified value
applyOnRootInport	Apply properties to Inport block

## Examples

### Create a fixed.DataSpecification object

Create a fixed.DataSpecification object with default property values and an int16 data type.

```
dataspec = fixed.DataSpecification('int16')
```

```
dataspec =
  fixed.DataSpecification with properties:
```

```
    DataTypeStr: 'int16'
      Intervals: [-32768,32767]
MandatoryValues: <empty>
    Complexity: 'real'
    Dimensions: 1
```

The default interval of the DataSpecification object is equal to the range of the data type specified in the constructor.

### Create a fixed.DataSpecification object from a fixed.Interval object

Create a fixed.Interval object specifying a range of  $-\pi$  to  $\pi$ .

```
interval = fixed.Interval(-pi,pi)
```

```
interval =
  [-3.1416,3.1416]
```

```
1x1 fixed.Interval with properties:
```

```
        LeftEnd: -3.1416
        RightEnd: 3.1416
        IsLeftClosed: true
        IsRightClosed: true
```

Create a `DataSpecification` object using this interval and a data type of `fixdt(1,16,10)`.

```
dataspec = fixed.DataSpecification('fixdt(1,16,10)', 'Intervals', interval)
```

```
dataspec =
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'sfix16_En10'
        Intervals: [-3.1416,3.1416]
        MandatoryValues: <empty>
        Complexity: 'real'
        Dimensions: 1
```

Alternatively, you can specify the interval as a cell array of inputs to the `fixed.Interval` constructor. The following code generates an equivalent `DataSpecification` object.

```
dataspec = fixed.DataSpecification('fixdt(1,16,10)', 'Intervals', {-pi, pi})
```

```
dataspec =
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'sfix16_En10'
        Intervals: [-3.1416,3.1416]
        MandatoryValues: <empty>
        Complexity: 'real'
        Dimensions: 1
```

### Create a `DataSpecification` object that includes NaN and Inf

You can include NaN and Inf values in the generated data by specifying these values as intervals in an `Interval` object.

The following code creates a `DataSpecification` object that references an array of interval objects that include the values `-Inf`, `Inf`, `NaN`, and the range `[-1, 1]`.

```
dataspec = fixed.DataSpecification('single', 'Intervals', ...  
    {-Inf}, {Inf}, {NaN}, {-1,1})
```

```
dataspec =
```

```
fixed.DataSpecification with properties:
```

```
    DataTypeStr: 'single'  
      Intervals: [-Inf] [-1,1] [Inf] [NaN]  
ExcludeDenormals: false  
ExcludeNegativeZero: false  
MandatoryValues: <empty>  
    Complexity: 'real'  
    Dimensions: 1
```

## See Also

### Objects

`fixed.DataGenerator` | `fixed.Interval`

### Introduced in R2019b

## fixed.Interval

Define interval of values

### Description

A `fixed.Interval` object defines an interval of real-world values. Use the `Interval` object to specify a range of values in a `fixed.DataSpecification` object.

### Creation

### Syntax

```
interval = fixed.Interval  
interval = fixed.Interval(a)  
interval = fixed.Interval(a, b)  
interval = fixed.Interval(a, b, endnotes)  
interval = fixed.Interval(a, b, Name, Value)  
interval = fixed.Interval(numerictype)  
interval = fixed.Interval({ __ }, ..., { __ })
```

### Description

`interval = fixed.Interval` creates a unit interval,  $[0,1]$ .

`interval = fixed.Interval(a)` creates a degenerate interval, containing only the value `a`.

`interval = fixed.Interval(a, b)` creates a closed interval from `a` to `b`.

`interval = fixed.Interval(a, b, endnotes)` creates an interval from `a` to `b`, with the `endnotes` argument specifying whether the interval is open or closed.

`interval = fixed.Interval(a, b, Name, Value)` creates an interval from `a` to `b` with the `IsLeftClosed` and `IsRightClosed` properties specified as `Name`, `Value` pair arguments.

`interval = fixed.Interval(numericType)` creates an interval or array of intervals with end points equal to the minimum and maximum representable values of the specified numeric type.

`interval = fixed.Interval({__}, ..., {__})` returns an array of `Interval` objects, where each cell array specifies the arguments for one or more of the objects.

## Input Arguments

### **a** — Left endpoint of interval

scalar | vector

Left endpoint of interval, specified as a scalar or vector.

### **b** — Right endpoint of interval

scalar | vector

Right endpoint of interval, specified as a scalar or vector.

### **endnotes** — Whether the interval is open or closed

'[]' (default) | '[' | '(' | ')' | ']'

Argument indicating whether the interval is closed, open, or half-open, specified as one of the following character vectors.

Endnotes	Description
'[]'	Generates a closed set, which includes both of its endpoints.
'['	Generates a half-open interval, in which the first endpoint is included, but the second is not included in the set.
'('	Generates a half-open interval, in which the first endpoint is not included, but the second is included in the set.

Endnotes	Description
'()'	Generates an open set, in which neither endpoint is included in the set.

Example: `interval = fixed.Interval(1, 10, '()');`

### **numerictype — Numeric data type**

`Simulink.Numerictype` object | `embedded.numerictype` object | character vector

Numeric data type whose range of representable values defines the `Interval` object, specified as a `Simulink.Numerictype` object, an `embedded.numerictype` object, or a character vector representing a numeric data type, for example, `'single'`.

When `numerictype` is `'double'`, `'single'`, or `'half'`, the output `Interval` object is an array of 4 `Interval` objects with intervals `[-Inf]`, `[Inf]`, `[NaN]`, and `[-realmax, realmax]`. For more information on representable values of a data type, see `realmax`.

Example: `interval = fixed.Interval('fixdt(1,16,8)');`

## **Properties**

### **LeftEnd — Left endpoint of interval**

0 (default) | scalar

Left endpoint of interval, specified as a scalar.

This property cannot be edited after object creation.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

### **RightEnd — Right endpoint of interval**

1 (default) | scalar

Right endpoint of interval, specified as a scalar.

This property cannot be edited after object creation.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

**IsLeftClosed — Whether the left end of the interval is closed**

true (default) | false

Whether the left end of the interval is closed, specified as a logical value.

This property cannot be edited after object creation.

Data Types: logical

**IsRightClosed — Whether the right end of the interval is closed**

true (default) | false

Whether the right end of the interval is closed, specified as a logical value.

This property cannot be edited after object creation.

Data Types: logical

**Object Functions**

contains	Determine if one fixed.Interval object contains another
intersect	Intersection of fixed.Interval objects
isDegenerate	Determine whether the left and right ends of a fixed.Interval object are degenerate
isLeftBounded	Determine whether a fixed.Interval object is left-bounded
isRightBounded	Determine whether the a fixed.Interval object is right-bounded
isnan	Determine whether a fixed.Interval object is NaN
overlaps	Determine if two fixed.Interval objects overlap
quantize	Quantize interval to range of numeric data type
setdiff	Set difference of fixed.Interval objects
union	Union of fixed.Interval objects
unique	Get set of unique values in fixed.Interval object

**Examples****Create a degenerate interval**

Create a degenerate interval, containing only a single point.

```
interval = fixed.Interval(pi)
```

```
interval =  
  [3.1416]  
  
1x1 fixed.Interval with properties:  
    LeftEnd: 3.1416  
    RightEnd: 3.1416  
    IsLeftClosed: true  
    IsRightClosed: true
```

This is equivalent to creating an interval with two equivalent endpoints.

```
interval = fixed.Interval(pi, pi)  
  
interval =  
  [3.1416]  
  
1x1 fixed.Interval with properties:  
    LeftEnd: 3.1416  
    RightEnd: 3.1416  
    IsLeftClosed: true  
    IsRightClosed: true
```

### Create an open interval

Specify end notes for an interval to create an open interval.

```
interval = fixed.Interval(-1, 1, '()')  
  
interval =  
  (-1,1)  
  
1x1 fixed.Interval with properties:  
    LeftEnd: -1  
    RightEnd: 1  
    IsLeftClosed: false  
    IsRightClosed: false
```

To create an interval that includes the first endpoint, but not the second, specify the end notes as '['



```

interval = fixed.Interval(-1, 1, '[')
interval =
    [-1,1)

1x1 fixed.Interval with properties:
    LeftEnd: -1
    RightEnd: 1
    IsLeftClosed: true
    IsRightClosed: false

```

### Create an interval with the range of a numeric data type

When you specify a numeric data type in the constructor of the `fixed.Interval` object, the range of the interval is set to the range of the data type.

Create an interval with the range of an `int8` data type.

```

interval_int8 = fixed.Interval('int8')
interval_int8 =
    [-128,127]

1x1 fixed.Interval with properties:
    LeftEnd: -128
    RightEnd: 127
    IsLeftClosed: true
    IsRightClosed: true

```

You can also specify a `Simulink.NumericType` to create an interval with the same range as the range representable by the `NumericType` object.

```

myNumericType = Simulink.NumericType;
myNumericType.DataTypeMode = "Fixed-point: binary point scaling";
myNumericType.Signedness = 'Unsigned';
myNumericType.WordLength = 16;
myNumericType.FractionLength = 14

myNumericType =
    NumericType with properties:

```

```
    DataTypeMode: 'Fixed-point: binary point scaling'  
    Signedness: 'Unsigned'  
    WordLength: 16  
    FractionLength: 14  
    IsAlias: 0  
    DataScope: 'Auto'  
    HeaderFile: ''  
    Description: ''
```

```
interval_16_14 = fixed.Interval(myNumericType)
```

```
interval_16_14 =  
    [0,3.9999]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 0  
    RightEnd: 3.9999  
    IsLeftClosed: true  
    IsRightClosed: true
```

### Create an array of fixed.Interval objects

To create an array of `fixed.Interval` objects, in the constructor of the `Interval` object, you can specify a series of cell arrays, each of which contain the arguments of an `Interval` object.

```
intervalarray = fixed.Interval({-1, 1}, {5, 10, '[]'},...  
    {1000, 1500, 'IsLeftClosed', 1, 'IsRightClosed', 0},...  
    {'int8'})
```

```
intervalarray=1x4 object  
    [-1,1]    [5,10)    [1000,1500)    [-128,127]
```

```
1x4 fixed.Interval with properties:
```

```
    LeftEnd  
    RightEnd  
    IsLeftClosed  
    IsRightClosed
```

## See Also

### Objects

`fixed.DataGenerator` | `fixed.DataSpecification`

**Introduced in R2019b**

## FunctionApproximation.LUTMemoryUsageCalculator class

**Package:** FunctionApproximation

Calculate total memory used by lookup table blocks in a model

### Description

The `FunctionApproximation.LUTMemoryUsageCalculator` class helps to calculate the total memory used by all lookup table blocks, including 1-D Lookup Table, 2-D Lookup Table, and n-D Lookup Table, used in a model.

### Construction

`calculator = FunctionApproximation.LUTMemoryUsageCalculator()` creates a `FunctionApproximation.LUTMemoryUsageCalculator` object. Use the `lutmemoryusage` method to calculate the memory used by all lookup table blocks in a model.

### Methods

`lutmemoryusage` Calculate total memory used by lookup table blocks in a model

### Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Calculate the Total Memory Used by Lookup Tables in a Model

Use the `FunctionApproximation.LUTMemoryUsageCalculator` class to calculate the total memory used by lookup table blocks in a model.

Create a `FunctionApproximation.LUTMemoryUsageCalculator` object.

```
calculator = FunctionApproximation.LUTMemoryUsageCalculator
```

Use the `lutmemoryusage` method to get the total memory used by the lookup table blocks in the `sldemo_fuelsys` model.

```
load_system('sldemo_fuelsys')
lutmemoryusage(calculator, 'sldemo_fuelsys')
```

```
ans =
```

```
6×1 table
```

```
sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throttle
sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estimati
sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Estim
sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki
Total
```

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

`FunctionApproximation.LUTSolution` | `FunctionApproximation.Options` | `FunctionApproximation.Problem`

**Functions**

approximate | compare | displayallsolutions | displayfeasiblesolutions |  
lutmemoryusage | solutionfromID | solve | totalmemoryusage

**Topics**

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

# FunctionApproximation.LUTSolution class

**Package:** FunctionApproximation

Optimized lookup table data or lookup table data approximating a math function

## Description

A `FunctionApproximation.LUTSolution` object contains optimized lookup table data or lookup table data approximating a math function. To create a `FunctionApproximation.LUTSolution` object, use the `solve` method on a `FunctionApproximation.Problem` object. To generate a subsystem containing the lookup table approximate or the optimized lookup table, use the `approximate` method of the `FunctionApproximation.LUTSolution` object.

You can save a `FunctionApproximation.LUTSolution` object to a MAT-file and restore the solution later.

## Construction

`solution = solve(problem)` solves the problem defined by the `FunctionApproximation.Problem` object, `problem`, and returns the approximation or optimization, `solution`, as a `FunctionApproximation.LUTSolution` object.

## Input Arguments

**problem** — **Function to approximate, or lookup table to optimize**

`FunctionApproximation.Problem` object

Function to approximate, or lookup table to optimize, and the constraints to consider during the optimization, specified as a `FunctionApproximation.Problem` object.

## Properties

### **ID — ID of the solution**

scalar integer

ID of the solution, specified as a scalar integer.

This property is read-only.

Data Types: double

### **Feasible — Whether the approximation meets the constraints**

true | false

Whether the approximation or optimization specified by the `FunctionApproximation.LUTSolution` object, `solution`, meets the constraints specified in the `FunctionApproximation.Problem` object, `problem`, and its associated `FunctionApproximation.Options`.

This property is read-only.

Data Types: logical

### **AllSolutions — All solutions, including infeasible solutions**

vector of `FunctionApproximation.LUTSolution` objects

All solutions found during the approximation, including infeasible solutions, specified as a vector of `FunctionApproximation.LUTSolution` objects.

This property is read-only.

### **FeasibleSolutions — All solutions that meet the constraints**

vector of `FunctionApproximation.LUTSolution` objects

All solutions meeting the specified constraints, specified as a vector of `FunctionApproximation.LUTSolution` objects.

This property is read-only.

### **PercentReduction — Reduction in memory of lookup table**

scalar

If the original `FunctionApproximation.Problem` object specified a lookup table block to optimize, the `PercentReduction` property indicates the reduction in memory from



the original lookup table. If the original `FunctionApproximation.Problem` object specified a math function or function handle, the `PercentReduction` is `-Inf`.

This property is read-only.

Data Types: `double`

### **SourceProblem — Problem object approximated by the solution**

`FunctionApproximation.Problem` object

`FunctionApproximation.Problem` object that the `FunctionApproximation.LUTSolution` object approximates.

This property is read-only.

### **TableData — Lookup table data**

struct

Struct containing data related to lookup table approximation. The struct has the following fields.

- `BreakpointValues` - Breakpoints of the lookup table
- `BreakpointDataTypes`- Data type of the lookup table breakpoints
- `TableValues` - Values in the lookup table
- `TableDataType` - Data type of the table data
- `IsEvenSpacing` - Boolean value indicating if the breakpoints are evenly spaced.

This property is read-only.

## Methods

approximate	Generate a Lookup Table block from a <code>FunctionApproximation.LUTSolution</code>
compare	Compare numerical results of <code>FunctionApproximation.LUTSolution</code> to original function or lookup table
displayallsolutions	Display all solutions found during function approximation
displayfeasiblesolutions	Display all feasible solutions found during function approximation
getErrorValue	Get the total error of the lookup table approximation
replaceWithApproximate	Replace block with the generated lookup table approximation
revertToOriginal	Revert the block that was replaced by the approximation back to its original state
solutionfromID	Access a solution found during the approximation process
totalmemoryusage	Calculate total memory used by a lookup table approximation

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

### Apps

[Lookup Table Optimizer](#)

### Classes

[FunctionApproximation.Problem](#) |  
[FunctionApproximation.LUTMemoryUsageCalculator](#) |  
[FunctionApproximation.Options](#)

### Functions

[approximate](#) | [compare](#) | [solve](#)

## **Topics**

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## FunctionApproximation.Options class

**Package:** FunctionApproximation

Specify additional options to use with `FunctionApproximation.Problem` object

### Description

The `FunctionApproximation.Options` object contains additional options for defining a `FunctionApproximation.Problem` object.

### Construction

`options = FunctionApproximation.Options()` creates a `FunctionApproximation.Options` object to use as an input to a `FunctionApproximation.Problem` object. The output, `options`, uses default property values.

`options = FunctionApproximation.Options(Name,Value)` creates a `FunctionApproximation.Options` object with property values specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes ( `'` ). You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### Properties

**AbsTol** — **Absolute tolerance of difference between original and approximate**  
non-negative scalar

Maximum tolerance of the absolute value of the difference between the original output value and the output value of the approximation, specified as a non-negative scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

**AllowUpdateDiagram — Whether to allow updating of the model diagram during the approximation process**

1 (default) | 0

Whether to allow updating of the model diagram during the approximation process, specified as a logical. This property is only relevant for `FunctionApproximation.Problem` objects that specify a Lookup Table block, or a Math Function block as the item to approximate.

Data Types: logical

**AUTOSARCompliant — Whether the generated lookup table block is an AUTOSAR block**

0 (default) | 1

Whether the generated lookup table is AUTOSAR compliant. When this property is set to 1 (true), the generated lookup table is a Curve or Map block from the AUTOSAR Blockset. When this property is set to 1, the data type of the table data must equal the output data type of the block.

Setting this property to 1 (true) checks out a AUTOSAR Blockset license when you use the `approximate` or `replaceWithApproximate` methods.

Data Types: logical

**BreakpointSpecification — Spacing of breakpoint data**

ExplicitValues (default) | EvenSpacing | EvenPow2Spacing

Spacing of breakpoint data, specified as one of the following values.

Breakpoint Specification	Description
ExplicitValues	Lookup table breakpoints are specified explicitly. Breakpoints can be closer together for some input ranges and farther apart in others.
EvenSpacing	Lookup table breakpoints are evenly spaced throughout.

Breakpoint Specification	Description
EvenPow2Spacing	Lookup table breakpoints use power-of-two spacing. This breakpoint specification boasts the fastest execution speed because a bit shift can replace the position search.

For more information on how breakpoint specification can affect performance, see “Effects of Spacing on Speed, Error, and Memory Usage”.

Data Types: char

**Display — Whether to display details of each iteration of the optimization**

1 (default) | 0

Whether to display details of each iteration of the optimization, specified as a logical. A value of 1 results in information in the command window at each iteration of the approximation process. A value of 0 does not display information until the approximation is complete.

Data Types: logical

**Interpolation — Method when an input falls between breakpoint values**

Linear (default) | Flat | Nearest | None

When an input falls between breakpoint values, the lookup table interpolates the output value using neighboring breakpoints.

Interpolation Method	Description
Linear	Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input.
Flat	Returns the output value corresponding to the breakpoint value that is immediately less than the input value. If no breakpoint value exists below the input value, it returns the breakpoint value nearest the input value.

Interpolation Method	Description
Nearest	Returns the value corresponding to the breakpoint that is closest to the input. If the input is equidistant from two adjacent breakpoints, the breakpoint with the higher index is chosen.
None	<p data-bbox="788 470 1335 621">Generates a Direct Lookup Table (n-D) block, which performs table lookups without any interpolation or extrapolation.</p> <p data-bbox="788 626 1335 722"><b>Note</b> When generating a Direct Lookup Table block, the maximum number of inputs is two.</p>

Data Types: char

**MaxMemoryUsage — Maximum amount of memory the generated lookup table can use**

80000000 (default) | scalar integer

The maximum amount of memory the generated lookup table can use, in bits, specified as a scalar integer. You can change the units of the option using the MemoryUnits property.

Data Types: double

**MaxTime — Maximum amount of time for the approximation to run (in seconds)**

Inf (default) | scalar

Maximum amount of time for the approximation to run, specified in seconds as a scalar number. The approximation runs until it reaches the time specified, finds an ideal solution, or reaches another stopping criteria.

Data Types: double

**MemoryUnits — Units for maximum memory usage**

'bits' (default) | 'bytes' | 'Kb' | 'Kibit' | 'KB' | 'KiB' | 'Mb' | 'Mibit' | 'MB' | 'MiB' | 'Gb' | 'Gibit' | 'GB' | 'GiB'

Units for MaxMemoryUsage property, specified as 'bits', 'bytes', or one of the other enumerated options.

Data Types: `char`

**OnCurveTableValues — Whether to constrain table values to the quantized output of the function being approximated**

0 (default) | 1

Whether to constrain table values to the quantized output of the function being approximated. By setting this property to 0 and allowing off-curve table values, you may be able to reduce the memory of the lookup table while maintaining the same error tolerances, or maintain the same memory while reducing the error tolerances.

Data Types: `logical`

**RelTol — Relative tolerance of difference between original and approximate**

non-negative scalar

Maximum tolerance of the relative difference between the original output value and the output value of the approximation, specified as a non-negative scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

**SaturateToOutputType — Saturate output of function to approximate to range of output type**

0 (default) | 1

Logical value specifying whether to automatically saturate the range of the output of the function to approximate to the range of the output data type.

Example: `options.SaturateToOutputType = 1;`

Data Types: `logical`

**WordLengths — Word lengths permitted in the lookup table approximate**

[8, 16, 32] (default) | integer scalar | integer vector

Specify the word lengths, in bits, that can be used in the lookup table approximate based on your intended hardware. For example, if you intend to target an embedded processor, you can restrict the data types in your lookup table to native types, 8, 16, and 32. The word lengths must be between 1 and 128.

Example: `options.WordLengths = [8,16,32];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`



## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Algorithms

When you set `BreakpointSpecification` to `'ExplicitValues'`, during the approximation process, the algorithm also attempts to find a solution using `'EvenSpacing'` and `'EvenPow2Spacing'`. Likewise, when you set `BreakpointSpecification` to `'EvenSpacing'`, the algorithm also attempts to find a solution using `'EvenPow2Spacing'`. If you set the property to `'EvenPow2Spacing'`, the algorithm only attempts to find a solution using this spacing.

In cases where the `BreakpointSpecification` property is set to `'EvenSpacing'`, but the `InputUpperBounds` or `InputLowerBounds` property of the `FunctionApproximation.Problem` object is equal to the range of the `InputTypes`, the algorithm does not attempt to find a solution using `'EvenPow2Spacing'`.

## See Also

### Apps

[Lookup Table Optimizer](#)

### Classes

[FunctionApproximation.LUTMemoryUsageCalculator](#) |  
[FunctionApproximation.LUTSolution](#) | [FunctionApproximation.Options](#) |  
[FunctionApproximation.Problem](#)

### Functions

[approximate](#) | [compare](#) | [displayallsolutions](#) | [displayfeasiblesolutions](#) |  
[lutmemoryusage](#) | [solutionfromID](#) | [solve](#) | [totalmemoryusage](#)

## Topics

[“Optimize Lookup Tables for Memory-Efficiency Programmatically”](#)

[“Optimize Lookup Tables for Memory-Efficiency”](#)

**Introduced in R2018a**

# FunctionApproximation.Problem class

**Package:** FunctionApproximation

Object defining the function to approximate, or the lookup table to optimize

## Description

The `FunctionApproximation.Problem` object defines the function to approximate with a lookup table, or the lookup table block to optimize. After defining the problem, use the `solve` method to generate a `FunctionApproximation.LUTSolution` object that contains the approximation.

## Construction

`approximationProblem = FunctionApproximation.Problem()` creates a `FunctionApproximation.Problem` object with default property values. When no function input is provided, the `FunctionToApproximate` property is set to `'sin'`.

`approximationProblem = FunctionApproximation.Problem(function)` creates a `FunctionApproximation.Problem` object to approximate the function, Math Function block, or lookup table specified by `function`.

## Input Arguments

**function** — **Function or block to approximate, or lookup table block to optimize**  
'sin' (default) | math function | function handle | Math Function block | Lookup Table block | Subsystem block

Function or block to approximate, or the lookup table block to optimize, specified as a function handle, a math function, a Simulink block or subsystem, or one of the lookup table blocks (for example, 1-D Lookup Table, n-D Lookup Table).

If you specify one of the lookup table blocks, the `solve` method generates an optimized lookup table.

If you specify a math function, a function handle, or a block, the `solve` method generates a lookup table approximation of the input function.

Function handles must be on the MATLAB search path, or approximation fails.

The MATLAB math functions supported for approximation are:

- `1/.x`
- `10.^x`
- `2.^x`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`
- `atanh`
- `cos`
- `cosh`
- `exp`
- `log`
- `log10`
- `log2`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `x.^2`

---

**Note** Functions and function handles that you approximate must be vectorized, meaning that for each input, there is exactly one output. For more information, see “Vectorization” (MATLAB).

---

---

**Tip** The process of generating a lookup table approximation is faster for a function handle than for a subsystem. If a subsystem can be represented by a function handle, it is faster to approximate the function handle.

---

Data Types: char | function\_handle

## Properties

### **FunctionToApproximate** — Function to approximate, or lookup table block to optimize

'sin' (default) | math function | function handle | Math Function block | Lookup Table block | Subsystem block

Function or block to approximate, or the lookup table block to optimize, specified as a function handle, a math function, a Simulink block or subsystem, or one of the lookup table blocks (for example, 1-D Lookup Table, n-D Lookup Table).

If you specify one of the lookup table blocks, the `solve` method generates an optimized lookup table.

If you specify a math function, a function handle, or a block, the `solve` method generates a lookup table approximation of the input function.

Function handles must be on the MATLAB search path, or approximation fails.

The MATLAB math functions supported for approximation are:

- `1/.x`
- `10.^x`
- `2.^x`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`

- `atanh`
- `cos`
- `cosh`
- `exp`
- `log`
- `log10`
- `log2`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `x.^2`

---

**Note** Functions and function handles that you approximate must be vectorized, meaning that for each input, there is exactly one output. For more information, see “Vectorization” (MATLAB).

---

---

**Tip** The process of generating a lookup table approximation is faster for a function handle than for a subsystem. If a subsystem can be represented by a function handle, it is faster to approximate the function handle.

---

Data Types: `char` | `function_handle`

### **NumberOfInputs — Number of inputs to function approximation**

1 | 2 | 3

Number of inputs to approximated function. This property is inferred from the `FunctionToApproximate` property, therefore it is not a writable property.

If you are generating a Direct Lookup Table, the function to approximate can have no more than two inputs.

Data Types: `double`

**InputTypes — Desired data types of inputs to function approximation**

numerictype object | vector of numerictype objects | Simulink.Numerictype object  
| vector of Simulink.Numerictype objects

Desired data types of the inputs to the approximated function, specified as a numerictype, Simulink.Numerictype, or a vector of numerictype or Simulink.Numerictype objects. The number of InputTypes specified must match the NumberOfInputs.

```
Example: problem.InputTypes = ["numerictype(1,16,13)",  
"numerictype(1,16,10)"];
```

**InputLowerBounds — Lower limit of range of inputs to function to approximate**

scalar | vector

Lower limit of range of inputs to function to approximate, specified as a scalar or vector. If you specify `inf`, the InputLowerBounds used during the approximation is derived from the InputTypes property. The dimensions of InputLowerBounds must match the NumberOfInputs.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | fi

**InputUpperBounds — Upper limit of range of inputs to function to approximate**

scalar | vector

Upper limit of range of inputs to function to approximate, specified as a scalar or vector. If you specify `inf`, the InputUpperBounds used during the approximation is derived from the InputTypes property. The dimensions of InputUpperBounds must match the NumberOfInputs.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64 | fi

**OutputType — Desired data type of the function approximation output**

numerictype | Simulink.Numerictype

Desired data type of the function approximation output, specified as a numerictype or Simulink.Numerictype. For example, to specify that you want the output to be a signed fixed-point data type with 16-bit word length and best-precision fraction length, set the OutputType property to `"numerictype(1,16)"`.

```
Example: problem.OutputType = "numerictype(1,16)";
```

### Options — Additional options and constraints to use in approximation

`FunctionApproximation.Options` object

Additional options and constraints to use in approximation, specified as a `FunctionApproximation.Options` object.

## Methods

`solve` Solve for optimized solution to function approximation problem

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Create Problem Object to Approximate a Function Handle

Create a `FunctionApproximation.Problem` object, specifying a function handle that you want to approximate.

```
problem = FunctionApproximation.Problem(@(x,y) sin(x)+cos(y))
```

```
problem =
```

```
FunctionApproximation.Problem with properties
```

```
FunctionToApproximate: @(x,y)sin(x)+cos(y)
NumberOfInputs: 2
InputTypes: ["numeric('double')" "numeric('double')"]
InputLowerBounds: [-Inf -Inf]
InputUpperBounds: [Inf Inf]
OutputType: "numeric('double')"
Options: [1x1 FunctionApproximation.Options]
```

The `FunctionApproximation.Problem` object, *problem*, uses default property values.

Set the range of the function inputs to be between zero and  $2\pi$ .



```

problem.InputLowerBounds = [0,0];
problem.InputUpperBounds = [2*pi, 2*pi]

problem =

    FunctionApproximation.Problem with properties

        FunctionToApproximate: @(x,y)sin(x)+cos(y)
        NumberOfInputs: 2
        InputTypes: ["numerictype('double')" "numerictype('double')"]
        InputLowerBounds: [0 0]
        InputUpperBounds: [6.2832 6.2832]
        OutputType: "numerictype('double')"
        Options: [1x1 FunctionApproximation.Options]

```

### Create Problem Object to Approximate a Math Function

Create a `FunctionApproximation.Problem` object, specifying a math function to approximate.

```

problem = FunctionApproximation.Problem('log')

problem =

```

```

    FunctionApproximation.Problem with properties

        FunctionToApproximate: @(x)log(x)
        NumberOfInputs: 1
        InputTypes: "numerictype(1,16,10)"
        InputLowerBounds: 0.6250
        InputUpperBounds: 15.6250
        OutputType: "numerictype(1,16,13)"
        Options: [1x1 FunctionApproximation.Options]

```

The math functions have appropriate input range, input data type, and output data type property defaults.

### Create Problem Object to Optimize a Lookup Table Block

Create a `FunctionApproximation.Problem` object to optimize an existing lookup table.

```

load_system('sldemo_fuelsys');
problem = FunctionApproximation.Problem('sldemo_fuelsys/fuel_rate_control/airflow_calc');

```

```
problem =  
  
    FunctionApproximation.Problem with properties  
  
        FunctionToApproximate: 'sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Const  
        NumberOfInputs: 2  
        InputTypes: ["numerictype('single')"    "numerictype('single')"]  
        InputLowerBounds: [50 0.0500]  
        InputUpperBounds: [1000 0.9500]  
        OutputType: "numerictype('single')"  
        Options: [1x1 FunctionApproximation.Options]
```

The software infers the properties of the `Problem` object from the model.

## Algorithms

### Required Specifications

Functions and function handles that you approximate must meet the following criteria.

- The function must be time-invariant.
- The function must operate element-wise, meaning for each input there is one output.
- The function must not contain states.

For more information, see “Vectorization” (MATLAB).

### Infinite Upper and Lower Input Bounds

When a `Problem` object specifies infinite input ranges and the input type is non-floating-point, during the approximation, the software infers upper and lower ranges based on the range of the input data type. The resulting `FunctionApproximation.LUTSolution` object specifies the bounds that the algorithm used during the approximation, not the originally specified infinite bounds.

### Upper and Lower Input Bounds and Input Data Type Range

If the `InputLowerBounds` or `InputUpperBounds` specified for a `Problem` object fall outside the range of the specified `InputTypes`, the algorithm uses the range of the data type specified by `InputTypes` for the approximation.

In cases where the `BreakpointSpecification` property of the `FunctionApproximation.Options` object is set to `'EvenSpacing'`, but the `InputUpperBounds` or `InputLowerBounds` property of the `FunctionApproximation.Problem` object is equal to the range of the `InputTypes`, the algorithm does not attempt to find a solution using `'EvenPow2Spacing'`.

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

`FunctionApproximation.LUTMemoryUsageCalculator` |  
`FunctionApproximation.LUTSolution` | `FunctionApproximation.Options`

### Functions

`approximate` | `compare` | `solve`

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## fxpOptimizationOptions class

Specify options for data type optimization

### Description

The `fxpOptimizationOptions` object enables you to specify options and constraints to use during the data type optimization process.

### Construction

`opt = fxpOptimizationOptions()` creates a `fxpOptimizationOptions` object with default values.

`opt = fxpOptimizationOptions(Name,Value)` creates an `fxpOptimizationOptions` object with property values specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Properties

#### **MaxIterations** — Maximum number of iterations to perform

50 (default) | scalar integer

Maximum number of iterations to perform, specified as a scalar integer. The optimization process iterates through different solutions until it finds an ideal solution, reaches the maximum number of iterations, or reaches another stopping criteria.

Example: `opt.MaxIterations = 75;`

Data Types: double

#### **MaxTime** — Maximum amount of time for the optimization to run (in seconds)

600 (default) | scalar

Maximum amount of time for the optimization to run, specified in seconds as a scalar number. The optimization runs until it reaches the time specified, an ideal solution, or another stopping criteria.

Example: `opt.MaxTime = 1000;`

Data Types: `double`

### **Patience — Maximum number of iterations where no new best solution is found**

10 (default) | scalar integer

Maximum number of iterations where no new best solution is found, specified as a scalar integer. The optimization continues as long as the algorithm continues to find new best solutions.

Example: `opt.Patience = 15;`

Data Types: `double`

### **Verbosity — Level of information displayed at the command line during the optimization**

'Moderate' (default) | 'High' | 'Silent'

The level of information displayed at the command line during the optimization process, specified as either 'High', 'Moderate', or 'Silent'.

- 'Silent' - Nothing is displayed at the command line until the optimization process is finished
- 'Moderate' - Information is displayed at each major step of the optimization process, including when the process is in the preprocessing, modeling, and optimization phases.
- 'High' - Information is displayed at the command line at each iteration of the optimization process, including whether a new best solution was found, and the cost of the solution.

Example: `opt.Verbosity = 'High';`

Data Types: `char` | `string`

### **AllowableWordLengths — Word lengths that can be used in your optimized system under design**

[1:128] (default) | scalar integer | vector of integers

Specify the word lengths that can be used in your optimized system under design. Use this property to target the neighborhood search of the optimization process. The final result of the optimization uses word lengths in the intersection of the `AllowableWordLengths` and word lengths compatible with hardware constraints specified in the **Hardware Implementation** pane of your model.

Example: `opt.AllowableWordLengths = [8:11,16,32];`

Data Types: `double`

### **UseParallel** — Whether to run iterations in parallel

`false` (default) | `true`

Whether to run iterations of the optimization in parallel, specified as a logical. Running the iterations in parallel requires a Parallel Computing Toolbox™ license. If you do not have a Parallel Computing Toolbox license, or if you specify `false`, the iterations run in serial.

Data Types: `logical`

### **AdvancedOptions** — Additional options for optimization

`struct`

Additional optimization options. `AdvancedOptions` is a struct containing four additional properties that can affect the optimization.

Property	Description
<code>PerformNeighborhoodSearch</code>	<ul style="list-style-type: none"><li>• 1 (default) - Perform a neighborhood search for the optimized solution.</li><li>• 0 - Do not perform a neighborhood search. Selecting this option can increase the speed of the optimization process, but also increases the chances of finding a less ideal solution.</li></ul>

Property	Description
EnforceLooseCoupling	<p>Some blocks have a parameter that forces inputs to share a data type, or forces the output to share the same data type as the input.</p> <ul style="list-style-type: none"> <li>• 1 (default) - Allow the optimizer to relax this restriction on all blocks in the system under design. Relaxing this restriction enables the optimizer to provide better fitting data types.</li> <li>• 0 - Do not allow the optimizer to relax this restriction on blocks in the system under design.</li> </ul>
UseDerivedRangeAnalysis	<ul style="list-style-type: none"> <li>• 0 (default) - The optimizer does not consider ranges derived from design ranges in the model when assessing a solution.</li> <li>• 1 - The optimizer considers both observed simulation ranges and ranges derived from design ranges in the model when assessing a solution.</li> </ul> <p>Depending on the model configuration, derived range analysis may take longer than simulation of the model.</p>
SimulationScenarios	<p>Define additional simulation scenarios to consider during optimization using a <code>Simulink.SimulationInput</code> object. For an example, see “Optimize Data Types Using Multiple Simulation Scenarios”.</p>

## Methods

addTolerance	Specify numeric tolerance for optimized system
showTolerances	Show tolerances specified for a system

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create an `fxpOptimizationOptions` object with default values

Create an `fxpOptimizationObject` with default property values.

```
options = fxpOptimizationOptions();
```

Edit the properties after creation using dot syntax.

```
options.Patience = 15;  
options.AllowableWordLengths = [8,16,32]
```

```
options =  
    fxpOptimizationOptions with properties:
```

```
        MaxIterations: 50  
           MaxTime: 600  
        Patience: 15  
       Verbosity: High  
AllowableWordLengths: [8 16 32]  
        UseParallel: 0
```

```
Advanced Options  
    AdvancedOptions: [1x1 struct]
```

## See Also

### Classes

[OptimizationResult](#) | [OptimizationSolution](#)

### Functions

[addTolerance](#) | [explore](#) | [fxpopt](#) | [showTolerances](#)



## **Topics**

“Optimize Fixed-Point Data Types for a System”

**Introduced in R2018a**

## OptimizationResult class

Result after optimizing fixed-point system

### Description

An `OptimizationResult` object contains the results after optimizing a fixed-point system. If the optimization process succeeds in finding a new fixed-point implementation, you can use this object to explore the different implementations that met the specified tolerances found during the process. Use the `explore` method to open the Simulation Data Inspector and view the behavior of the optimized system.

### Construction

`result = fxpopt(model, sud, options)` optimizes the data types in the system specified by `sud` in the model, `model`, with additional options specified in the `fxpOptimizationOptions` object, `options`.

### Input Arguments

**model** — Model containing system under design

character vector

Name of the model containing the system that you want to optimize.

Data Types: char

**sud** — System whose data types you want to optimize

character vector

System whose data types you want to optimize, specified as a character vector containing the path to the system.

Data Types: char

**options** — Additional optimization options

`fxpOptimizationOptions` object

`fxpOptimizationOptions` object specifying additional options to use during the data type optimization process.

## Properties

### **FinalOutcome** — Message specifying whether a new optimal solution was found

character vector

Message specifying whether the optimization process found a new optimal solution, returned as a character vector.

Data Types: `char`

### **OptimizationOptions** — `fxpOptimizationOptions` object associated with the result

`fxpOptimizationOptions` object

The `fxpOptimizationOptions` object used as an input to the `fxpopt` function used to generate the `OptimizationResult`.

### **Solutions** — Vector of `OptimizationSolution` objects

`OptimizationSolution` object | vector of `OptimizationSolution` objects

A vector of `OptimizationSolution` objects found during the optimization process. If the optimization finds a feasible solution, the vector is sorted by cost, with the lowest cost (most optimal) solution as the first element of the vector. If the optimization does not find a feasible solution, the vector is sorted by maximum difference from the original design.

## Methods

`explore` Explore fixed-point implementations found during optimization process

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

### Classes

OptimizationSolution | fxpOptimizationOptions

### Functions

addTolerance | explore | fxpopt | showTolerances

### Topics

“Optimize Fixed-Point Data Types for a System”

**Introduced in R2018a**

# OptimizationSolution class

Optimized fixed-point implementation of system

## Description

An `OptimizationSolution` object is a fixed-point implementation of a system whose data types were optimized using the `fxpopt` function.

## Construction

`solution = explore(result)` opens the Simulation Data Inspector. If the optimization found a solution, it returns the `OptimizationSolution` object with the lowest cost out of the vector of `OptimizationSolution` objects contained in the `OptimizationResult` object, `result`. If the optimization did not find a solution, it returns the `OptimizationSolution` object with the smallest `MaxDifference`.

You can also access a `OptimizationSolution` object by indexing the `Solutions` property of an `OptimizationResult` object. For example, to access the solution with the second lowest cost contained in the `OptimizationResult` object, `result`, enter

```
solution = result.Solutions(2)
```

## Input Arguments

**result** — `OptimizationResult` containing the solution

`OptimizationResult` object

The `Solutions` property of the `OptimizationResult` object is a vector of `OptimizationSolution` objects found during the optimization process. If the optimization found a feasible solution, the vector is sorted by cost, with the lowest cost (most optimal) solution as the first element of the vector. If the optimization did not find a feasible solution, the vector is sorted by `MaxDifference`, with the solution with the smallest `MaxDifference` as the first element.

## Properties

### **Cost — Sum of word lengths used in the system under design**

scalar integer

Sum of all word lengths used in the solution in the system under design. The most optimal solution is the solution with the smallest cost.

Data Types: double

### **Pass — Whether the solution meets specified criteria**

1 | 0

Whether the solution meets the criteria specified by the associated `fxpOptimizationOptions` object, specified as a logical.

Data Types: logical

### **MaxDifference — Maximum absolute difference between baseline solution run**

scalar

The maximum absolute difference between the baseline the solution.

Data Types: double

### **RunID — Run identifier**

scalar integer

Unique numerical identification for the run used by the Simulation Data Inspector. For more information, see “Inspect and Compare Data Programmatically” (Simulink).

Data Types: double

### **RunName — Name of the run**

character vector

Name of the run in Simulation Data Inspector.

Data Types: char

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

### Classes

OptimizationResult | fxpOptimizationOptions

### Functions

addTolerance | explore | fxpopt | showTolerances

### Topics

“Optimize Fixed-Point Data Types for a System”

**Introduced in R2018a**





# Methods — Alphabetical List

---

## isHeterogeneous

**Class:** coder.CellType

**Package:** coder

Determine whether cell array type represents a heterogeneous cell array

### Syntax

```
tf = isHeterogeneous(t)
```

### Description

`tf = isHeterogeneous(t)` returns `true` if the `coder.CellType` object `t` is heterogeneous. Otherwise, it returns `false`.

### Examples

#### Determine Whether Cell Array Type Is Heterogeneous

Create a `coder.CellType` object for a cell array whose elements have different classes.

```
t = coder.typeof({'a', 1})
```

```
t =
```

```
coder.CellType  
  1x2 heterogeneous cell  
    f0: 1x1 char  
    f1: 1x1 double
```

Determine whether the `coder.CellType` object represents a heterogeneous cell array.

```
isHeterogeneous(t)
```

```
ans =
```

```
    1
```

## Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

## See Also

`coder.newtype` | `coder.typeof`

## Topics

“Code Generation for Cell Arrays”

“Specify Cell Array Inputs at the Command Line”

**Introduced in R2015b**

## isHomogeneous

**Class:** coder.CellType

**Package:** coder

Determine whether cell array type represents a homogeneous cell array

### Syntax

```
tf = isHomogeneous(t)
```

### Description

`tf = isHomogeneous(t)` returns `true` if the `coder.CellType` object `t` represents a homogeneous cell array. Otherwise, it returns `false`.

### Examples

#### Determine Whether Cell Array Type Is Homogeneous.

Create a `coder.CellType` object for a cell array whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t =
```

```
coder.CellType  
  1x3 homogeneous cell  
  base: 1x1 double
```

Determine whether the `coder.CellType` object represents a homogeneous cell array.

```
isHomogeneous(t)
```

```
ans =
     1
```

### Test for a Homogeneous Cell Array Type Before Executing Code

Write a function `make_varsize`. If the input type `t` is homogeneous, the function returns a variable-size copy of `t`.

```
function c = make_varsize(t, n)
assert(isHomogeneous(t));
c = coder.typeof(t, [n n], [1 1]);
end
```

Create a heterogeneous type `tc`.

```
tc = coder.typeof({'a', 1});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

The assertion fails because `tc` is heterogeneous.

Create a homogeneous type `tc`.

```
tc = coder.typeof({1 2 3});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

```
tc1 =
coder.CellType
   :5x:5 homogeneous cell
   base: 1x1 double
```

## Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size,

`coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

## See Also

`coder.newtype` | `coder.typeof`

## Topics

“Code Generation for Cell Arrays”

“Specify Cell Array Inputs at the Command Line”

**Introduced in R2015b**

# makeHeterogeneous

**Class:** `coder.CellType`

**Package:** `coder`

Make a heterogeneous copy of a cell array type

## Syntax

```
newt = makeHeterogeneous(t)
```

```
t = makeHeterogeneous(t)
```

## Description

`newt = makeHeterogeneous(t)` creates a `coder.CellType` object for a heterogeneous cell array from the `coder.CellType` object `t`. `t` cannot represent a variable-size cell array.

The classification as heterogeneous is permanent. You cannot later create a homogeneous `coder.CellType` object from `newt`.

`t = makeHeterogeneous(t)` creates a heterogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

## Examples

### Replace a Homogeneous Cell Array Type with a Heterogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t =
```

```
coder.CellType
```

```
1x3 homogeneous cell
  base: 1x1 double
```

The cell array type is homogeneous.

Replace `t` with a cell array type for a heterogeneous cell array.

```
t = makeHeterogeneous(t)
```

```
t =
```

```
coder.CellType
  1x3 locked heterogeneous cell
    f1: 1x1 double
    f2: 1x1 double
    f3: 1x1 doublee
```

The cell array type is heterogeneous. The elements have the size and class of the original homogeneous cell array type.

## Tips

- In the display of a `coder.CellType` object, the terms `locked` `heterogeneous` or `locked` `homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a `1x2` heterogeneous type. The first element is `double` and the second element is `1x2 double`. The type can also be a `1x3` homogeneous type in which the elements have class `double` and size `1x2`. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

## See Also

`coder.newtype` | `coder.typeof`



## **Topics**

“Code Generation for Cell Arrays”

“Specify Cell Array Inputs at the Command Line”

**Introduced in R2015b**

## makeHomogeneous

**Class:** `coder.CellType`

**Package:** `coder`

Create a homogeneous copy of a cell array type

### Syntax

```
newt = makeHomogeneous(t)  
t = makeHomogeneous(t)
```

### Description

`newt = makeHomogeneous(t)` creates a `coder.CellType` object for a homogeneous cell array `newt` from the `coder.CellType` object `t`.

To create `newt`, the `makeHomogeneous` method must determine a size and class that represent all elements of `t`:

- If the elements of `t` have the same class, but different sizes, the elements of `newt` are variable size with upper bounds that accommodate the elements of `t`.
- If the elements of `t` have different classes, for example, `char` and `double`, the `makeHomogeneous` method cannot create a `coder.CellType` object for a homogeneous cell array.

The classification as homogeneous is permanent. You cannot later create a heterogeneous `coder.CellType` object from `newt`.

`t = makeHomogeneous(t)` creates a homogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

### Examples

## Replace a Heterogeneous Cell Array Type with a Homogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class, but different sizes.

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

```
t =
```

```
coder.CellType
  1x2 heterogeneous cell
    f0: 1x1 double
    f1: 1x2 double
```

The cell array type is heterogeneous.

Replace `t` with a cell array type for a homogeneous cell array.

```
t = makeHomogeneous(t)
```

```
t =
```

```
coder.CellType
  1x2 locked homogeneous cell
    base: 1x:2 double
```

The new cell array type is homogeneous.

## Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a `1x2` heterogeneous type. The first element is double and the second element is `1x2` double. The type can also be a `1x3` homogeneous type in which the elements have class double and size `1x:2`. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous.

If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

### **See Also**

`coder.newtype` | `coder.typeof`

### **Topics**

“Code Generation for Cell Arrays”

“Specify Cell Array Inputs at the Command Line”

**Introduced in R2015b**

# addApproximation

Replace floating-point function with lookup table during fixed-point conversion

## Syntax

```
addApproximation(approximationObject)
```

## Description

`addApproximation(approximationObject)` specifies a lookup table replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion, the conversion process generates a lookup table approximation for the function specified in the `approximationObject`.

## Input Arguments

### **approximationObject** — Function replacement configuration object

`coder.mathfcngenerator.LookupTable` configuration object

Function replacement configuration object that specifies how to create an approximation for a MATLAB function. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `fiaccel` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point MATLAB code.

## Examples

### **Replace log function with an optimized lookup table replacement**

Create a function replacement configuration object that specifies to replace the log function with an optimized lookup table.

```
logAppx = coder.approximation('Function','log','OptimizeLUTSize',...  
    true,'InputRange',[0.1,1000],'InterpolationDegree',1,...  
    'ErrorThreshold',1e-3,...  
    'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Create a fixed-point configuration object and associate the function replacement configuration object with it.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.addApproximation(logAppx);
```

You can now generate fixed-point code using the `fiaccel` function.

## See Also

`coder.FixptConfig` | `fiaccel`

## Topics

“Replace the exp Function with a Lookup Table”

“Replace a Custom Function with a Lookup Table”

“Replacing Functions Using Lookup Table Approximations”

# addDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Add design range specification to parameter

## Syntax

```
addDesignRangeSpecification(fcnName,paramName,designMin, designMax)
```

## Description

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)` specifies the minimum and maximum values allowed for the parameter, `paramName`, in function, `fcnName`. The fixed-point conversion process uses this design range information to derive ranges for downstream variables in the code.

## Input Arguments

**fcnName — Function name**

string

Function name, specified as a string.

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Data Types: char

**designMin — Minimum value allowed for this parameter**

scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: double

**designMax — Maximum value allowed for this parameter**

scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: double

## Examples

### Add a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;

% Derive ranges and generate fixed-point code
fiaccel -float2fixed cfg dti
```

### See Also

```
coder.FixptConfig | coder.FixptConfig.clearDesignRangeSpecifications |
coder.FixptConfig.getDesignRangeSpecification |
coder.FixptConfig.hasDesignRangeSpecification |
coder.FixptConfig.removeDesignRangeSpecification | fiaccel
```



# addFunctionReplacement

**Class:** `coder.FixptConfig`

**Package:** `coder`

Replace floating-point function with fixed-point function during fixed-point conversion

## Syntax

```
addFunctionReplacement(floatFn, fixedFn)
```

## Description

`addFunctionReplacement(floatFn, fixedFn)` specifies a function replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion, the conversion process replaces the specified floating-point function with the specified fixed-point function. The fixed-point function must be in the same folder as the floating-point function or on the MATLAB path.

## Input Arguments

**floatFn** — Name of floating-point function

' ' (default) | string

Name of floating-point function, specified as a string.

**fixedFn** — Name of fixed-point function

' ' (default) | string

Name of fixed-point function, specified as a string.

## Examples

## Specify Function Replacement in Fixed-Point Conversion Configuration Object

Suppose that:

- The function `myfunc` calls a local function `myadd`.
- The test function `mytest` calls `myfunc`.
- You want to replace calls to `myadd` with the fixed-point function `fi_myadd`.

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mytest`.

```
fixptcfg.TestBenchName = 'mytest';
```

Specify that the floating-point function, `myadd`, should be replaced with the fixed-point function, `fi_myadd`.

```
fixptcfg.addFunctionReplacement('myadd', 'fi_myadd');
```

Convert the floating-point MATLAB function, `myfunc`, to fixed-point.

```
fiaccel -float2fixed fixptcfg myfunc
```

`fiaccel` replaces `myadd` with `fi_myadd` during floating-point to fixed-point conversion.

## See Also

`coder.FixptConfig` | `fiaccel`

# addFunctionReplacement

**Class:** `coder.SingleConfig`

**Package:** `coder`

Replace double-precision function with single-precision function during single-precision conversion

## Syntax

```
addFunctionReplacement(doubleFn,singleFn)
```

## Description

`addFunctionReplacement(doubleFn,singleFn)` specifies a function replacement in a `coder.SingleConfig` object. During double-precision to single-precision conversion, the conversion process replaces the specified double-precision function with the specified single-precision function. The single-precision function must be in the same folder as the double-precision function or on the MATLAB path. It is a best practice to provide unique names to local functions that a replacement function calls. If a replacement function calls a local function, do not give that local function the same name as a local function in a different replacement function file.

## Input Arguments

**doubleFn — Name of double-precision function**

' ' (default) | string

Name of double-precision function, specified as a string.

**singleFn — Name of single-precision function**

' ' (default) | string

Name of single-precision function, specified as a string.

## Examples

### Specify Function Replacement in Single-Precision Conversion Configuration Object

Suppose that:

- The function `myfunc` calls a local function `myadd`.
- The test function `mytest` calls `myfunc`.
- You want to replace calls to `myadd` with the single-precision function `single_myadd`.

Create a `coder.SingleConfig` object, `scfg`, with default settings.

```
scfg = coder.config('single');
```

Set the test file name. In this example, the test file function name is `mytest`.

```
scfg.TestBenchName = 'mytest';
```

Specify that you want to replace the double-precision function, `myadd`, with the single-precision function, `single_myadd`.

```
scfg.addFunctionReplacement('myadd', 'single_myadd');
```

Convert the double-precision MATLAB function, `myfunc` to a single-precision MATLAB function.

```
convertToSingle -config scfg myfunc
```

The double-precision to single-precision conversion replaces instances of `myadd` with `single_myadd`.

## See Also

**Introduced in R2015b**

# clearDesignRangeSpecifications

**Class:** coder.FixptConfig

**Package:** coder

Clear all design range specifications

## Syntax

```
clearDesignRangeSpecifications()
```

## Description

`clearDesignRangeSpecifications()` clears all design range specifications.

## Examples

### Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

## See Also

[coder.FixptConfig](#) | [coder.FixptConfig.addDesignRangeSpecification](#) |  
[coder.FixptConfig.getDesignRangeSpecification](#) |

```
coder.FixptConfig.hasDesignRangeSpecification |  
coder.FixptConfig.removeDesignRangeSpecification | fiaccel
```

# getDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Get design range specifications for parameter

## Syntax

```
[designMin, designMax] = getDesignRangeSpecification(fcnName,  
paramName)
```

## Description

[designMin, designMax] = getDesignRangeSpecification(fcnName, paramName) gets the minimum and maximum values specified for the parameter, paramName, in function, fcnName.

## Input Arguments

**fcnName — Function name**

string

Function name, specified as a string.

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Data Types: char

## Output Arguments

### **designMin** — Minimum value allowed for this parameter

scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: double

### **designMax** — Maximum value allowed for this parameter

scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: double

## Examples

### Get Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')

designMin =

    -1

designMax =

     1
```

## See Also

[coder.FixptConfig](#) | [coder.FixptConfig.addDesignRangeSpecification](#) | [coder.FixptConfig.clearDesignRangeSpecifications](#) |



```
coder.FixptConfig.hasDesignRangeSpecification |  
coder.FixptConfig.removeDesignRangeSpecification | fiaccel
```

## hasDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Determine whether parameter has design range

### Syntax

```
hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)
```

### Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)`  
returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

### Input Arguments

**fcnName — Name of function**

string

Function name, specified as a string.

Example: 'dti'

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Example: 'dti'

Data Types: char

## Output Arguments

### hasDesignRange — Parameter has design range

true | false

Parameter has design range, returned as a boolean.

Data Types: logical

## Examples

### Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')

hasDesignRanges =

     1
```

## See Also

[coder.FixptConfig](#) | [coder.FixptConfig.addDesignRangeSpecification](#) |  
[coder.FixptConfig.clearDesignRangeSpecifications](#) |  
[coder.FixptConfig.getDesignRangeSpecification](#) |  
[coder.FixptConfig.removeDesignRangeSpecification](#) | [fiaccl](#)

## removeDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Remove design range specification from parameter

### Syntax

```
removeDesignRangeSpecification(fcnName,paramName)
```

### Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

### Input Arguments

**fcnName — Name of function**

string

Function name, specified as a string.

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Data Types: char

## Examples

### Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

### See Also

[coder.FixptConfig](#) | [coder.FixptConfig.addDesignRangeSpecification](#) |  
[coder.FixptConfig.clearDesignRangeSpecifications](#) |  
[coder.FixptConfig.getDesignRangeSpecification](#) |  
[coder.FixptConfig.hasDesignRangeSpecification](#) | [fiaccl](#)

## applyDataTypes

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Apply proposed data types to model

### Syntax

```
converter.applyDataTypes(RunName)
```

### Description

`converter.applyDataTypes(RunName)` applies the proposed data types for the specified run to the converter's system.

### Input Arguments

**RunName — Name of run**

character vector


Name of run to apply data types to, specified as a character vector.

Example: `converter.applyDataTypes('Run1')`

Data Types: char

### Alternatives

`DataTypeWorkflow.Converter.applyDataTypes` provides functionality similar to the

Fixed-Point Tool button **Apply accepted fraction lengths** . For more information, see `fxptdlg`.

## See Also

`DataTypeWorkflow.Converter.proposeDataTypes` |  
`DataTypeWorkflow.ProposalSettings`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

## applySettingsFromRun

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Apply system settings used in previous run to model

### Syntax

```
converter.applySettingsFromRun(RunName)
```

### Description

`converter.applySettingsFromRun(RunName)` applies the data type override and instrumentation settings used in a previous run to the model.

### Input Arguments

**RunName — Name of run**

character vector

Name of run that has the settings to apply, specified as a character vector.

Example: `converter.applySettingsFromRun('Run1')`

Data Types: char

### See Also

`DataTypeWorkflow.Converter.applySettingsFromShortcut`

### Topics

“Convert a Model to Fixed Point Using the Command Line”



# applySettingsFromShortcut

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Apply settings from shortcut to model

## Syntax

```
converter.applySettingsFromShortcut(shortcutName)
```

## Description

`converter.applySettingsFromShortcut(shortcutName)` applies the settings from the specified configuration to the model.

## Input Arguments

**shortcutName — Name of shortcut**

character vector

Name of shortcut that specifies which settings to use, specified as a character vector.

Example: `converter.applySettingsFromShortcut('Range collection using double override')`

Data Types: char

## Tips

- You can create additional configurations using the Fixed-Point Tool. For more information, see “Use Shortcuts to Manage Runs”.

## Alternatives

`DataTypeWorkflow.Converter.applySettingsFromShortcut` provides functionality similar to the Fixed-Point Tool button group **Configure model settings**



. For more information, see `fxptdlg`.

## See Also

`DataTypeWorkflow.Converter.applySettingsFromRun` | `fxptdlg`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

# deriveMinMax

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Derive range information for model

## Syntax

```
converter.deriveMinMax()
```

## Description


`converter.deriveMinMax()` derives the minimum and maximum values for each block based on design minimum and maximum values.

## Tips

- If any issues come up during the derivation, they can be queried using the `DataTypeWorkflow.Converter.proposalIssues` method.

## Alternatives

The `DataTypeWorkflow.Converter.deriveMinMax` method is equivalent to the

**Derive min/max values for selected system** button () in the Fixed-Point Tool. See `fxptdlg` for more information.

## See Also

`DataTypeWorkflow.Converter.simulateSystem` | `fxptdlg`

**Topics**

“Convert a Model to Fixed Point Using the Command Line”

# proposeDataTypes

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Propose data types for system

## Syntax

```
converter.proposeDataTypes(RunName, propSettings)
```

## Description

`converter.proposeDataTypes(RunName, propSettings)` proposes data types for the system based on the range results stored in `RunName` and the settings specified in `propSettings`.

## Input Arguments

### **RunName — Name of run**

character vector

Name of run to propose data types for, specified as a character vector.


Data Types: char

### **propSettings — Proposed data type settings**

`DataTypeWorkflow.ProposalSettings` object

Proposed data type settings specified as a `DataTypeWorkflow.ProposalSettings` object. Use this object to specify proposal settings such as the default data type for all floating point signals.

## Alternatives

`DataTypeWorkflow.Converter.proposeDataTypes` provides functionality similar to the Fixed-Point Tool button **Propose Data Types** . For more information, see `fxptdlg`.

## See Also

`DataTypeWorkflow.Converter.applyDataTypes` |  
`DataTypeWorkflow.ProposalSettings`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

---

## results

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Find results for selected system in converter object

## Syntax

```
results = converter.results(RunName)
results = converter.results(RunName, filterFunc)
```

## Description

`results = converter.results(RunName)` returns all results in the specified run.

`results = converter.results(RunName, filterFunc)` returns the results in the specified run which match the criteria specified by `filterFunc`.

## Input Arguments

### **RunName — Name of run**

character vector

Name of the run to query, specified as a character vector.

Data Types: `char`

### **filterFunc — Function to use to filter results**

function handle

Function to use to filter results, specified as a function handle with a `DataTypeWorkflow.Result` object as its input.

Data Types: `function_handle`

## Output Arguments

### **results** — Filtered results

array of `Result` objects

Array of `DataTypeWorkflow.Result` objects from `RunName` filtered by `filterFunc`

## Alternatives

The `DataTypeWorkflow.Converter.results` method offers a command-line approach to using the Fixed-Point Tool. See `fxptdlg` for more information.

## See Also

`DataTypeWorkflow.Converter.proposalIssues` |  
`DataTypeWorkflow.Converter.saturationOverflows` |  
`DataTypeWorkflow.Converter.wrapOverflows`

## Topics

“Convert a Model to Fixed Point Using the Command Line”



# proposalIssues

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Get results which have comments associated with them

## Syntax

```
results = converter.proposalIssues(RunName)
```

## Description

`results = converter.proposalIssues(RunName)` returns all results in `RunName` that have associated comments. The `comments` field of the returned results can provide information related to any issues found.

## Input Arguments

**RunName — Name of run**

character vector

Name of run to look for comments in, specified as a character vector.

Data Types: char

## Output Arguments

**results — Results that have associated comments**

DataTypeWorkflow.Result object

A `DataTypeWorkflow.Result` object containing all signals in `RunName` with associated comments.

## Alternatives

The `DataTypeWorkflow.Converter.proposalIssues` method offers a command-line approach to using the Fixed-Point Tool. See `fxptdlg` for more information.

## See Also

`DataTypeWorkflow.Converter.results` |  
`DataTypeWorkflow.Converter.saturationOverflows` |  
`DataTypeWorkflow.Converter.wrapOverflows`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

# saturationOverflows

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Get results where saturation occurred

## Syntax

```
results = converter.saturationOverflows(RunName)
```

## Description

`results = converter.saturationOverflows(RunName)` all results in RunName that saturated during simulation.

## Input Arguments

**RunName — Name of run**

character vector

Name of run to look for saturations in, specified as a character vector.

Data Types: char

## Output Arguments

**results — Results that saturated**

DataTypeWorkflow.Result object

DataTypeWorkflow.Result object containing all of the signals that saturated during the specified run.

## **See Also**

`DataTypeWorkflow.Converter.proposalIssues` |  
`DataTypeWorkflow.Converter.results` |  
`DataTypeWorkflow.Converter.wrapOverflows`

## **Topics**

“Convert a Model to Fixed Point Using the Command Line”

# simulateSystem

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Simulate converter's system

## Syntax

```
simOut = converter.simulateSystem()  
simOut = converter.simulateSystem(Name,Value)  
simOut = converter.simulateSystem(simIn)  
simOut = converter.simulateSystem(ParameterStruct)  
simOut = converter.simulateSystem(ConfigSet)
```

## Description

`simOut = converter.simulateSystem()` simulates the converter's selected system.

`simOut = converter.simulateSystem(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. This method accepts the same `Name,Value` pairs as the `sim` function.

`simOut = converter.simulateSystem(simIn)` simulates the converter's selected system using the inputs specified in the `Simulink.SimulationInput` object `simIn`.

`simOut = converter.simulateSystem(ParameterStruct)` simulates the converter's selected system using the parameter values specified in the structure, `ParameterStruct`.

`simOut = converter.simulateSystem(ConfigSet)` simulates the converter's selected system using the configuration settings specified in the model configuration set, `ConfigSet`.

---

### Note

- The `SimulationMode` property must be set to `normal`. The Fixed-Point Designer software does collect simulation ranges in Rapid accelerator or Hot restart modes.
- The `SrcWorkspace` parameter must be set to either `base` or `current`.

## Input Arguments

### **simIn** — Simulation input for the system

`Simulink.SimulationInput` object | array of `Simulink.SimulationInput` objects

Simulation input for the system, specified as a `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects.

When you use a `SimulationInput` object, you can also specify the following Name, Value pair arguments.

Parameter	Values
<code>ShowSimulationManager</code>	<ul style="list-style-type: none"> <li>• 'on' - Opens the <b>Simulation Manager</b>.</li> <li>• 'off' (default) - Does not open the Simulation Manager.</li> </ul>
<code>ShowProgress</code>	<ul style="list-style-type: none"> <li>• 'on' - View the progress of the simulations in the command window.</li> <li>• 'off' (default) - The progress of the simulation does not display in the command window.</li> </ul>

### **ParameterStruct** — Structure of parameter settings

structure

Structure with fields that are the names of the configuration parameters for the simulation. The corresponding values are the parameter values.

Data Types: `struct`

### **ConfigSet** — Configuration set

`Simulink.ConfigSet`

Configuration set, specified as a `Simulink.ConfigSet`, containing the values of the model parameters.

## Output Arguments

### **simOut — Simulation output**

`Simulink.SimulationOutput` object

`Simulink.SimulationOutput` object containing the simulation outputs: logged time, states, and signals.

## Tips

- To correspond your simulation to a specific run name, before simulation, change the `CurrentRunName` property of the `DataTypeWorkflow.Converter` object.
- `DataTypeWorkflow.Converter.simulateSystem` provides functionality similar to the `sim` command, except that `simulateSystem` preserves the model-wide data type override and instrumentation settings of each run.

## See Also

`sim`

## Topics

“Convert a Model to Fixed Point Using the Command Line”

## verify

**Class:** `DataTypeWorkflow.Converter`

**Package:** `DataTypeWorkflow`

Compare behavior of baseline and autoscaled systems

## Syntax

```
output_args = verify(converter, baselineRun, verificationRunName)
```

## Description

`output_args = verify(converter, baselineRun, verificationRunName)` simulates the system under design specified by the `DataTypeWorkflow.Converter` object, `converter`, and stores the run information in a new run, `verificationRun`. It returns a `DataTypeWorkflow.VerificationResult` object which compares the baseline and verification runs.

## Input Arguments

**converter** — `DataTypeWorkflow.Converter` object to verify

`DataTypeWorkflow.Converter` object

`DataTypeWorkflow.Converter` object to verify. The `DataTypeWorkflow.Converter` object contains instrumentation data from the `baselineRun`, as well as the tolerances specified on the associated `DataTypeWorkflow.ProposalSettings` object. The software determines if the behavior of the verification run is acceptable using the tolerances specified on the `ProposalSettings` object.

**baselineRun** — Run to compare against

character vector

Baseline run to compare against, specified as a character vector

Data Types: `char` | `string`



**verificationRunName** — Name of the verification run to create

character vector

Name of the run to create during the embedded simulation, specified as a character vector.

Data Types: char | string

## Output Arguments

**verificationResult** — Object comparing the behavior of the baseline run and the verification run

DataTypeWorkflow.VerificationResult object

Object comparing the behavior of the baseline run and the verification run, returned as a DataTypeWorkflow.VerificationResult object.

## See Also

DataTypeWorkflow.Converter | DataTypeWorkflow.ProposalSettings |  
DataTypeWorkflow.VerificationResult

## Topics

“Convert a Model to Fixed Point Using the Command Line”

**Introduced in R2019a**

## wrapOverflows

**Class:** DataTypeWorkflow.Converter

**Package:** DataTypeWorkflow

Get results where wrapping occurred

### Syntax

```
results = converter.wrapOverflows(RunName)
```

### Description

`results = converter.wrapOverflows(RunName)` returns all results in RunName that wrapped during simulation.

### Input Arguments

**RunName — Name of run**

character vector

Name of run in which to look for wrap overflows, specified as a character vector.

Example: `converter.WrapOverflows('Run3')`

Data Types: char

### Output Arguments

**results — Result object**

DataTypeWorkflow.Result object

DataTypeWorkflow.Result object containing all of the signals that wrapped during the specified run.

## See Also

[DataTypeWorkflow.Converter.proposalIssues](#) |  
[DataTypeWorkflow.Converter.results](#) |  
[DataTypeWorkflow.Converter.saturationOverflows](#)

## Topics

[“Convert a Model to Fixed Point Using the Command Line”](#)

## addTolerance

**Class:** `DataTypeWorkflow.ProposalSettings`

**Package:** `DataTypeWorkflow`

Specify numeric tolerance for converted system

## Syntax

```
addTolerance(proposalSettings,block_path,port_index,tolerance_type,  
tolerance_value)
```

## Description

`addTolerance(proposalSettings,block_path,port_index,tolerance_type, tolerance_value)` specifies a numeric tolerance for the output signal specified by `block_path` and `port_index`, with the tolerance type specified by `tolerance_type` and value specified by `tolerance_value`.

## Input Arguments

**proposalSettings** — **Associated `DataTypeWorkflow.ProposalSettings` object**  
`DataTypeWorkflow.ProposalSettings` object

`DataTypeWorkflow.ProposalSettings` object to add a tolerance specification.

**block\_path** — **Path to block for which to add tolerance**

block path name

Path to the block to add a tolerance to, specified as a character vector.

Data Types: `char` | `string`

**port\_index** — **Index of output port of block**

scalar integer

Index of output port of the block specified by `block_path` for which you want to specify a tolerance, specified as a scalar integer.

Data Types: `double`

### **tolerance\_type** — Type of tolerance to specify

`'AbsTol' | 'RelTol' | 'TimeTol'`

Type of tolerance to add to the port indicated, specified as either absolute tolerance, `AbsTol`, relative tolerance, `RelTol`, or time tolerance, `TimeTol`.

Data Types: `char`

### **tolerance\_value** — Difference between the original output and the output of the new design

scalar `double`

Acceptable level of tolerance for the signal specified by `block_path` and `port_index`.

If `tolerance_type` is set to `'AbsTol'`, then `tolerance_value` represents the absolute value of the maximum acceptable difference between the original output, and the output of the new design.

If `tolerance_type` is set to `'RelTol'`, then `tolerance_value` represents the maximum relative difference, specified as a percentage, between the original output, and the output of the new design. For example, a value of `1e-2` indicates a maximum difference of one percent between the original output, and the output of the new design.

If `tolerance_type` is set to `'TimeTol'`, then `tolerance_value` defines a time interval, in seconds, in which the maximum and minimum values define the upper and lower values to compare against. For more information, see “How the Simulation Data Inspector Compares Data” (Simulink).

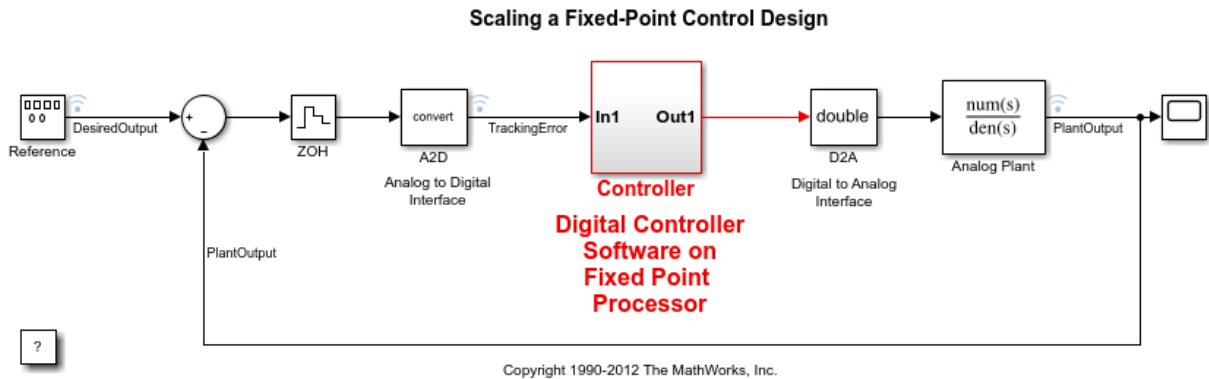
Data Types: `double`

## **Examples**

### **Specify Signal Tolerances**

This example shows how to apply tolerances to signals in a system. In this example, you apply tolerances to the `fxpdemo_feedback` model.

```
model = 'fxpdemo_feedback';
open_system(model);
```



Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'AbsTol', 5e-2)
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'RelTol', 1e-2)
```

Use the `showTolerances` method to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

Path	Port_Index	Tolerance_Type	Tolerance
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'AbsTol'}	0.05
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'RelTol'}	0.01

Clear the tolerances associated with the `ProposalSettings` object.

`clearTolerances(propSettings)`

Using the `showTolerances` method, verify that there are no longer any tolerances associated with the `ProposalSettings` object.

`showTolerances(propSettings)`

## See Also

`clearTolerances` | `showTolerances`

## Topics

“The Command-Line Interface for the Fixed-Point Tool”

**Introduced in R2019a**

## clearTolerances

**Class:** DataTypeWorkflow.ProposalSettings

**Package:** DataTypeWorkflow

Clear all tolerances specified on a system

### Syntax

```
clearTolerances(proposalSettings)
```

### Description

`clearTolerances(proposalSettings)` clears the absolute, relative, and time tolerances specified for a system using the `addTolerance` method of the `DataTypeWorkflow.ProposalSettings` class.

### Input Arguments

**proposalSettings** — Proposal settings object specifying tolerances

`DataTypeWorkflow.ProposalSettings` object

`DataTypeWorkflow.ProposalSettings` object specifying tolerances and settings to use during the data type proposal process.

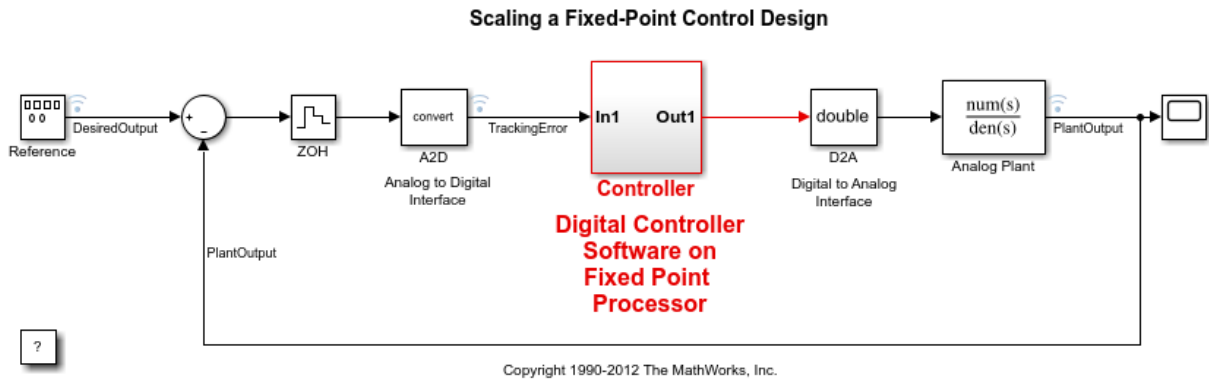
### Examples

#### Specify Signal Tolerances

This example shows how to apply tolerances to signals in a system. In this example, you apply tolerances to the `fxpdemo_feedback` model.

```
model = 'fxpdemo_feedback';  
open_system(model);
```





Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'AbsTol', 5e-2)
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'RelTol', 1e-2)
```

Use the `showTolerances` method to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

Path	Port_Index	Tolerance_Type	Tolerance_Value
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'AbsTol'}	0.05
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'RelTol'}	0.01

Clear the tolerances associated with the `ProposalSettings` object.

```
clearTolerances(propSettings)
```

Using the `showTolerances` method, verify that there are no longer any tolerances associated with the `ProposalSettings` object.

```
showTolerances(propSettings)
```

### **See Also**

`addTolerance` | `showTolerances`

### **Topics**

“The Command-Line Interface for the Fixed-Point Tool”

**Introduced in R2019a**

# showTolerances

**Class:** `DataTypeWorkflow.ProposalSettings`

**Package:** `DataTypeWorkflow`

Show tolerances specified for a system

## Syntax

```
showTolerances(proposalSettings)
```

## Description

`showTolerances(proposalSettings)` displays the absolute, relative, and time tolerances specified for a system using the `addTolerance` method of the `DataTypeWorkflow.ProposalSettings` class. If the `ProposalSettings` object has no tolerances specified, the `showTolerances` method does not display anything.

## Input Arguments

**proposalSettings** — Proposal settings object

`DataTypeWorkflow.ProposalSettings` object

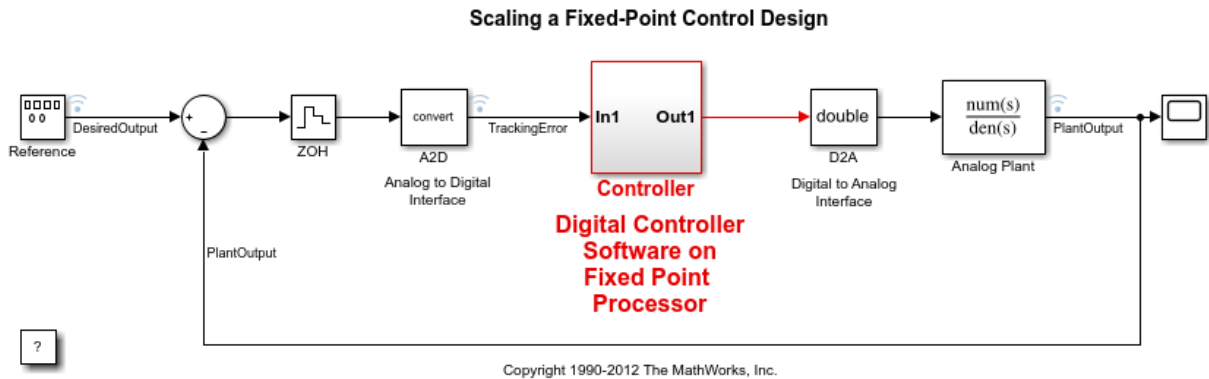
`DataTypeWorkflow.ProposalSettings` object specifying tolerances and settings to use during the data type proposal process.

## Examples

### Specify Signal Tolerances

This example shows how to apply tolerances to signals in a system. In this example, you apply tolerances to the `fxpdemo_feedback` model.

```
model = 'fxpdemo_feedback';
open_system(model);
```



Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'AbsTol', 5e-2)
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast', 1, 'RelTol', 1e-2)
```

Use the `showTolerances` method to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

Path	Port_Index	Tolerance_Type	Tolerance
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'AbsTol'}	0.05
{'fxpdemo_feedback/Controller/Down Cast'}	1	{'RelTol'}	0.01

Clear the tolerances associated with the `ProposalSettings` object.

```
clearTolerances(propSettings)
```

Using the `showTolerances` method, verify that there are no longer any tolerances associated with the `ProposalSettings` object.

```
showTolerances(propSettings)
```

## See Also

`addTolerance` | `clearTolerances`

## Topics

“The Command-Line Interface for the Fixed-Point Tool”

**Introduced in R2019a**

## convertToSingle

Convert a double-precision system to single precision

### Syntax

```
ConversionReport = DataTypeWorkflow.Single.convertToSingle(  
systemToConvert)
```

### Description

`ConversionReport = DataTypeWorkflow.Single.convertToSingle(systemToConvert)` converts the system specified by `systemToConvert` to single precision and returns a report. Data types that are specified as Boolean, fixed point, or one of the built-in integers are not affected by conversion.

### Input Arguments

**systemToConvert** — System to convert to single precision

character vector

The system to convert from double-precision to single-precision, specified as a character vector. The system must be open before using this method.

Data Types: char

### Output Arguments

**ConversionReport** — Report containing results from the conversion

report

Report containing results from the conversion.

## Examples

### Convert a system to single precision

- 1 Open the system to convert to single precision.

```
addpath(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))  
ex_fuel_rate_calculation
```

- 2 Use the `DataTypeWorkflow.Single.convertToSingle` method to convert the system from double precision to single precision.

```
report = DataTypeWorkflow.Single.convertToSingle('ex_fuel_rate_calculation')
```

The specified system now uses single-precision data types instead of double-precision data types. Data types in the model that were specified as Boolean, fixed-point, or one of the built-in integers remain the same after conversion.

## Alternatives

You can also use the Single Precision Converter app to convert a system from double precision to single precision. To open the Converter, from the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**. For more information, see “Getting Started with Single Precision Converter”.

## See Also

Single Precision Converter

## Topics

“Convert a System to Single Precision”

“Getting Started with Single Precision Converter”

**Introduced in R2016b**

## explore

**Class:** `DataTypeWorkflow.VerificationResult`

**Package:** `DataTypeWorkflow`

Explore comparison of baseline and fixed-point implementations

## Syntax

```
explore(verificationResult)
```

## Description

`explore(verificationResult)` opens the Simulation Data Inspector with logging data displayed for the `DataTypeWorkflow.VerificationResult` object specified by `result`.

## Input Arguments

**verificationResult** — Object comparing the behavior of a baseline run and a verification run

`DataTypeWorkflow.VerificationResult` object

Object comparing the behavior of a baseline run and a verification run, specified as a `DataTypeWorkflow.VerificationResult` object.

## See Also

`DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings` | `DataTypeWorkflow.VerificationResult`

## Topics

“Convert a Model to Fixed Point Using the Command Line”



**Introduced in R2019a**

## getNumDataPointsInfo

**Package:** fixed

Get information about number of data points in generated data

### Syntax

```
datainfo = getNumDataPointsInfo(datagenerator)
```

### Description

`datainfo = getNumDataPointsInfo(datagenerator)` returns information about the data points generated by the `fixed.DataGenerator` object, `datagenerator`.

### Examples

#### Get information about number of data points in generated data

The `getNumDataPointsInfo` function returns information related to the number of data points in the data generated from a `fixed.DataGenerator` object.

```
dataspec = fixed.DataSpecification('fixdt(1,16,13)',...  
    'Intervals', {-1,1})
```

```
dataspec =
```

```
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'sfix16_En13'  
        Intervals: [-1,1]  
        MandatoryValues: <empty>  
        Complexity: 'real'  
        Dimensions: 1
```

```

datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
    'NumDataPointsLimit', 20);
getNumDataPointsInfo(datagen)

ans =

    struct with fields:

        Current: 20
        Next: 21
        Min: 5
        Max: 75

```

The output indicates that there are currently 20 data combinations in the generated data. The maximum number of combinations that the `DataGenerator` object would produce is 75.

### Get information about number of data points for multidimensional data

When the dimension of the generated data is greater than one, it can be useful to find the next possible size of generated data.

Create a `DataGenerator` object where the associated `DataSpecification` object specifies 2-dimensional data.

```

dataspec = fixed.DataSpecification('single', 'Dimensions', 2);
datagen = fixed.DataGenerator('DataSpecifications', dataspec)

datagen =

```

```

    fixed.DataGenerator with properties:

        DataSpecifications: {[1x1 fixed.DataSpecification]}
        NumDataPointsLimit: 100000

```

The `DataGenerator` object uses the default limit of 100000 data points in the generated data.

Get information about the number of data points generated.

```

getNumDataPointsInfo(datagen)

ans =

```

```
struct with fields:
```

```
Current: 99856
Next: 100489
Min: 81
Max: 130321
```

The current size of the generated data is 99856 points. By setting the `NumDataPointsLimit` property of the `DataGenerator` object to the value specified in `Max`, you can get the maximum possible number of data combinations.

Set the `NumDataPointsLimit` property of the `DataGenerator` object to the maximum possible number of data points.

```
datagen.NumDataPointsLimit = 130321;
getNumDataPointsInfo(datagen)
```

```
ans =
```

```
struct with fields:
```

```
Current: 130321
Next: 130321
Min: 81
Max: 130321
```

## Input Arguments

### **datagenerator** — Object from which you want to get information

`fixed.DataGenerator` object

Object from which you want to get information, specified as a `fixed.DataGenerator` object.

## Output Arguments

### **datainfo** — Information about the number of data points

struct

Information about the number of data points in the data generated from a `fixed.DataGenerator` object, returned as a struct with the following fields.

Field	Description
Current	The number of data combinations in the generated data.
Next	Next possible size of data combinations.
Min	Minimum number of combinations of data required to be in the generated data.  This number is equal to the number of boundary values and mandatory values in the <code>DataSpecification</code> objects associated with the <code>DataGenerator</code> object.
Max	Maximum number of combinations that could be in the generated data.

## See Also

`fixed.DataGenerator` | `getUniqueValues` | `outputAllData`

**Introduced in R2019b**

## getUniqueValues

**Package:** fixed

Get unique values from `fixed.DataGenerator` object

### Syntax

```
data = getUniqueValues(datagenerator)
```

### Description

`data = getUniqueValues(datagenerator)` returns all unique values in the data generated by the `fixed.DataGenerator` object, `datagenerator`.

### Examples

#### Get unique values in data from DataGenerator object

In data generated from a `fixed.DataGenerator` object, there can be repeated values. Use the `getUniqueValues` function to get all of the unique values in the data set.

```
dataspec = fixed.DataSpecification('fixdt(1,16,13)', ...  
    'Intervals', {-1,1})
```

```
dataspec =
```

```
fixed.DataSpecification with properties:
```

```
    DataTypeStr: 'sfix16_En13'  
    Intervals: [-1,1]  
    MandatoryValues: <empty>  
    Complexity: 'real'  
    Dimensions: 1
```

```
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...  
    'NumDataPointsLimit', 20);  
getUniqueValues(datagen)
```

```
ans =
```

```
-1.0000  
-0.9999  
-0.4999  
-0.2500  
-0.0624  
-0.0313  
-0.0039  
-0.0021  
-0.0005  
-0.0002  
    0  
    0.0010  
    0.0018  
    0.0078  
    0.0155  
    0.0157  
    0.1249  
    0.1251  
    0.9999  
    1.0000
```

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

## Input Arguments

**datagenerator** — Input `fixed.DataGenerator` object

`fixed.DataGenerator` object

Input `fixed.DataGenerator` object to get unique values from.

## Output Arguments

### **data** — Unique set of values in data

scalar | vector | matrix

Unique set of data generated by the input `fixed.DataGenerator` object, returned as a scalar, vector, or matrix.

## See Also

`fixed.DataGenerator` | `getNumDataPointsInfo` | `outputAllData`

**Introduced in R2019b**



# outputAllData

**Package:** fixed

Get data from `fixed.DataGenerator` object

## Syntax

```
data = outputAllData(datagenerator)
data = outputAllData(datagenerator, format)
```

## Description

`data = outputAllData(datagenerator)` returns the data generated by the `fixed.DataGenerator` object, `datagenerator`.

`data = outputAllData(datagenerator, format)` returns the data generated by the `fixed.DataGenerator` object, `datagenerator`, in the format specified by `format`.

## Examples

### Get data as an array

Get the data from a `fixed.DataGenerator` object, returned as an array of values.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1});
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
    'NumDataPointsLimit', 20)
```

```
datagen =
```

```
fixed.DataGenerator with properties:
```

```
DataSpecifications: {[1x1 fixed.DataSpecification]}
NumDataPointsLimit: 20
```

Use the `outputAllData` function to access the data in the `DataGenerator` object.

```
data = outputAllData(datagen)
```

```
data =  
  
1×3 int8 row vector  
  
-1    0    1
```

The function returns the data in an array with the type specified by the `fixed.DataSpecification` object.

### Get data as a timeseries object

Get the data from a `fixed.DataGenerator` object, returned as a `timeseries` object.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1});  
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...  
    'NumDataPointsLimit', 2000)
```

```
datagen =  
  
fixed.DataGenerator with properties:  
  
DataSpecifications: {[1×1 fixed.DataSpecification]}  
NumDataPointsLimit: 20000
```

Specify the format of the output type to get a `timeseries` object.

```
data = outputAllData(datagen, 'timeseries')
```

```
timeseries  
  
Common Properties:  
    Name: 'unnamed'  
    Time: [3×1 double]  
    TimeInfo: [1×1 tsdata.timemetadata]
```

```
Data: [3x1 int8]
DataInfo: [1x1 tsdata.datametadate]
```

## Input Arguments

### **datagenerator** — Object from which you want to get data

`fixed.DataGenerator` object

Object from which you want to get data, specified as a `fixed.DataGenerator` object.

### **format** — Format in which you want data returned

'array' (default) | 'timeseries'

Format in which you want data returned, specified as either 'array', or 'timeseries'.

Example: `data = outputAllData(datagen, 'timeseries');`

Data Types: char

## Output Arguments

### **data** — Data from the DataGenerator object

scalar | vector | matrix | `timeseries` object

Data from the `DataGenerator` object, returned as either a scalar, vector, matrix, or `timeseries` object.

## See Also

`fixed.DataGenerator` | `getNumDataPointsInfo` | `getUniqueValues`

**Introduced in R2019b**

# applyOnRootInport

**Package:** fixed

Apply properties to Inport block

## Syntax

```
applyOnRootInport(dataspec, model, inportnumber)
```

## Description

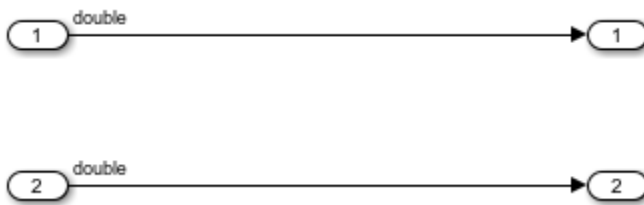
`applyOnRootInport(dataspec, model, inportnumber)` applies the properties specified in `fixed.DataSpecification` object, `dataspec` to the specified Inport block in `model`.

## Examples

### Apply properties of `fixed.DataSpecification` object to Inport blocks

The `ex_dataspecification` model contains two Inport blocks. Create an array of `fixed.DataSpecification` objects and apply the properties to the Inport blocks in the model.

```
model = 'ex_dataspecification';  
open_system(model);  
set_param(model, 'SimulationCommand', 'update')
```



Create a `fixed.DataSpecification` object and apply its properties to the Inport blocks in the model.

```
dataspec1 = fixed.DataSpecification('int8', 'Intervals', {-1,1})  
dataspec2 = fixed.DataSpecification('fixdt(1,16,10)', 'Intervals', {-5,5})
```

```
applyOnRootInport(dataspec1, 'ex_dataspecification', 1)  
applyOnRootInport(dataspec2, 'ex_dataspecification', 2)
```

```
dataspec1 =
```

```
    fixed.DataSpecification with properties:
```

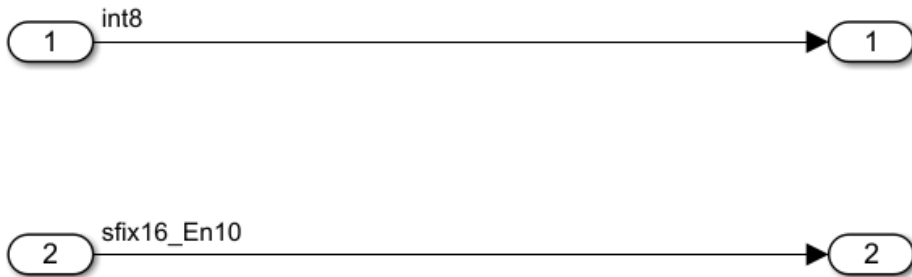
```
        DataTypeStr: 'int8'  
        Intervals: [-1,1]  
        MandatoryValues: <empty>  
        Complexity: 'real'  
        Dimensions: 1
```

```
dataspec2 =
```

```
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'sfix16_En10'  
        Intervals: [-5,5]  
        MandatoryValues: <empty>  
        Complexity: 'real'  
        Dimensions: 1
```

You can see that the signal from the Inport block In1 now specifies an int8 data type, and the signal from the In2 block specifies a data type of sfix16\_En10.



## Input Arguments

### **dataspec** — Properties to apply to Inport block

`fixed.DataSpecification` object

Properties to apply to Inport block, specified as a `fixed.DataSpecification` object.

### **model** — Model containing Inport block

character vector

Name of the model containing the Inport block to apply settings to, specified as a character vector.

Data Types: `char`

### **inportnumber** — Number of Inport block

scalar integer

Port number of root-level Inport block on which you want to apply properties from the `fixed.DataSpecification` object. The following properties of the `DataSpecification` object are applied to the block:

- Data type
- Complexity
- Dimensions

Data Types: `double`

## See Also

`contains` | `fixed.DataSpecification`

**Introduced in R2019b**

## contains

**Package:** fixed

Determine whether value domain of a `DataSpecification` object contains a specified value

## Syntax

```
bool = contains(dataspec, value)
```

## Description

`bool = contains(dataspec, value)` returns a boolean value indicating whether the value domain of the `fixed.DataSpecification` object, `dataspec`, contains the value, `value`.

## Examples

### Determine whether a `fixed.DataSpecification` object contains a value

Use the `contains` function to determine whether a `fixed.DataSpecification` object contains a specified value.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1})
```

```
dataspec =
```

```
fixed.DataSpecification with properties:
```

```
    DataTypeStr: 'int8'  
    Intervals: [-1,1]  
MandatoryValues: <empty>  
    Complexity: 'real'  
    Dimensions: 1
```



Determine whether `dataspec` contains the value 0.

```
bool = contains(dataspec, 0)
```

```
bool =
```

```
    logical
```

```
    1
```

## Input Arguments

**dataspec** — **fixed.DataSpecification object**

`fixed.DataSpecification` object

Input `fixed.DataSpecification` object.

**value** — **Value**

scalar | vector

Value or values to check for in the `fixed.DataSpecification` object, specified as a scalar, or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

**bool** — **Whether the `fixed.DataSpecification` object contains the value**

`true` | `false` | vector of logical values

Whether the `fixed.DataSpecification` object contains the value, returned as a boolean value.

If the `value` argument is a vector, the output is a boolean vector of the same length.

## See Also

`applyOnRootInport` | `fixed.DataSpecification`

**Introduced in R2019b**

# contains

**Package:** fixed

Determine if one `fixed.Interval` object contains another

## Syntax

```
bool = contains(A, B)
```

## Description

`bool = contains(A, B)` returns a boolean indicating whether `fixed.Interval` object A contains the `fixed.Interval` object B.

## Examples

### Determine if a `fixed.Interval` object contains another

Create two `fixed.Interval` objects. Use the `contains` function to determine if the intervals in `interval2` are contained within the corresponding intervals in `interval1`.

```
interval1 = fixed.Interval({0,1}, {2,3}, {3,4});  
interval2 = fixed.Interval({0,0.5}, {2.5, 3}, {4,5});  
bool = contains(interval1, interval2)
```

```
bool = 1x3 logical array
```

```
1 1 0
```

When the second input is a scalar `Interval` object, `contains` determines whether each interval of the first input contains the interval of the second input.

```
interval2 = fixed.Interval(0,1);  
bool = contains(interval1, interval2)
```

```
bool = 1x3 logical array  
  1  0  0
```

## Input Arguments

### **A, B — Input fixed.Interval objects**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

If A is an array of `Interval` objects, B must be a scalar `Interval` object or an `Interval` object with the same dimensions as A.

## Output Arguments

### **bool — Whether B is contained in A**

`true` | `false` | logical array

Whether `fixed.Interval` object B is contained in `fixed.Interval` object A, returned as a logical value.

When A is an array of `Interval` objects, the output is an array of logical values of the same size as A.

## See Also

`fixed.Interval` | `intersect` | `overlaps` | `setdiff` | `union` | `unique`

**Introduced in R2019b**

# intersect

**Package:** fixed

Intersection of `fixed.Interval` objects

## Syntax

```
C = intersect(A, B)
```

## Description

`C = intersect(A, B)` returns the intersection of `fixed.Interval` objects A and B.

## Examples

### Get intersection of two `fixed.Interval` objects

Create two `fixed.Interval` objects.

```
interval1 = fixed.Interval(-10, 10)
```

```
interval1 =  
  [-10,10]
```

1x1 `fixed.Interval` with properties:

```
      LeftEnd: -10  
      RightEnd: 10  
      IsLeftClosed: true  
      IsRightClosed: true
```

```
interval2 = fixed.Interval(0, 20)
```

```
interval2 =  
  [0,20]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 0
    RightEnd: 20
    IsLeftClosed: true
    IsRightClosed: true
```

Find the intersection of the two `Interval` objects.

```
intervalIntersection = intersect(interval1, interval2)
```

```
intervalIntersection =
  [0,10]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 0
    RightEnd: 10
    IsLeftClosed: true
    IsRightClosed: true
```

The output is an `Interval` object whose range is the intersection of the ranges of the two input `Interval` objects.

When the ranges of the two input `Interval` objects do not overlap, the output is an empty `Interval` object.

```
interval3 = fixed.Interval(100,200)
```

```
interval3 =
  [100,200]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 100
    RightEnd: 200
    IsLeftClosed: true
    IsRightClosed: true
```

```
intervalIntersection = intersect(interval1, interval3)
```

```
intervalIntersection =
```

```
1x0 fixed.Interval with properties:
```

```
LeftEnd  
RightEnd  
IsLeftClosed  
IsRightClosed
```

## Input Arguments

### A, B — Input `fixed.Interval` objects

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

## Output Arguments

### C — Intersection of `fixed.Interval` objects

`fixed.Interval` object | array of `fixed.Interval` objects

Intersection of input `fixed.Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

The output `Interval` object contains all values in both inputs, A and B.

## See Also

`contains` | `fixed.Interval` | `overlaps` | `setdiff` | `union` | `unique`

**Introduced in R2019b**

## isDegenerate

**Package:** fixed

Determine whether the left and right ends of a `fixed.Interval` object are degenerate

## Syntax

```
bool = isDegenerate(A)
```

## Description

`bool = isDegenerate(A)` returns a boolean indicating whether the left and right ends of the `fixed.Interval` object `A` are the same, or equivalently, whether the interval contains only one point.

## Examples

### Determine if a `fixed.Interval` object has degenerate end points

Create a `fixed.Interval` object. Use the `isDegenerate` function to determine whether the left and right ends of the `Interval` object are the same.

```
interval = fixed.Interval({-pi,pi},{1,1});  
bool = isDegenerate(interval)
```

```
bool = 1x2 logical array
```

```
  0  1
```

The output is a logical `0` when the left and right ends of the interval are different, and `1` when they are the same.



## Input Arguments

### **A** — **fixed.Interval object**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

## Output Arguments

### **bool** — **Indicates whether left and right ends of A are degenerate**

`true` | `false` | logical array

Indicates whether the `fixed.Interval` object A has degenerate end points. Returns 1 (true) when the left and right ends of A are the same, or equivalently, when the interval contains only one point, and 0 (false) otherwise.

When A is an array of `Interval` objects, the output is an array of logical values of the same size as A.

## See Also

`fixed.Interval` | `isLeftBounded` | `isRightBounded` | `isnan`

**Introduced in R2019b**

## isLeftBounded

**Package:** fixed

Determine whether a `fixed.Interval` object is left-bounded

### Syntax

```
bool = isLeftBounded(A)
```

### Description

`bool = isLeftBounded(A)` returns a boolean indicating whether the `fixed.Interval` object `A` is left-bounded.

### Examples

#### Determine if a `fixed.Interval` object is left bounded

Create a `fixed.Interval` object. Use the `isLeftBounded` function to determine whether the interval is bounded on the left.

```
interval = fixed.Interval({-pi,pi},{-inf,1});  
bool = isLeftBounded(interval)
```

`bool = 1x2 logical array`

```
1  0
```

The output is a logical 1 when the left end of the interval is bounded, and 0 otherwise.

## Input Arguments

### **A** — **fixed.Interval object**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

## Output Arguments

### **bool** — **Indicates whether left end of A is bounded**

`true` | `false` | logical array

Indicates whether the `fixed.Interval` object A is left-bounded, returned as a logical value. Returns 0 (false) when A contains `-inf`, and 1 (true) otherwise.

When A is an array of `Interval` objects, the output is an array of logical values of the same size as A.

## See Also

`fixed.Interval` | `isDegenerate` | `isRightBounded` | `isnan`

**Introduced in R2019b**

## isnan

**Package:** fixed

Determine whether a `fixed.Interval` object is NaN

## Syntax

```
bool = isnan(A)
```

## Description

`bool = isnan(A)` returns a boolean indicating whether a `fixed.Interval` object `A` is NaN.

## Examples

### Determine if a `fixed.Interval` object is NaN

Create a `fixed.Interval` object. Use the `isnan` function to determine whether the `Interval` object is not a number.

```
interval = fixed.Interval({-pi, pi}, {nan, 1}, {nan, nan});  
bool = isnan(interval)
```

```
bool = 1x3 logical array
```

```
    0    1    1
```

The output is a logical 1 when the interval contains one or more NaN elements, and 0 otherwise.

## Input Arguments

### **A** — **fixed.Interval object**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

## Output Arguments

### **bool** — **Indicates whether elements of A are NaN**

`true` | `false` | logical array

Indicates whether the `fixed.Interval` object A is NaN, returned as a logical value.

When A is an array of `Interval` objects, the output is an array of logical values of the same size as A.

## See Also

`fixed.Interval` | `isDegenerate` | `isLeftBounded` | `isRightBounded`

**Introduced in R2019b**

## isRightBounded

**Package:** fixed

Determine whether the a `fixed.Interval` object is right-bounded

### Syntax

```
bool = isRightBounded(A)
```

### Description

`bool = isRightBounded(A)` returns a boolean indicating whether the `fixed.Interval` object `A` is right-bounded.

### Examples

#### Determine if a `fixed.Interval` object is right bounded

Create a `fixed.Interval` object. Use the `isRightBounded` function to determine whether the interval is bounded on the right.

```
interval = fixed.Interval({-pi,pi},{-1,inf});  
bool = isRightBounded(interval)
```

*bool = 1x2 logical array*

```
1  0
```

The output is logical 1 when the right end of the interval is bounded, and 0 otherwise.

## Input Arguments

### **A** — **fixed.Interval object**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

## Output Arguments

### **bool** — **Indicates whether right end of A is bounded**

Boolean scalar | Boolean array

Indicates whether the `fixed.Interval` object A is right-bounded, returned as a logical value. Returns 0 (false) when A contains `inf`, and 1 (true) otherwise.

When A is an array of `Interval` objects, the output is an array of logical values of the same size as A.

## See Also

`fixed.Interval` | `isDegenerate` | `isLeftBounded` | `isnan`

**Introduced in R2019b**

## overlaps

**Package:** `fixed`

Determine if two `fixed.Interval` objects overlap

## Syntax

```
bool = overlaps(A, B)
```

## Description

`bool = overlaps(A, B)` returns a boolean indicating whether two `fixed.Interval` objects overlap.

## Examples

### Determine if two `fixed.Interval` objects overlap

Create two `fixed.Interval` objects and determine if their ranges overlap.

```
interval1 = fixed.Interval(-1, 1);  
interval2 = fixed.Interval(0, 1);  
overlaps(interval1, interval2)
```

```
ans =
```

```
logical
```

```
1
```



When the ranges of the `Interval` objects overlap, the `overlaps` function returns a value of 1, or `true`.

## Input Arguments

### **A, B** – Input `fixed.Interval` objects

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

## Output Arguments

### **bool** – Whether the intervals overlap

`true` | `false` | vector of logical values

Whether the input `fixed.Interval` objects overlap, returned as a logical value or a vector of logical values.

## See Also

`contains` | `fixed.Interval` | `intersect` | `setdiff` | `union` | `unique`

**Introduced in R2019b**

## quantize

**Package:** fixed

Quantize interval to range of numeric data type

### Syntax

```
quantizedinterval = quantize(interval, numerictype)
quantizedinterval = quantize(interval, numerictype, Name, Value)
```

### Description

`quantizedinterval = quantize(interval, numerictype)` returns the quantized range of `fixed.Interval` object, `interval`, quantized to the numeric type specified by `numerictype`.

`quantizedinterval = quantize(interval, numerictype, Name, Value)` returns the quantized range of `fixed.Interval` object, `interval`, with additional properties specified as name-value pairs.

### Examples

#### Quantize a numeric interval to uint8

Create a `fixed.Interval` object and find the range of the `Interval` object quantized to an unsigned 8-bit integer.

```
interval = fixed.Interval(-200,200);
quantizedInterval = quantize(interval, 'fixdt(0,8,0)')
```

```
quantizedInterval =
    1×2 uint8 row vector
    0    200
```

Because `fixdt(0,8,0)` is equivalent to `uint8`, the `quantize` function returns the quantized range as a `uint8` row vector with the endpoints within the representable range of the numeric type.

To return the quantized row vector as a fixed-point data type, set the `'PreferBuiltIn'` property to `false`.

```
quantizedInterval = quantize(interval, 'fixdt(0,8,0)',...
    'PreferBuiltIn', false)
```

```
quantizedInterval =
```

```
    0    200
```

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

## Input Arguments

### **interval** – Input fixed.Interval objects to quantize

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

### **numerictype** – Numeric data type

`Simulink.Numerictype` object | `embedded.numerictype` object | character vector

Numeric data type to quantize the `Interval`, specified as a `Simulink.Numerictype` object, an `embedded.numerictype` object, or a character vector representing a numeric data type, for example, `'single'`.

Example: `quantizedinterval = quantize(interval, 'fixdt(1,16,8)');`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

```
Example: interval = quantize(interval, 'fixdt(1,16,0)',  
'PreferBuiltIn', false, 'PreferStrict', true);
```

**PreferBuiltIn — Quantize to built-in data type when possible**

true (default) | false

When this property is true, if the specified `numerictype` has an equivalent built-in integer type the software returns the built-in type. For example, when this property is true, a specified `numerictype` of `'fixdt(1,8,0)'` would return an `int8`.

Data Types: logical

**PreferStrict — Quantize end points to numeric type**

false (default) | true

When this property is true, all ends are quantized to the closest representable values within original intervals regardless of whether the intervals are closed or open.

Data Types: logical

## Output Arguments

**quantizedinterval — Quantized interval range**

*N*-by-2 matrix

*N*-by-2 matrix with rows consisting of endpoints of input `Interval` objects quantized to the numeric data type specified by `numerictype`.

When the `'PreferStrict'` property is set to `false`, the end points after quantization may lie outside the original interval.

## See Also

`contains` | `fixed.Interval` | `intersect` | `overlaps` | `union` | `unique`

**Introduced in R2019b**

# setdiff

**Package:** fixed

Set difference of `fixed.Interval` objects

## Syntax

```
C = setdiff(A, B)
```

## Description

`C = setdiff(A, B)` returns a `fixed.Interval` object containing the values in `fixed.Interval` object A, but not in B.

## Examples

### Get set difference of two `fixed.Interval` objects

Create two `fixed.Interval` objects. Use the `setdiff` function to find the values that are in `Interval` object `interval1` but not in `interval2`. In this example, `interval1` contains all values between 0 and 1, but `interval2` only contains values from 0 to 0.5, so the output `Interval` object has an interval from 0.5 to 1.

```
interval1 = fixed.Interval(0,1);  
interval2 = fixed.Interval(0,0.5);  
intervaldiff = setdiff(interval1, interval2)
```

```
intervaldiff =  
    (0.5000,1]
```

1x1 `fixed.Interval` with properties:

```
    LeftEnd: 0.5000  
    RightEnd: 1
```

```
IsLeftClosed: false
IsRightClosed: true
```

### Create an interval object that excludes zero

You can use the `setdiff` function to create an interval object based on another interval, while excluding zero.

Create an `Interval` object that contains zero.

```
myInterval = fixed.Interval(-1,1);
```

To create an interval based on the `Interval` object, `myInterval`, use the `setdiff` function. Include the constructor for a degenerate `Interval` object containing only zero as the second argument.

```
myInterval_nozero = setdiff(myInterval, {0});
```

```
myInterval_nozero =
```

```
[-1,0)  (0,1]
```

```
1x2 fixed.Interval with properties:
```

```
LeftEnd
RightEnd
IsLeftClosed
IsRightClosed
```

The output `Interval` object, `myInterval_nozero`, contains two intervals, each with an open end point at zero. Therefore, the interval contains all values between -1 and 1, except 0.

## Input Arguments

### A, B — Input `fixed.Interval` objects

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

## Output Arguments

### **C — Set difference of fixed.Interval objects**

`fixed.Interval` object | array of `fixed.Interval` objects

Set difference of input `fixed.Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

The output `Interval` object contains all values in first input, A, but not in B.

## See Also

`contains` | `fixed.Interval` | `intersect` | `overlaps` | `union`

**Introduced in R2019b**

## union

**Package:** fixed

Union of `fixed.Interval` objects

## Syntax

```
C = union(A, B)
```

## Description

`C = union(A, B)` returns the union of `fixed.Interval` objects A and B.

## Examples

### Get the union of two `fixed.Interval` objects

Create two `fixed.Interval` objects.

```
interval1 = fixed.Interval(-10, 10)
```

```
interval1 =  
  [-10,10]
```

1x1 `fixed.Interval` with properties:

```
      LeftEnd: -10  
      RightEnd: 10  
      IsLeftClosed: true  
      IsRightClosed: true
```

```
interval2 = fixed.Interval(0,20)
```

```
interval2 =  
  [0,20]
```



```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 0
    RightEnd: 20
    IsLeftClosed: true
    IsRightClosed: true
```

Find the union of the two `Interval` objects.

```
intervalUnion = union(interval1, interval2)
```

```
intervalUnion =
  [-10,20]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: -10
    RightEnd: 20
    IsLeftClosed: true
    IsRightClosed: true
```

The output is an `Interval` object whose range is the union of the ranges of the two input objects.

When the ranges of the two input `Interval` objects do not overlap, the output is an array of `Interval` objects covering the union of the ranges of the inputs.

```
interval3 = fixed.Interval(100, 200)
```

```
interval3 =
  [100,200]
```

```
1x1 fixed.Interval with properties:
```

```
    LeftEnd: 100
    RightEnd: 200
    IsLeftClosed: true
    IsRightClosed: true
```

```
intervalUnion = union(interval1, interval3)
```

```
intervalUnion=1x2 object
  [-10,10]  [100,200]
```

1x2 `fixed.Interval` with properties:

```
    LeftEnd  
    RightEnd  
    IsLeftClosed  
    IsRightClosed
```

## Input Arguments

### **A, B — Input `fixed.Interval` objects**

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

## Output Arguments

### **C — Union of `fixed.Interval` objects**

`fixed.Interval` object | array of `fixed.Interval` objects

Union of input `fixed.Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

The output `Interval` object contains all values in A or B.

## See Also

`contains` | `fixed.Interval` | `intersect` | `overlaps` | `setdiff`

**Introduced in R2019b**

# unique

**Package:** fixed

Get set of unique values in `fixed.Interval` object

## Syntax

```
uniqueinterval = unique(interval)
```

## Description

`uniqueinterval = unique(interval)` returns a vector of incrementally sorted and non overlapping intervals that represent an equivalent value set as `fixed.Interval` object, `interval`.

## Examples

### Create a non-overlapping set of intervals from an array of `Interval` objects

Use the `unique` function to get a non-overlapping set of intervals from an array of `Interval` objects.

```
intervals = fixed.Interval({-5,5},{-10,10},{4,20},{50,100})
```

```
[-5,5]    [-10,10]    [4,20]    [50,100]
```

1x4 `fixed.Interval` with properties:

```
LeftEnd
RightEnd
IsLeftClosed
IsRightClosed
```

The first three intervals represented in the object overlap with one another. The fourth interval is disjointed from the set.

```
uniqueInterval = unique(intervals)
uniqueInterval =
    [-10,20]    [50,100]
1x2 fixed.Interval with properties:
    LeftEnd
    RightEnd
    IsLeftClosed
    IsRightClosed
```

The output, `uniqueInterval`, an array of two `Interval` objects, merges the three overlapping intervals into a single `Interval` object.

## Input Arguments

### **interval** — **fixed.Interval** object

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

## Output Arguments

### **uniqueinterval** — **Non-overlapping set of Interval** objects

`fixed.Interval` object | array of `fixed.Interval` objects

Non-overlapping set of `Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

When `interval` is a scalar `Interval` object, the output is the same as the input.

## See Also

`contains` | `fixed.Interval` | `intersect` | `overlaps` | `setdiff` | `union`

**Introduced in R2019b**

## lutmemoryusage

**Class:** FunctionApproximation.LUTMemoryUsageCalculator

**Package:** FunctionApproximation

Calculate total memory used by lookup table blocks in a model

### Syntax

```
memory = lutmemoryusage(calculator,model)
```

### Description

`memory = lutmemoryusage(calculator,model)` calculates the memory used by all lookup table blocks in the specified model.

### Input Arguments

**calculator** — **FunctionApproximation.LUTMemoryUsageCalculator object**  
FunctionApproximation.LUTMemoryUsageCalculator

FunctionApproximation.LUTMemoryUsageCalculator object.

**model** — **Model containing lookup table blocks**

character vector

Model containing lookup table blocks, specified as a character vector.

Data Types: char

### Output Arguments

**memory** — **Memory used by the model**

table

Table displaying the memory, in bits, used by each lookup table block in the specified model.

## Examples

### Calculate the Total Memory Used by Lookup Tables in a Model

Use the `FunctionApproximation.LUTMemoryUsageCalculator` class to calculate the total memory used by lookup table blocks in a model.

Create a `FunctionApproximation.LUTMemoryUsageCalculator` object.

```
calculator = FunctionApproximation.LUTMemoryUsageCalculator
```

Use the `lutmemoryusage` method to get the total memory used by the lookup table blocks in the `sldemo_fuelsys` model.

```
load_system('sldemo_fuelsys')
lutmemoryusage(calculator, 'sldemo_fuelsys')
```

```
ans =
```

```
6x1 table
```

```
sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throttle
sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estimati
sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Estir
sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki
Total
```

## See Also

**Apps**  
**Lookup Table Optimizer**

**Classes**

FunctionApproximation.LUTMemoryUsageCalculator |  
FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
FunctionApproximation.Problem

**Topics**

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**



# approximate

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Generate a Lookup Table block from a FunctionApproximation.LUTSolution

## Syntax

```
approximate(solution)
```

## Description

`approximate(solution)` generates a Simulink model containing a subsystem made up of the Lookup Table block with data and breakpoints specified by the `FunctionApproximation.LUTSolution` object, `solution`. The generated Lookup Table block is surrounded with Data Type Conversion blocks.

## Input Arguments

**solution** — Solution to generate lookup table from  
`FunctionApproximation.LUTSolution` object

The solution to generate a lookup table from, specified as a `FunctionApproximation.LUTSolution` object.

## Examples

### Generate a Lookup Table Approximating a Function

Create a `FunctionApproximation.Problem` object defining the function you want to approximate.

```
problem = FunctionApproximation.Problem('tanh')
```

```
problem =
```

```
1x1 FunctionApproximation.Problem with properties:
```

```
FunctionToApproximate: @(x)tanh(x)
  NumberOfInputs: 1
    InputTypes: "numerictype(1,16,12)"
  InputLowerBounds: -8
  InputUpperBounds: 8
    OutputType: "numerictype(1,16,15)"
    Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. Approximate the `tanh` function using the `solve` method.

```
solution = solve(problem)
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints	WLs	TableData	WL	Break
0	64	0	2		16		16	
1	1248	1	76		16		16	
2	1232	1	75		16		16	
3	944	1	57		16		16	
4	928	1	56		16		16	
5	656	0	39		16		16	
6	640	0	38		16		16	
7	784	0	47		16		16	
8	848	1	51		16		16	
9	816	1	49		16		16	
10	800	0	48		16		16	
11	384	0	12		16		16	
12	448	1	14		16		16	
13	384	1	12		16		16	

```
Best Solution
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints	WLs	TableData	WL	Break
13	384	1	12		16		16	

```
solution =
```

```
1x1 FunctionApproximation.LUTSolution with properties:
```

```
ID: 13
```

Feasible: "true"

Generate a Simulink™ subsystem containing a Lookup Table block approximating the tanh function.

`approximate(solution)`



## See Also

### Apps

**Lookup Table Optimizer**

### Classes

FunctionApproximation.LUTMemoryUsageCalculator |  
 FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
 FunctionApproximation.Problem

### Functions

`approximate` | `compare` | `solve`

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
 “Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## compare

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Compare numerical results of FunctionApproximation.LUTSolution to original function or lookup table

## Syntax

```
data = compare(solution)
```

## Description

`data = compare(solution)` plots the difference between the data contained in the FunctionApproximation.LUTSolution object, `solution`, and the original lookup table, function, or Math Function block.

## Input Arguments

**solution** — Solution to compare original behavior against

FunctionApproximation.LUTSolution object

The solution to compare original behavior against, specified as a FunctionApproximation.LUTSolution object.

## Output Arguments

**data** — Struct containing data comparing original and the solution

struct

Struct containing data comparing the original function or lookup table and the approximation contained in the solution.

## Examples

### Compare Function Approximation to Original Function

Create a `FunctionApproximation.Problem` object defining the function you want to approximate.

```
problem = FunctionApproximation.Problem('tanh')

problem =
  1x1 FunctionApproximation.Problem with properties:

    FunctionToApproximate: @(x)tanh(x)
      NumberOfInputs: 1
        InputTypes: "numeric(1,16,12)"
    InputLowerBounds: -8
    InputUpperBounds: 8
      OutputType: "numeric(1,16,15)"
      Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. Approximate the `tanh` function using the `solve` method.

```
solution = solve(problem)
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
0	64	0	2	16	16	
1	1248	1	76	16	16	
2	1232	1	75	16	16	
3	944	1	57	16	16	
4	928	1	56	16	16	
5	656	0	39	16	16	
6	640	0	38	16	16	
7	784	0	47	16	16	
8	848	1	51	16	16	
9	816	1	49	16	16	
10	800	0	48	16	16	
11	384	0	12	16	16	
12	448	1	14	16	16	
13	384	1	12	16	16	

Best Solution

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
13	384	1	12	16	16	

```

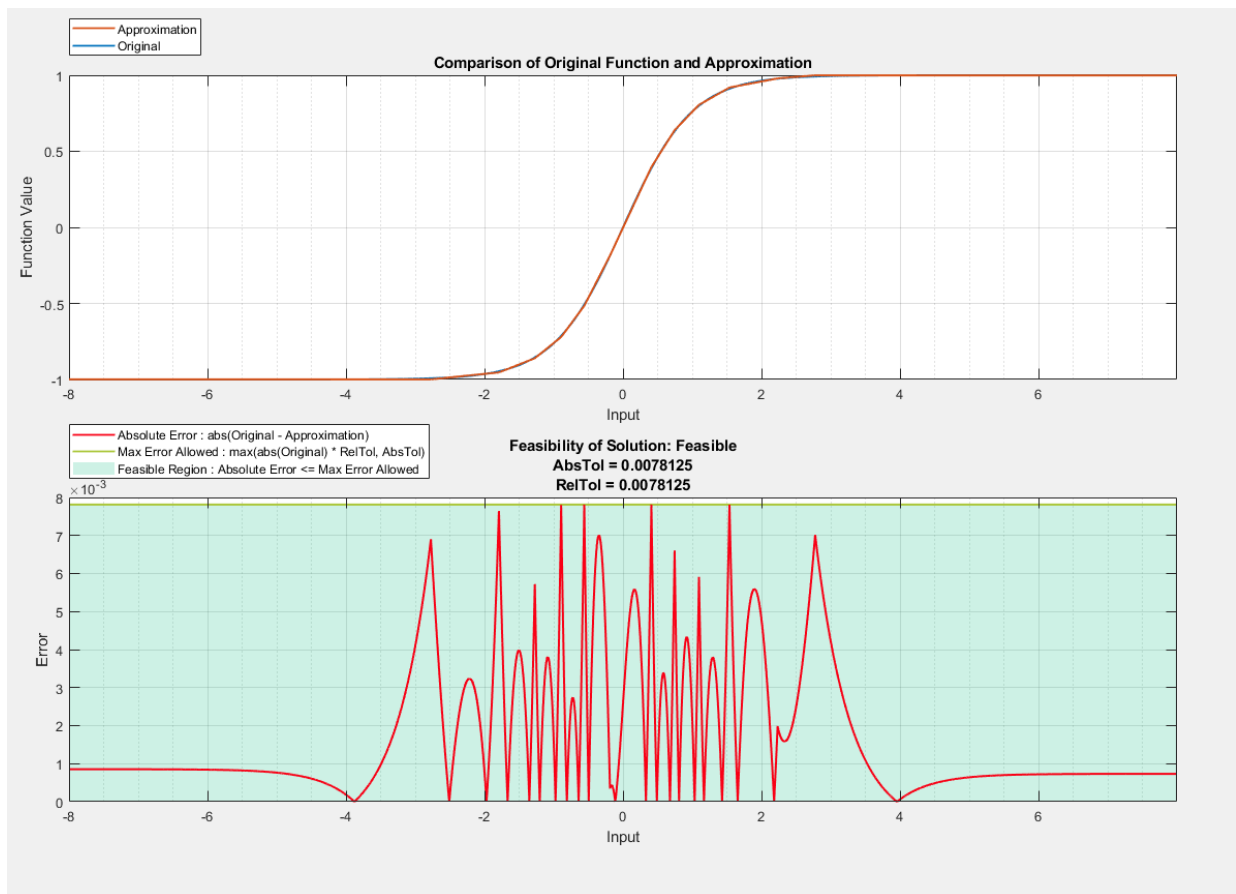
solution =
  1x1 FunctionApproximation.LUTSolution with properties:

      ID: 13
  Feasible: "true"

```

Compare the original function and the function approximation.

```
data = compare(solution)
```



```

data = struct with fields:
  Breakpoints: [65536x1 double]

```

Original: [65536x1 double]  
Approximate: [65536x1 double]

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

FunctionApproximation.LUTMemoryUsageCalculator |  
FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
FunctionApproximation.Problem

### Functions

approximate | compare | solve

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## displayallsolutions

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Display all solutions found during function approximation

### Syntax

```
displayallsolutions(solution)
```

### Description

`displayallsolutions(solution)` displays all solutions, including the non-feasible solutions, associated with a `FunctionApproximation.LUTSolution` object.

### Input Arguments

**solution** — **Solution object from which to display all associated solutions**

`FunctionApproximation.LUTSolution` object

`FunctionApproximation.LUTSolution` object from which to display all associated solutions.

### Examples

#### Display All Solutions Found During Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Display all solutions found during the approximation process using the `displayallsolutions` method.



```
problem = FunctionApproximation.Problem('sin')
```

```
problem =
```

```
FunctionApproximation.Problem with properties
```

```
FunctionToApproximate: @(x)sin(x)
NumberOfInputs: 1
InputTypes: "numeric(0,16,13)"
InputLowerBounds: 0
InputUpperBounds: 6.2832
OutputType: "numeric(1,16,14)"
Options: [1x1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution =
```

```
FunctionApproximation.LUTSolution with properties
```

```
ID: 8
Feasible: "true"
```

```
displayallsolutions(solution)
```

ID	Memory (bits)	ConstraintMet	Table Size	Breakpoints	WLs	Ta
0	64	0	2		16	
1	464	0	27		16	
2	864	1	52		16	
3	64	0	2		16	
4	560	1	33		16	
5	304	0	17		16	
6	432	0	25		16	
7	496	1	29		16	
8	464	1	27		16	
9	448	0	26		16	
10	704	1	22		16	

```
Best Solution
```

	ID		Memory (bits)		ConstraintMet		Table Size		Breakpoints WLS		Ta
	8		464		1		27		16		

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

FunctionApproximation.LUTMemoryUsageCalculator |  
FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
FunctionApproximation.Problem

### Functions

displayfeasiblesolutions | solutionfromID | totalmemoryusage

## Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

# displayfeasiblesolutions

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Display all feasible solutions found during function approximation

## Syntax

```
displayfeasiblesolutions(solution)
```

## Description

`displayfeasiblesolutions(solution)` displays all feasible solutions found during the approximation process, including the best solution. Feasible solutions are defined as any solutions to the original `FunctionApproximation.Problem` object that met the constraints defined in the associated `FunctionApproximation.Options` object.

## Input Arguments

**solution** — **Solution object from which to display all associated feasible solutions**

`FunctionApproximation.LUTSolution` object

`FunctionApproximation.LUTSolution` object from which to display all associated feasible solutions.

## Examples

### Display All Feasible Solutions Found During Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Display all feasible solutions found during the approximation process using the `displayfeasiblesolutions` method.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem =
```

```
FunctionApproximation.Problem with properties
```

```
FunctionToApproximate: @(x)sin(x)
NumberOfInputs: 1
InputTypes: "numeric(0,16,13)"
InputLowerBounds: 0
InputUpperBounds: 6.2832
OutputType: "numeric(1,16,14)"
Options: [1x1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution =
```

```
FunctionApproximation.LUTSolution with properties
```

```
ID: 8
Feasible: "true"
```

```
displayfeasiblesolutions(solution)
```

ID	Memory (bits)	ConstraintMet	Table Size	Breakpoints	WLs
2	864	1	52		16
4	560	1	33		16
7	496	1	29		16
8	464	1	27		16
10	704	1	22		16

```
Best Solution
```

ID	Memory (bits)	ConstraintMet	Table Size	Breakpoints	WLs
8	464	1	27		16

## See Also

**Apps**  
**Lookup Table Optimizer**

### **Classes**

FunctionApproximation.LUTMemoryUsageCalculator |  
FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
FunctionApproximation.Problem

### **Functions**

compare | displayallsolutions | solutionfromID | totalmemoryusage

### **Topics**

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

### **Introduced in R2018a**

## getErrorValue

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Get the total error of the lookup table approximation

## Syntax

```
memory = getErrorValue(solution)
```

## Description

`memory = getErrorValue(solution)` returns the total error of the lookup table approximation specified by `solution`.

## Input Arguments

**solution** — Solution to get error of

FunctionApproximation.LUTSolution object

Solution to get error of, specified as a FunctionApproximation.LUTSolution object.

## Output Arguments

**error** — Total error of the lookup table approximation

struct

Total error of the lookup table approximation, returned as a struct.

The struct contains two fields. The `MaxErrorInSolution` field specifies the maximum difference between the original function or block and the lookup table approximation. The `ErrorUpperBound` field displays the maximum error that was acceptable according to the tolerances specified on the `FunctionApproximation.Options` object.

## Examples

### Calculate the Total Error of a Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Calculate the total error of the `FunctionApproximation.LUTSolution` object using the `getErrorValue` method.

```
problem = FunctionApproximation.Problem('sin')
problem =
    FunctionApproximation.Problem with properties
        FunctionToApproximate: @(x)sin(x)
        NumberOfInputs: 1
        InputTypes: "numeric(0,16,13)"
        InputLowerBounds: 0
        InputUpperBounds: 6.2832
        OutputType: "numeric(1,16,14)"
        Options: [1x1 FunctionApproximation.Options]

solution = solve(problem)
solution =
    FunctionApproximation.LUTSolution with properties
        ID: 8
        Feasible: "true"

error = getErrorValue(solution)
error =
    struct with fields:
```

```
MaxErrorInSolution: 0.0073  
ErrorUpperBound: 0.0078
```

## **See Also**

`FunctionApproximation.LUTSolution`

## **Topics**

“Approximate Functions with a Direct Lookup Table”

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

**Introduced in R2019a**



# replaceWithApproximate

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Replace block with the generated lookup table approximation

## Syntax

```
replaceWithApproximate(solution)
```

## Description

`replaceWithApproximate(solution)` replaces the simulink block with its lookup table approximation, generated using the `approximate` method of the `FunctionApproximation.LUTSolution` object.

## Input Arguments

**solution** — Solution to use to replace the source block

`FunctionApproximation.LUTSolution` object

Solution to replace the source block, specified as a `FunctionApproximation.LUTSolution` object.

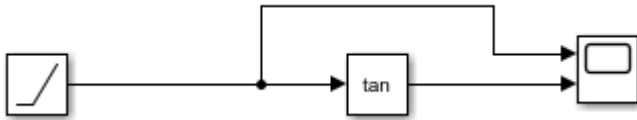
## Examples

### Replace a Block with an Approximation

This example shows how to approximate a block using a lookup table approximation, replace the original block with the approximation, and then revert the block back to its original state.

Open the model containing the block to approximate. In this example, replace the tan block with a lookup table approximation.

```
open_system('ex_luto_approx')
```



Create a `FunctionApproximation.Problem` object specifying what you want to approximate.

```
problem = FunctionApproximation.Problem('ex_luto_approx/Trigonometric Function')
```

```
problem =
```

```
1x1 FunctionApproximation.Problem with properties:
```

```
FunctionToApproximate: 'ex_luto_approx/Trigonometric Function'
NumberOfInputs: 1
InputTypes: "numerictype('double')"
InputLowerBounds: -1.5083
InputUpperBounds: 1.5083
OutputType: "numerictype('double')"
Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. To approximate the block use the `solve` method.

```
solution = solve(problem)
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints	WLs	TableData	WL	Break
0	48	0	2		8		16	
1	800	0	49		8		16	
2	1584	1	98		8		16	
3	640	0	39		8		16	
4	416	0	25		8		16	
5	1056	0	65		8		16	
6	64	0	2		16		16	
7	768	1	46		16		16	
8	752	1	45		16		16	
9	592	1	35		16		16	

10	576	1	34	16	16
11	416	0	24	16	16
12	400	0	23	16	16
13	496	0	29	16	16
14	528	1	31	16	16
15	512	0	30	16	16
16	80	0	2	8	32
17	48	0	2	8	16
18	64	0	2	16	16
19	80	0	2	8	32
20	96	0	2	16	32
21	128	0	2	32	32
22	96	0	2	32	16
23	216	0	9	8	16
24	216	1	9	8	16
25	216	0	9	8	16
26	216	0	9	8	16
27	216	1	9	8	16

Best Solution

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
24	216	1	9	8	16	

solution =

1x1 FunctionApproximation.LUTSolution with properties:

ID: 24  
Feasible: "true"

Generate a Simulink™ subsystem containing the lookup table approximation using the approximate method.

approximate(solution)



Replace the original block with the approximation.

```
replaceWithApproximate(solution)
```

You can revert the system back to its original state using the `revertToOriginal` method.

```
revertToOriginal(solution)
```

### See Also

`approximate` | `revertToOriginal`

### Topics

“Approximate Functions with a Direct Lookup Table”

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

**Introduced in R2018b**

# revertToOriginal

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Revert the block that was replaced by the approximation back to its original state

## Syntax

```
revertToOriginal(solution)
```

## Description

`revertToOriginal(solution)` reverts the block that was replaced by a lookup table approximation back to its original state.

---

**Note** You can only revert a block back to its original state within a single MATLAB session.

---

## Input Arguments

**solution** — Solution approximating the block you want to revert to its original state

FunctionApproximation.LUTSolution object

The solution approximating the block you want to revert to its original state, specified as a FunctionApproximation.LUTSolution object.

## Examples

## Replace a Block with an Approximation

This example shows how to approximate a block using a lookup table approximation, replace the original block with the approximation, and then revert the block back to its original state.

Open the model containing the block to approximate. In this example, replace the tan block with a lookup table approximation.

```
open_system('ex_luto_approx')
```



Create a `FunctionApproximation.Problem` object specifying what you want to approximate.

```
problem = FunctionApproximation.Problem('ex_luto_approx/Trigonometric Function')
```

```
problem =
```

```
1x1 FunctionApproximation.Problem with properties:
```

```
FunctionToApproximate: 'ex_luto_approx/Trigonometric Function'
NumberOfInputs: 1
InputTypes: "numerictype('double')"
InputLowerBounds: -1.5083
InputUpperBounds: 1.5083
OutputType: "numerictype('double')"
Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. To approximate the block use the `solve` method.

```
solution = solve(problem)
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints Ws	TableData WL	Break
0	48	0	2	8	16	
1	800	0	49	8	16	
2	1584	1	98	8	16	

3	640	0	39	8	16
4	416	0	25	8	16
5	1056	0	65	8	16
6	64	0	2	16	16
7	768	1	46	16	16
8	752	1	45	16	16
9	592	1	35	16	16
10	576	1	34	16	16
11	416	0	24	16	16
12	400	0	23	16	16
13	496	0	29	16	16
14	528	1	31	16	16
15	512	0	30	16	16
16	80	0	2	8	32
17	48	0	2	8	16
18	64	0	2	16	16
19	80	0	2	8	32
20	96	0	2	16	32
21	128	0	2	32	32
22	96	0	2	32	16
23	216	0	9	8	16
24	216	1	9	8	16
25	216	0	9	8	16
26	216	0	9	8	16
27	216	1	9	8	16

Best Solution

ID	Memory (bits)	Feasible	Table Size	Breakpoints Ws	TableData WL	Break
24	216	1	9	8	16	

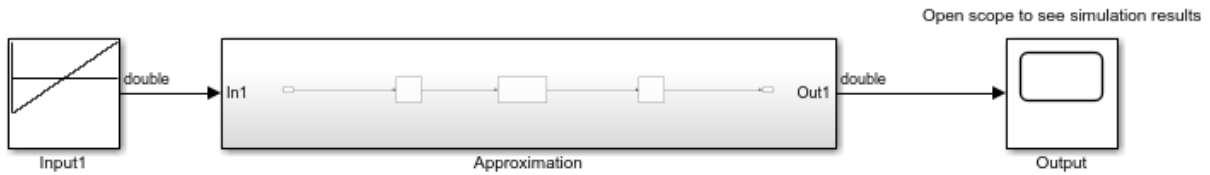
solution =

1x1 FunctionApproximation.LUTSolution with properties:

ID: 24  
Feasible: "true"

Generate a Simulink™ subsystem containing the lookup table approximation using the approximate method.

approximate(solution)



Replace the original block with the approximation.

```
replaceWithApproximate(solution)
```

You can revert the system back to its original state using the `revertToOriginal` method.

```
revertToOriginal(solution)
```

## See Also

`approximate` | `replaceWithApproximate`

## Topics

“Approximate Functions with a Direct Lookup Table”

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

**Introduced in R2018b**



# solutionfromID

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Access a solution found during the approximation process

## Syntax

```
other_solution = solutionfromID(solution,id)
```

## Description

`other_solution = solutionfromID(solution,id)` returns the solution associated with the `FunctionApproximation.LUTSolution` object, `solution`, with the ID specified by `id`.

## Input Arguments

### **solution** — Solution object

`FunctionApproximation.LUTSolution` object

The solution object containing the solution you want to explore, specified as a `FunctionApproximation.LUTSolution` object.

### **id** — ID of the solution

scalar integer

ID of the solution that you want to explore, specified as a scalar integer.

Data Types: `double`

## Output Arguments

**other\_solution** — **FunctionApproximation.LUTSolution** specified by **id**  
 FunctionApproximation.LUTSolution object

FunctionApproximation.LUTSolution object associated with the specified ID.

## Examples

### Examine Infeasible Function Approximation Solution

This example shows how to use the `solutionfromID` method of the `FunctionApproximation.LUTSolution` object to examine other approximation solutions.

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem =
```

```
  1x1 FunctionApproximation.Problem with properties:
```

```

    FunctionToApproximate: @(x)sin(x)
      NumberOfInputs: 1
        InputTypes: "numeric(0,16,13)"
    InputLowerBounds: 0
    InputUpperBounds: 6.2832
      OutputType: "numeric(1,16,14)"
      Options: [1x1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
0	64	0	2	16	16	
1	784	1	47	16	16	
2	768	1	46	16	16	
3	608	1	36	16	16	
4	592	1	35	16	16	
5	416	1	24	16	16	

6	400	1	23	16	16
7	64	0	2	16	16
8	576	0	18	16	16
9	640	1	20	16	16
10	576	0	18	16	16
11	576	0	18	16	16
12	640	1	20	16	16

Best Solution

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
6	400	1	23	16	16	

solution =

1x1 FunctionApproximation.LUTSolution with properties:

ID: 6  
Feasible: "true"

Display all feasible solutions found during the approximation process.

displayfeasiblesolutions(solution)

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
1	784	1	47	16	16	
2	768	1	46	16	16	
3	608	1	36	16	16	
4	592	1	35	16	16	
5	416	1	24	16	16	
6	400	1	23	16	16	
9	640	1	20	16	16	
12	640	1	20	16	16	

Best Solution

ID	Memory (bits)	Feasible	Table Size	Breakpoints WLS	TableData WL	Break
6	400	1	23	16	16	

Solution with ID 5 is not listed as a feasible solution in the table. Explore this solution to see why it is not feasible.

solution5 = solutionfromID(solution, 5)

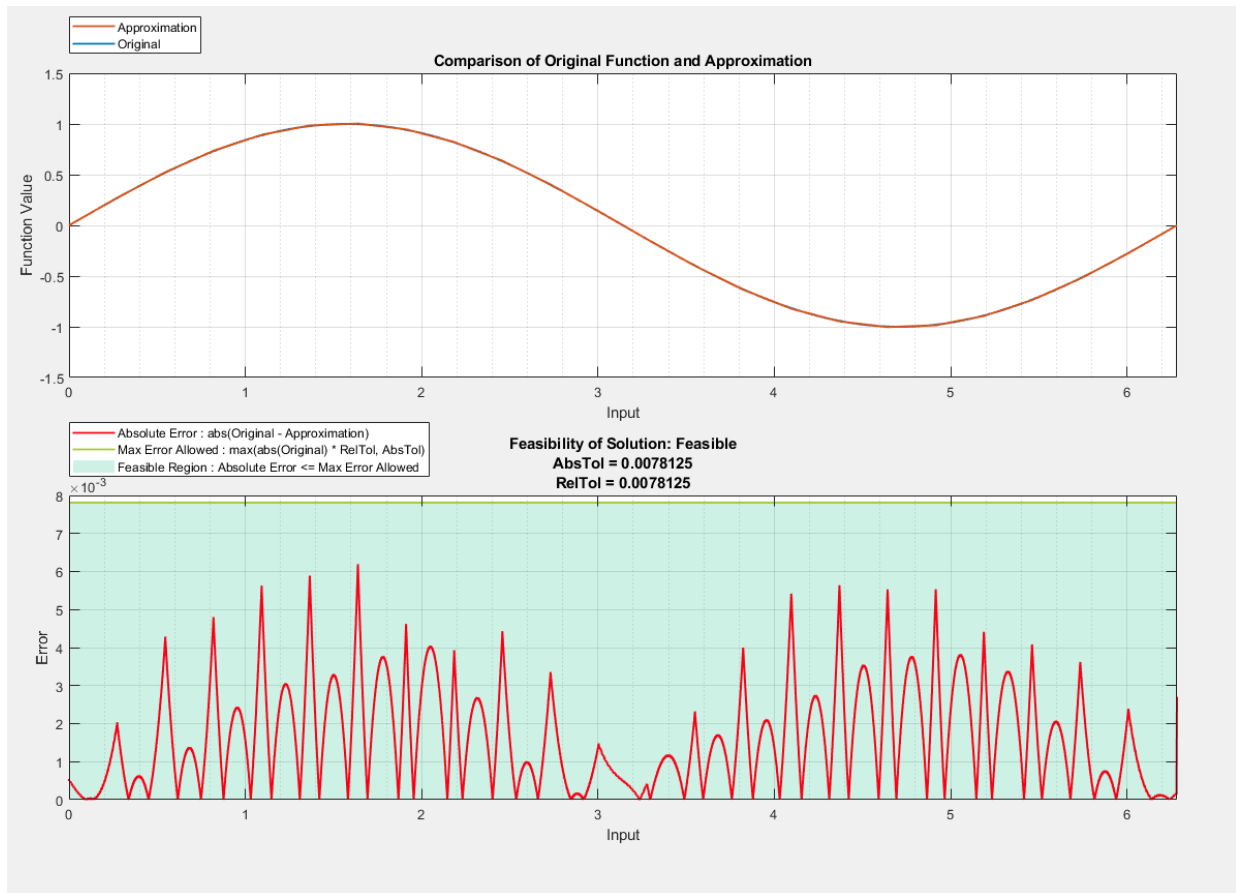
solution5 =

1x1 FunctionApproximation.LUTSolution with properties:

```
ID: 5
Feasible: "true"
```

Compare the numerical behavior of the solution with ID 5.

```
compare(solution5)
```



```
ans = struct with fields:
  Breakpoints: [51473x1 double]
  Original: [51473x1 double]
  Approximate: [51473x1 double]
```

You can see from the plot that the solution does not meet the required tolerances.

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

`FunctionApproximation.LUTMemoryUsageCalculator` |  
`FunctionApproximation.LUTSolution` | `FunctionApproximation.Options` |  
`FunctionApproximation.Problem`

### Functions

`displayallsolutions` | `displayfeasiblesolutions` | `totalmemoryusage`

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## totalmemoryusage

**Class:** FunctionApproximation.LUTSolution

**Package:** FunctionApproximation

Calculate total memory used by a lookup table approximation

### Syntax

```
memory = totalmemoryusage(solution,units)
```

### Description

`memory = totalmemoryusage(solution,units)` returns the total memory used by the lookup table approximation specified by `solution`, in the units specified by `units`.

### Input Arguments

**solution** — Solution to get memory of

FunctionApproximation.LUTSolution object

Solution to get memory of, specified as a FunctionApproximation.LUTSolution object.

**units** — Units in which to display the total memory used

'bits' (default) | 'bytes' | 'GiB' | 'KiB' | 'MiB'

Units in which to display the total memory used, specified as a character vector.

Data Types: char

### Output Arguments

**memory** — total memory used by a lookup table approximation

scalar

Total memory used by a lookup table approximation, returned as a scalar.

## Examples

### Calculate the Total Memory Used by a Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Calculate the total memory used by the `FunctionApproximation.LUTSolution` object using the `totalmemoryusage` method.

```
problem = FunctionApproximation.Problem('sin')
problem =
    FunctionApproximation.Problem with properties
        FunctionToApproximate: @(x)sin(x)
        NumberOfInputs: 1
        InputTypes: "numeric(0,16,13)"
        InputLowerBounds: 0
        InputUpperBounds: 6.2832
        OutputType: "numeric(1,16,14)"
        Options: [1x1 FunctionApproximation.Options]

solution = solve(problem)
solution =
    FunctionApproximation.LUTSolution with properties
        ID: 8
        Feasible: "true"

totalmemoryusage(solution, 'bytes')
```

ans =

58

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

FunctionApproximation.LUTMemoryUsageCalculator |  
FunctionApproximation.LUTSolution | FunctionApproximation.Options |  
FunctionApproximation.Problem

### Functions

compare | displayallsolutions | displayfeasiblesolutions | solutionfromID

## Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**



---

# solve

**Class:** `FunctionApproximation.Problem`

**Package:** `FunctionApproximation`

Solve for optimized solution to function approximation problem

## Syntax

```
solution = solve(problem)
```

## Description

`solution = solve(problem)` solves the optimization problem defined by the `FunctionApproximation.Problem` object, `problem`, and returns the optimized result, `solution`, as a `FunctionApproximation.LUTSolution` object.

## Input Arguments

**problem** — Optimization problem

`FunctionApproximation.Problem`

Optimization problem specified as a `FunctionApproximation.Problem` object defining the function or Math Function block to approximate, or the Lookup Table block to optimize, and other parameters and constraints to use during the optimization process.

## Output Arguments

**solution** — Approximation solution

`FunctionApproximation.LUTSolution` object

Approximation solution, returned as a `FunctionApproximation.LUTSolution` object.

## Examples

### Approximate a Math Function

Create a `FunctionApproximation.Problem` object, specifying a math function to approximate.

```
problem = FunctionApproximation.Problem('log')
```

```
problem =
```

```
FunctionApproximation.Problem with properties
```

```
FunctionToApproximate: @(x)log(x)
NumberOfInputs: 1
InputTypes: "numerictype(1,16,10)"
InputLowerBounds: 0.6250
InputUpperBounds: 15.6250
OutputType: "numerictype(1,16,13)"
Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options.

Use the `solve` method to generate an approximation of the function.

```
solution = solve(problem)
```

ID	Memory (bits)	ConstraintMet	Table Size	Breakpoints	WLs
0	64	0	2		16
1	1984	1	122		16
2	1024	0	62		16
3	1968	1	121		16
4	64	0	2		16
5	416	1	13		16

```
Best Solution
```

ID	Memory (bits)	ConstraintMet	Table Size	Breakpoints	WLs
5	416	1	13		16

```
solution =
```

```
FunctionApproximation.LUTSolution with properties
```

```
ID: 5  
Feasible: "true"
```

You can then use the `approximate` method to generate a subsystem containing the lookup table approximation.

## See Also

### Apps

**Lookup Table Optimizer**

### Classes

`FunctionApproximation.LUTMemoryUsageCalculator` |  
`FunctionApproximation.LUTSolution` | `FunctionApproximation.Options` |  
`FunctionApproximation.Problem`

### Functions

`approximate` | `compare`

### Topics

“Optimize Lookup Tables for Memory-Efficiency Programmatically”  
“Optimize Lookup Tables for Memory-Efficiency”

**Introduced in R2018a**

## addTolerance

**Class:** `fxpOptimizationOptions`

Specify numeric tolerance for optimized system

### Syntax

```
addTolerance(options,block_path,port_index,tolerance_type,  
tolerance_value)
```

### Description

`addTolerance(options,block_path,port_index,tolerance_type,tolerance_value)` specifies a numeric tolerance for the output signal specified by `block_path` and `port_index`, with the tolerance type specified by `tolerance_type` and value specified by `tolerance_value`.

### Input Arguments

**options** — Associated `fxpOptimizationOptions` object

`fxpOptimizationOptions`

`fxpOptimizationOptions` object to add a tolerance specification.

**block\_path** — Path to block for which to add tolerance

block path name

Path to the block to add a tolerance to, specified as a character vector.

Data Types: `char` | `string`

**port\_index** — Index of output port of block

scalar integer

Index of output port of the block specified by `block_path` for which you want to specify a tolerance, specified as a scalar integer.

Data Types: double

### **tolerance\_type — Type of tolerance to specify**

'AbsTol' | 'RelTol' | 'TimeTol'

Type of tolerance to add to the port indicated specified as either absolute tolerance, `AbsTol`, relative tolerance, `RelTol`, or time tolerance, `TimeTol`.

Data Types: char

### **tolerance\_value — Difference between the original output and the output of the new design**

scalar double

Acceptable level of tolerance for the signal specified by `block_path` and `port_index`.

If `tolerance_type` is set to `'AbsTol'`, then `tolerance_value` represents the absolute value of the maximum acceptable difference between the original output, and the output of the new design.

If `tolerance_type` is set to `'RelTol'`, then `tolerance_value` represents the maximum relative difference, specified as a percentage, between the original output, and the output of the new design. For example, a value of `1e-2` indicates a maximum difference of one percent between the original output, and the output of the new design.

If `tolerance_type` is set to `'TimeTol'`, then `tolerance_value` defines a time interval, in seconds, in which the maximum and minimum values define the upper and lower values to compare against. For more information, see “How the Simulation Data Inspector Compares Data” (Simulink).

Data Types: double

## **Examples**

### **Specify required numeric tolerance for optimized system**

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

Create a `fxpOptimizationOptions` object with default property values.

```
options = fxpOptimizationOptions;
```

To specify a required numeric tolerance to use during the optimization process, use the `addTolerance` method of the `fxpOptimizationOptions` object. To specify several tolerance constraints, call the method once per constraint. You can specify either relative, or absolute tolerance constraints.

```
addTolerance(options, 'ex_auto_gain_controller/output_signal', 1, 'AbsTol', 5e-2);
addTolerance(options, 'ex_auto_gain_controller/input_signal', 1, 'RelTol', 1e-2);
```

Use the `showTolerances` method to display all tolerance constraints added to a specified `fxpOptimizationOptions` object.

```
showTolerances(options)
```

Path	Port_Index	Tolerance_Type	Tolerance
{'ex_auto_gain_controller/output_signal'}	1	{'AbsTol'}	5e-2
{'ex_auto_gain_controller/input_signal' }	1	{'RelTol'}	1e-2

## See Also

### Classes

`OptimizationResult` | `OptimizationSolution` | `fxpOptimizationOptions`

### Functions

`addTolerance` | `explore` | `fxpopt` | `showTolerances`

### Topics

“Optimize Fixed-Point Data Types for a System”

### Introduced in R2018a

# showTolerances

**Class:** `fxpOptimizationOptions`

Show tolerances specified for a system

## Syntax

```
showTolerances(options)
```

## Description

`showTolerances(options)` displays the absolute and relative tolerances specified for a system using the `addTolerance` method of the `fxpOptimizationOptions` class. If the `options` object has no tolerances specified, the `showTolerances` method does not display anything.

## Input Arguments

**options** — Optimization options

`fxpOptimizationOptions` object

`fxpOptimizationOptions` object specifying options and tolerances to use during the data type optimization process.

## Examples

### Specify required numeric tolerance for optimized system

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

Create a `fxpOptimizationOptions` object with default property values.

```
options = fxpOptimizationOptions;
```

To specify a required numeric tolerance to use during the optimization process, use the `addTolerance` method of the `fxpOptimizationOptions` object. To specify several tolerance constraints, call the method once per constraint. You can specify either relative, or absolute tolerance constraints.

```
addTolerance(options, 'ex_auto_gain_controller/output_signal', 1, 'AbsTol', 5e-2);
addTolerance(options, 'ex_auto_gain_controller/input_signal', 1, 'RelTol', 1e-2);
```

Use the `showTolerances` method to display all tolerance constraints added to a specified `fxpOptimizationOptions` object.

```
showTolerances(options)
```

Path	Port_Index	Tolerance_Type	Tolerance
{'ex_auto_gain_controller/output_signal'}	1	{'AbsTol'}	5e-2
{'ex_auto_gain_controller/input_signal' }	1	{'RelTol'}	1e-2

## See Also

### Classes

[OptimizationResult](#) | [OptimizationSolution](#) | [fxpOptimizationOptions](#)

### Functions

[addTolerance](#) | [explore](#) | [fxpopt](#) | [showTolerances](#)

### Topics

[“Optimize Fixed-Point Data Types for a System”](#)

### Introduced in R2018a



# explore

**Class:** OptimizationResult

Explore fixed-point implementations found during optimization process

## Syntax

```
explore(result)
solution = explore(result)
solution = explore(result, n)
solution = explore(result, n, i)
```

## Description

`explore(result)` opens the Simulation Data Inspector with logging data displayed for the `OptimizationResult` object specified by `result`.

`solution = explore(result)` opens the Simulation Data Inspector and returns an `OptimizationSolution` object, `solution`.

`solution = explore(result, n)` returns the  $n^{\text{th}}$  `OptimizationSolution` object contained in `result`.

`solution = explore(result, n, i)` returns the  $n^{\text{th}}$  `OptimizationSolution` object, using the  $i^{\text{th}}$  simulation scenario contained in `result`.

## Input Arguments

**result — OptimizationResult to explore**

`OptimizationResult`

`OptimizationResult` object to explore.

If the optimization finds a feasible solution, the vector of `OptimizationSolution` objects contained in the `result` object is sorted by cost, with the lowest cost (most

optimal) solution as the first element of the vector. If the optimization does not find a feasible solution, the vector is sorted by maximum difference from the original design.

**n — Index of solution to explore**

scalar integer

Index of the solution to explore, specified as a scalar integer. For example, if the optimization found a solution, `solution = explore(result, 3)` returns the solution with the 3rd lowest cost.

Data Types: double

**i — Index of simulation scenario to explore**

scalar integer

Index of the simulation scenario to explore, specified as a scalar integer. For example, if the optimization found a solution, `solution = explore(result, 3, 2)` returns the solution with the 3rd lowest cost, using the simulation scenario with index two.

Data Types: double

## Output Arguments

**solution — OptimizationSolution containing information related to fixed-point implementation for system**

OptimizationSolution

OptimizationSolution object containing information related to the optimal fixed-point implementation for the system, including total cost of the implementation and the maximum difference between the baseline and the solution.

## See Also

**Classes**

OptimizationResult | OptimizationSolution | fxpOptimizationOptions

**Functions**

addTolerance | fxpopt | showTolerances

## **Topics**

“Optimize Fixed-Point Data Types for a System”

**Introduced in R2018a**



## Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe MathWorks products that have fixed-point support.

### **arithmetic shift**

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

*See also* binary point, binary word, bit, logical shift, most significant bit

### **bias**

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{fractionalslope} \times 2^{\text{exponent}}$$

*See also* fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

### **binary number**

Value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

*See also* bit

### **binary point**

Symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

*See also* binary number, bit, fraction, integer, radix point

**binary point-only scaling**

Scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

*See also* binary number, binary point, scaling

**binary word**

Fixed-length sequence of bits (1's and 0's). In digital hardware, numbers are stored in binary words. The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

*See also* bit, data type, word

**bit**

Smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

**ceiling (round toward)**

Rounding mode that rounds to the closest representable number in the direction of positive infinity. This is equivalent to the `ceil` mode in Fixed-Point Designer software.

*See also* convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

**contiguous binary point**

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000  
0.000  
00.00  
000.0  
0000.

*See also* data type, noncontiguous binary point, word length

- convergent rounding** Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 0.
- See also* ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)
- data type** Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.
- See also* fixed-point representation, floating-point representation, fraction length, signedness, word length
- data type override** Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.
- See also* data type, scaling
- exponent** Part of the numerical representation used to express a floating-point or fixed-point number.
1. Floating-point numbers are typically represented as
 
$$real - worldvalue = mantissa \times 2^{exponent}$$
  2. Fixed-point numbers can be represented as
 
$$real-worldvalue = (slope \times storedinteger) + bias$$
 where the slope can be expressed as
 
$$slope = fractionalslope \times 2^{exponent}$$
- The exponent of a fixed-point number is equal to the negative of the fraction length:

$$\text{exponent} = -1 \times \text{fractionlength}$$

*See also* bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

**fixed-point representation**

Method for representing numerical values and data types that have a set range and precision.

1. Fixed-point numbers can be represented as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{fractionalslope} \times 2^{\text{exponent}}$$

The slope and the bias together represent the scaling of the fixed-point number.

2. Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

*See also* bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

**floating-point representation**

Method for representing numerical values and data types that can have changing range and precision.

1. Floating-point numbers can be represented as

$$\text{real - worldvalue} = \text{mantissa} \times 2^{\text{exponent}}$$

2. Floating-point data types are defined by their word length.

*See also* data type, exponent, mantissa, precision, range, word length

**floor (round toward)**

Rounding mode that rounds to the closest representable number in the direction of negative infinity.



- See also* ceiling (round toward), convergent rounding, nearest (round toward), rounding, truncation, zero (round toward)
- fraction** Part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.
- See also* binary point, bit, fixed-point representation
- fraction length** Number of bits to the right of the binary point in a fixed-point representation of a number.
- See also* binary point, bit, fixed-point representation, fraction
- fractional slope** Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as
- $$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$
- where the slope can be expressed as
- $$\text{slope} = \text{fractionalslope} \times 2^{\text{exponent}}$$
- The term *slope adjustment* is sometimes used as a synonym for fractional slope.
- See also* bias, exponent, fixed-point representation, integer, slope
- full range** The broadest range available for a data type. From  $-\infty$  to  $\infty$  for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8-bit integer. Also known as representable range.
- guard bits** Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

	<i>See also</i> binary word, bit, overflow
<b>incorrect range</b>	A range that is too restrictive and does not include values that can actually occur in the model element. A range that is too broad is not considered incorrect because it will not lead to overflow.  <i>See also</i> range analysis
<b>integer</b>	<ol style="list-style-type: none"><li>1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.</li><li>2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as <math display="block">\text{real-worldvalue} = 2^{-\text{fractionlength}} \times \text{storedinteger}</math>or <math display="block">\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}</math>where the slope can be expressed as <math display="block">\text{slope} = \text{fractionalslope} \times 2^{\text{exponent}}</math> <i>See also</i> bias, fixed-point representation, fractional slope, integer, real-world value, slope</li></ol>
<b>integer length</b>	Number of bits to the left of the binary point in a fixed-point representation of a number.  <i>See also</i> binary point, bit, fixed-point representation, fraction length, integer
<b>least significant bit (LSB)</b>	Bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word. The weight of the LSB is related to the fraction length according to

$$\text{weightofLSB} = 2^{-\text{fractionlength}}$$

*See also* big-endian, binary word, bit, most significant bit

**logical shift**

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

*See also* arithmetic shift, binary point, binary word, bit, most significant bit

**mantissa**

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

$$\text{real - worldvalue} = \text{mantissa} \times 2^{\text{exponent}}$$

*See also* exponent, floating-point representation

**model element**

Entities in a model that range analysis software tracks, for example, blocks, signals, parameters, block internal data (such as accumulators, products).

*See also* range analysis

**most significant bit (MSB)**

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

*See also* binary word, bit, least significant bit

**nearest (round toward)**

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Designer software.

*See also* ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

**noncontiguous binary point**

0000\_\_.

$2^5 2^4 2^3 2^2$  \_\_.

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

thereby giving the bits of the word the following potential values:

*See also* binary point, data type, word length

**one's complement representation**

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

*See also* binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

**overflow**

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

*See also* saturation, wrapping

**padding**

Extending the least significant bit of a binary word with one or more zeros.

*See also* least significant bit

**precision**

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional

bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

*See also* data type, fraction, least significant bit, quantization, quantization error, range, slope

### **Q format**

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

$Qm.n$

where

- $Q$  indicates that the number is in Q format.
- $m$  is the number of bits used to designate the two's complement integer part of the number.
- $n$  is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

*See also* binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

### **quantization**

Representation of a value by a data type that has too few bits to represent it exactly.

*See also* bit, data type, quantization error

### **quantization error**

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

	<i>See also</i> bit, data type, quantization
<b>radix point</b>	Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.  <i>See also</i> binary point, bit, fraction, integer, sign bit
<b>range</b>	Span of numbers that a certain data type can represent.  <i>See also</i> data type, full range, precision, representable range
<b>range analysis</b>	Static analysis of model to derive minimum and maximum range values for elements in the model. The software statically analyzes the ranges of the individual computations in the model based on specified design ranges, inputs, and the semantics of the calculation.
<b>real-world value</b>	Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as  $real - worldvalue = 2^{-fractionlength} \times storedinteger$ or $real-worldvalue = (slope \times storedinteger) + bias$ where the slope can be expressed as $slope = fractionalslope \times 2^{exponent}$  <i>See also</i> integer
<b>representable range</b>	The broadest range available for a data type. From $-\infty$ to $\infty$ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8-bit integer. Also known as full range.

---

<b>resolution</b>	<i>See</i> <b>precision</b>
<b>rounding</b>	<p>Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.</p> <p><i>See also</i> bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)</p>
<b>saturation</b>	<p>Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.</p> <p><i>See also</i> overflow, wrapping</p>
<b>scaled double</b>	<p>A double data type that retains fixed-point scaling information. For example, in Simulink and Fixed-Point Designer software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.</p>
<b>scaling</b>	<ol style="list-style-type: none"><li>1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.</li><li>2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.</li></ol> <p><i>See also</i> bias, fixed-point representation, integer, slope</p>
<b>shift</b>	<p>Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be</p>

either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

*See also* arithmetic shift, logical shift, sign extension

**sign bit**

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

*See also* binary number, bit

**sign extension**

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

*See also* binary number, guard bits, most significant bit, two's complement representation, word

**sign/magnitude representation**

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

*See also* binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, signedness, two's complement representation

**signed fixed-point**

Fixed-point number or data type that can represent both positive and negative numbers.

*See also* data type, fixed-point representation, signedness, unsigned fixed-point

**signedness**

The signedness of a number or data type can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned



numbers and data types can only represent values that are greater than or equal to zero.

*See also* data type, sign bit, sign/magnitude representation, signed fixed-point, unsigned fixed-point

### **slope**

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{fractionalslope} \times 2^{\text{exponent}}$$

*See also* bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

### **slope adjustment**

*See* **fractional slope**

### **[Slope Bias]**

Representation used to define the scaling of a fixed-point number.

*See also* bias, scaling, slope

### **stored integer**

*See* **integer**

### **trivial scaling**

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$\text{real - worldvalue} = \text{storedinteger}$$

In [Slope Bias] representation, fixed-point numbers can be represented as

$$\text{real-worldvalue} = (\text{slope} \times \text{storedinteger}) + \text{bias}$$

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real - worldvalue = storedinteger \times 2^{-fractionlength} = storedinteger \times 2^0$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

*See also* bias, binary point, binary point-only scaling, fixed-point representation, fraction length, integer, least significant bit, scaling, slope, [Slope Bias]

**truncation**

Rounding mode that drops one or more least significant bits from a number.

*See also* ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, zero (round toward)

**two's complement representation**

Common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

*See also* binary word, one's complement representation, sign/magnitude representation, signed fixed-point

**unsigned fixed-point**

Fixed-point number or data type that can only represent numbers greater than or equal to zero.

*See also* data type, fixed-point representation, signed fixed-point, signedness

**word**

Fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

*See also* binary word, data type

**word length**

Number of bits in a binary word or data type.

*See also* binary word, bit, data type

**wrapping**

Method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range the data type being used back into the representable range.

*See also* data type, overflow, range, saturation

**zero (round toward)**

Rounding mode that rounds to the closest representable number in the direction of zero. This is equivalent to the `fix` mode in Fixed-Point Designer software.

*See also* ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation



# Selected Bibliography

- [1] Burrus, C.S., J.H. McClellan, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [2] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems, Second Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
- [3] *Handbook For Digital Signal Processing*, edited by S.K. Mitra and J.F. Kaiser, John Wiley & Sons, Inc., New York, 1993.
- [4] Hanselmann, H., "Implementation of Digital Controllers — A Survey," *Automatica*, Vol. 23, No. 1, pp. 7-32, 1987.
- [5] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*, Kluwer Academic Publishers, Seventh Printing, Norwell, Massachusetts, 1993.
- [6] Middleton, R. and G. Goodwin, *Digital Control and Estimation — A Unified Approach*, Prentice Hall, Englewood Cliffs, New Jersey. 1990.
- [7] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at [https://www.mathworks.com/company/newsletters/news\\_notes/clevescorner/index.html](https://www.mathworks.com/company/newsletters/news_notes/clevescorner/index.html).
- [8] Ogata, K., *Discrete-Time Control Systems, Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [9] Roberts, R.A. and C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

