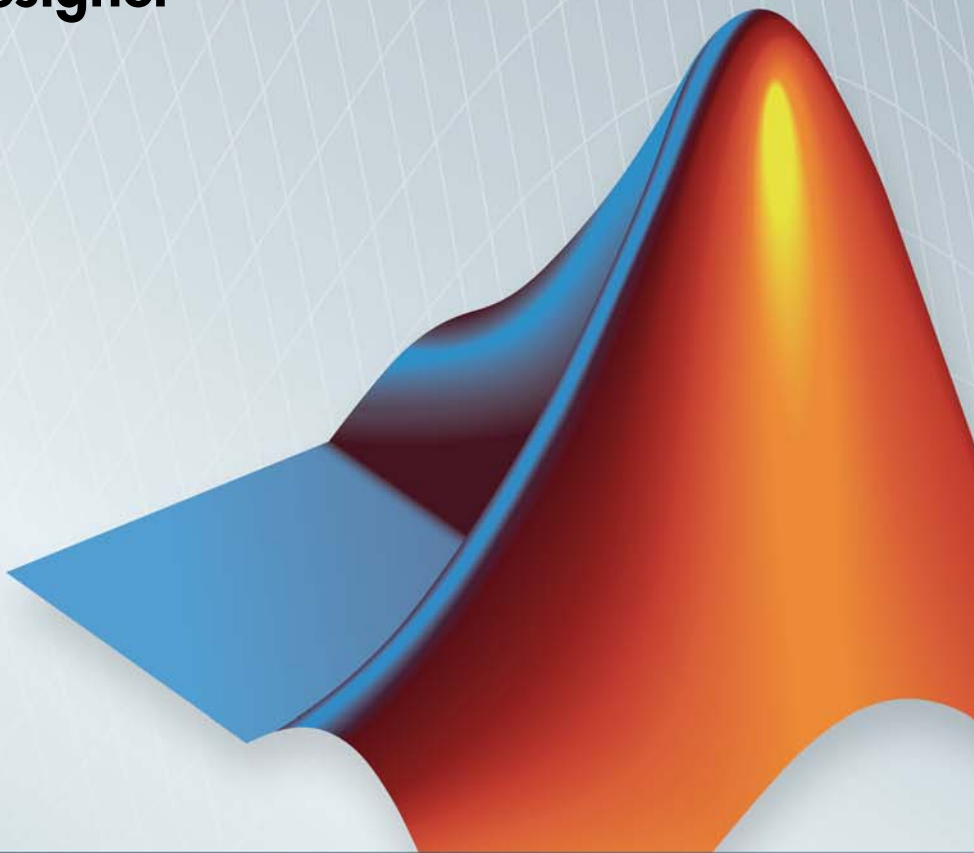


Fixed-Point Designer™

Reference

R2014a



MATLAB®



How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Fixed-Point Designer™ Reference

© COPYRIGHT 2013–2014 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (R2013b)
March 2014	Online only	Revised for Version 4.2 (R2014a)

Property Reference

1

fi Object Properties	1-2
bin	1-2
data	1-2
dec	1-2
double	1-2
fimath	1-2
hex	1-3
int	1-3
NumericType	1-3
oct	1-3
fipref Object Properties	1-4
DataTypeOverride	1-4
DataTypeOverrideAppliesTo	1-4
FimathDisplay	1-4
LoggingMode	1-5
NumericTypeDisplay	1-5
NumberDisplay	1-6
quantizer Object Properties	1-7
DataMode	1-7
Format	1-7
OverflowAction	1-8
RoundingMethod	1-9

Functions — Alphabetical List

2

Glossary

Selected Bibliography

A

Property Reference

- “fi Object Properties” on page 1-2
- “fipref Object Properties” on page 1-4
- “quantizer Object Properties” on page 1-7

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.

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
- `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

$$\frac{1}{(2^{\text{fixed-point exponent}})} \times \text{stored integer}$$

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.

DataMode Property Value	Interpreting the Format Property Values
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>

OverflowAction

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

- 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`.

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

The `OverflowMode` property value is set to `saturate` and becomes a read-only property when you set the value of the `DataMode` property to `float`, `double`, or `single`.

RoundingMethod

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

- `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`.

Functions — Alphabetical List

abs

Purpose 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 2-3.

`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 2-3.

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 `OverflowMode` property is set to `saturate`. If `OverflowMode` 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 `OverflowMode` 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.OverflowMode = '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 `OverflowMode` 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.numerictype, fimath('OverflowMode','wrap'))

ans =
    1.0000

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

abs(re,re.numerictype, fimath('OverflowMode','wrap'))

ans =
    -1

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

FractionLength: 15

Example 3

The following example shows how to specify `numericType` 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,'OverflowMode','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('OverflowMode','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,'OverflowMode','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('OverflowMode','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:

$$y = a \text{ if } a \geq 0$$

$$y = -a \text{ if } a < 0$$

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:

$$\text{re} = \text{real}(a)$$

$$\text{im} = \text{imag}(a)$$

- 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 `OverflowMode` property is set to `wrap` then an error will occur when taking the square root in step five.

Purpose	Subtract two <code>fi</code> objects or values
Syntax	<pre>c = accumneg(a,b) c = accumneg(a,b,RoundingMethod) c = accumneg(a,b,RoundingMethod,OverflowAction)</pre>
Description	<p><code>c = accumneg(a,b)</code> subtracts <code>b</code> from <code>a</code> using <code>a</code>'s data type. <code>b</code> is cast into <code>a</code>'s data type. If <code>a</code> is a <code>fi</code> object, the default 'Floor' rounding method and default 'Wrap' overflow action are used. The <code>fimath</code> properties of <code>a</code> and <code>b</code> are ignored.</p> <p><code>c = accumneg(a,b,RoundingMethod)</code> uses the rounding method specified in <code>RoundingMethod</code>.</p> <p><code>c = accumneg(a,b,RoundingMethod,OverflowAction)</code> uses the overflow action specified in <code>OverflowAction</code>.</p>
Input Arguments	<p>a</p> <p>Number from which to subtract. <code>a</code> can be <code>fi</code> object or double, single, logical, or integer value. The data type of <code>a</code> is used to compute the output data type.</p> <p>b</p> <p>Number to subtract. <code>b</code> can be <code>fi</code> object or double, single, logical, or integer value. .</p> <p>RoundingMethod</p> <p>Rounding method to use if <code>a</code> is a <code>fi</code> object. Valid values are 'Ceiling', 'Convergent', 'Floor', 'Nearest', 'Round' and 'Zero'.</p> <p>Default: Floor</p> <p>OverflowAction</p> <p>Overflow action to take if <code>a</code> is a <code>fi</code> object. Valid values are 'Saturate' and 'Wrap',</p>

accumneg

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');
```

See Also

`accumpos`

Related Examples

- “Avoid Multiword Operations in Generated Code”

Purpose

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 `fimath` 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');
```

See Also

`accumneg`

Related Examples

- “Avoid Multiword Operations in Generated Code”

Purpose

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

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 =
```

add

5.8599

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

RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
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.

See Also

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

Purpose Determine whether all array elements are nonzero

Description Refer to the MATLAB `all` reference page for more information.

and

Purpose Find logical AND of array or scalar inputs

Description Refer to the MATLAB and reference page for more information.

Purpose Determine whether any array elements are nonzero

Description Refer to the MATLAB `any` reference page for more information.

area

Purpose Create filled area 2-D plot

Description Refer to the MATLAB area reference page for more information.

Purpose 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`

atan2

Purpose Four-quadrant inverse tangent of fixed-point values

Syntax $z = \text{atan2}(y, x)$

Description $z = \text{atan2}(y, x)$ returns the four-quadrant arctangent 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.

Definitions **Four-Quadrant Arctangent**

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

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

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
```

atan2

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

```
z =
```

```
-3.0309
```

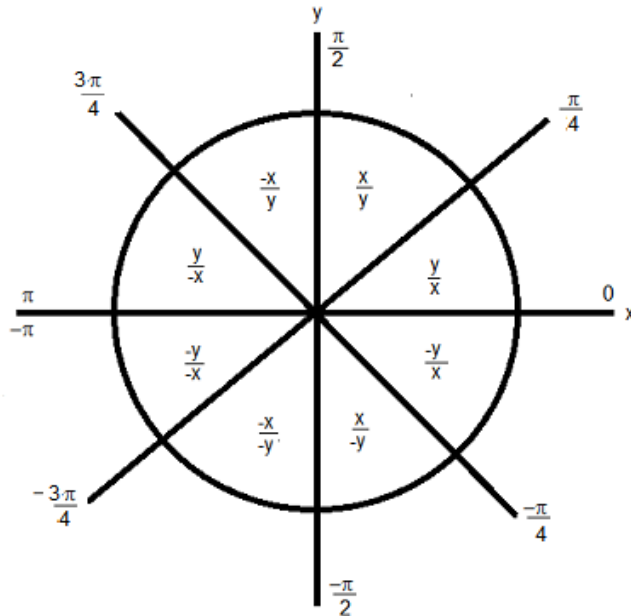
```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 16  
FractionLength: 13
```

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.

atan2

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`.

See Also

`atan2` | `sin` | `angle` | `cos` | `cordicatan2`

Related Examples

- Demo: Fixed-Point Arctangent Calculation

Purpose	Automatically change scaling of fixed-point data types
Syntax	autofixexp
Description	<p>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.</p> <p>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.</p> <p>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” 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.</p> <p>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.</p>

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.

For more information, see “Fixed-Point Tool”.

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

See Also

`fxptdlg`

Purpose Create vertical bar graph

Description Refer to the MATLAB bar reference page for more information.

barh

Purpose Create horizontal bar graph

Description Refer to the MATLAB barh reference page for more information.

Purpose 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 string. `bin(a)` is equivalent to `a.bin`.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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 = bin(a)
z = a.bin
```

returns

y =

```
10000000  01111111
```

z =

```
10000000  01111111
```

See Also `dec` | `hex` | `storedInteger` | `oct`

bin2num

Purpose 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 binary string `b` to numeric array `y`. When `b` is a cell array containing binary strings, `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 strings in a column.

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

```
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](#)

bitand

Purpose 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.

See Also `bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor`

Purpose 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 `l`.

`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.WordLength >= lidx >= ridx >= 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

bitandreduce

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.

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 =

```

```

    0    0    0    1

```

```

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

```

The only entry in output `c` with a value of 1 is the 4th entry. This is because only entry 4 of `a` 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.

bitandreduce

```
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 =
```

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

See Also

[bitconcat](#) | [bitorreduce](#) | [bitsliceget](#) | [bitxorreduce](#)

Purpose	Bitwise complement of <code>fi</code> object
Syntax	<code>c = bitcmp(a)</code>
Description	<p><code>c = bitcmp(a)</code> returns the bitwise complement of <code>fi</code> object <code>a</code>. If <code>a</code> has a signed <code>numericType</code>, the bit representation of the stored integer is in two's complement representation.</p> <p><code>bitcmp</code> only supports <code>fi</code> objects with fixed-point data types. <code>a</code> can be a scalar <code>fi</code> object or a vector <code>fi</code> object.</p>
Examples	<p>This example shows how to get the bitwise complement of a <code>fi</code> object. Consider the following unsigned fixed-point <code>fi</code> object with a value of 10, word length 4, and fraction length 0:</p> <pre>a = fi(10,0,4,0); disp(bin(a))</pre> <p>1010</p> <p>Complement the values of the bits in <code>a</code>:</p> <pre>c = bitcmp(a); disp(bin(c))</pre> <p>0101</p>
See Also	<code>bitand</code> <code>bitget</code> <code>bitor</code> <code>bitset</code> <code>bitxor</code>

bitconcat

Purpose 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.

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. When `a` is the only input, `length(a)` must be greater than 1. `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.

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 `WordLength` of 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 `WordLength` 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 =
```

```
    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 WordLength 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);
```

bitconcat

```
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 =
```

```
    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 WordLength equal to the sum of the word lengths of a and b.

See Also

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

bitget

Purpose 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.

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`.

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);
```

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

MATLAB 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 =  
  
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 `WordLength` 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
```

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

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

```
bit =
```

```
     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 =
```

```
     0     0     1     0     0
```

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

MATLAB 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 `WordLength` 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 =
```

```
    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`.

See Also

[bitand](#) | [bitcmp](#) | [bitor](#) | [bitset](#) | [bitxor](#)

Purpose 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 `fi`math, their local `fi`math 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`.

bitor

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

```
binary_a =
```

```
100010
```

```
binary_b =
```

```
001100
```

```
binary_c =
```

```
101110
```

See Also

[bitand](#) | [bitcmp](#) | [bitget](#) | [bitset](#) | [bitxor](#)

Purpose	Reduce consecutive slice of bits to one bit by performing bitwise OR operation
Syntax	<pre>c = bitorreduce(a) c = bitorreduce(a, lidx) c = bitorreduce(a, lidx, ridx)</pre>
Description	<p><code>c = bitorreduce(a)</code> performs a bitwise OR operation on the entire set of bits in the fixed-point input, <code>a</code>, and returns the result as an unsigned integer of word length 1.</p> <p><code>c = bitorreduce(a, lidx)</code> performs a bitwise OR operation on a consecutive range of bits, starting at position <code>lidx</code> and ending at the LSB (the bit at position 1).</p> <p><code>c = bitorreduce(a, lidx, ridx)</code> performs a bitwise OR operation on a consecutive range of bits, starting at position <code>lidx</code> and ending at position <code>ridx</code>.</p> <p>The <code>bitorreduce</code> arguments must satisfy the following condition:</p> $a.\text{WordLength} \geq lidx \geq ridx \geq 1$
Input Arguments	<p>a - Input array scalar vector matrix multidimensional array</p> <p>Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point <code>fi</code> objects.</p> <p><code>bitorreduce</code> supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. <code>bitorreduce</code> performs the operation on a two's complement bit representation of the stored integer.</p> <p>Data Types: fixed-point <code>fi</code></p> <p>lidx - Start position of range</p>

bitorreduce

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.

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 =
    
```

```

     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.

bitorreduce

```
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 =
```

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

See Also

[bitandreduce](#) | [bitconcat](#) | [bitsliceget](#) | [bitxorreduce](#)

Purpose 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
```

bitreplicate

See Also

bitand | bitconcat | bitget | bitset | bitor | bitsliceget |
bitxor

Purpose	Bitwise rotate left
Syntax	<code>c = bitrol(a, k)</code>
Description	<p><code>c = bitrol(a, k)</code> returns the value of the fixed-point <code>fi</code> object, <code>a</code>, rotated left by <code>k</code> bits. <code>bitrol</code> 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 <code>a</code>.</p> <p><code>bitrol</code> does not check overflow or underflow. It ignores <code>fimath</code> properties such as <code>RoundingMode</code> and <code>OverflowAction</code>.</p> <p><code>a</code> and <code>c</code> have the same <code>fimath</code> and <code>numericType</code> properties.</p>
Input Arguments	<p>a - Data that you want to rotate scalar vector matrix multidimensional array</p> <p>Data that you want to rotate, specified as a scalar, vector, matrix, or multidimensional array of <code>fi</code> objects. <code>a</code> can be signed or unsigned.</p> <p>Data Types: fixed-point <code>fi</code></p> <p>Complex Number Support: Yes</p> <p>k - Number of bits to rotate non-negative, integer-valued scalar</p> <p>Number of bits to rotate, specified as a non-negative integer-valued scalar <code>fi</code> object or built-in numeric type. <code>k</code> can be greater than the word length of <code>a</code>. This value is always normalized to <code>mod(a.WordLength,k)</code>.</p> <p>Data Types: <code>fi</code> <code>single</code> <code>double</code> <code>int8</code> <code>int16</code> <code>int32</code> <code>int64</code> <code>uint8</code> <code>uint16</code> <code>uint32</code> <code>uint64</code></p>
Examples	<p>Rotate the Bits of a <code>fi</code> Object Left</p> <p>Create an unsigned fixed-point <code>fi</code> object with a value of 10, word length 4, and fraction length 0.</p> <pre>a = fi(10,0,4,0);</pre>

```
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)
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 5, 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
```

See Also

[bitconcat](#) | [bitror](#) | [bitshift](#) | [bitsliceget](#) | [bitsll](#) | [bitsra](#) | [bitsrl](#)

bitror

Purpose 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 `fimath` properties such as `RoundingMode` and `OverflowAction`.

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

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`

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
```

bitror

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

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

```
1010
```

See Also

[bitrol](#) | [bitconcat](#) | [bitshift](#) | [bitsliceget](#) | [bitsll](#) | [bitsra](#) | [bitsrl](#)

Purpose 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`.

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
```

bitset

	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.

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
```

See Also

[bitand](#) | [bitcmp](#) | [bitget](#) | [bitor](#) | [bitxor](#)

Purpose

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 `fi` 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` `fi` 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
```

bitshift

```
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

See Also

bitand | bitcmp | bitget | bitor | bitset | bitsll | bitsra |
bitsrl | bitxor | pow2

bitsliceget

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

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)`.

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
```

See Also

[bitand](#) | [bitcmp](#) | [bitget](#) | [bitor](#) | [bitset](#) | [bitxor](#)

bitshiftl

Purpose Bit shift left logical

Syntax `c = bitshiftl(a, k)`

Description `c = bitshiftl(a, k)` returns the result of a logical left shift by k bits on input `a` for fixed-point operations. `bitshiftl` shifts zeros into the positions of bits that it shifts left. The function does not check overflow or underflow. For floating-point operations, `bitshiftl` performs a multiply by 2^k .

`bitshiftl` ignores `fi` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fi` and `numericType` objects.

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`

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 =
```

```
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 =
```

```
16
```

Shift Left a Floating-Point Input

Scale a floating-point double input by 2^3 .

```
a = double(16);  
bitsll(a,3)
```

```
ans =  
    128
```

See Also

[bitsrl](#) | [bitsra](#) | [bitshift](#) | [pow2](#) | [bitconcat](#) | [bitrol](#) | [bitror](#)

Purpose	Bit shift right arithmetic
Syntax	<code>c=bitsra(a,k)</code>
Description	<p><code>c=bitsra(a,k)</code> returns the result of an arithmetic right shift by k bits on input <code>a</code> for fixed-point operations. For floating-point operations, it performs a multiply by 2^{-k}.</p> <p>If the input is unsigned, <code>bitsra</code> shifts zeros into the positions of bits that it shifts right. If the input is signed, <code>bitsra</code> shifts the most significant bit (MSB) into the positions of bits that it shifts right.</p> <p><code>bitsra</code> ignores <code>fimath</code> properties such as <code>RoundingMode</code> and <code>OverflowAction</code>.</p> <p>When <code>a</code> is a <code>fi</code> object, <code>a</code> and <code>c</code> have the same associated <code>fimath</code> and <code>numerictype</code> objects.</p>
Input Arguments	<p><code>a</code> - Data that you want to shift scalar vector matrix multidimensional array</p> <p>Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array of <code>fi</code> objects or built-in numeric types.</p> <p>Data Types: <code>fi</code> <code>single</code> <code>double</code> <code>int8</code> <code>int16</code> <code>int32</code> <code>int64</code> <code>uint8</code> <code>uint16</code> <code>uint32</code> <code>uint64</code></p> <p>Complex Number Support: Yes</p> <p><code>k</code> - Number of bits to shift non-negative integer-valued scalar</p> <p>Number of bits to shift, specified as a non-negative integer-valued scalar <code>fi</code> object or built-in numeric type.</p> <p>Data Types: <code>fi</code> <code>single</code> <code>double</code> <code>int8</code> <code>int16</code> <code>int32</code> <code>int64</code> <code>uint8</code> <code>uint16</code> <code>uint32</code> <code>uint64</code></p>

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 =  
    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 =  
    16
```


Shift Right a Floating-Point Input

Scale a floating-point double input by 2^{-3} .

```
a = double(128);  
bitsra(a,3)
```

```
ans =  
    16
```

See Also

[bitsll](#) | [bitsrl](#) | [bitshift](#) | [pow2](#)

bitrsl

Purpose Bit shift right logical

Syntax `c = bitrsl(a, k)`

Description `c = bitrsl(a, k)` returns the result of a logical right shift by `k` bits on input `a` for fixed-point operations. `bitrsl` shifts zeros into the positions of bits that it shifts right. It does not check overflow or underflow.

`bitrsl` 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.

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`

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(bitshr(a,1)))
```

```
0100
```

`bitshr` 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);  
bitshr(a,k)
```

```
ans =
```

```
16
```

Shift right a built-in uint8 input

Use `bitshr` to shift an `uint8` input right by 2 bits.

```
a = uint8(64);  
bitshr(a,2)
```

```
ans =
```

```
16
```

See Also

[bitconcat](#) | [bitrotl](#) | [bitror](#) | [bitshift](#) | [bitsliceget](#) | [bitshll](#) | [bitsra](#) | [pow2](#)

bitxor

Purpose 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 `fi`math, their local `fi`math 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
```

See Also

```
bitand | bitcmp | bitget | bitor | bitset
```

bitxorreduce

Purpose 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 `l`.

`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 `l`).

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

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.

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 =
```

bitxorreduce

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 =
```

```
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 =
```

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

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

See Also

[bitandreduce](#) | [bitconcat](#) | [bitorreduce](#) | [bitsliceget](#)

buffer

Purpose Buffer signal vector into matrix of data frames

Description Refer to the DSP System Toolbox™ `buffer` function reference page for more information.

Purpose Generate compiled C code function including logging instrumentation

Syntax `buildInstrumentedMex fcn -options`

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.

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
```

buildInstrumentedMex

```
        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
```

Input Arguments

fcn

MATLAB function to be instrumented. *fcn* must be suitable for code generation. For more information, see “Making the MATLAB Code Suitable for Code Generation”.

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 `fiaccl` function. This option removes `fiaccl` 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:

# buildInstrumentedMex

---

*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.

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.

`-histogram`

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.

# buildInstrumentedMex

---

`-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. 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 Codercodegen` function.

---

```
buildInstrumentedMex testfft -o testfft_instrumented...
-args {x,W} -histogram
```

- 4 Run a test file to record instrumentation results. Call `showInstrumentationResults` to open the Code Generation Report. View the simulation minimum and maximum values and whole number status by hovering 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)));
```


# buildInstrumentedMex

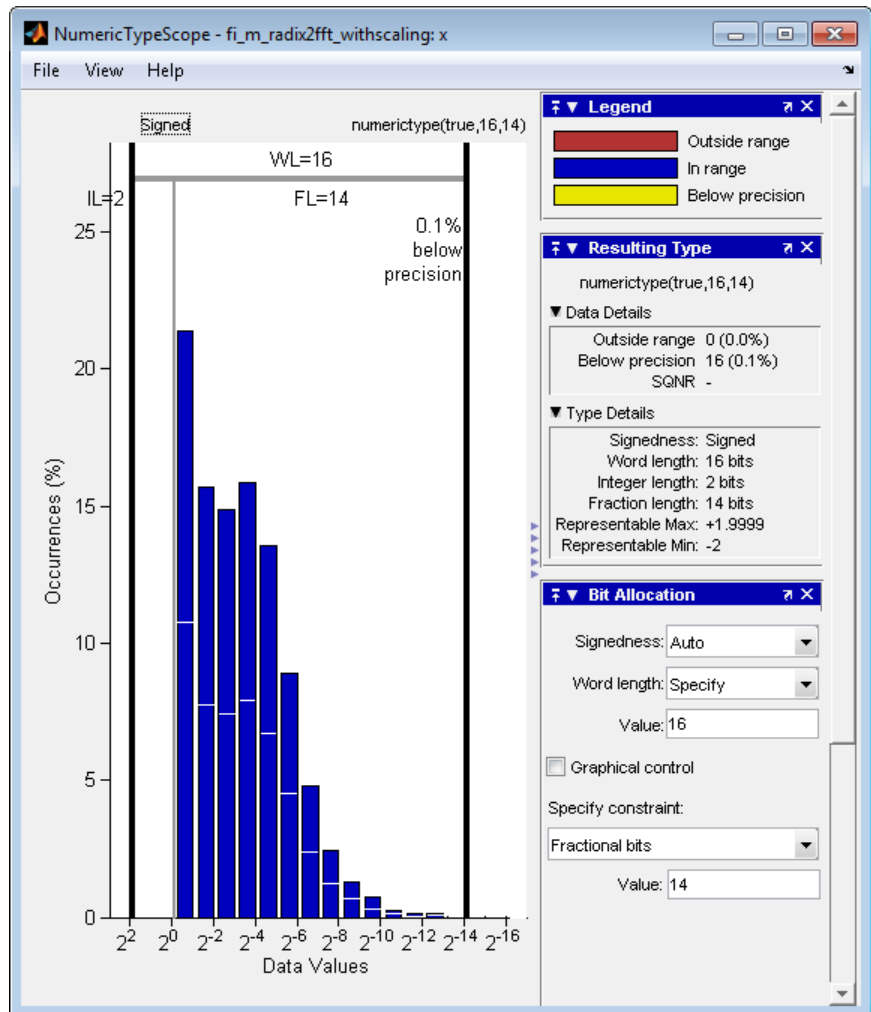
```
end
```

```
showInstrumentationResults testfft_instrumented
```

The screenshot shows the Code Generation Report window for the function `fi_m_radix2fft_withscaling`. The MATLAB code is displayed in the editor, and the Variable tab is selected in the bottom panel. A tooltip for the variable `L2` is visible, showing its properties: Size 1 x 1, Class int32, Complex No, Always Yes, Whole Number, SimMin 1, and SimMax 64. The Variable table below lists the variables and their properties.

| Order | Variable | Type  | Size    | Class  | Complex | Signedness | WL | FL | Always Whole Number | SimMin             | SimMax             | Histogram |
|-------|----------|-------|---------|--------|---------|------------|----|----|---------------------|--------------------|--------------------|-----------|
| 1     | x        | I/O   | 128 x 1 | double | Yes     | -          | -  | -  | No                  | -3.232037795940007 | 3.5783969397257605 |           |
| 2     | w        | Input | 127 x 1 | double | Yes     | -          | -  | -  | No                  | -1                 | 1                  |           |
| 3     | n        | Local | 1 x 1   | double | No      | -          | -  | -  | Yes                 | 128                | 128                |           |
| 4     | t        | Local | 1 x 1   | double | No      | -          | -  | -  | Yes                 | 7                  | 7                  |           |
| 5     | LL       | Local | 1 x 7   | int32  | No      | -          | -  | -  | Yes                 | 2                  | 128                |           |
| 6     | rr       | Local | 1 x 7   | int32  | No      | -          | -  | -  | Yes                 | 1                  | 64                 |           |
| 7     | LL2      | Local | 1 x 7   | int32  | No      | -          | -  | -  | Yes                 | 1                  | 64                 |           |
| 8     | q        | Local | 1 x 1   | double | No      | -          | -  | -  | Yes                 | 1                  | 7                  |           |
| 9     | L        | Local | 1 x 1   | int32  | No      | -          | -  | -  | Yes                 | 2                  | 128                |           |
| 10    | r        | Local | 1 x 1   | int32  | No      | -          | -  | -  | Yes                 | 1                  | 64                 |           |
| 11    | L2       | Local | 1 x 1   | int32  | No      | -          | -  | -  | 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.

# buildInstrumentedMex

---

```
clearInstrumentationResults testfft_instrumented;
```

**7** Clear the MEX function, then delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanUp;
```

## See Also

[fiaccel](#) | [clearInstrumentationResults](#) |  
[showInstrumentationResults](#) | [NumericTypeScope](#) | [codegen](#) | [mex](#)

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Cast variable to different data type                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>          | <code>b = cast(a, 'like', p)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b>     | <code>b = cast(a, 'like', p)</code> converts <code>a</code> to the same numeric type, complexity (real or complex), and <code>fimath</code> as <code>p</code> . If <code>a</code> and <code>p</code> are both real, then <code>b</code> is also real. Otherwise, <code>b</code> is complex.                                                                                                                                                                                                                                                                                                                    |
| <b>Input Arguments</b> | <p><b>a - Variable that you want to cast to a different data type</b><br/><code>fi</code> object   numeric variable</p> <p>Variable, specified as a <code>fi</code> object or numeric variable.</p> <p>Complex Number Support: Yes</p> <p><b>p - Prototype</b><br/><code>fi</code> object   numeric variable</p> <p>Prototype, specified as a <code>fi</code> 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.</p> <p>Complex Number Support: Yes</p>                                  |
| <b>Tips</b>            | <p>Using the <code>b = cast(a, 'like', p)</code> syntax to specify data types separately from algorithm code allows you to:</p> <ul style="list-style-type: none"><li>• Reuse your algorithm code with different data types.</li><li>• Keep your algorithm uncluttered with data type specifications and switch statements for different data types.</li><li>• Improve readability of your algorithm code.</li><li>• Switch between fixed-point and floating-point data types to compare baselines.</li><li>• Switch between variations of fixed-point settings without changing the algorithm code.</li></ul> |

## 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 `fimath` 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
```

### Match Data Type and Complex Nature of p

Define a complex `fi` object.

```
p = fi([1+2i 3i],1,24,12);
```

Define a scalar 8-bit integer.

```
a = int8(5);
```

Convert `a` to the same data type and complexity as `p`.

```
b = cast(a, 'like', p)
```

```
b =
```

```
5.0000 + 0.0000i
```

```
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 =
```

```
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)
```

# cast

---

end

## See Also

ones | zeros | cast

## Related Examples

- “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros”

## Concepts

- “Manual Fixed-Point Conversion Workflow”
- “Manual Fixed-Point Conversion Best Practices”

**Purpose** 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 numeric type 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               |

### See Also

`convergent` | `fix` | `floor` | `nearest` | `round`

# clabel

---

**Purpose** Create contour plot elevation labels

**Description** Refer to the MATLAB `clabel` reference page for more information.

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Clear results logged by instrumented, compiled C code function                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>          | <pre>clearInstrumentationResults('mex_fcn') clearInstrumentationResults mex_fcn clearInstrumentationResults all</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>     | <p><code>clearInstrumentationResults('mex_fcn')</code> clears the results logged from calling the instrumented MEX function <code>mex_fcn</code>.</p> <p><code>clearInstrumentationResults mex_fcn</code> is alternative syntax for clearing the log.</p> <p><code>clearInstrumentationResults all</code> clears the results from all instrumented MEX functions.</p>                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Input Arguments</b> | <p><b>mex_fcn</b></p> <p>Instrumented MEX function created using <code>buildInstrumentedMex</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Examples</b>        | <p>Run a test bench to log instrumentation, then use <code>clearInstrumentationResults</code> to clear the log.</p> <ol style="list-style-type: none"><li>1 Create a temporary directory, then import an example function from Fixed-Point Designer.<br/><pre>tempdirObj=fidemo.fiTempdir('showInstrumentationResults') copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...     'fidemos','fi_m_radix2fft_withscaling.m'),...     'testfft.m','f')</pre></li><li>2 Define prototype input arguments.<br/><pre>n = 128; x = complex(fi(zeros(n,1),'DataType','ScaledDouble')); W = coder.Constant(fi(fidemo.fi_radix2twiddles(n)));</pre></li><li>3 Generate an instrumented MEX function. Use the <code>-o</code> option to specify the MEX function name.</li></ol> |

# clearInstrumentationResults

```
buildInstrumentedMex testfft -o testfft_instrumented -args {x,W}
```

- 4 Run a test bench to record instrumentation results. Call `showInstrumentationResults` to open the Code Generation Report. View the simulation minimum and maximum values and whole number status by hovering 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
```

```
21 %
22 %#codegen
23 n = length(x); t = log2(n);
24 x = fidemo.fi_bitreverse(x,n);
25
26 % Generate index variables
27 % the loop.
28 LL = int32(2.^(1:t));
29 rr = int32(n./LL);
30 LL2 = int32(LL./2);
31 for q=1:t
32 L = LL(q); r = rr(q); L2 = LL2(q);
33 for k=0:(r-1)
34 for j=0:(L2-1)
35 temp = x(k*L+j+L2+1);
36 x(k*L+j+L2+1) = x(k*L+j+1);
37 x(k*L+j+1) = temp;
38 end
39 end
40 end
```

| Information for the selected variable: |                     |
|----------------------------------------|---------------------|
| Size                                   | 128 × 1             |
| Class                                  | embedded.fi         |
| Complex                                | Yes                 |
| DT Mode                                | ScaledDouble        |
| Signedness                             | Signed              |
| WL                                     | 16                  |
| FL                                     | 15                  |
| Percent of Current Range               | 100                 |
| Always Whole Number                    | No                  |
| SimMin                                 | -0.9995378696011665 |
| SimMax                                 | 0.9995851288895032  |

computed in

- 5 Clear the results log.

```
clearInstrumentationResults testfft_instrumented
```



- 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
```

```
22 %#codegen
23 n = length(x); t = log2(n);
24 x = fidemo.fi_bitreverse(x,n);
25
26 % Generate index variables
27 % the loop.
28 LL = int32(2.^(1:t));
29 rr = int32(n./LL);
30 LL2 = int32(LL./2);
31 for q=1:t
32 L = LL(q); r = rr(q); L
33 for k=0:(r-1)
34 for j=0:(L2-1)
35 temp
36 x(k*L+j+L2+1)
37 x(k*L+j+1)
38 end
39 end
40 end
```

Information for the selected variable:

|                          |                      |
|--------------------------|----------------------|
| Size                     | 128 x 1              |
| Class                    | embedded.fi          |
| Complex                  | Yes                  |
| DT Mode                  | ScaledDouble         |
| Signedness               | Signed               |
| WL                       | 16                   |
| FL                       | 15                   |
| Percent of Current Range | 50                   |
| Always Whole Number      | No                   |
| SimMin                   | -0.49963437623669427 |
| SimMax                   | 0.4996693794649575   |

computed in

- Clear the MEX function and delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanUp;
```

# clearInstrumentationResults

---

## See Also

`fiaccel` | `showInstrumentationResults` | `buildInstrumentedMex` |  
`codegen` | `mex`

**Purpose** 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

[pcode](#) | [codegen](#)

|                       |                                                                                                                                                                                                                                                                                                                           |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>        | Represent set of MATLAB arrays                                                                                                                                                                                                                                                                                            |
| <b>Description</b>    | Specifies the set of arrays that the generated code accepts. Use only with the <code>fiaccel -args</code> option. Do not pass as an input to a generated MEX function.                                                                                                                                                    |
| <b>Construction</b>   | <code>coder.ArrayType</code> is an abstract class. You cannot create instances of it directly. You can create <code>coder.EnumType</code> , <code>coder.FiType</code> , <code>coder.PrimitiveType</code> , and <code>coder.StructType</code> objects that derive from this class.                                         |
| <b>Properties</b>     | <b>ClassName</b><br>Class of values in this set<br><b>SizeVector</b><br>The upper-bound size of arrays in this set.<br><b>VariableDims</b><br>A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is <code>true</code> , the corresponding dimension is variable size. |
| <b>Copy Semantics</b> | Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.                                                                                                                                                                                                                |
| <b>See Also</b>       | <code>coder.Type</code>   <code>coder.EnumType</code>   <code>coder.FiType</code>   <code>coder.PrimitiveType</code>   <code>coder.StructType</code>   <code>coder.newtype</code>   <code>coder.typeof</code>   <code>coder.resize</code>   <code>fiaccel</code>                                                          |

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Fold expressions into constants in generated code                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Syntax</b>          | <pre>out = coder.const(expression) [out1,...,outN] = coder.const(handle,arg1,...,argN)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b>     | <p><code>out = coder.const(expression)</code> evaluates <code>expression</code> and replaces <code>out</code> with the result of the evaluation in generated code.</p> <p><code>[out1,...,outN] = coder.const(handle,arg1,...,argN)</code> evaluates the multi-output function having <code>handle</code> <code>handle</code>. It then replaces <code>out1,...,outN</code> with the results of the evaluation in the generated code.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Input Arguments</b> | <p><b>expression - MATLAB expression or user-written function</b><br/>expression with constants   single-output function with constant arguments</p> <p>MATLAB expression or user-defined single-output function.</p> <p>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 <code>x</code> is not a compile-time constant.</p> <pre>function y=func(x)     y=coder.const(log10(x));</pre> <p>To fix the error, assign <code>x</code> to a constant in the MATLAB code. Alternatively, during code generation, you can use <code>coder.Constant</code> to define input type as follows:</p> <pre>codegen -config:lib func -args coder.Constant(10)</pre> <p><b>Example:</b> <code>2*pi, factorial(10)</code></p> <p><b>handle - Function handle</b><br/>function handle</p> <p>Handle to built-in or user-written function.</p> |

**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.

**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 generation software generates code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
 int k;
 for (k = 0; k < 10; k++) {
 y[k] = (double)((1 + k) * (1 + k)) + Shift;
 }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```



The code generation software creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
 int i0;
 static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
 49, 64, 81, 100 };

 for (i0 = 0; i0 < 10; i0++) {
 y[i0] = (double)iv0[i0] + Shift;
 }
}
```

### 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
```

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 generation software generates 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 generation software 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 generation software 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.

## Tips

- The code generation software constant-folds expressions automatically when possible. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generation software does not constant-fold expressions on its own.

# coder.Constant

---

<b>Superclasses</b>	Type
<b>Purpose</b>	Represent set containing one MATLAB value
<b>Description</b>	Use a <code>coder.Constant</code> object to define values that are constant during code generation. Use only with the <code>fiaccl -args</code> options. Do not pass as an input to a generated MEX function.
<b>Construction</b>	<code>const_type=coder.Constant(v)</code> creates a <code>coder.Constant</code> type from the value <code>v</code> .  <code>const_type=coder.newtype('constant', v)</code> creates a <code>coder.Constant</code> type from the value <code>v</code> .
	<b>Input Arguments</b>
	<b>v</b> Constant value used to construct the type.
<b>Properties</b>	<b>Value</b> The actual value of the constant.
<b>Copy Semantics</b>	Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.
<b>Examples</b>	Create a constant with value 42.  <pre>k = coder.Constant(42);</pre> <hr/> Create a new constant type for use in code generation.  <pre>k = coder.newtype('constant', 42);</pre>
<b>See Also</b>	<code>coder.Type</code>   <code>coder.newtype</code>   <code>fiaccl</code>

## Purpose

Specify structure name in generated code

## Syntax

```
coder.cstructname(structVar, 'structName')
coder.cstructname(structVar, 'structName', 'extern')
coder.cstructname(structVar, 'structName', 'extern', Name, Value)
coder.cstructname(structType, 'structName')
coder.cstructname(structType, 'structName', 'extern')
coder.cstructname(structType, 'structName', 'extern', Name, Value)
```

## Description

`coder.cstructname(structVar, 'structName')` allows you to specify the name of a structure in generated code. *structVar* is the structure variable. *structName* specifies the name to use for the structure variable *structVar* in the generated code. Use `coder.cstructname(structVar, 'structName')` in a function that is compiled using `fiaccel`. Before the first use of the structure variable in your function, you must call `coder.cstructname`.

`coder.cstructname(structVar, 'structName', 'extern')` declares an externally defined structure. It does not generate the definition of the structure type. Provide the definition in a custom include file.

`coder.cstructname(structVar, 'structName', 'extern', Name, Value)` uses additional options specified by one or more *Name, Value* pair arguments.

`coder.cstructname(structType, 'structName')` returns a `coder.StructType` with the name *structName*. When the first argument is *structType*, `coder.cstructname` is a MATLAB function. You cannot use `coder.cstructname(structType, 'structName')` in a function that is compiled using `fiaccel`. Use the returned type with the `fiaccel -args` option.

`coder.cstructname(structType, 'structName', 'extern')` returns a `coder.StructType` that uses an externally defined structure. Provide the structure definition in a custom include file.

`coder.cstructname(structType, 'structName', 'extern', Name, Value)` uses additional options specified by one or more *Name, Value* pair arguments.

## Tips

- `coder.cstructname(structVar, 'structName')` is ignored outside of code generation. Using `coder.cstructname` at the MATLAB command line, and then calling `codegen` does not assign a name to a structure in the generated code. For example, if function `foo` does not use `coder.cstructname` to assign a name to structure `S`, the following commands do not assign the name `myStruct` to the structure variable `S` in generated code.

```
coder.cstructname(S, 'myStruct');
codegen foo -args {S}
```

- To assign a structure name outside of code generation, use `coder.cstructname(structType, 'structName')`. `coder.cstructname` returns a `coder.StructType` object you can use with the `-args` option. For more information, see “Create a `coder.StructType` Object” on page 2-134.
- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot use `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.
- Use of `coder.cstructname` with global variables is not supported.
- If you use `coder.cstructname` on an array, it sets the name of the base type of the array, not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname, myStructName`.

```
% Define scalar structure with field a
myStruct = struct('a', 0);
coder.cstructname(mystruct, 'myStructName');
% Define array of structure with field a
myStructArray = repmat(myStruct, k, n);
coder.cstructname(myStructArray, 'myStructArrayName');
```



- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure. Use the `HeaderFile` input argument.
- 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. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.
- You can also use `coder.cstructname` to assign a name to a substructure, which is a structure that appears as a field of another structure. For more information, see “Assign a Name to a SubStructure” on page 2-133.

## Input Arguments

### **structName**

The name to use for the structure in the generated code.

### **structType**

`coder.StructType` object.

### **structVar**

Structure variable.

## **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 single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **'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. Hence it is not matched by CRL functions that require alignment.

Alignment must be either -1 or a power of 2 that is not greater than 128.

**Default:** -1

### **'HeaderFile'**

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 string.

## Examples

### **Apply coder.cstructname to Top-Level Inputs**

Generate code for a MATLAB function that takes structure inputs.

- 1 Write a MATLAB function, `topfun`, that assigns the name `MyStruct` to its input parameter.

```
function y = topfun(x) %#codegen
% Assign the name 'MyStruct' to the input variable in
% the generated code
coder.cstructname(x, 'MyStruct');
y = x;
end
```

- 2 Declare a structure `s` in MATLAB. `s` is the structure definition for the input variable `x`.

```
s = struct('a',42,'b',4711);
```

- 3 Generate a MEX function for `topfun`, using the `-args` option to specify that the input parameter is a structure.

```
fiaccel topfun.m -args { s }
```

codegen generates a MEX function in the default folder `codegen\mex\topfun`. In this folder, the structure definition is in `topfun_types.h`.

```
typedef struct
{
 double a;
 double b;
} MyStruct;
```

## Assign a Name to a SubStructure

Use `coder.cstructname` to assign a name to a substructure.

- 1 Define a MATLAB structure, `top`, that has another structure, `lower`, as a field.

```
% Define structure top with field lower,
% which is a structure with fields a and b
top.lower = struct('a',1,'b',1);
top.c = 1;
```

- 2 Define a function, `MyFunc`, which takes an argument, `TopVar`, as input. Mark the function for code generation using `codegen`.

```
function out = MyFunc(TopVar) %codegen
```

- 3 Inside `MyFunc`, include the following lines

```
coder.cstructname(TopVar, 'topType');
coder.cstructname(TopVar.lower, 'lowerType');
```

- 4 Generate C code for MyFunc with an argument having the same type as top. This ensures that TopVar has the same type as top.

```
codegen -config:lib MyFunc -args coder.typeof(top)
```

In the generated C code, the field variable TopVar.lower is assigned the type name lowerType. For instance, the structure declaration of the variable TopVar.lower appears in the C code as:

```
typedef struct
{
 /* Definitions of a and b appear here */
} lowerType;
```

and the structure declaration of the variable TopVar appears as:

```
typedef struct
{
 lowerType lower;
 /* Definition of c appears here */
} topType;
```

### Create a coder.StructType Object

Create a coder.StructType object and pass it as argument. .

```
S = struct('a',double(0),'b',single(0))
T = coder.typeof(S);
T = coder.cstructname(T,'mytype');
codegen -config:lib MyFile -args T
```

In this example, you create a coder.StructType object T. The object is passed as a codegen argument. However, because of the coder.cstructname statement, T is replaced with mytype in the generated C code. For instance, the declaration of T appears in the C code as:

```
typedef struct
{
```

```

 /* Field definitions appear here */
} mytype;

```

## Create a coder.StructType Object Using an Externally Defined Type

Create a C header file, `MyFile.h`, containing the definition of a structure type, `mytype`.

```

typedef struct {
 /* Field definitions */
 double a;
 float b;
} mytype;

```

Save the file in the folder, `C:\MyHeaders`.

Define a `coder.StructType` object, `T`, with the same fields as `mytype`.

```
T = coder.typeof(struct('a',double(0),'b',single(0)));
```

Using `coder.cstructname`, rename `T` as `mytype`. Specify that the definition of `mytype` is in `MyFile.h`.

```
T = coder.cstructname(T,'mytype','extern','HeaderFile','MyFile.h');
```

Generate code for MATLAB function, `MyFunc`, which takes a structure of type, `T`, as input argument. Add the folder, `C:\MyHeaders`, to the include path during code generation.

```
codegen -config:lib MyFunc -args T -I C:\MyHeaders
```

In the generated code, the structure, `T`, is assigned the name, `mytype`. The code generation software does not generate the definition of `mytype`. Instead the software includes the header file, `MyFile.h`, in the generated code.

```
fiaccel | coder.StructType
```

## How To

- “Structures”

# coder.EnumType

---

**Superclasses**    ArrayType

**Purpose**            Represent set of MATLAB enumerations

**Description**      Specifies the set of MATLAB enumerations that the generated code should accept. Use only with the `fiaccl -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 a numeration 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 in the MATLAB documentation.

## **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(Enumeration) MyColors < int32
 enumeration
 green(1),
 red(2),
```

## coder.EnumType

---

```
 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(Enumeration) 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.Type](#) | [coder.ArrayType](#) | [coder.typeof](#) | [coder.newtype](#) | [coder.resize](#) | [fiaccel](#)

### How To

- “Enumerated Data”



## Purpose

Declare extrinsic function or functions

## Syntax

```
coder.extrinsic('function_name');
coder.extrinsic('function_name_1', ... ,
'function_name_n');
coder.extrinsic('-sync:on', 'function_name');
coder.extrinsic('-sync:on', 'function_name_1', ... ,
'function_name_n');
coder.extrinsic('-sync:off', 'function_name');
coder.extrinsic('-sync:off', 'function_name_1', ... ,
'function_name_n');
```

## Arguments

*function\_name*

*function\_name\_1*, ... , *function\_name\_n*

Declares *function\_name* or *function\_name\_1* through *function\_name\_n* as extrinsic functions.

`-sync:on`

*function\_name* or *function\_name\_1* through *function\_name\_n*.

Enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function\_name* or *function\_name\_1* through *function\_name\_n*. If only a few extrinsic calls 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.

`-sync:off`

Disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function\_name* or *function\_name\_1* through *function\_name\_n*. If most extrinsic calls modify global data, but a few do not, you

can use the `-sync:off` option to turn off synchronization for the extrinsic calls that *do not* modify global data.

## Description

`coder.extrinsic` declares extrinsic functions. During simulation, the code generation software generates code for the call to an extrinsic function, but does not generate the function's internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

You cannot use `coder.ceval` on functions that you declare extrinsic by using `coder.extrinsic`.

`coder.extrinsic` is ignored outside of code generation.

## Tips

- The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions, but you do not have to declare them extrinsic using the `coder.extrinsic` function.
- Use the `coder.screener` function to detect which functions you must declare extrinsic. This function opens the code generations readiness tool that detects code generation issues in your MATLAB code.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called—for example, by returning `mxArrays` to an output variable. Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, a MATLAB issues a compiler error.

## Examples

The following code declares the MATLAB functions `patch` and `axis` extrinsic in the MATLAB local function `create_plot`:

```
function c = pythagoras(a,b,color) %#codegen
```

```

% Calculates the hypotenuse of a right triangle
% and displays the triangle as a patch object.

c = sqrt(a^2 + b^2);

create_plot(a, b, color);

function create_plot(a, b, color)
%Declare patch and axis as extrinsic

coder.extrinsic('patch', 'axis');

x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');

```

By declaring these functions extrinsic, you instruct the software not to compile or generate code for `patch` and `axis`. Instead it dispatches these functions to MATLAB for execution.

**See Also**

`coder.screener`

**How To**

- “Resolution of Function Calls for Code Generation”
- “Restrictions on Extrinsic Functions for Code Generation”

# coder.FiType

---

<b>Superclasses</b>	ArrayType
<b>Purpose</b>	Represent set of MATLAB fixed-point arrays
<b>Description</b>	Specifies the set of fixed-point array values that the generated code should accept. Use only with the <code>fiaccel</code> -args options. Do not pass as an input to the generated MEX function.
<b>Construction</b>	<p><code>t=coder.typeof(v)</code> creates a <code>coder.FiType</code> object representing a set of fixed-point values whose properties are based on the fixed-point input <code>v</code>.</p> <p><code>t=coder.typeof(v, sz, variable_dims)</code> returns a modified copy of <code>coder.typeof(v)</code> with (upper bound) size specified by <code>sz</code> and variable dimensions <code>variable_dims</code>. If <code>sz</code> specifies <code>inf</code> for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When <code>sz</code> is <code>[]</code>, the (upper bound) sizes of <code>v</code> do not change. If you do not specify the <code>variable_dims</code> input parameter, the bounded dimensions of the type are fixed. When <code>variable_dims</code> is a scalar, it applies to the bounded dimensions that are not 1 or 0 (which are fixed).</p> <p><code>t=coder.newtype('embedded.fi', numerictype, sz, variable_dims)</code> creates a <code>coder.Type</code> object representing a set of fixed-point values with <code>numerictype</code> and (upper bound) sizes <code>sz</code> and variable dimensions <code>variable_dims</code>. If <code>sz</code> specifies <code>inf</code> for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When you do not specify <code>variable_dims</code>, the bounded dimensions of the type are fixed. When <code>variable_dims</code> is a scalar, it applies to the bounded dimensions that are not 1 or 0 (which are fixed).</p> <p><code>t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)</code> creates a <code>coder.Type</code> object representing a set of fixed-point values with <code>numerictype</code> and additional options specified by one or more <code>Name, Value</code> pair arguments. <code>Name</code> can also be a property name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (<code>' '</code>). You can specify several name-value pair arguments in any order as <code>Name1,Value1, ,NameN,ValueN</code>.</p>

## 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 single 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 in the MATLAB documentation](#).

## 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.Type](#) | [coder.ArrayType](#) | [coder.typeof](#) | [coder.resize](#) |  
[coder.newtype](#) | [fiaccel](#)

# coder.inline

---

**Purpose** Control inlining in generated code

**Syntax**

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

**Description**

`coder.inline('always')` forces inlining of the current function in generated code.

`coder.inline('never')` prevents inlining of the current function in generated code. For example, you may want to prevent inlining to simplify the mapping between the MATLAB source code and the generated code.

`coder.inline('default')` uses internal heuristics to determine whether or not to inline the current function.

In most cases, the heuristics used produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

Place the `coder.inline` directive inside the function to which it applies. MATLAB CoderFixed-Point Designer does not inline entry-point functions.

`coder.inline('always')` does not inline functions called from `parfor`-loops. MATLAB Coder does not inline functions into `parfor`-loops.

## Examples

- “Preventing Function Inlining” on page 2-146
- “Using `coder.inline` In Control Flow Statements” on page 2-147

### Preventing Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
 coder.inline('never');
 y = x;
end
```



## Using 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 you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
 coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
 coder.inline('never');
end

if any(divisor == 0)
 error('Can not divide by 0');
end

y = dividend / divisor;
```

# coder.load

---

**Purpose** 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.

`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', '-regexp', expr1, ..., exprN)`  
treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

## Input Arguments

### **filename - Name of file**

string

Name of file, specified as a string 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'`

### **Data Types**

char

### **var1,...,varN - Names of variables to load**

string

Names of variables, specified as string constants. Use the `*` wildcard to match patterns.

**Example:** `load('myFile.mat', 'A*')` loads all variables in the file whose names start with `A`.

### **Data Types**

char

### **expr1,...,exprN - Regular expressions indicating which variables to load**

string

Regular expressions indicating which variables to load, specified as string constants.

**Example:** `load('myFile.mat', '^A', '^B')` loads only variables whose names begin with A or B.

## Data Types

char

## 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.

## 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.

## Limitations

- `coder.load` does not support loading objects.
- Arguments to `coder.load` must be compile-time constant strings.
- 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 generation software does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save 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 generation software 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](#)

## Concepts

- “Regular Expressions”

<b>Purpose</b>	Code acceleration configuration object
<b>Syntax</b>	<code>config_obj = coder.mexconfig</code>
<b>Description</b>	<code>config_obj = coder.mexconfig</code> creates a <code>coder.MexConfig</code> code generation configuration object for use with <code>fiaccl</code> , which generates a MEX function.
<b>Output Arguments</b>	<b>config_obj</b> Code generation configuration object for use when generating MEX functions using <code>fiaccl</code> .
<b>Examples</b>	Create a configuration object to disable run-time checks  <pre>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</pre>
<b>See Also</b>	<code>coder.ArrayType</code>   <code>coder.Constant</code>   <code>coder.EnumType</code>   <code>coder.FiType</code>   <code>coder.MexConfig</code>   <code>coder.PrimitiveType</code>   <code>coder.StructType</code>   <code>coder.Type</code>   <code>coder.newtype</code>   <code>coder.resize</code>   <code>coder.typeof</code>   <code>fiaccl</code>

# coder.MexConfig

---

- Purpose** 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

### 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 `fiaccel` allocates memory on the heap.

**Default:** 65536

## **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`

## **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. `fiaccel` does not compile or generate code for extrinsic functions. Set this property to `true` to have `fiaccel` 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.

**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`

## **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 in the MATLAB documentation.

**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

```

## **coder.MexConfig**

---

### **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`

**Purpose** Create a new `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('embedded.fi', numericity, sz,
variable_dims, Name,
Value)
t=coder.newtype(enum_value, sz, variable_dims)
```

**Description**

---

**Note** `coder.newtype` is an advanced function. Consider using `coder.typeof` instead.

---

`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 should 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 of the given `sz` and `variable_dims` information with the same fields as the scalar structure `struct_fields`.

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.FiType` object representing a set of fixed-point values with `numerictype` 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`.

## Input Arguments

### **numeric\_class**

Class of the set of values represented by the type object

### **struct\_fields**

Scalar structure used to specify the fields in a new structure type

### **sz**

Size vector specifying each dimension of type object

**Default:** [1 1]

### **variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false)

**Default:** false(size(sz)) | sz==Inf

## **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 single 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 `fimath` is not specified, uses default `fimath` values.

Use only with `t=coder.newtype('embedded.fi', numericType,SZ, variable_dims, Name, Value)`.

**'sparse'**

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)`

**Default:** `false`

**Output Arguments**

**t**  
New `coder.Type` object.

**Examples**

Create a new type for use in code generation.

```
t=coder.newtype('double',[2 3 4],[1 1 0])
% Returns double :2x:3x4
% ':' 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])
% returns double:inf x 3
```

```
coder.newtype('double', [inf, 3], [1 0])
% also returns double :inf x3
% ':' indicates variable-size dimensions
```

---

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size with an upper bound of 3

```
coder.newtype('double', [inf,3],[0 1])
% returns double :inf x :3
% ':' indicates variable-size dimensions
```

---

Create a new structure type for 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))
% returns struct 1x1
% a: int8 1x1
% b: double :1x:2
% ':' indicates variable-size dimensions
```

---

Create a new constant type for use in code generation.

```
k = coder.newtype('constant', 42);
% Returns
% k =
%
% coder.Constant
% 42
```

---

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(Enumeration) MyColors < int32
 enumeration
 green(1),
 red(2),
 end
end
```

- 2 Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors');
```

Create a new fixed-point type for use in code generation. The fixed-point type uses default `fimath` values.

```
t = coder.newtype('embedded.fi',...
 numericity(1, 16, 15), [1 2])

t =
% Returns
% coder.FiType
% 1x2 embedded.fi
% DataTypeMode: Fixed-point: binary point scaling
% Signedness: Signed
% WordLength: 16
% FractionLength: 15
```

**Alternatives** `coder.typeof`

# coder.newtype

---

## See Also

[coder.resize](#) | [coder.Type](#) | [coder.ArrayType](#) | [coder.EnumType](#) |  
[coder.FiType](#) | [coder.PrimitiveType](#) | [coder.StructType](#) | [fiaccel](#)

**Purpose** Declare uninitialized variables

**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. Preallocates memory for `X` without incurring the overhead of initializing memory.

`coder.nullcopy` does not support MATLAB classes as inputs.

### Use With Caution

Use this function with caution. See “How to Eliminate Redundant Copies by Defining Uninitialized Variables”.

### Examples

The following example shows how to declare variable `X` as a 1-by-5 vector of real doubles without performing an unnecessary initialization:

```
function X = foo

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.

### How To

- “Eliminate Redundant Copies of Variables in Generated Code”

# coder.PrimitiveType

---

**Superclasses**    ArrayType

**Purpose**            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. `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**

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 single 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.

# coder.PrimitiveType

---

**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 in the MATLAB documentation.

## **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.Type` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `fiaccel`



**Purpose**                      Resize a coder.Type object

**Syntax**

```
t_out = coder.resize(t, sz, variable_dims)
t_out = coder.resize(t, sz)
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, 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, they are applied 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. `coder.resize` ignores `variable_dims` for dimensions with size `inf`. These dimensions are always variable size. `t` can be a cell array, in which case, `coder.resize` resizes all elements of the cell array.

`t_out = coder.resize(t, sz)` resizes `t` to have size `sz`.

`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` using additional options specified by one or more `Name, Value` pair arguments.

`t_out = coder.resize(t, 'sizelimits', limits)` resizes `t` with dimensions automatically 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 unbounded variable size.

**Input Arguments**

**limits**

Two-element vector (or a scalar-expanded one-element vector) of variable-sizing thresholds. 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 unbounded variable size.

**sz**

New size for `coder.Type` object, `t_out`

**t**

`coder.Type` object that you want to resize

**variable\_dims**

Specify whether each dimension of `t_out` should be fixed or variable size.

**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 single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'recursive'**

Setting `recursive` to `true` resizes `t` and all types contained within it

**Default:** `false`

**'uniform'**

Setting `uniform` to `true` resizes `t` but does not apply the heuristic for dimensions of size one.

**Default:** `false`

**Output Arguments****t\_out**

Resized `coder.Type` object

## Examples

Change a fixed-size array to a bounded variable-size array

```
t = coder.typeof(ones(3,3))
% t is 3x3
coder.resize(t, [4 5], 1)
% returns :4 x :5
% ':' indicates variable-size dimensions
```

---

Change a fixed-size array to an unbounded variable-size array

```
t = coder.typeof(ones(3,3))
% t is 3x3
coder.resize(t, inf)
% returns :inf x :inf
% ':' indicates variable-size dimensions
% 'inf' indicates unbounded dimensions
```

---

Resize a structure field

```
ts = coder.typeof(struct('a', ones(3, 3)))
% returns field a as 3x3
coder.resize(ts, [5, 5], 'recursive', 1)
% returns field a as 5x5
```

---

Make a fixed-sized array variable size based on bounded and unbounded thresholds

```
t = coder.typeof(ones(100,200))
% t is 100x200
coder.resize(t,'sizelimits', [99 199])
% returns :100x:inf
% ':' indicates variable-size dimensions
% :inf is unbounded variable size
```

# coder.resize

---

## See Also

`coder.typeof` | `coder.newtype` | `fiaccel`

<b>Purpose</b>	Determine if function is suitable for code generation
<b>Syntax</b>	<pre>coder.screener(fcn) coder.screener(fcn_1,...,fcn_n )</pre>
<b>Description</b>	<p><code>coder.screener(fcn)</code> analyzes the entry-point MATLAB function, <code>fcn</code>. It identifies unsupported functions and language features, such as recursion, cell arrays, nested functions, and function handles as code generation compliance issues and displays them in a report. If <code>fcn</code> calls other functions directly or indirectly that are not MathWorks® functions, <code>coder.screener</code> analyzes these functions too. It does not analyze MathWorks functions. <code>coder.screener</code> might not detect all code generation issues. Under certain circumstances, it might report false errors.</p> <p><code>coder.screener(fcn_1,...,fcn_n )</code> analyzes entry-point functions (<code>fcn_1,...,fcn_n</code>).</p>
<b>Tips</b>	<ul style="list-style-type: none"> <li>• Before using <code>coder.screener</code>, fix issues identified by the code analyzer.</li> <li>• Before generating code, use <code>coder.screener</code> to check that a function is suitable for code generation. Fix all the issues that it detects.</li> </ul>
<b>Input Arguments</b>	<p><b>fcn</b> Name of entry-point MATLAB function that you want to analyze.</p> <p><b>fcn_1,...,fcn_n</b> Comma-separated list of names of entry-point MATLAB functions that you want to analyze.</p>
<b>Examples</b>	<p><b>Identify Unsupported Functions</b></p> <p>The <code>coder.screener</code> function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, <code>foo1</code>, and the function <code>foo2</code> that <code>foo1</code> calls.</p>

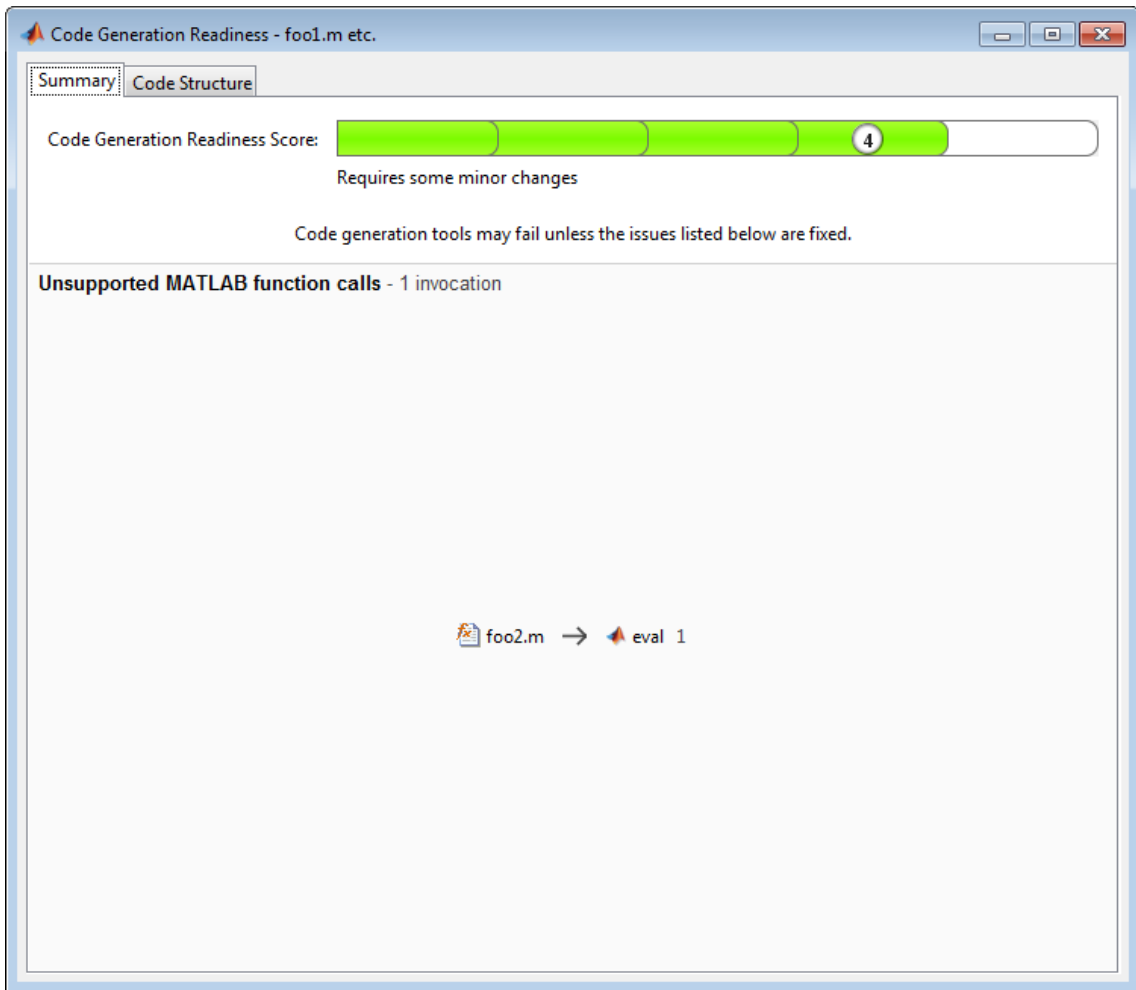
Analyze the MATLAB function `foo1` that calls `foo2`.

```
function out = foo1(in)
 out = foo2(in);
 disp(out);
end
```

```
function out = foo2(in)
 out = eval(in);
end
```

```
coder.screener('foo1')
```

The code generation readiness report opens. It provides a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.

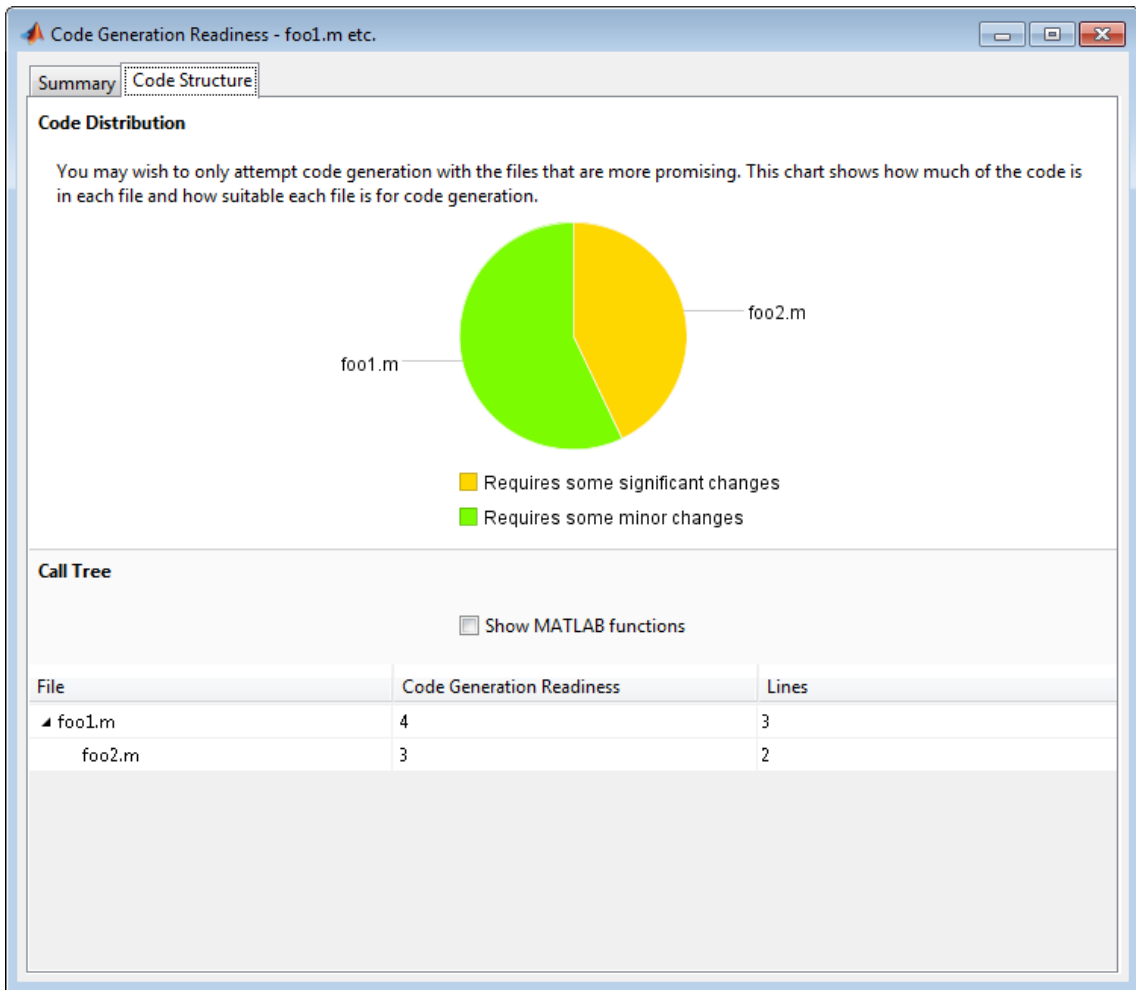


In the report, click the **Code Structure** tab and select **Show MATLAB functions**.

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 some significant changes are required.
- 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 to 5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.
- Displays a call tree.





The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function which is not supported for code generation. 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 the `eval` function is not supported for code generation. When you generate a MEX function for `foo1`, the code generation software automatically calls out to MATLAB for `eval`. For standalone code generation, it does not generate code for it.

## Identify Unsupported Data Types

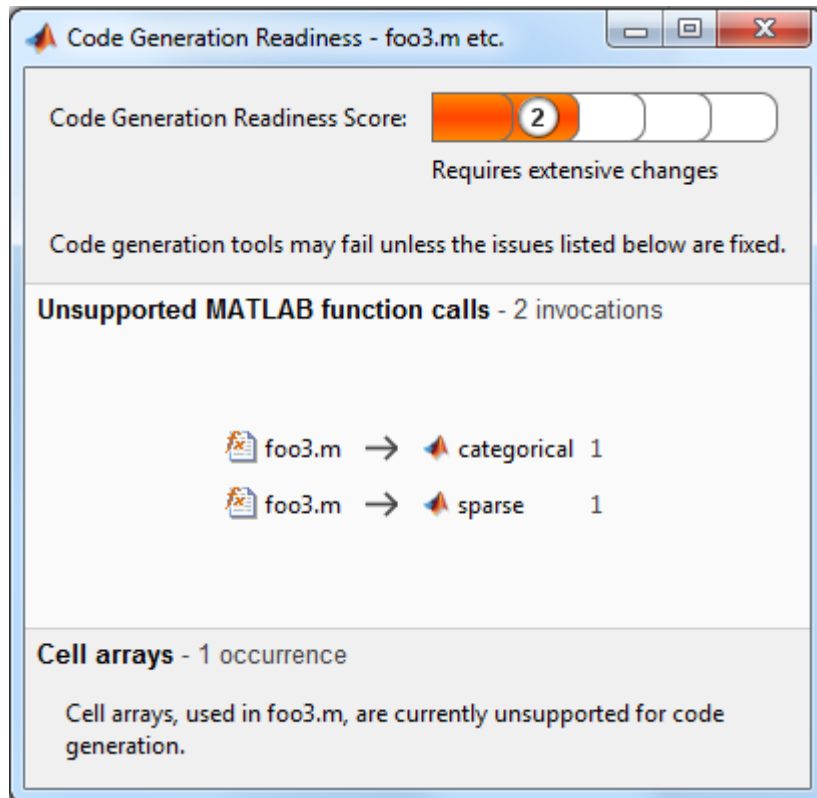
The `coder.screener` function identifies data types that are not supported for code generation.

Analyze the MATLAB function `foo3` that uses unsupported data types.

```
function [outSparse,outCategorical] = foo3(inVal,inStr1,inStr2)
 outSparse = sparse(inVal);
 cellArray = {inStr1,inStr2};
 outCategorical = categorical(cellArray);
end
```

```
coder.screener('foo3')
```

The code generation readiness report opens. It provides a summary of the unsupported data types.



The report assigns the code a code readiness score of 2, indicating that the code requires extensive changes.

Before generating code, you must fix the reported issues.

### Determine code generation readiness for multiple entry-point functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks the entry-point functions `foo4` and `foo5`.

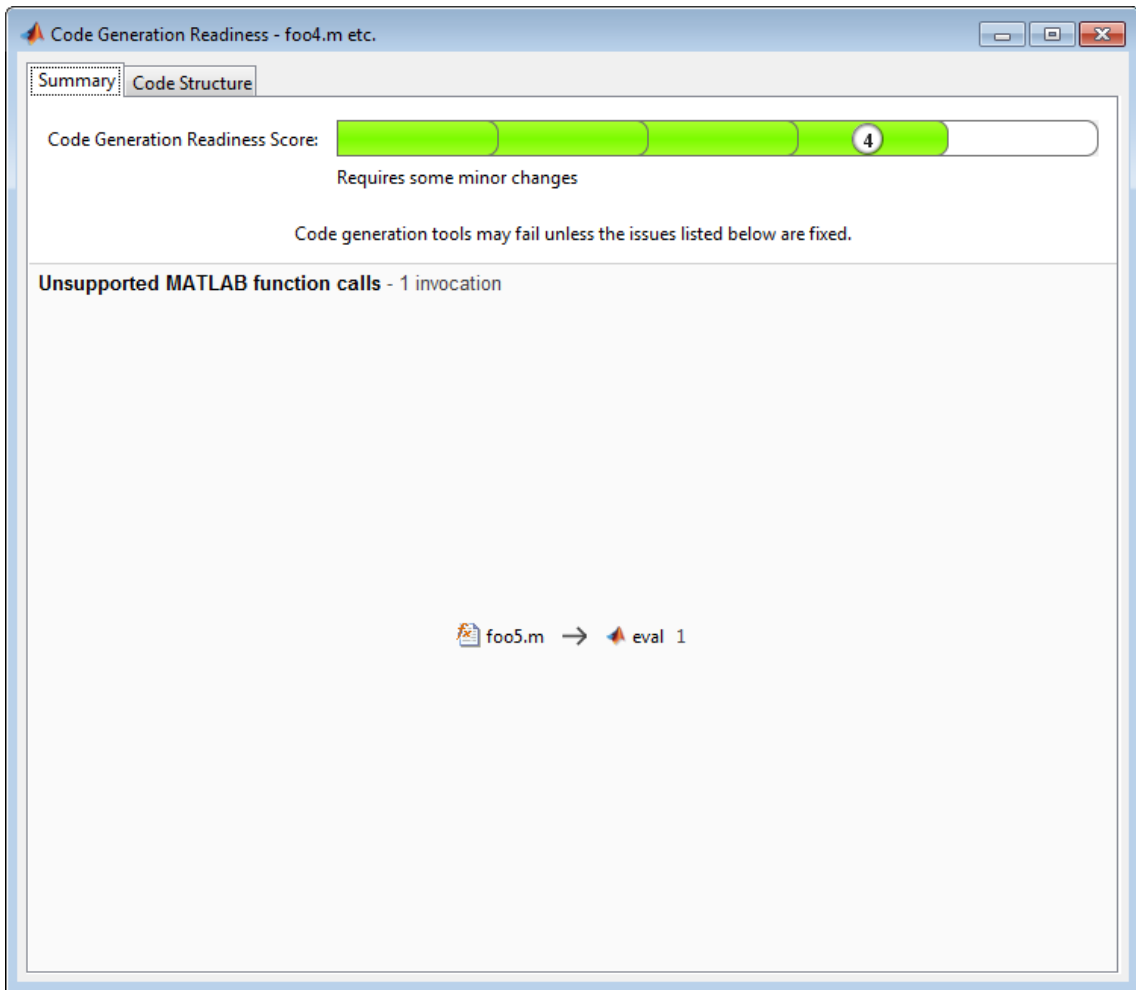
Analyze the MATLAB functions `foo4` and `foo5`.

```
function out = foo4(in)
 out = in;
 disp(out);
end
```

```
function out = foo5(in)
 out = eval(in);
end
```

```
coder.screener('foo4', 'foo5')
```

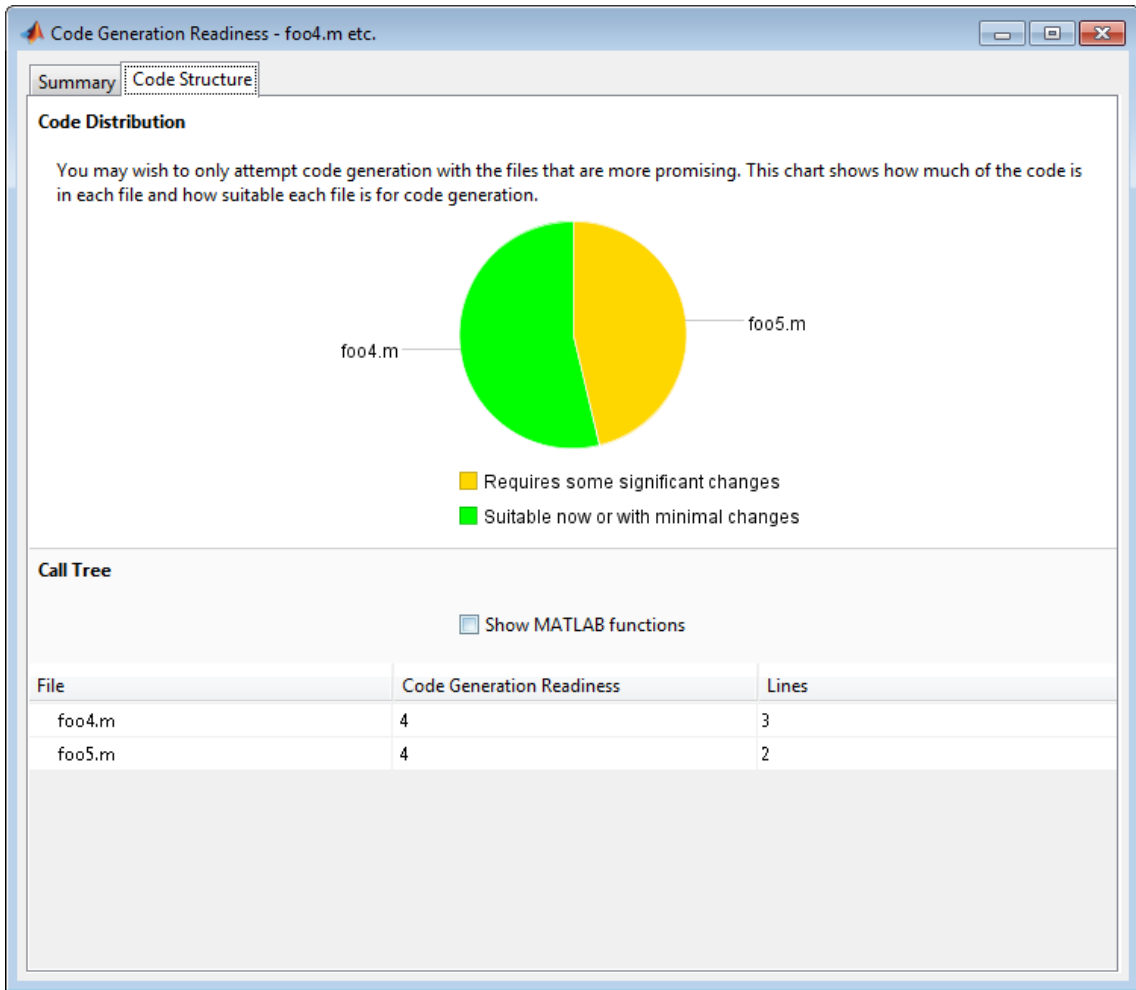
The code generation readiness report opens. It provides a summary of the unsupported MATLAB function calls. The function `foo5` calls one unsupported MATLAB function.



In the report, click the **Code Structure** tab and select **Show MATLAB functions**.

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 some significant changes are required.
- 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 to 5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.
- Displays a call tree.



- Alternatives**
- “Run the Code Generation Readiness Tool From the Current Folder Browser”

**See Also**      `fiaccel`

## **Concepts**

- “Functions Supported for Code Acceleration or C Code Generation”
- “Code Generation Readiness Tool”



**Superclasses**    ArrayType

**Purpose**            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.

# coder.StructType

---

## **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 string.

## 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 in the MATLAB documentation.

## 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
```

---

# coder.StructType

---

Create a `coder.StructType` object that uses an externally-defined structure type.

- 1 Create a type that uses an externally-defined structure type.

```
S.a = coder.typeof(double(0));
S.b = coder.typeof(single(0));
T = coder.typeof(S);
T = coder.cstructname(T, 'mytype', 'extern', 'HeaderFile', 'myheader.h');
```

```
T =
```

```
coder.StructType
 1x1 extern mytype (myheader.h) struct
 a: 1x1 double
 b: 1x1 single
```

- 2 View the types of the structure fields.

```
T.Fields
```

```
ans =
```

```
 a: [1x1 coder.PrimitiveType]
 b: [1x1 coder.PrimitiveType]
```

## See Also

`coder.Type` | `coder.PrimitiveType` | `coder.EnumType` |  
`coder.FiType` | `coder.Constant` | `coder.ArrayType` | `coder.newtype`  
| `coder.typeof` | `coder.resize` | `fiaccel` |

**Purpose** 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.

**Input Arguments**

**target - code generation target**

string

Code generation target specified as one of the following strings:

- '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')`

**Data Types**

char

**Examples**

**Use `coder.target` to parameterize a MATLAB function**

Parameterize a MATLAB function so that it works in MATLAB or 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 the C library for `myabsval.m`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

`codegen` creates the library `myabsval.lib` and header file `myabsval.h` in the folder `/codegen/lib/myabsval`. It also 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 %#codegen
y = -2.75;
% 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
 % 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
```

Convert `callmyabsval.m` to the MEX function `callmyabsval_mex`.

```
codegen -config:mex callmyabsval codegen/lib/myabsval/myabsval.lib...
 codegen/lib/myabsval/myabsval.h
```

Run the MATLAB function `callmyabsval` .

```
callmyabsval
```

```
ans =
```

```
 2.7500
```

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex
```

```
ans =
```

```
 2.7500
```

# coder.Type

---

<b>Purpose</b>	Represent set of MATLAB values
<b>Description</b>	Specifies the set of values that the generated code should accept. Use only with the <code>fiaccl -args</code> option. Do not pass as an input to a generated MEX function.
<b>Construction</b>	<code>coder.Type</code> is an abstract class, and you cannot create instances of it directly. You can create <code>coder.Constant</code> , <code>coder.EnumType</code> , <code>coder.FiType</code> , <code>coder.PrimitiveType</code> , and <code>coder.StructType</code> objects that are derived from this class.
<b>Properties</b>	<b>ClassName</b> Class of values in this set
<b>Copy Semantics</b>	Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.
<b>See Also</b>	<code>coder.StructType</code>   <code>coder.PrimitiveType</code>   <code>coder.EnumType</code>   <code>coder.FiType</code>   <code>coder.Constant</code>   <code>coder.ArrayType</code>   <code>coder.newtype</code>   <code>coder.typeof</code>   <code>coder.resize</code>   <code>fiaccl</code>



**Purpose** Convert MATLAB value into its canonical type

**Syntax**

```
t=coder.typeof(v)
t=coder.typeof(v, sz, variable_dims)
t=coder.typeof(t)
```

**Description** `t=coder.typeof(v)` creates a `coder.Type` object denoting the smallest nonconstant type that contains `v`. `v` must be a MATLAB numeric, logical, char, enumeration or fixed-point array, or a struct constructed from the preceding types. 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.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `t=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` remain unchanged. 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 is applied to bounded dimensions or dimensions that are 1 or 0, which are fixed.

`t=coder.typeof(t)`, where `t` is a `coder.Type` object, returns `t` itself.

**Input Arguments**

**sz**  
Size vector specifying each dimension of type object

**t**  
`coder.Type` object

**v**  
MATLAB expression that describes the set of values represented by this type.

# coder.typeof

---

v must be a MATLAB numeric, logical, char, enumeration or fixed-point array, or a struct constructed from the preceding types.

## **variable\_dims**

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false).

**Default:** false(size(sz)) | sz==Inf

## **Output Arguments**

**t**  
coder.Type object

## **Examples**

Create a type for a simple fixed-size 5x6 matrix of doubles.

```
coder.typeof(ones(5, 6))
% returns 5x6 double
coder.typeof(0, [5 6])
% also returns 5x6 double
```

---

Create a type for a variable-size matrix of doubles.

```
coder.typeof(ones(3,3), [], 1)
% returns :3 x :3 double
% ':' indicates variable-size dimensions
```

---

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
```

---

Create a type for a matrix with fixed-size and variable-size dimensions.

```
coder.typeof(0, [2,3,4], [1 0 1]);
% Returns :2x3x:4 double
% ':' indicates variable-size dimensions
```

---

```
coder.typeof(10, [1 5], 1)
% returns double 1 x :5
% ':' 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])
% returns double:inf x 3
% ':' indicates variable-size dimensions
```

---

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size with an upper bound of 3.

```
coder.typeof(10, [inf,3],[0 1])
% returns double :inf x :3
% ':' indicates variable-size dimensions
```

---

Convert a fixed-sized matrix to a variable-sized matrix.

```
coder.typeof(ones(5,5), [], 1)
% returns double :5x:5
```

# coder.typeof

---

% ':' 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)
% Returns
% coder.StructType
% SuperS: 1x1 struct
% with fields
% x: 1x1 struct
% with fields
% 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)
% Returns
% coder.StructType
% SuperS: 1x1 struct
% with fields
% x: 1x:inf struct
% with fields
% a: 1x1 double
% b: 1x1 single
% y: 1x1 single
```

% ':' indicates variable-size dimensions

## Tips

- If you are already specifying the type of an input variable 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)`.

## See Also

[coder.newtype](#) | [coder.resize](#) | [fiaccel](#)

# coder.unroll

---

**Purpose** Copy body of for-loop in generated code for each iteration

**Syntax**  
`for i = coder.unroll(range)`  
`for i = coder.unroll(range, flag)`

**Description** `for i = coder.unroll(range)` copies the body of a for-loop (unrolls a for-loop) in generated code for each iteration specified by the bounds in *range*. *i* is the loop counter variable.

`for i = coder.unroll(range, flag)` unrolls a for-loop as specified in *range* if *flag* is true.

You must use `coder.unroll` in a for-loop header. `coder.unroll` modifies the generated code, but does not change the computed results.

`coder.unroll` must be able to evaluate the bounds of the for-loop at compile time. The number of iterations cannot exceed 1024; unrolling large loops can increase compile time significantly and generate inefficient code

This function is ignored outside of code generation.

## Input Arguments

### flag

Boolean expression that indicates whether to unroll the for-loop:

true Unroll the for-loop

false Do not unroll the for-loop

### range

Specifies the bounds of the for-loop iteration:

<code>init_val : end_val</code>	Iterate from <code>init_val</code> to <code>end_val</code> , using an increment of 1
<code>init_val : step_val : end_val</code>	Iterate from <code>init_val</code> to <code>end_val</code> , using <code>step_val</code> as an increment if positive or as a decrement if negative
Matrix variable	Iterate for a number of times equal to the number of columns in the matrix

## Examples

To limit the number of times to copy the body of a `for`-loop in generated code:

- 1 Write a MATLAB function `getrand(n)` that uses a `for`-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the `for`-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Calling getrand 8 times triggers unroll
y1 = getrand(8);
% Calling getrand 50 times does not trigger unroll
y2 = getrand(50);
```

```
function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
coder.inline('never');

% Set flag variable dounroll to repeat loop body
% only for fewer than 10 iterations
dounroll = n < 10;
```

```
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = coder.unroll(1:2:n, dounroll)
 if (i > 2) && (i < n-2)
 y(i) = rand();
 end;
end;
% Loop body ends
```

- 2** In the default output folder, `codegen/lib/test_unroll`, generate C static library code for `test_unroll` :

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the for-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void getrand(double y[8])
{
 /* Turn off inlining to make */
 /* generated code easier to read */
 /* Set flag variable dounroll to repeat loop body */
 /* only for fewer than 10 iterations */
 /* Declare size, class, and complexity */
 /* of variable y by assignment */
 memset(&y[0], 0, sizeof(double) << 3);

 /* Loop body begins */
 y[2] = b_rand();
 y[4] = b_rand();

 /* Loop body ends */
}
```



The generated C code for `getrand(50)` does not unroll the for-loop because the number of iterations is greater than 10:

```
static void b_getrand(double y[50])
{
 int i;
 int b_i;

 /* Turn off inlining to make */
 /* generated code easier to read */
 /* Set flag variable dounroll to repeat loop body */
 /* only for fewer than 10 iterations */
 /* Declare size, class, and complexity */
 /* of variable y by assignment */
 memset(&y[0], 0, 50U * sizeof(double));

 /* Loop body begins */
 for (i = 0; i < 25; i++) {
 b_i = (i << 1) + 1;
 if ((b_i > 2) && (b_i < 48)) {
 y[b_i - 1] = b_rand();
 }
 }
}
```

**See Also**

`coder.inline` | `coder.nullcopy` | `for`

**How To**

- “Using Logicals in Array Indexing”
-

# coder.versize

---

**Purpose** Declare variable-size data

**Syntax**

```
coder.versize('var1', 'var2', ...)
coder.versize('var1', 'var2', ..., ubound)
coder.versize('var1', 'var2', ..., ubound, dims)
coder.versize('var1', 'var2', ..., [], dims)
```

**Description** `coder.versize('var1', 'var2', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each 'var<sub>n</sub>' must be a quoted string that represents a variable or structure field. If the 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. For example, the expression `coder.versize('data(:).A')` declares that the field A inside each element of data is variable sized.

`coder.versize('var1', 'var2', ..., ubound)` declares one or more variables as variable-size data with an explicit upper bound specified in *ubound*. The argument *ubound* must be a constant, integer-valued vector of upper bound sizes for every dimension of each 'var<sub>n</sub>'. If you specify more than one 'var<sub>n</sub>', each variable must have the same number of dimensions.

`coder.versize('var1', 'var2', ..., ubound, dims)` declares one or more variables as variable-sized with an explicit upper bound and a mix of fixed and varying dimensions specified in *dims*. The argument *dims* is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or false in *dims* have fixed size; dimensions that correspond to ones or true vary in size. If you specify more than one variable, each fixed dimension must have the same value across all 'var<sub>n</sub>'.

`coder.versize('var1', 'var2', ..., [], dims)` declares one or more variables as variable-sized with a mix of fixed and varying dimensions. The empty vector [] means that you do not specify an explicit upper bound.

When you do *not* specify *ubound*, the upper bound is computed for each 'var<sub>n</sub>' in generated code.

When you do *not* specify *dims*, dimensions are assumed to be variable except the singleton ones. A singleton dimension is a dimension for which `size(A,dim) = 1`.

You must add the `coder.versize` declaration before each '`varn`' is used (read). You may add the declaration before the first assignment to each '`varn`'.

`coder.versize` cannot be applied to global variables.

`coder.versize` is not supported for MATLAB class properties.

You cannot use `coder.versize` outside the MATLAB code intended for code generation. For example, the following code does not declare the variable, `var`, as variable-size data:

```
coder.versize('var',10);
codegen -config:lib MyFile -args var
```

Instead, include the `coder.versize` statement inside `MyFile` to declare `var` as variable-size data. Alternatively, you can use `coder.typeof` to declare `var` as variable-size outside `MyFile`. It can then be passed to `MyFile` during code generation using the `-args` option. For more information, see `coder.typeof`.

## Examples

### Develop a simple stack that varies in size up to 32 elements as you push and pop data at run time.

Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack.

```
function test_stack %#codegen
 % The directive %#codegen indicates that the function
 % is intended for code generation
 stack('init', 32);
 for i = 1 : 20
 stack('push', i);
 end
 for i = 1 : 10
 value = stack('pop');
```

## coder.varsize

---

```
 % Display popped value
 value
 end
end
```

Write local function `stack` to execute the push and pop commands.

```
function y = stack(command, varargin)
 persistent data;
 if isempty(data)
 data = ones(1,0);
 end
 y = 0;
 switch (command)
 case {'init'}
 coder.varsize('data', [1, varargin{1}], [0 1]);
 data = ones(1,0);
 case {'pop'}
 y = data(1);
 data = data(2:size(data, 2));
 case {'push'}
 data = [varargin{1}, data];
 otherwise
 assert(false, ['Wrong command: ', command]);
 end
end
```

The variable `data` is the stack. The statement `coder.varsize('data', [1, varargin{1}], [0 1])` declares that:

- `data` is a row vector
- Its first dimension has a fixed size
- Its second dimension can grow to an upper bound of 32

Generate a MEX function for `test_stack`:

```
codegen -config:mex test_stack
```

codegen generates a MEX function in the current folder.

Run `test_stack` to get these results:

```
value =
 20
```

```
value =
 19
```

```
value =
 18
```

```
value =
 17
```

```
value =
 16
```

```
value =
 15
```

```
value =
 14
```

```
value =
 13
```

```
value =
 12
```

```
value =
 11
```

At run time, the number of items in the stack grows from zero to 20, and then shrinks to 10.

## Declare a variable-size structure field.

Write a function `struct_example` that declares an array `data`, where each element is a structure that contains a variable-size field:

```
function y=struct_example() %#codegen

 d = struct('values', zeros(1,0), 'color', 0);
 data = repmat(d, [3 3]);
 coder.sizes('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
end
```

The statement `coder.sizes('data(:).values')` marks as variable-size the field `values` inside each element of the matrix `data`.

Generate a MEX function for `struct_example`:

```
codegen -config:mex struct_example
```

Run `struct_example`.

Each time you run `struct_example` you get a different answer because the function loads the array with random numbers.

## Tips

- If you use input variables (or 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 use `coder.versize` on the array again, unless you also want to specify an upper bound for its size.

- Using `coder.versize` on an array without explicit upper bounds causes dynamic memory allocation of the array. This can reduce speed of generated code. To avoid this, use the syntax `coder.versize('var1', 'var2', ..., ubound)` to specify an upper bound for the array size (if you know it in advance).

## See Also

[assert](#) | [fiaccl](#) | [size](#) | [varargin](#)

## How To

- “Variable-Size Data Definition for Code Generation”
- “Defining Variable-Size Structure Fields”
- “Compilation Directive `%#codegen`”
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation”

# colon

---

**Purpose** Create vectors, array subscripting

**Syntax**  
`y = j:k`  
`y = j:i:k`

**Description** `y = j:k` returns a regularly-spaced vector,  $[j, j+1, \dots, k]$ . `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,  $[j, j+i, j+2i, \dots, j+m*i]$ , where  $m = \text{fix}((k-j)/i)$ . `y = j:i:k` returns an empty matrix when  $i == 0$ ,  $i > 0$  and  $j > k$ , or  $i < 0$  and  $j < k$ .

## 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`.

## 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 =
```

```
 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
signedness = c.Signedness
```

```
len =
```

```
 9
```

```
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 =
 4 3 2 1
 DataTypeMode: Fixed-point: binary point scaling
 Signedness: Unsigned
 WordLength: 16
 FractionLength: 0
```

### Use 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 =

Columns 1 through 6
 1 2 3 4 5 6
Columns 7 through 10
 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.

## See Also

`colon` | `fi`

**Purpose** Create 2-D comet plot

**Description** Refer to the MATLAB comet reference page for more information.

## comet3

---

**Purpose** Create 3-D comet plot

**Description** Refer to the MATLAB comet3 reference page for more information.

**Purpose** Plot arrows emanating from origin

**Description** Refer to the MATLAB compass reference page for more information.

# complex

---

**Purpose** Construct complex `fi` object from real and imaginary parts

**Syntax**  
`c = complex(a,b)`  
`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.

The output `fi` object `c` has the same `numericType` and `fiMath` properties as the input `fi` object `a`.

**See Also** `imag` | `real`



**Purpose** Plot velocity vectors as cones in 3-D vector field

**Description** Refer to the MATLAB `coneplot` reference page for more information.

# conj

---

**Purpose** 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.

**See Also** `complex` | `imag` | `real`

**Purpose** Create contour graph of matrix

**Description** Refer to the MATLAB contour reference page for more information.

# contour3

---

**Purpose** Create 3-D contour plot

**Description** Refer to the MATLAB contour3 reference page for more information.

**Purpose** Create two-level contour plot computation

**Description** Refer to the MATLAB `contourc` reference page for more information.

# contourf

---

**Purpose** Create filled 2-D contour plot

**Description** Refer to the MATLAB `contourf` reference page for more information.

**Purpose**

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(length(a)-max(0,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 the 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 = fir1s(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`.

### See Also

`conv`



**Purpose** 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 =
```

# convergent

---

```
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

### See Also

`ceil` | `fix` | `floor` | `nearest` | `round`

# copyobj

---

**Purpose** 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** `quantizer` | `get` | `set`

<b>Purpose</b>	CORDIC-based absolute value
<b>Syntax</b>	<pre>r = cordicabs(c) r = cordicabs(c,niters) r = cordicabs(c,niters,'ScaleOutput',b) r = cordicabs(c,'ScaleOutput',b)</pre>
<b>Description</b>	<p><code>r = cordicabs(c)</code> returns the magnitude of the complex elements of <code>C</code>.</p> <p><code>r = cordicabs(c,niters)</code> performs <code>niters</code> iterations of the algorithm.</p> <p><code>r = cordicabs(c,niters,'ScaleOutput',b)</code> specifies both the number of iterations and, depending on the Boolean value of <code>b</code>, whether to scale the output by the inverse CORDIC gain value.</p> <p><code>r = cordicabs(c,'ScaleOutput',b)</code> scales the output depending on the Boolean value of <code>b</code>.</p>
<b>Input Arguments</b>	<p><b>c</b></p> <p><code>c</code> is a vector of complex values.</p> <p><b>niters</b></p> <p><code>niters</code> is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, <code>niters</code> must be a positive, integer-valued scalar. If you do not specify <code>niters</code>, 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 <code>r</code> or one less than the word length of <code>theta</code>, 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.</p>

## 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.

## Definitions

### 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.

## 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)
```

---

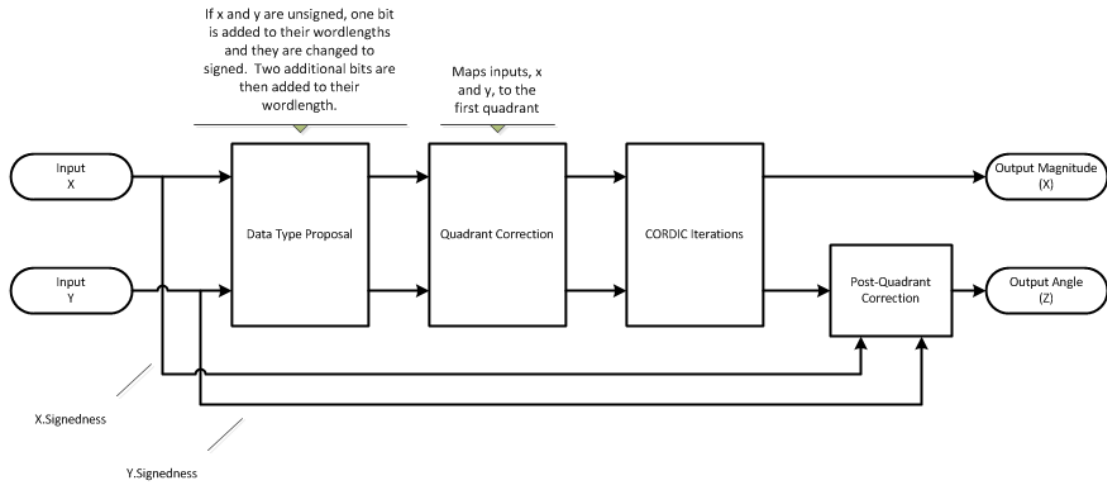
Compute absolute values of fixed-point inputs.

```
fxpValues = fi(dblValues);
r_fxp_cdc = cordicabs(fxpValues)
```

## References

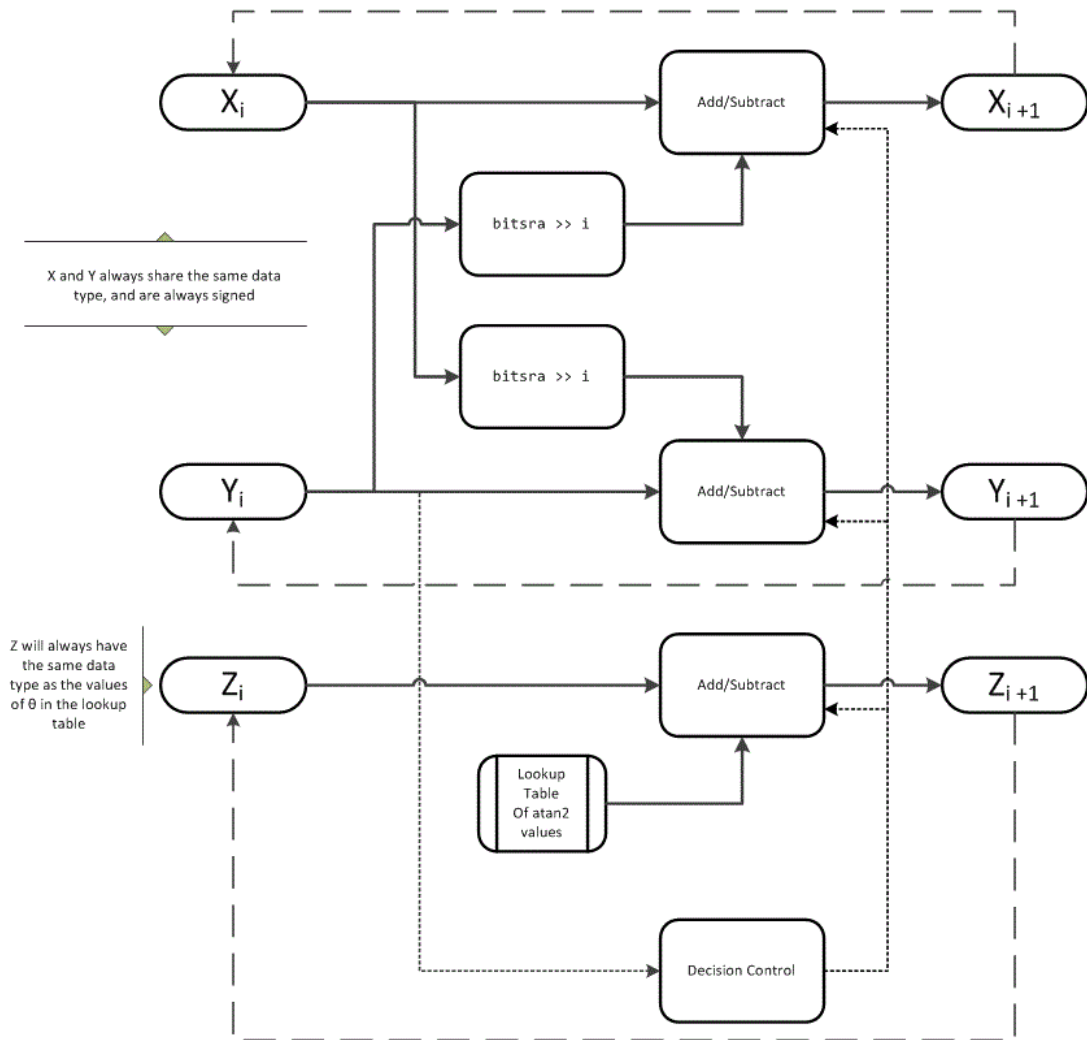
- [1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.
- [2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable 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.

## Algorithms      Signal Flow Diagrams





**CORDIC VECTING 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`.

## **See Also**

`cordiccart2pol` | `cordicangle` | `abs`

<b>Purpose</b>	CORDIC-based phase angle
<b>Syntax</b>	<pre>theta = cordicangle(c) theta = cordicangle(c,niters)</pre>
<b>Description</b>	<p><code>theta = cordicangle(c)</code> returns the phase angles, in radians, of matrix <code>C</code>, which contains complex elements.</p> <p><code>theta = cordicangle(c,niters)</code> performs <code>niters</code> iterations of the algorithm.</p>
<b>Input Arguments</b>	<p><b>c</b> Matrix of complex numbers</p> <p><b>niters</b> <code>niters</code> is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, <code>niters</code> must be a positive, integer-valued scalar. If you do not specify <code>niters</code>, 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 <code>r</code> or one less than the word length of <code>theta</code>, 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.</p>
<b>Output Arguments</b>	<p><b>theta</b> <code>theta</code> contains the polar coordinates angle values, which are in the range <math>[-\pi, \pi]</math> radians. If <code>x</code> and <code>y</code> are floating-point, then <code>theta</code> has the same data type as <code>x</code> and <code>y</code>. Otherwise, <code>theta</code> is a fixed-point data type with the same word length as <code>x</code> and <code>y</code> and with a best-precision fraction length for the <math>[-\pi, \pi]</math> range.</p>

## Definitions

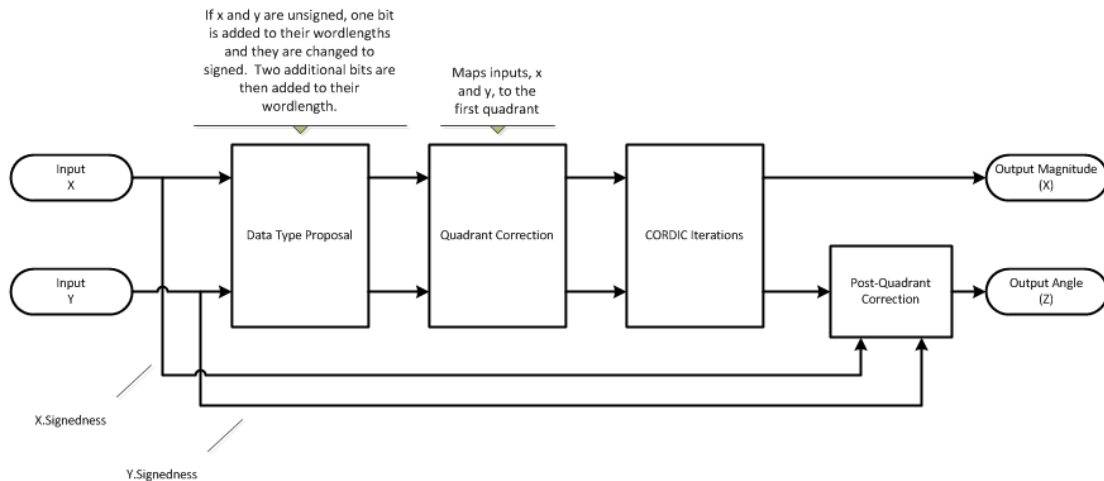
### CORDIC

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.

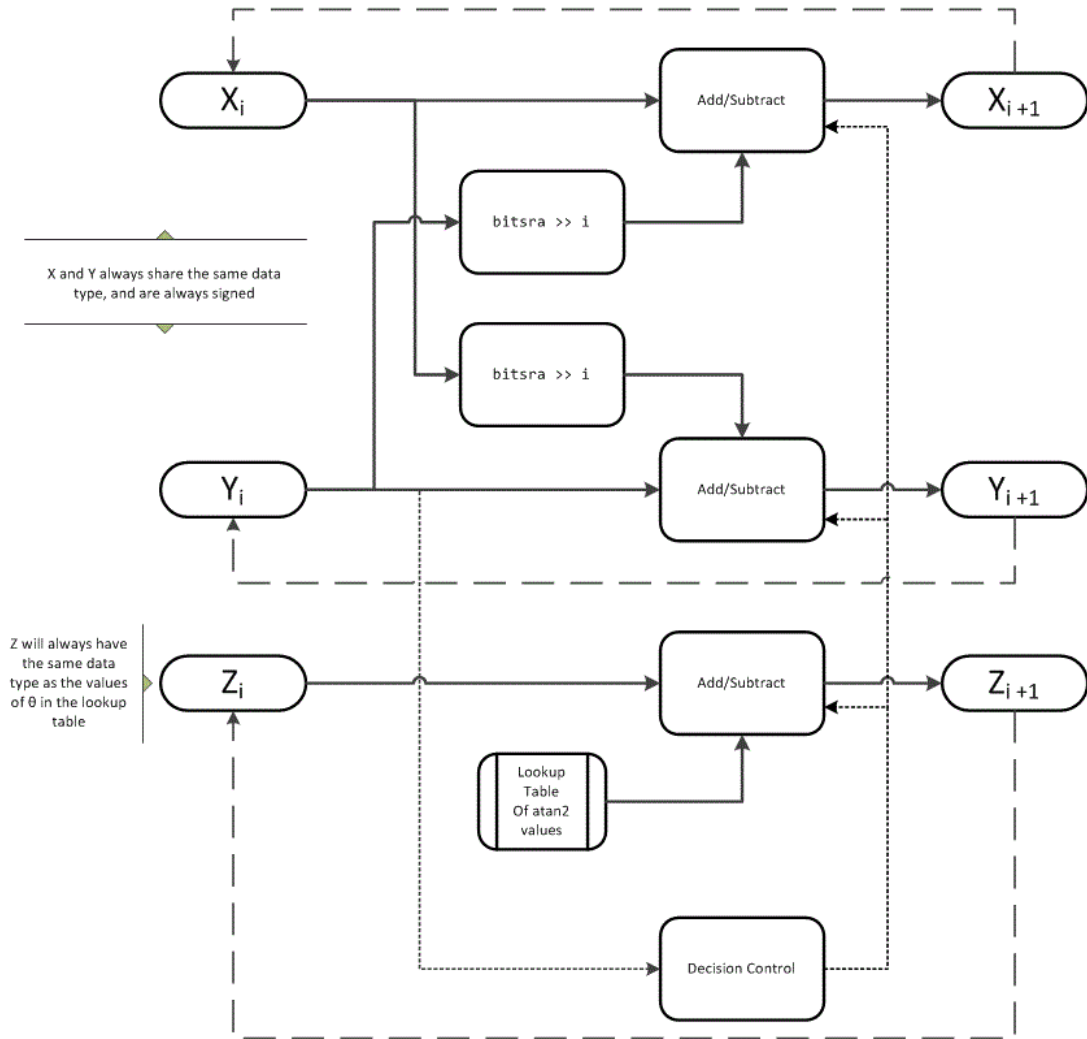
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

## Algorithms

### Signal Flow Diagrams



**CORDIC VECTING 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`.

## 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

## 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.

## See Also

cordicatan2 | cordiccart2po1 | cordicabs | angle

# cordicatan2

---

**Purpose** 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 2-297 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.

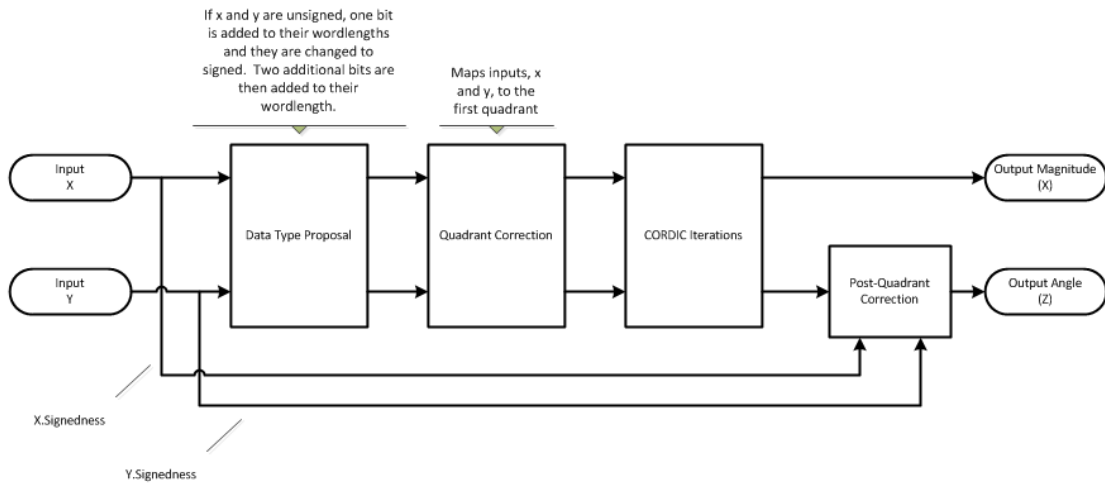


**Definitions**      **CORDIC**

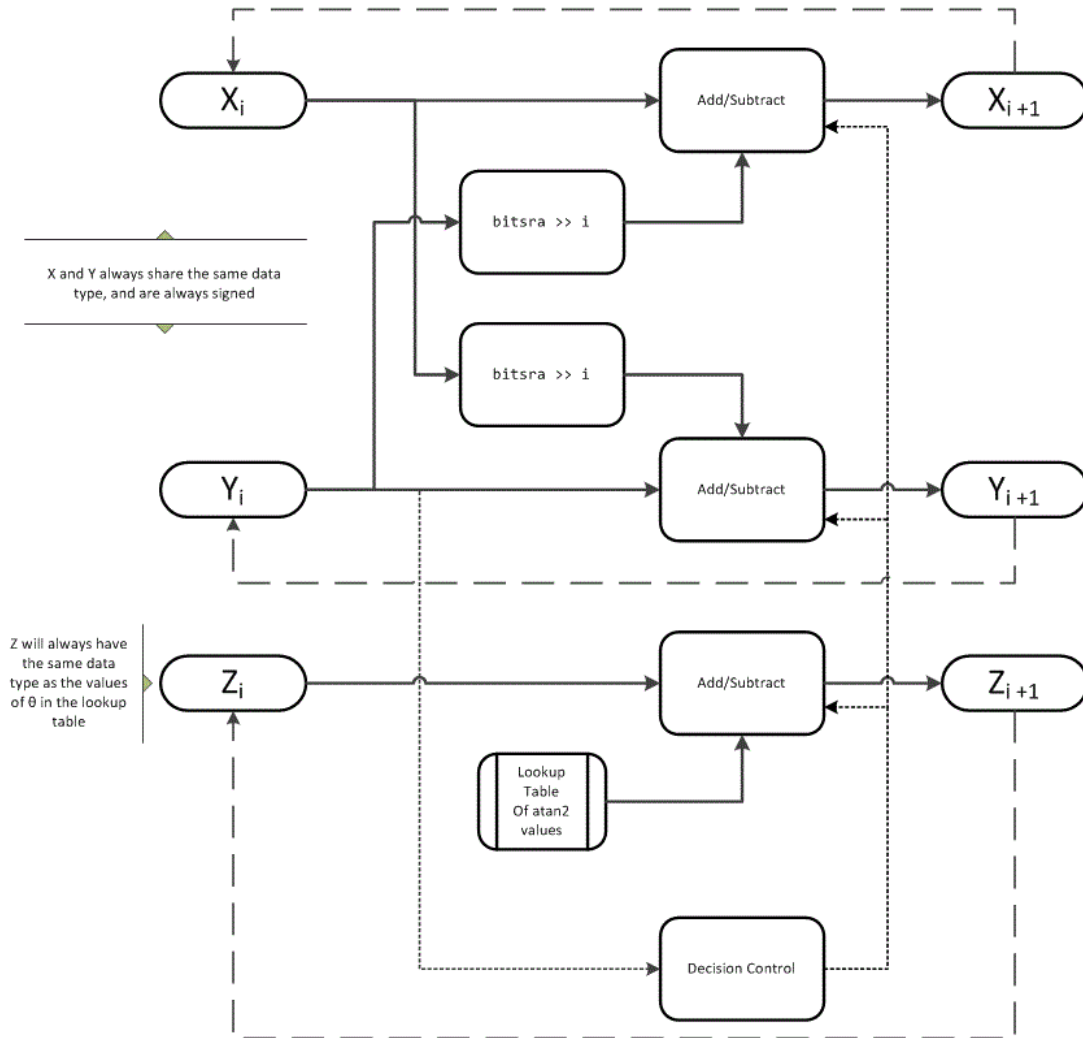
CORDIC is an acronym for COordinate Rotation DIGital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. 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`.

## 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
```

## 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.

## See Also

`atan2` | `atan2` | `cordicsin` | `cordiccos`

## Tutorials

- Demo: Fixed-Point Arctangent Calculation

**Purpose** 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**

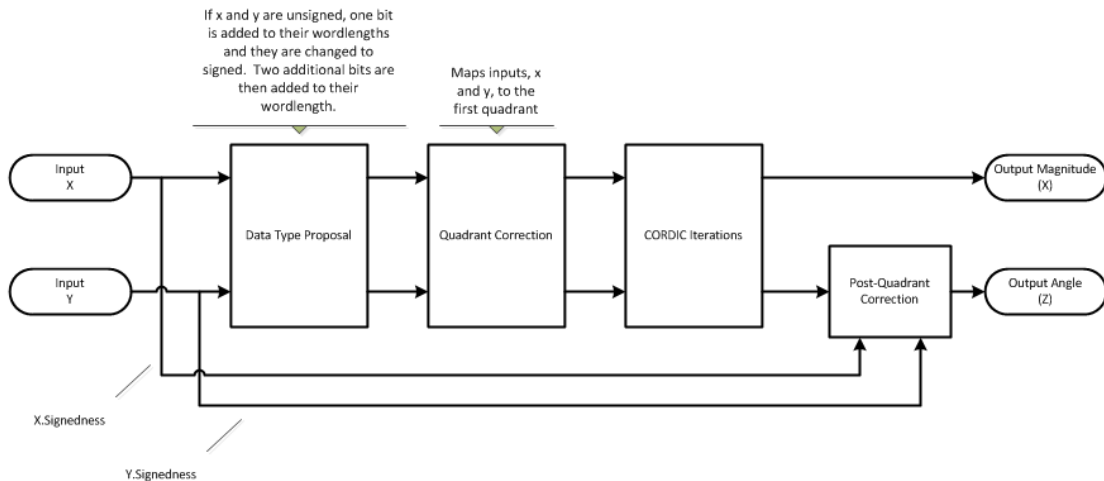
**r** contains the polar coordinates radius magnitude values. **r** is real-valued and can be a scalar value or have the same dimensions as **theta**. If the inputs **x**, **y** are fixed-point values, **r** is also fixed point (and is always signed, with binary point scaling). Both **x**, **y** input 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.

## Definitions **CORDIC**

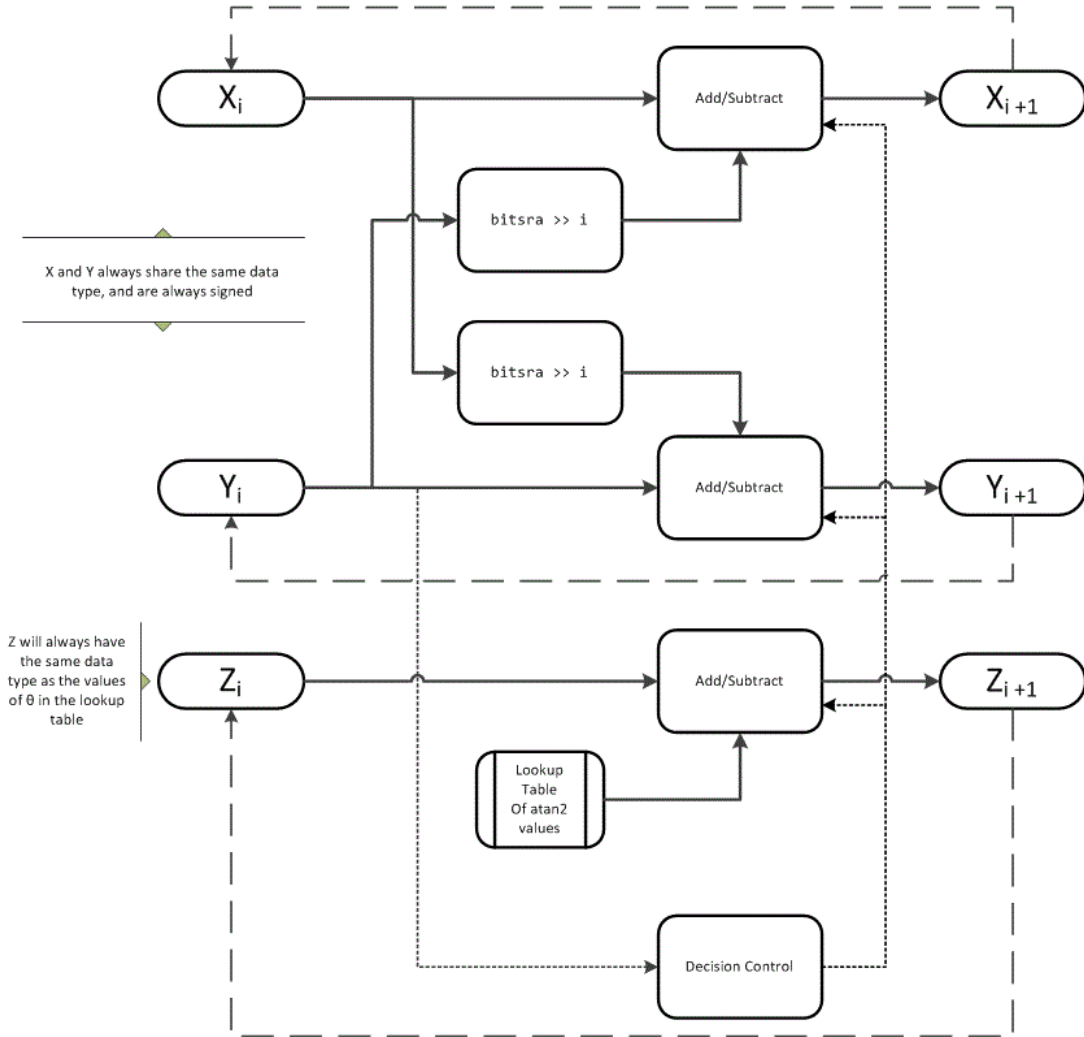
CORDIC is an acronym for COordinate Rotation DIGital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. 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`.

## 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
```

# cordiccart2pol

---

```
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
```

## 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.

## See Also

[cordicatan2](#) | [cordicpol2cart](#) | [cart2pol](#)

# cordicexp

---

<b>Purpose</b>	CORDIC-based approximation of complex exponential
<b>Syntax</b>	<code>y = cordicexp(theta, niters)</code>
<b>Description</b>	<code>y = cordicexp(theta, niters)</code> computes $\cos(\theta) + j\sin(\theta)$ using a “CORDIC” on page 2-297 algorithm approximation. <code>y</code> contains the approximated complex result.
<b>Input Arguments</b>	<p><b>theta</b></p> <p><code>theta</code> can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of <code>theta</code> must be real and in the range <math>[-2\pi, 2\pi]</math>.</p> <p><b>niters</b></p> <p><code>niters</code> is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, <code>niters</code> must be a positive, integer-valued scalar. If you do not specify <code>niters</code> 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 <code>theta</code>. 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.</p>
<b>Output Arguments</b>	<p><b>y</b></p> <p><code>y</code> is the approximated complex result of the <code>cordicexp</code> 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 <code>WordLength - 2</code>.</p>
<b>Definitions</b>	<p><b>CORDIC</b></p> <p>CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most</p>

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.

## Examples

The following example illustrates the effect of the number of iterations on the result of the `cordicexp` 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 = cordicexp(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

## References

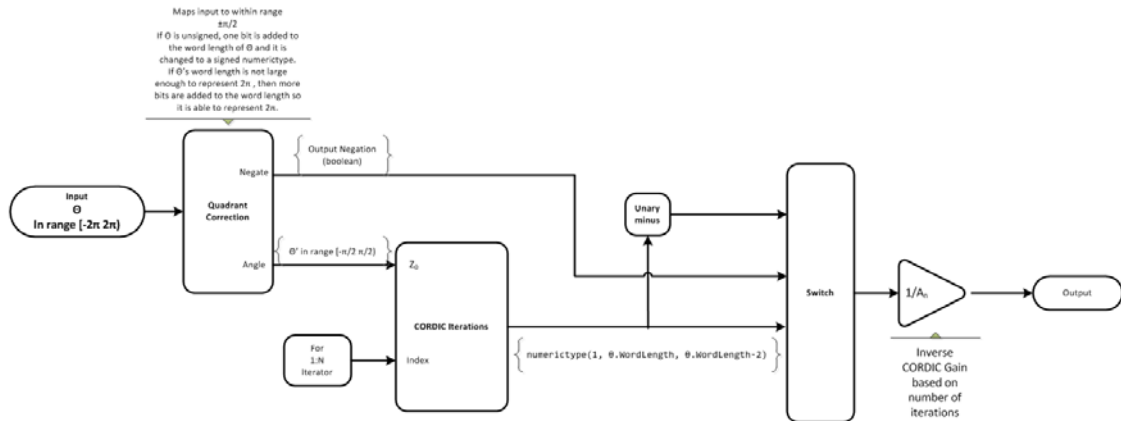
[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[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.

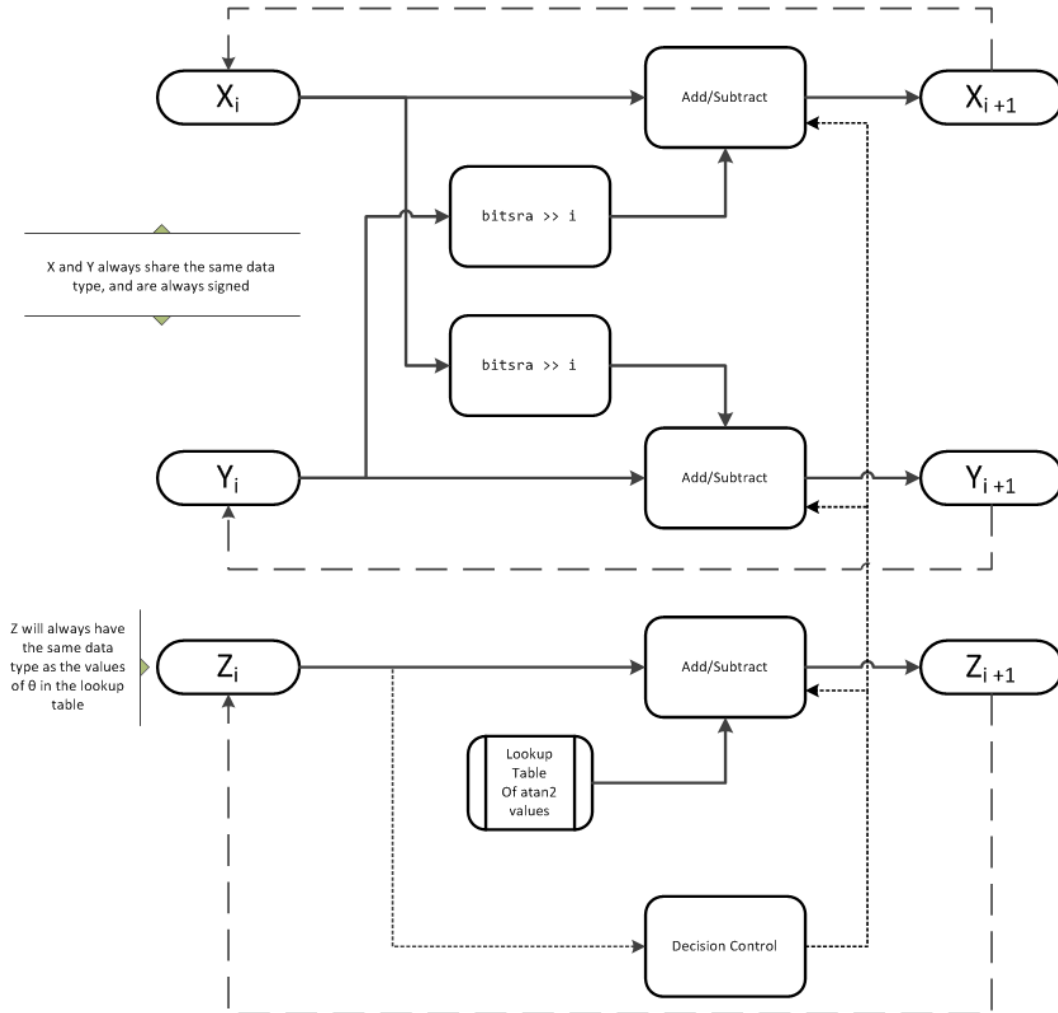
[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

## 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`.

### **See Also**

`cordiccos` | `cordicsin` | `cordicsincos`

### **Tutorials**

- Demo: Fixed-Point Sine and Cosine Calculation
- Demo: Fixed-Point Arctangent Calculation

# cordiccos

---

**Purpose** 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 2-297 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`.

**Definitions**

**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.

## Example

### 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.

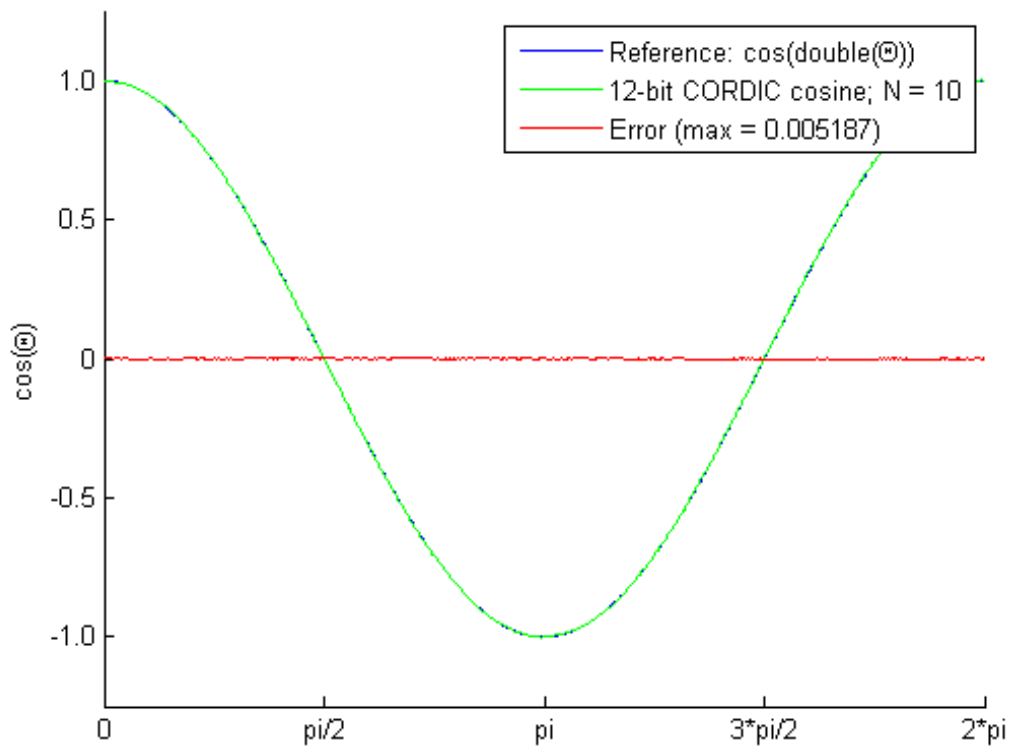
```
% Create 1024 points between [0, 2*pi)
stepSize = pi/512;
thRadDb1 = 0:stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDb1, 12); % signed, 12-bit fixed-point
cosThRef = cos(double(thRadFxp)); % reference results

% 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)');
set(gca, 'XTick', 0:pi/2:2*pi);
set(gca, 'XTickLabel', {'0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
set(gca, 'YTick', -1:0.5:1);
```

# cordiccos

```
set(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.

## References

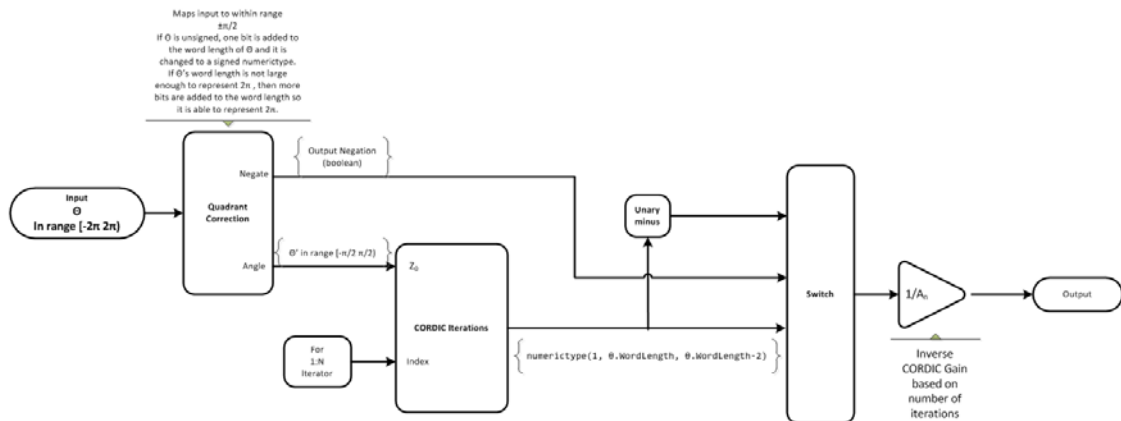
[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[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.

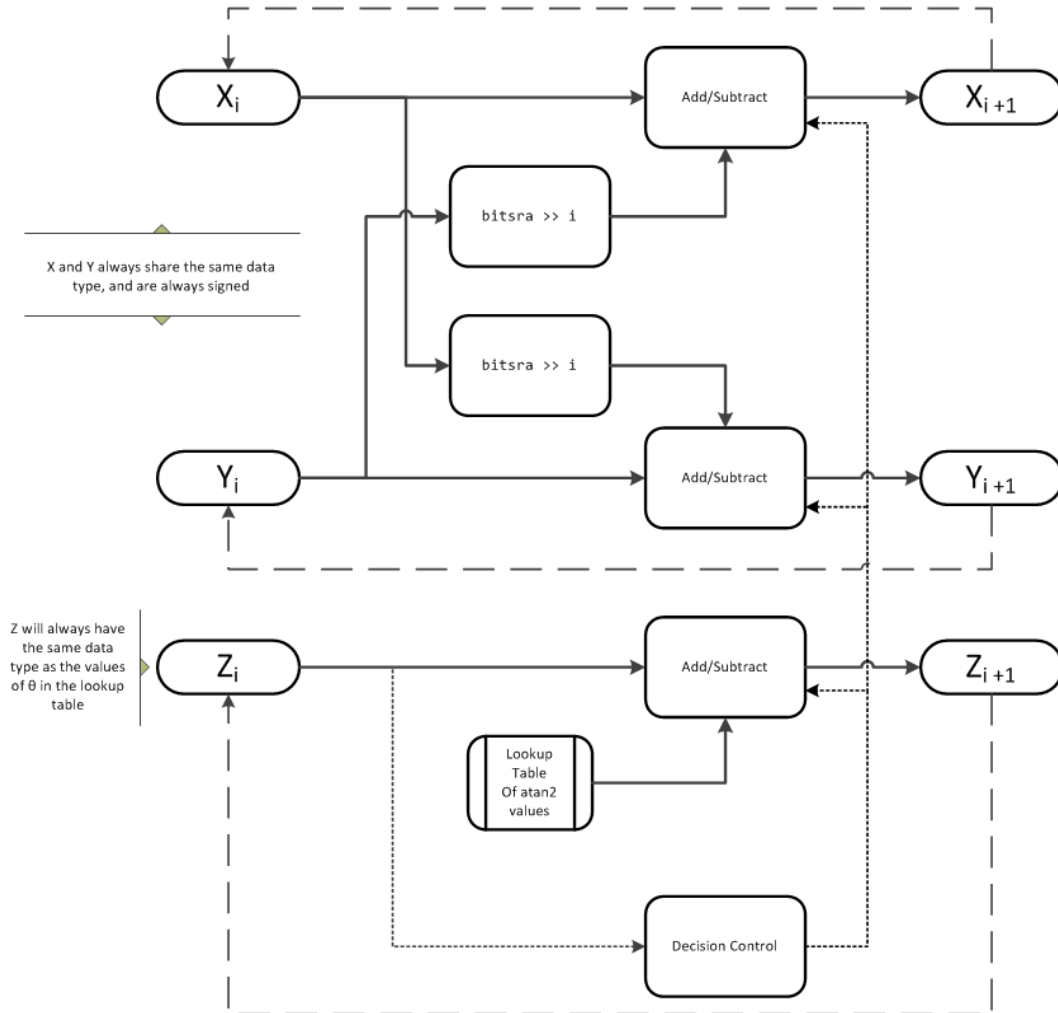
[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum).  
[www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

## 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`.

### **See Also**

`cordicexp` | `cordicsin` | `cordicsincos` | `sin` | `cos`

### **Tutorials**

- Demo: Fixed-Point Sine and Cosine Calculation
- Demo: Fixed-Point Arctangent Calculation

# cordicpol2cart

---

**Purpose** 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`.

## Definitions

### 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.

## 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

## References

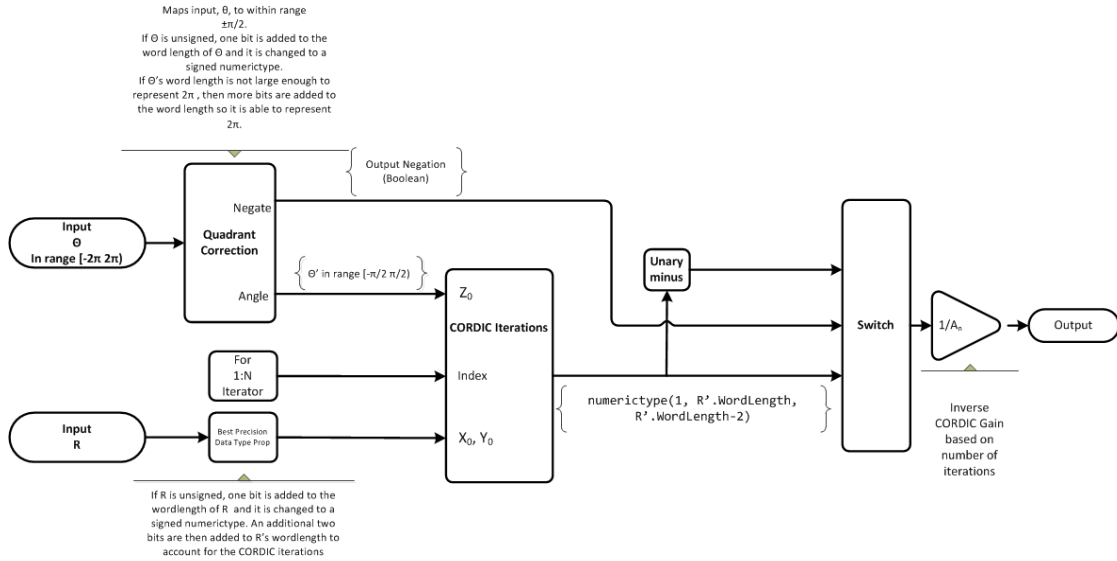
[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[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.

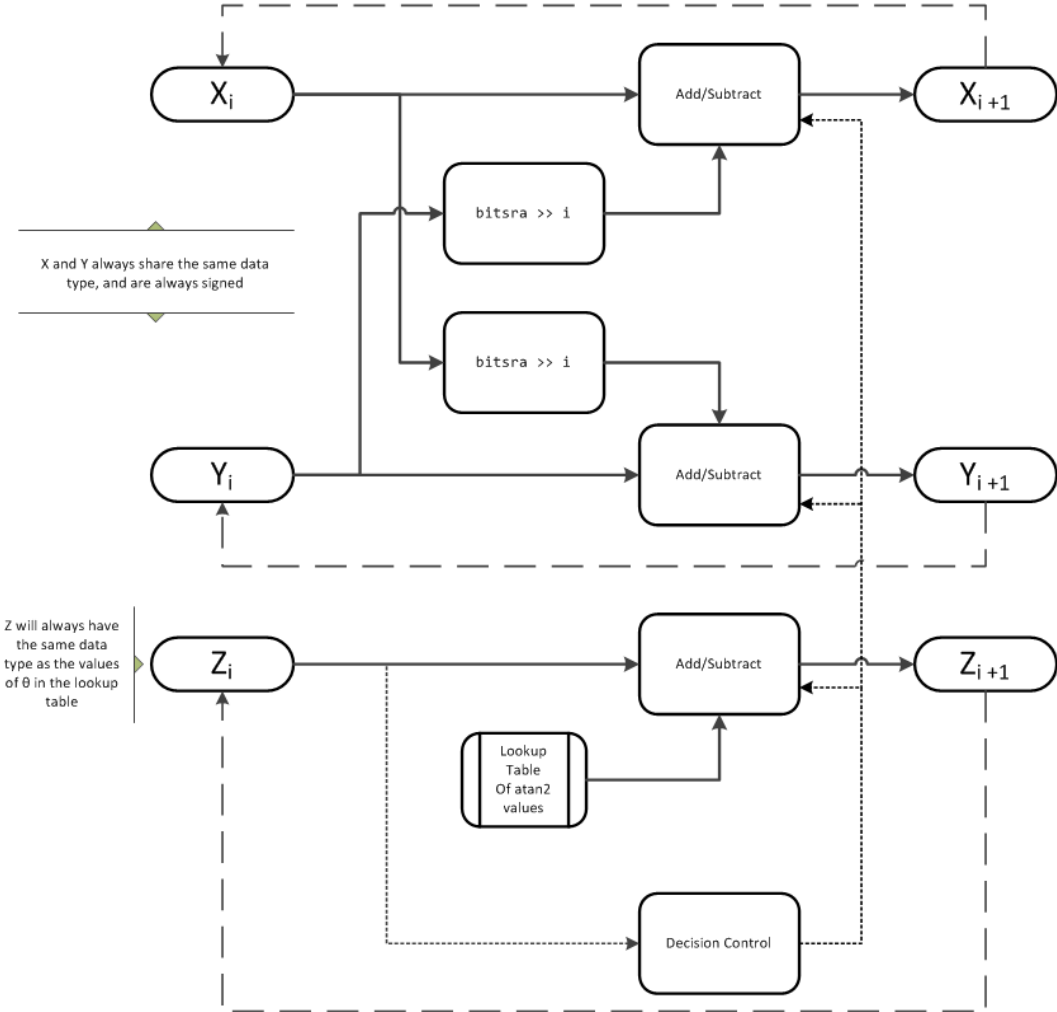
[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

## 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,  $r$  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`.

## **See Also**

`cordicrotate` | `cordicsincos` | `pol2cart`

<b>Purpose</b>	Rotate input using CORDIC-based approximation
<b>Syntax</b>	<pre>v = cordicrotate(theta,u) v = cordicrotate(theta,u,niters) v = cordicrotate(theta,u,Name,Value) v = cordicrotate(theta,u,niters,Name,Value)</pre>
<b>Description</b>	<p><code>v = cordicrotate(theta,u)</code> rotates the input <code>u</code> by <code>theta</code> using a CORDIC algorithm approximation. The function returns the result of <code>u .* e^(j*theta)</code>.</p> <p><code>v = cordicrotate(theta,u,niters)</code> performs <code>niters</code> iterations of the algorithm.</p> <p><code>v = cordicrotate(theta,u,Name,Value)</code> scales the output depending on the Boolean value, <code>b</code>.</p> <p><code>v = cordicrotate(theta,u,niters,Name,Value)</code> specifies both the number of iterations and the <code>NAME,Value</code> pair for whether to scale the output.</p>
<b>Input Arguments</b>	<p><b>theta</b></p> <p><code>theta</code> can be a signed or unsigned scalar, vector, matrix, or <math>N</math>-dimensional array containing the angle values in radians. All values of <code>theta</code> must be in the range <math>[-2\pi, 2\pi)</math>.</p> <p><b>u</b></p> <p><code>u</code> can be a signed or unsigned scalar value or have the same dimensions as <code>theta</code>. <code>u</code> can be real or complex valued.</p> <p><b>niters</b></p> <p><code>niters</code> is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, <code>niters</code> must be a positive, integer-valued scalar. If you do not specify <code>niters</code>, 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</p>

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`.



**Definitions****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.

**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-----\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');

```

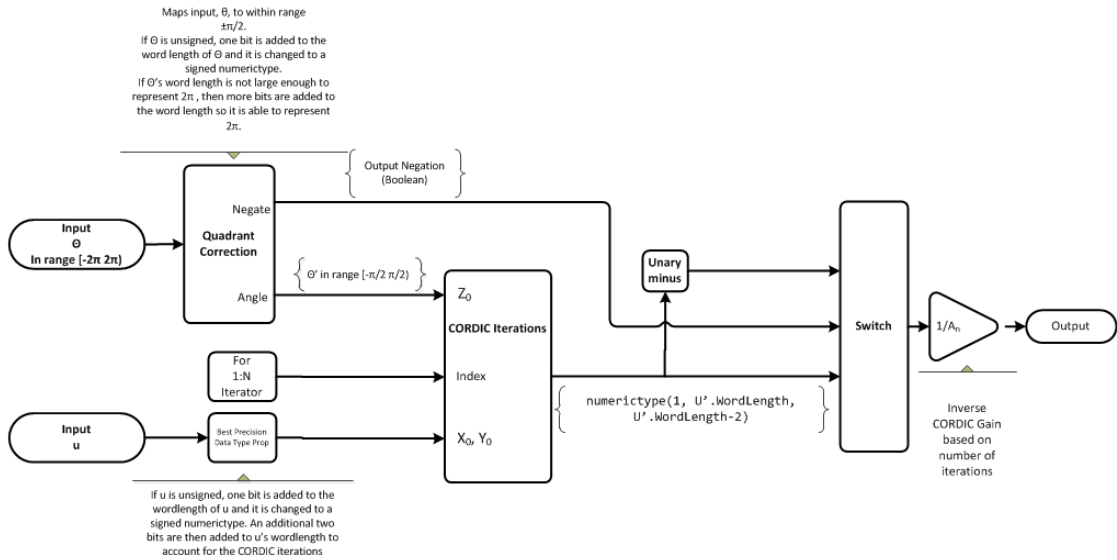
The output table appears as follows:

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

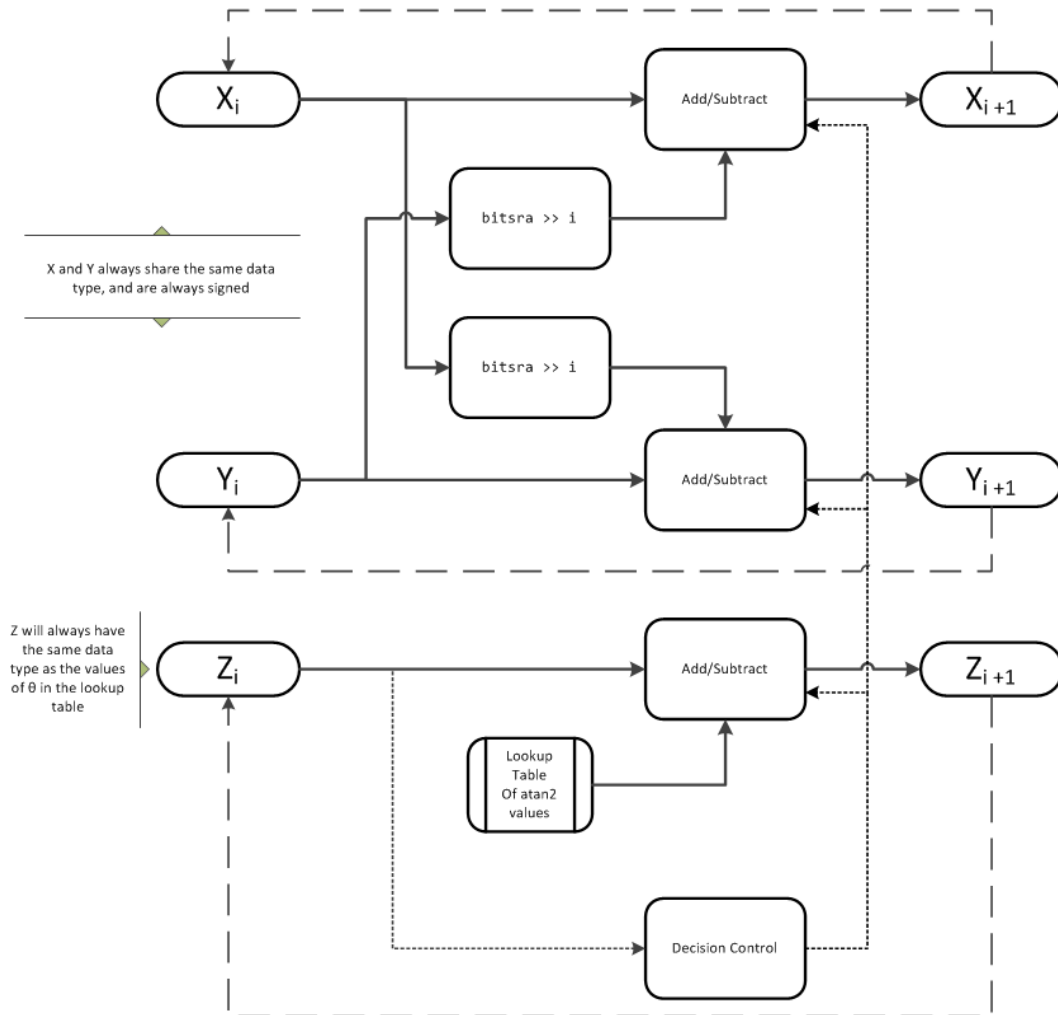
## References

- [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.

## 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`.

### **See Also**

`cordicpol2cart` | `cordicexp`

# cordicsin

---

<b>Purpose</b>	CORDIC-based approximation of sine
<b>Syntax</b>	<code>y = cordicsin(theta, niters)</code>
<b>Description</b>	<code>y = cordicsin(theta, niters)</code> computes the sine of <code>theta</code> using a “CORDIC” on page 2-297 algorithm approximation.
<b>Input Arguments</b>	<p><b>theta</b></p> <p><code>theta</code> can be a signed or unsigned scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of <code>theta</code> must be real and in the range <math>[-2\pi, 2\pi]</math>.</p> <p><b>niters</b></p> <p><code>niters</code> is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, <code>niters</code> must be a positive, integer-valued scalar. If you do not specify <code>niters</code> 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 <code>theta</code>. 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.</p>
<b>Output Arguments</b>	<p><b>y</b></p> <p><code>y</code> is the CORDIC-based approximation of the sine of <code>theta</code>. 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 <code>WordLength - 2</code>.</p>
<b>Definitions</b>	<p><b>CORDIC</b></p> <p>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</p>

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.

## 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

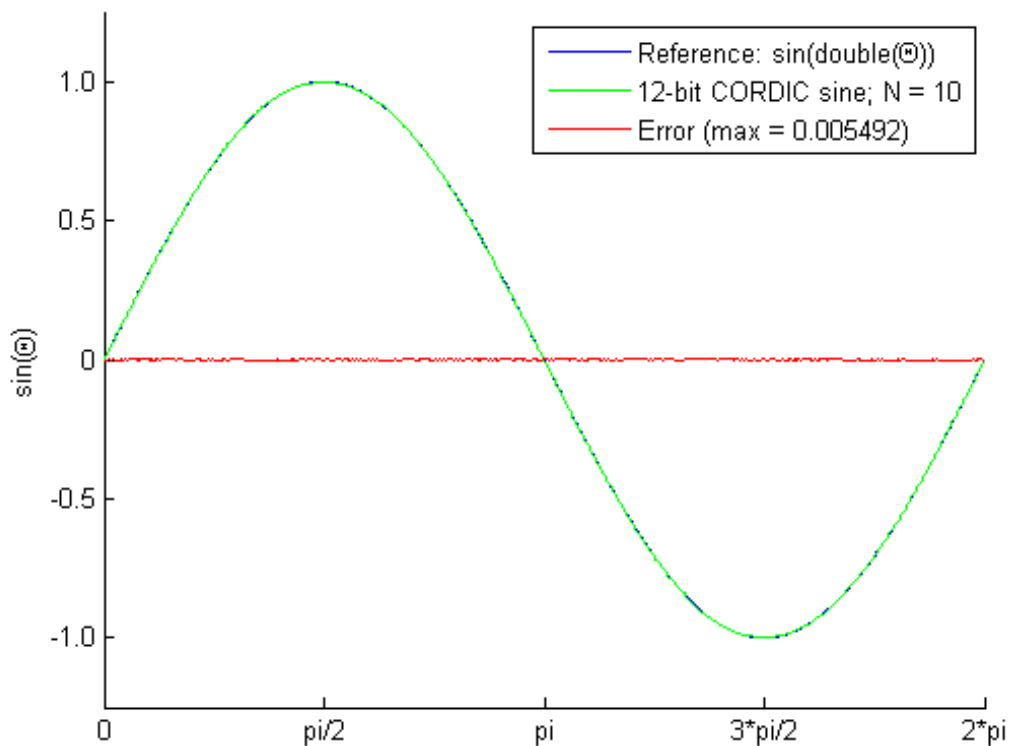
% Use 12-bit quantized inputs and vary the number of iterations
% from 2 to 10.
% Compare the fixed-point cordicsin function results to the
% results of the double-precision sin function.

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)');
set(gca, 'XTick', 0:pi/2:2*pi);
set(gca, 'XTickLabel', {'0', 'pi/2', 'pi', '3*pi/2', '2*pi'});
```

# cordicsin

```
set(gca,'YTick',-1:0.5:1);
set(gca,'YTickLabel',{'-1.0','-0.5','0','0.5','1.0'});
ref_str = 'Reference: sin(double(\Theta))';
cdc_str = sprintf('12-bit 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.



## References

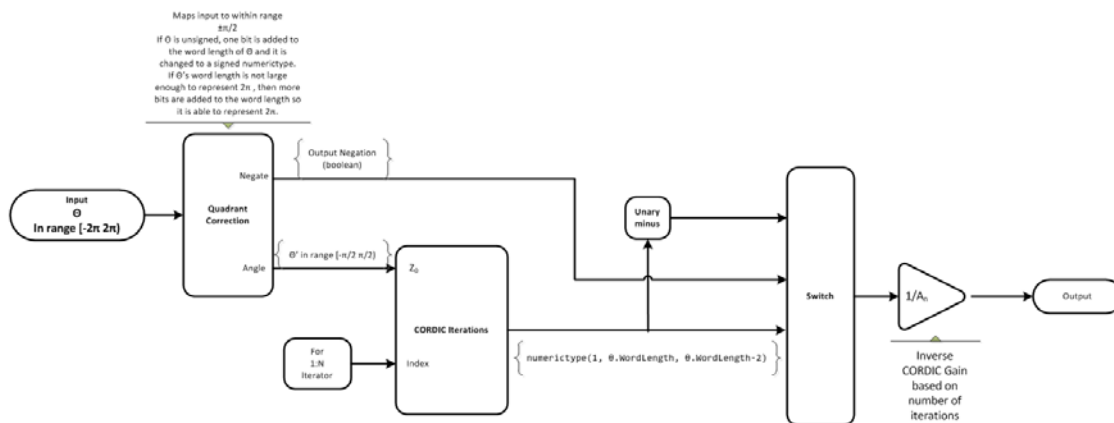
[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[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.

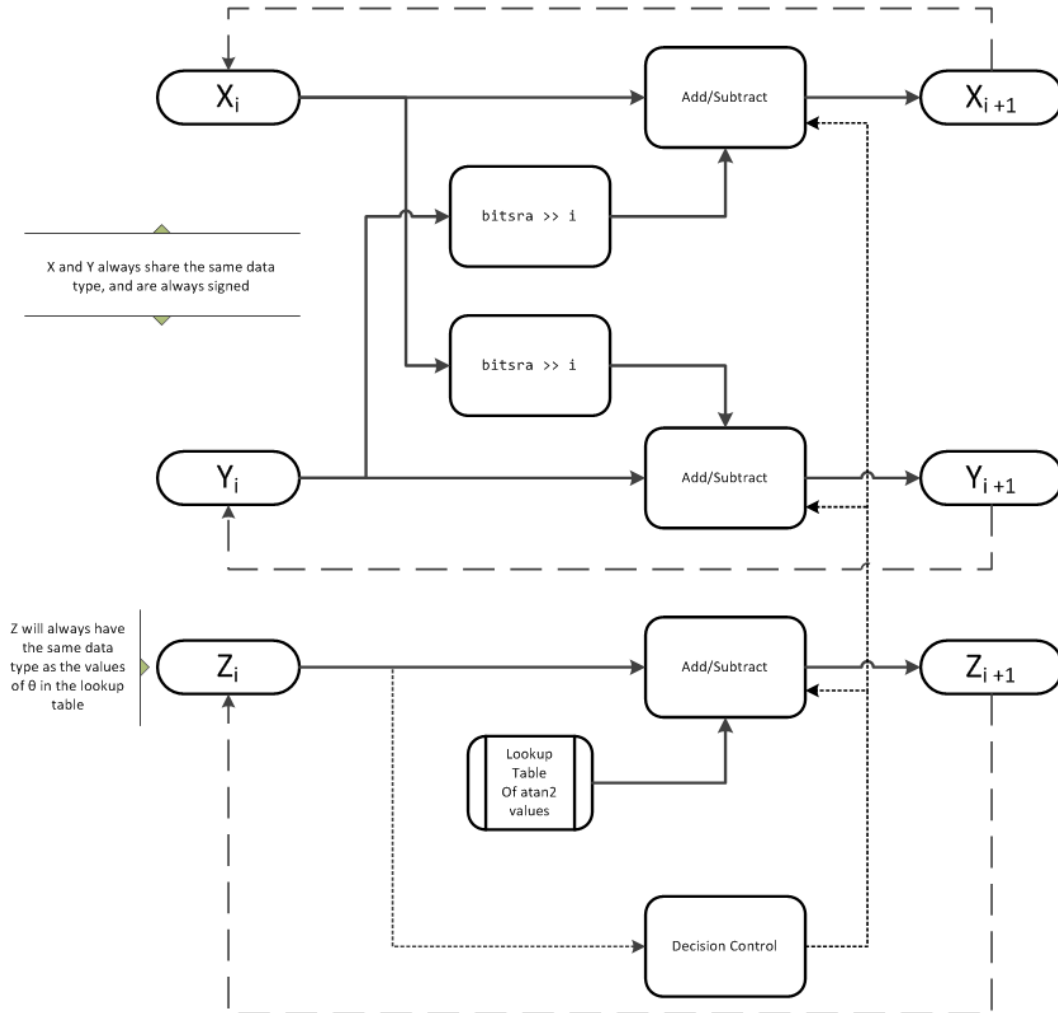
[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum).  
[www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

## 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`.

### **See Also**

`cordiccxp` | `cordiccos` | `cordicsincos` | `sin` | `cos`

### **Tutorials**

- Demo: Fixed-Point Sine and Cosine Calculation
- Demo: Fixed-Point Arctangent Calculation

# cordicsincos

---

**Purpose**                   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 2-297 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`.

## Definitions

### 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.

## 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

## References

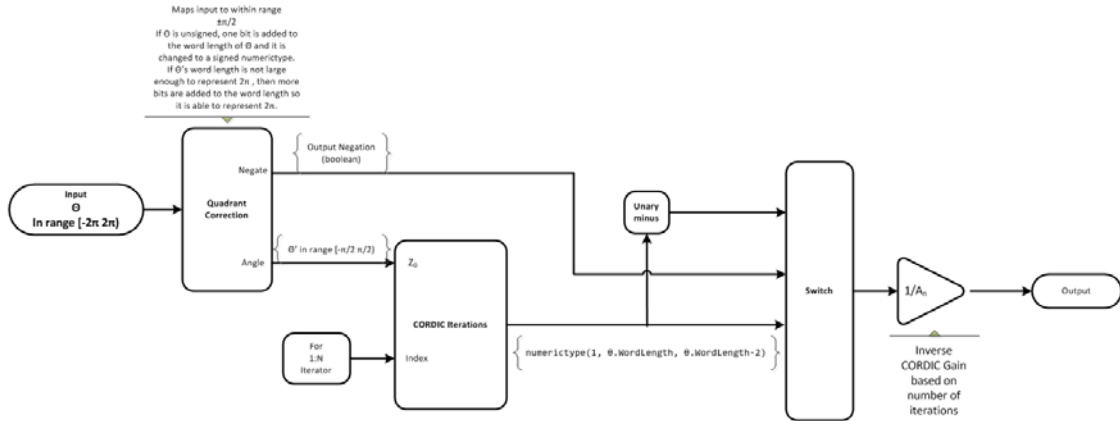
[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[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.

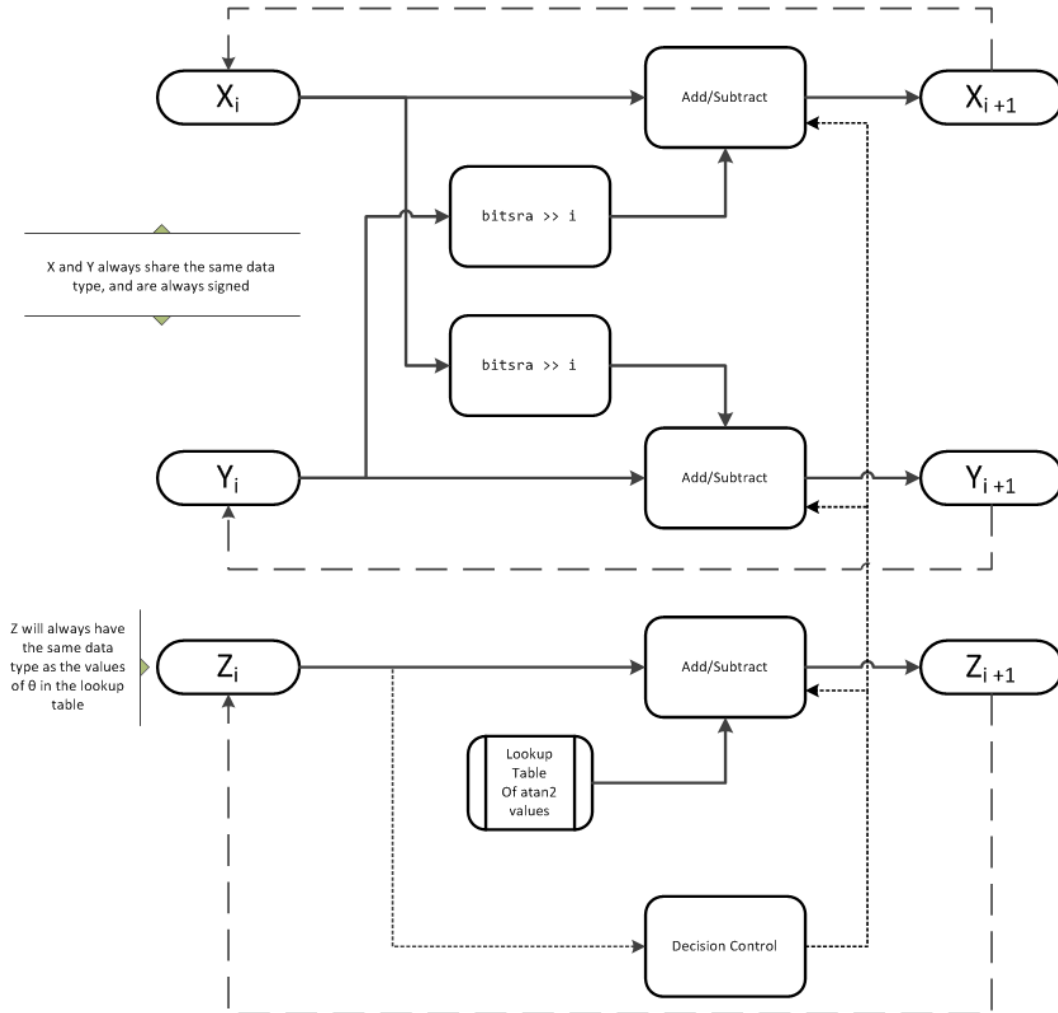
[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). [www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf](http://www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf)

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

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`.

### **See Also**

`cordicexp` | `cordiccos` | `cordicsin`

### **Tutorials**

- Demo: Fixed-Point Sine and Cosine Calculation
- Demo: Fixed-Point Arctangent Calculation

# cordicsqrt

---

**Purpose** 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`.

## 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 2-300.

**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 single 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.

## 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

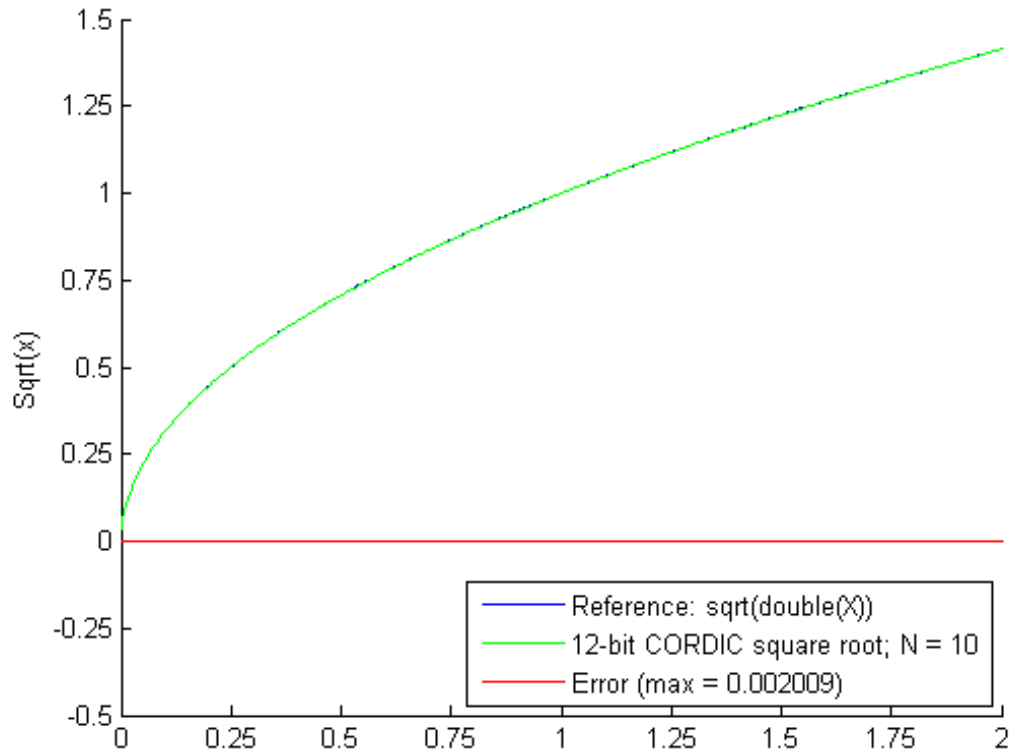
Compare the results produced by 10 iterations of the `cordicsqrt` algorithm to the results of the double-precision `sqrt` function.

```
% 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.

nitters = 10;
cdcSqrtX = cordicsqrt(XFxp, nitters);
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)');
 set(gca, 'XTick', 0:0.25:2);
 set(gca, 'XTickLabel', {'0', '0.25', '0.5', '0.75', '1', '1.25', '1.5', '1.75'});
 set(gca, 'YTick', -.5:.25:1.5);
 set(gca, 'YTickLabel', {'-0.5', '-0.25', '0', '0.25', '0.5', '0.75', '1', '1.25'});
 ref_str = 'Reference: sqrt(double(X))';
 cdc_str = sprintf('12-bit CORDIC square root; N = %d', nitters);
 err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
 legend(ref_str, cdc_str, err_str, 'Location', 'southeast');
```



## Definitions

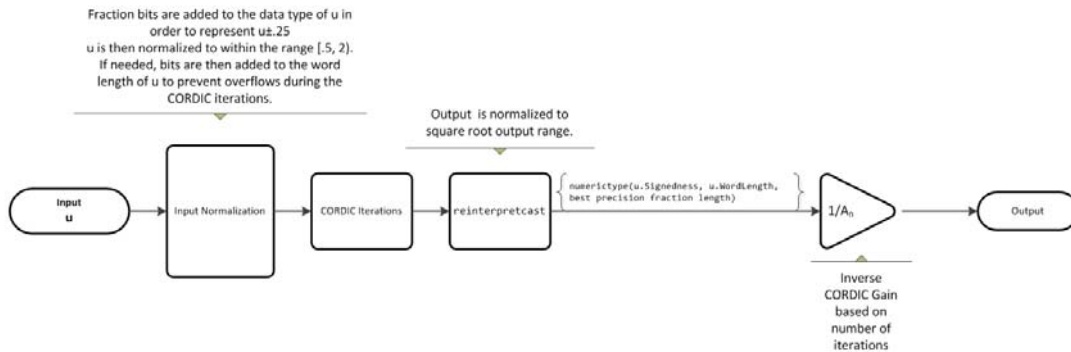
### CORDIC

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.

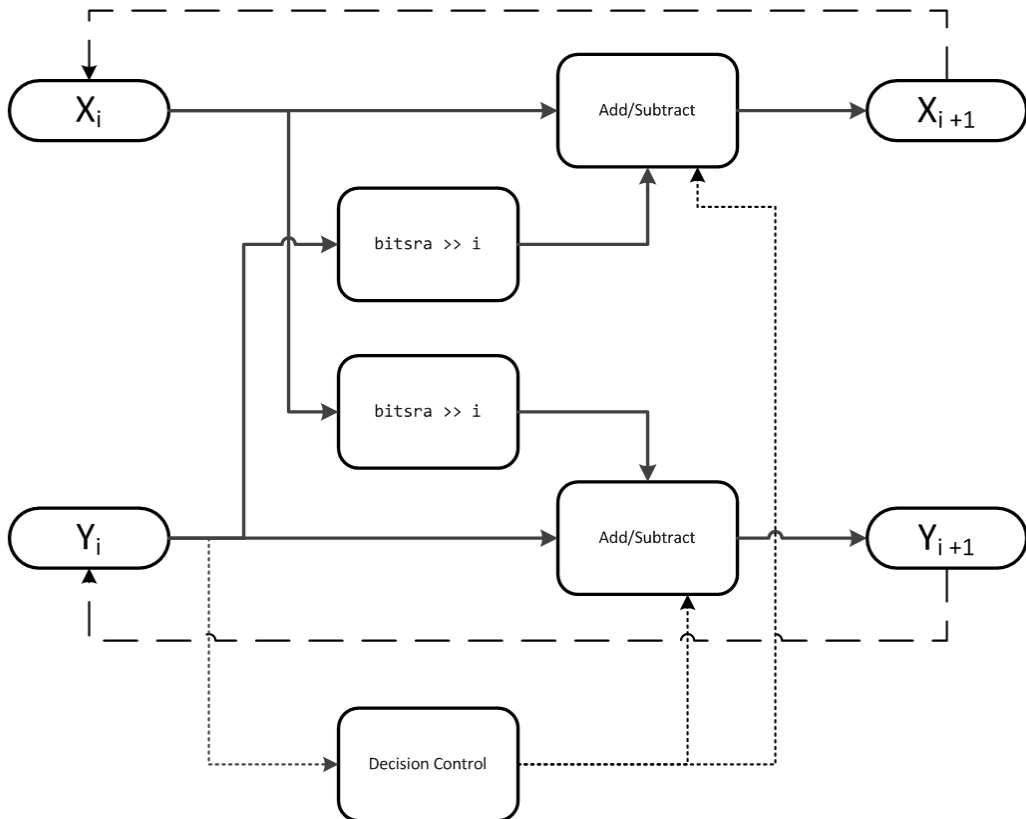
## Algorithms      Signal Flow Diagrams



For further details on the pre- and post-normalization process, see “Pre- and Post-Normalization” on page 2-300.



## 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.

**See Also**

sqrt

**Related Examples**

- “Compute Square Root Using CORDIC”

**Purpose** Cosine of `fi` object

**Syntax** `y = cos(theta)`

**Description** `y = cos(theta)` returns the cosine 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)`). This cosine calculation is accurate only to within the top 16 most-significant bits of the input.

**Definitions**

**Cosine**

The cosine of angle  $\Theta$  is defined as

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}$$

**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

```

## Algorithms

The cos function computes the cosine of fixed-point input using an 8-bit lookup table as follows:

- 1** Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
- 2** Perform a modulo  $2\pi$ , so the input is in the range  $[0, 2\pi)$  radians.
- 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`.

## **See Also**

`cos` | `angle` | `sin` | `atan2` | `cordiccos` | `cordicsin`

## **Related Examples**

- Demo: Fixed-Point Sine and Cosine Calculation

- Purpose** Complex conjugate transpose of `fi` object
- Syntax** `ctranspose(a)`
- Description** `ctranspose(a)` returns the complex conjugate transpose of `fi` object `a`. It is also called for the syntax `a'`.
- See Also** `transpose`

# dec

---

**Purpose** 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 string. `dec(a)` is equivalent to `a.dec`.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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 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` | `storedInteger` | `oct` | `sdec`



**Purpose** 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`

# denormalmin

---

**Purpose** 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`

**Purpose** Diagonal matrices or diagonals of matrix

**Description** Refer to the MATLAB `diag` reference page for more information.

# disp

---

**Purpose**            Display object

**Description**       Refer to the MATLAB disp reference page for more information.

---

<b>Purpose</b>	Divide two objects
<b>Syntax</b>	<code>c = divide(T,a,b)</code>
<b>Description</b>	<p><code>c = divide(T,a,b)</code> performs division on the elements of <code>a</code> by the elements of <code>b</code>. The result <code>c</code> has the <code>numericType</code> object <code>T</code>.</p> <p>If <code>a</code> and <code>b</code> are both <code>fi</code> objects, <code>c</code> has the same <code>fimath</code> object as <code>a</code>. If <code>c</code> has a <code>fi</code> Fixed data type, and any one of the inputs have <code>fi</code> floating point data types, then the <code>fi</code> floating point is converted into a fixed-point value. Intermediate quantities are calculated using the <code>fimath</code> object of <code>a</code>. See “Data Type Propagation Rules” on page 2-311.</p> <p><code>a</code> and <code>b</code> must have the same dimensions unless one is a scalar. If either <code>a</code> or <code>b</code> is scalar, then <code>c</code> has the dimensions of the nonscalar object.</p> <p>If either <code>a</code> or <code>b</code> is a <code>fi</code> object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the <code>fi</code> object, preserving best-precision fraction length. Intermediate quantities are calculated using the <code>fimath</code> object of the input <code>fi</code> object. See “Data Type Propagation Rules” on page 2-311.</p> <p>If <code>a</code> and <code>b</code> are both MATLAB built-in doubles, then <code>c</code> is the floating-point quotient <code>a./b</code>, and <code>numericType T</code> is ignored.</p>

---

**Note** The `divide` function is not currently supported for [Slope Bias] signals.

---

## Data Type Propagation Rules

For syntaxes for which Fixed-Point Designer software uses the `numericType` object `T`, the `divide` 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.

# divide

Data Type of Input <code>fi</code> Objects <code>a</code> and <code>b</code>		Data Type of numerictype object <code>T</code>	Data Type of Output <code>c</code>
Built-in double	Built-in double	Any	Built-in double
<code>fi</code> Fixed	<code>fi</code> Fixed	<code>fi</code> Fixed	Data type of numerictype object <code>T</code>
<code>fi</code> Fixed	<code>fi</code> Fixed	<code>fi</code> double	<code>fi</code> double
<code>fi</code> Fixed	<code>fi</code> Fixed	<code>fi</code> single	<code>fi</code> single
<code>fi</code> Fixed	<code>fi</code> Fixed	<code>fi</code> ScaledDouble	<code>fi</code> ScaledDouble with properties of numerictype object <code>T</code>
<code>fi</code> double	<code>fi</code> double	<code>fi</code> Fixed	<code>fi</code> double
<code>fi</code> double	<code>fi</code> double	<code>fi</code> double	<code>fi</code> double
<code>fi</code> double	<code>fi</code> double	<code>fi</code> single	<code>fi</code> single
<code>fi</code> double	<code>fi</code> double	<code>fi</code> ScaledDouble	<code>fi</code> double
<code>fi</code> single	<code>fi</code> single	<code>fi</code> Fixed	<code>fi</code> single
<code>fi</code> single	<code>fi</code> single	<code>fi</code> double	<code>fi</code> double
<code>fi</code> single	<code>fi</code> single	<code>fi</code> single	<code>fi</code> single
<code>fi</code> single	<code>fi</code> single	<code>fi</code> ScaledDouble	<code>fi</code> single
<code>fi</code> ScaledDouble	<code>fi</code> ScaledDouble	<code>fi</code> Fixed	<code>fi</code> ScaledDouble with properties of numerictype object <code>T</code>

Data Type of Input <i>fi</i> Objects <i>a</i> and <i>b</i>		Data Type of numeric type object <i>T</i>	Data Type of Output <i>c</i>
<i>fi</i> ScaledDouble	<i>fi</i> ScaledDouble	<i>fi</i> double	<i>fi</i> double
<i>fi</i> ScaledDouble	<i>fi</i> ScaledDouble	<i>fi</i> single	<i>fi</i> single
<i>fi</i> ScaledDouble	<i>fi</i> ScaledDouble	<i>fi</i> ScaledDouble	<i>fi</i> ScaledDouble with properties of numeric type object <i>T</i>

## Examples

This example highlights the precision of the *fi* divide function.

First, create an unsigned *fi* object with an 80-bit word length and  $2^{83}$  scaling, which puts the leading 1 of the representation into the most significant bit. Initialize the object with double-precision floating-point value 0.1, and examine the binary representation:

```
P = ...
fipref('NumberDisplay','bin',...
 'NumericTypeDisplay','short',...
 'FimathDisplay','none');
a = fi(0.1, false, 80, 83)
```

```
a =

1100110011001100110011001100110011001100110011001100110011001101010000
00000000000000000000000000000000
 u80,83
```

Notice that the infinite repeating representation is truncated after 52 bits, because the mantissa of an IEEE standard double-precision floating-point number has 52 bits.

# divide

---

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 =

110011001100110011001100110011001100110011001100110011001100110011001100
110011001100110011001100
```

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 IEEE standard double-precision floating-point number can.

With 1000 bits of precision,

```
T = numerictype('Signed',false,'WordLength',1000,...
 'FractionLength',1003);
a = fi(1);
b = fi(10);
c = divide(T,a,b);
```





# double

---

**Purpose** 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

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

or, equivalently as

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

## Examples

The code

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

returns

```
y =

 -1 0.9922
z =

 -1 0.9922
```

**See Also** `single`

- Purpose** Fixed-point numeric object
- Description** Use the `fi` function to create an `embedded.fi` object.
- See Also** `fi` | `embedded.fimath` | `embedded.numerictype`
- Concepts**
- Class Attributes
  - Property Attributes

# embedded.fimath

---

**Purpose**            `fimath` object

**Description**      Use the `fimath` function to create an `embedded.fimath` object.

**See Also**            `fimath` | `embedded.fi` | `embedded.numericType`

**Concepts**

- Class Attributes
- Property Attributes

**Purpose** numerictype object

**Description** Use the numerictype function to create an embedded.numerictype object.

**See Also** numerictype | embedded.fi | embedded.fimath

**Concepts**

- Class Attributes
- Property Attributes

# end

---

**Purpose** Last index of array

**Description** Refer to the MATLAB end reference page for more information.

<b>Purpose</b>	Quantized relative accuracy for <code>fi</code> or quantizer objects
<b>Syntax</b>	<code>eps(obj)</code>
<b>Description</b>	<code>eps(obj)</code> returns the value of the least significant bit of the value of the <code>fi</code> object or quantizer object <code>obj</code> . The result of this function is equivalent to that given by the Fixed-Point Designer function <code>lsb</code> .
<b>See Also</b>	<code>intmax</code>   <code>intmin</code>   <code>lowerbound</code>   <code>lsb</code>   <code>range</code>   <code>realmax</code>   <code>realmin</code>   <code>upperbound</code>

**Purpose** 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.

**See Also** `ge` | `gt` | `isequal` | `le` | `lt` | `ne`



**Purpose** 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`

# errorbar

---

**Purpose** Plot error bars along curve

**Description** Refer to the MATLAB errorbar reference page for more information.

**Purpose** 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`.

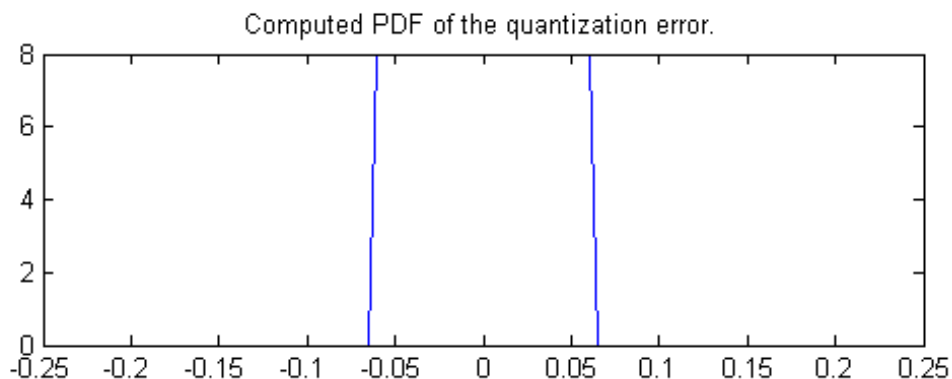
---

**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**

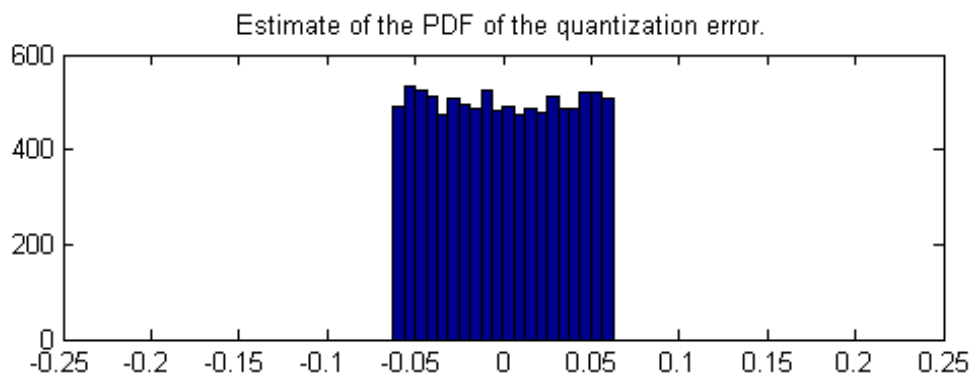
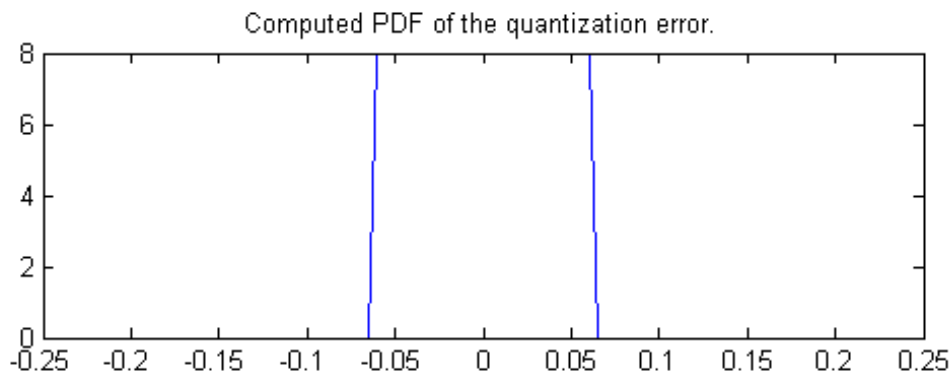
```
q = quantizer('nearest',[4 3]);
[f,x] = errpdf(q);
subplot(211)
plot(x,f)
title('Computed PDF of the quantization error.')
```



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);set(gca,'xlim',[min(x) max(x)])
```

```
title('Estimate of the PDF of the quantization error.')
```



**See Also** [errmean](#) | [errvar](#) | [quantize](#)

# errvar

---

**Purpose** 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`

**Purpose** Plot elimination tree

**Description** Refer to the MATLAB `etreeplot` reference page for more information.

# exponentbias

---

**Purpose** 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`



**Purpose** 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`

# exponentmax

---

**Purpose** 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`

**Purpose** 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`

## ezcontour

---

**Purpose** Easy-to-use contour plotter

**Description** Refer to the MATLAB ezcontour reference page for more information.

**Purpose** Easy-to-use filled contour plotter

**Description** Refer to the MATLAB ezcontourf reference page for more information.

# ezmesh

---

**Purpose** Easy-to-use 3-D mesh plotter

**Description** Refer to the MATLAB ezmesh reference page for more information.

**Purpose** Easy-to-use function plotter

**Description** Refer to the MATLAB ezplot reference page for more information.

# ezplot3

---

**Purpose** Easy-to-use 3-D parametric curve plotter

**Description** Refer to the MATLAB `ezplot3` reference page for more information.



**Purpose** Easy-to-use polar coordinate plotter

**Description** Refer to the MATLAB `ezpolar` reference page for more information.

# ezsurf

---

**Purpose** Easy-to-use 3-D colored surface plotter

**Description** Refer to the MATLAB `ezsurf` reference page for more information.

**Purpose** Easy-to-use combination surface/contour plotter

**Description** Refer to the MATLAB `ezsurf` reference page for more information.

# feather

---

**Purpose** Plot velocity vectors

**Description** Refer to the MATLAB `feather` reference page for more information.

**Purpose**

Construct fixed-point numeric object

**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,fixexponent,bias)
a = fi(v,T)
a = fi(v,F)
b = fi(a,F)
a = fi(v,T,F)
a = fi(v,s,F)
a = fi(v,s,w,F)
a = fi(v,s,w,f,F)
a = fi(v,s,w,slope,bias,F)
a = fi(v,s,w,slopeadjustmentfactor,fixexponent,bias,F)
a = fi(... 'PropertyName',PropertyValue...)
a = fi('PropertyName',PropertyValue...)
```

**Description**

You can use the `fi` constructor function in the following ways:

- `a = fi` is the default constructor and returns a `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = fi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = fi(v,s)` returns a fixed-point object with value `v`, Signed property value `s`, 16-bit word length, and best-precision fraction length. `s` can be 0 (false) for unsigned or 1 (true) for signed.
- `a = fi(v,s,w)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, and best-precision fraction length.
- `a = fi(v,s,w,f)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, and fraction length `f`. Fraction length can be greater than word length or negative, see “Create a `fi`

Object With Fraction Length Greater Than Word Length” on page 2-350 and “Create a fi Object With Negative Fraction Length” on page 2-352.

- `a = fi(v,s,w,slope,bias)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, `slope`, and `bias`.
- `a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.
- `a = fi(v,T)` returns a fixed-point object with value `v` and `embedded.numericitytype T`. Refer to “numericitytype Object Construction” for more information on `numericitytype` objects.
- `a = fi(v,F)` returns a fixed-point object with value `v`, `embedded.fimath F`, 16-bit word length, and best-precision fraction length. Refer to “fimath Object Construction” for more information on `fimath` objects.
- `b = fi(a,F)` allows you to maintain the value and `numericitytype` object of `fi` object `a`, while changing its `fimath` object to `F`.
- `a = fi(v,T,F)` returns a fixed-point object with value `v`, `embedded.numericitytype T`, and `embedded.fimath F`. The syntax `a = fi(v,T,F)` is equivalent to `a = fi(v,F,T)`.
- `a = fi(v,s,F)` returns a fixed-point object with value `v`, Signed property value `s`, 16-bit word length, best-precision fraction length, and `embedded.fimath F`.
- `a = fi(v,s,w,F)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, best-precision fraction length, and `embedded.fimath F`.
- `a = fi(v,s,w,f,F)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, fraction length `f`, and `embedded.fimath F`.
- `a = fi(v,s,w,slope,bias,F)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, `slope`, `bias`, and `embedded.fimath F`.

- `a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias,F)` returns a fixed-point object with value `v`, Signed property value `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, `bias`, and embedded `fi` math `F`.
- `a = fi(...'PropertyName',PropertyValue...)` and `a = fi('PropertyName',PropertyValue...)` allow you to set fixed-point objects for a `fi` object by property name/property value pairs.

The `fi` object has the following three general types of properties:

- “Data Properties” on page 2-345
- “`fi`math Properties” on page 2-346
- “`numeric`type Properties” on page 2-347

---

**Note** These properties are described in detail in “`fi` Object Properties” on page 1-2 in the Properties Reference.

---

## 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 1-2.

### **fimath Properties**

When you create a `fi` object and specify `fimath` object properties in the `fi` constructor, a `fimath` object is created as a property of the `fi` object. If you do not specify any `fimath` properties in the `fi` constructor, the resulting `fi` has no attached `fimath` object.

- `fimath` — `fimath` properties associated with a `fi` object

The following `fimath` properties are, by transitivity, also properties of a `fi` object. The properties of the `fimath` object listed below are always writable.

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

---

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

---

- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow mode
- `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 mode
- `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
- `DataTypeOverride` — Data type override for applying `fipref` data type override settings to `fi` objects. This property provides a

convenient way to ignore a global `fipref` data type override setting. Note that this property is not visible when its value is the default, `Inherit`. When this property is set to `Off`, the `fi` object uses the `numerictype` data type settings and ignores `fipref` settings.

- `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”.

---

### Create a fi Object

Create a signed `fi` object with a value of `pi`, a word length of 8 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

```
a = fi((magic(3)/10), 1, 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
```

### Create a fi Object With Default Precision

If you omit the argument `f`, the fraction length is set automatically to achieve the best precision possible.

```
a = fi(pi, 1, 8)
```

```
a =

 3.1563

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

### **Create a fi Object With Default Word Length and Precision**

If you omit *w* and *f*, the word length is set automatically to 16 bits and the fraction length is set to achieve the best precision possible.

```
a = fi(pi, 1)

a =

 3.1416

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

### **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. We can calculate the real-world value.

```
RealWorldValue = StoredInteger * 2 ^ -FractionLength
RealWorldValue = 5 * 2 ^ -10 = 0.0048828125
```

---

Create a signed `fi` object with a value of `0.0048828125`, a word length of 8 bits, and a fraction length of 10 bits.

```
a = fi(0.0048828125, true, 8, 10)
```

```
a =
```

```
0.004882812500000
```

```
 DataTypeMode: Fixed-point: binary point scaling
 Signedness: Signed
 WordLength: 8
 FractionLength: 10
```

Get the stored integer value of `a`.

```
a.int
```

```
ans =
```

```
5
```

Get the binary value of the stored integer.

```
a.bin
```

```
ans =
```

```
00000101
```

Because the fraction length is 2 bits longer than the word length, the binary value of the stored integer is `x.xx00000101`, where `x` is a placeholder for implicit zeros. `0.0000000101` (binary) is equivalent to `0.0048828125` (decimal).

## 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 value with a word length of 8, fraction length of  $-2$  and a stored integer value of 5. We can calculate the real-world value.

```
RealWorldValue = StoredInteger * 2 ^ -FractionLength
RealWorldValue = 5 * 2 ^ 2 = 20
```

Create a signed `fi` object with a value of 20, a word length of 8 bits, and a fraction length of  $-2$  bits.

```
a = fi(20, true, 8, -2)
```

```
a =
```

```
20
```

```
 DataTypeMode: Fixed-point: binary point scaling
 Signedness: Signed
 WordLength: 8
 FractionLength: 10
```

Get the stored integer value of `a`.

```
a.int
```

```
ans =
```

```
5
```

Get the binary value of the stored integer.

```
a.bin
```

```
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. 000000010100 (binary) is equivalent to 20 (decimal).

### Create a fi Object Specifying Rounding and Overflow

You can use property name/property value pairs to set fi properties, such as rounding method and overflow action, when you create the 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
```

### Remove Local fimath

You can remove a local fimath object from a fi object at any time using the removefimath function.

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

```
a =
```

```
3.1415
```

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

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

```
a =
 3.1415
```

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

**fi** object **a** now has no local **fimath**. To reassign it a local **fimath** object, use the **setfimath** function.

```
a = setfimath(a, fimath('ProductMode', 'KeepLSB'))
```

```
a =
 3.1415
```

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

```
 RoundingMethod: Nearest
 OverflowAction: Saturate
 ProductMode: KeepLSB
 ProductWordLength: 32
 SumMode: FullPrecision
```



`fi` object `a` now has a local `fi` object with a `ProductMode` of `KeepLSB`. The values of the remaining `fi` object properties are default `fi` values.

### Use `fi` as an Indexing Argument

Set up an array to be indexed.

```
x = 10:-1:1
x =
 10 9 8 7 6 5 4 3 2 1
```

Create a `fi` object and use it to index into `x`.

```
k = fi(3);
y = x(k)

y =
 8
```

### Use `fi` in a Switch Statement

You can use a `fi` object as the switch condition and as one or more of the cases in the switch expression.

```
function y = test_switch(u, v)
 cExpr = fi(u + v, 0, 2, 0);
 t = 1;

 switch cExpr % condition expression type: ufix2
 case 0
 y = t * 2;
 case fi(1,0,2,0)
 y = t * 3;
 case 2
 y = t * 4;
 case 3
```

```
 y = t * 3;
 otherwise
 y = 0;
 end
end

y = test_switch(1,2.0)

y =
 3
```

### **Use fi as a Colon Operator**

Use a `fi` object 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 =

 1 3 5 7 9 11

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

### **Create Fixed-point Vector With Non-integer Spacing**

To create a fixed-point vector with non-integer spacing, first create the vector, then cast the vector to fixed-point.

```
x = fi(0:0.1:10);
```

Alternatively, use the `linspace` function.

```
x = fi(linspace(0,10, 101));
```

The following code, where one of the colon operands is not an integer, generates an error.

```
a = fi(0);
b = fi(0.1);
c = fi(10);
z = a:b:c
```

### Set Data Type Override on a fi Object

Set the `DataTypeOverride` property of a `fi` object so that the `fi` object does not use the data type override settings of the `fipref` object.

Set up `fipref` with data type override set to 'TrueDoubles' for all numeric types.

```
fipref('DataTypeOverride', 'TrueDoubles')
```

```
ans =
```

```
 NumberDisplay: 'RealWorldValue'
NumericTypeDisplay: 'full'
 FimathDisplay: 'full'
 LoggingMode: 'Off'
 DataTypeOverride: 'TrueDoubles'
DataTypeOverrideAppliesTo: 'AllNumericTypes'
```

Create a new `fi` object without specifying its `DataTypeOverride` property so that it uses the data type override settings specified using `fipref`.

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

```
x =
```

```
3.1416
```

```
DataTypeMode: Double
```

Now create a `fi` object and set its `DataTypeOverride` property to `'Off'` so that it ignores the data type override settings specified using `fipref`.

```
y = fi(pi, 1, 16, 13, 'DataTypeOverride','Off')
```

```
y =
```

```
3.1416
```

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

**See Also**

`fimath` | `fipref` | `isfimathlocal` | `numerictype` | `quantizer` | `sfi` | `ufi`

**Concepts**

- “fi Object Functions”
- “Binary Point Interpretation”

**Purpose** Accelerate fixed-point code

**Syntax** `fiaccel -options 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.

---

## 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. `fiaccel` 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.

`-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, `fiaccel` creates it for you.

If you do not specify the folder location, `fiaccel` 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, `fiaccel` 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.

`-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.

`-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.



<code>-O optimization_option</code>	<p>Optimize generated MEX code, based on the value of <code>optimization_option</code>:</p> <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> <p>If not specified, <code>fiaccl</code> uses inlining for optimization.</p>
<code>-report</code>	<p>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.</p>
<code>-?</code>	<p>Display help for <code>fiaccl</code> command.</p>

## Examples

Create a test file and compute the moving average. Then, use `fiaccl` 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
```

## See Also

[coder.ArrayType](#) | [coder.Constant](#) | [coder.EnumType](#) |  
[coder.FiType](#) | [coder.newtype](#) | [coder.PrimitiveType](#) |  
[coder.resize](#) | [coder.StructType](#) | [coder.Type](#) | [coder.typeof](#)

**Purpose**

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`.

**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.

# filter

---

## 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 `numericType` 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 `numericType` 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.

## Definitions

### ***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  $L-1$ .

## Examples

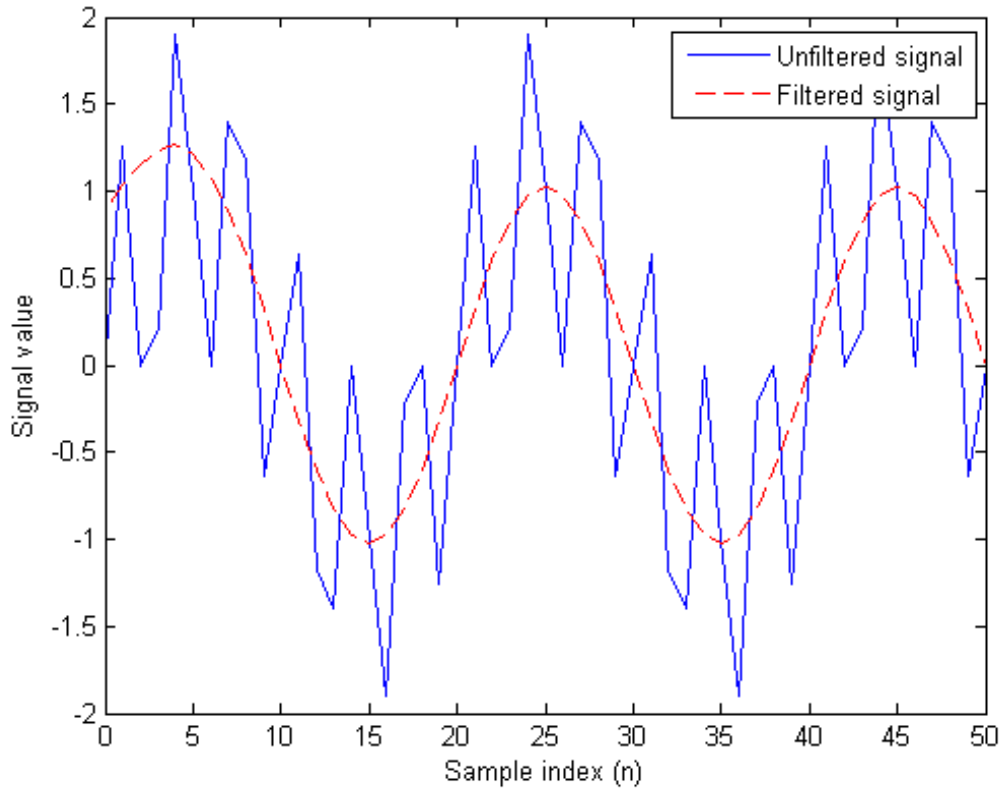
### Filter a high-frequency fixed-point sinusoid from a signal

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')
```

# filter



The resulting plot shows both the unfiltered and filtered signals.

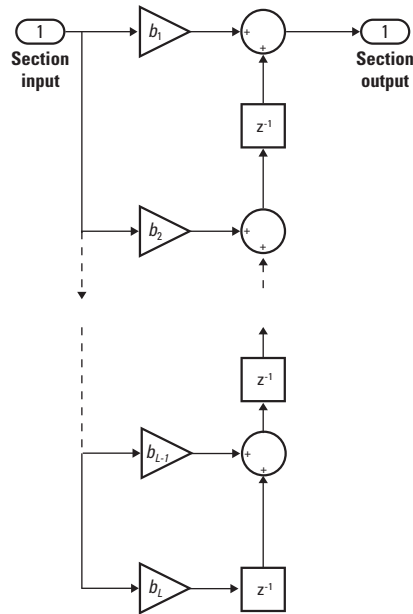
## Algorithms

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 and  $N$  is the filter order.

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`.

# filter

---

## See Also

[conv | filter](#)



**Purpose**

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 fmath Object

Set properties of a fmath object at the time of object creation by including properties after the arguments of the fmath constructor function. For example, set the overflow action to Saturate and the rounding method to Convergent.

```
F = fmath('OverflowAction','Saturate','RoundingMethod','Convergent')
```

```
F =
```

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

### See Also

[fi](#) | [fipref](#) | [globalfmath](#) | [numerictype](#) | [quantizer](#) | [removefmath](#) | [setfmath](#)

### Related Examples

- “fmath Object Construction”

### Concepts

- “fmath Object Properties”
- How Functions Use fmath
- “fmath Properties Usage for Fixed-Point Arithmetic”

**Purpose** 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 1-4.

- `FimathDisplay` — Display options for the local `fimath` attributes of `fi` objects. When `fi` objects to 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

## Related Examples

- “fipref Object Construction”

## **Concepts**

- “fipref Object Properties”

**Purpose** 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

### See Also

`ceil` | `convergent` | `floor` | `nearest` | `round`

# fixed.aggregateType

---

<b>Purpose</b>	Compute aggregate numerictype
<b>Syntax</b>	<code>aggNT = fixed.aggregateType(A,B)</code>
<b>Description</b>	<code>aggNT = fixed.aggregateType(A,B)</code> computes the smallest binary point scaled numerictype that is able to represent both the full range and precision of inputs A and B.
<b>Input Arguments</b>	<b>A</b> An integer, binary point scaled fixed-point <code>fi</code> object, or numerictype object. <b>B</b> An integer, binary point scaled fixed-point <code>fi</code> object, or numerictype object.
<b>Output Arguments</b>	<b>aggNT</b> A numerictype object.
<b>Examples</b>	Compute the aggregate numerictype of two numerictype objects.  <pre>% can represent range [-4,4) and precision 2^-13 a_nt = numerictype(true,16,13); % can represent range [-2,2) and precision 2^-16 b_nt = numerictype(true,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</pre>

---

Compute the aggregate numeric type 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 numeric type 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
```

# fixed.aggregateType

---

FractionLength: 14

**See Also**      `numericType | fi`

## Purpose

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`.

## 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`.

# fixed.Quantizer

---

- 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.

## Input Arguments

### **nt**

Binary-point, scaled numeric type object or slope-bias scaled, fixed-point numeric type object. If your `fixed.Quantizer` uses a numeric type 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

## **fi**

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 single 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. 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.

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.

# fixed.Quantizer

---

**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

## **Scaling**

Scaling mode of the object. The possible values of this property are:

- `BinaryPoint` — Scaling for the `fi` object is defined by the fraction length.
- `SlopeBias` — Scaling for the `fi` object is defined by the slope and bias.
- `Unspecified` — A temporary setting that is only allowed at `fi` object creation, to allow for the automatic assignment of a binary point best-precision scaling.

**Default:** `BinaryPoint`

## **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. 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

# fixed.Quantizer

---

## 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

## Definitions

### Fixed-point numbers

Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

## 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));
```

**See Also**      `fi | numerictype | quantizer`

**How To**      • “Set numerictype Object Properties”

# fixpt\_instrument\_purge

---

**Purpose** Remove corrupt fixed-point instrumentation from model

---

**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`

**Purpose**

Flip order of elements

**Description**

Refer to the MATLAB `flip` reference page for more information.

# flipdim

---

**Purpose** Flip array along specified dimension

**Description**

---

**Note** `flipdim` will be removed in a future release. Use `flip` instead.

---

Refer to the MATLAB `flipdim` reference page for more information.

**Purpose** Flip matrix left to right

**Description** Refer to the MATLAB `fliplr` reference page for more information.

# flipud

---

**Purpose** Flip matrix up to down

**Description** Refer to the MATLAB `flipud` reference page for more information.



**Purpose**

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 numeric type 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

**See Also**

`ceil` | `convergent` | `fix` | `nearest` | `round`

# fplot

---

**Purpose** Plot function between specified limits

**Description** Refer to the MATLAB `fplot` reference page for more information.

**Purpose** 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`

**Purpose** 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.

**See Also** `eq` | `gt` | `le` | `lt` | `ne`

**Purpose** 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 the string '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.

**See Also** `set`

# getlsb

---

**Purpose** 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
```

**See Also** `bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` | `bitxor` | `bitxorreduce` | `getmsb`



**Purpose** 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
```

**See Also** `bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` | `bitxor` | `bitxorreduce` | `getlsb`

# globalfimath

---

**Purpose** Configure global fimath and return handle object

**Syntax**

```
G = globalfimath
G = globalfimath('PropertyName1',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('PropertyName1',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.

**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.

<b>Goal</b>	<b>Issue Using globalfimath</b>	<b>Solution</b>
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”.
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 “Separate Data Type Definitions from Algorithm”.

# globalfimath

---

## See Also

`fimath` | `codegen` | `fiaccel` | `removeglobalfimathpref` |  
`resetglobalfimath`

**Purpose** Plot set of nodes using adjacency matrix

**Description** Refer to the MATLAB `gplot` reference page for more information.

**Purpose** 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.

**See Also** `eq` | `ge` | `le` | `lt` | `ne`



**Purpose** Hankel matrix

**Description** Refer to the MATLAB `hankel` reference page for more information.

# hdlram

---

**Purpose** Single, simple dual, or dual-port RAM for memory read/write access

**Description** hdlram reads from and writes to memory locations for a single, simple dual, or dual-port RAM. The output data is delayed one step.

The RAM size, or number of locations, is inferred from the data type of the address variable as shown in the following table.

Data type of address variable	RAM address size (bits)
single or double	16
uint <i>N</i>	<i>N</i>
embedded.fi	WordLength

The maximum RAM address size is 32 bits.

**Construction** `H = hdlram` creates a single port RAM System object. This object allows you to read from or write to a memory location in the RAM. The output data port corresponds to the read/write address passed in with the `step` method.

`H = hdlram(Name, Value)` creates a single, simple dual, or dual port hdlram System object, `H`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`. See “Properties” on page 2-412 for the list of available property names.

## Properties

### RAMType

Type of RAM to be created

Default: Single port

Specify the type of RAM to be created. Values of this property are:

## Single Port

Create a single port RAM, with 3 inputs and 1 output.

Inputs:

- Write Data
- Write address
- Write enable

Outputs: Read/Write Data

## Simple dual port

Create a simple dual-port RAM, with 4 inputs and 1 output.

Inputs:

- Write Data
- Write address
- Write enable
- Read address

Outputs: Read Data

## Dual port

Create a dual-port RAM, with 4 inputs and 2 outputs.

Inputs:

- Write Data
- Read/Write address
- Write enable
- Read address

Outputs:

- Write Data
- Read Data

## WriteOutputValue

Behavior for Write output

Default: New data

Specify the behavior for Write output for single-port and dual-port RAMs. Values of this property are:

New data	Send out new data at the address to the output.
Old data	Send out old data at the address to the output.

## Methods

clone	Construct hdlram System object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow changes to property values and input characteristics
step	Read or write input value to memory location

## Examples

### Create Single-Port RAM System Object

Construct System object to read from or write to a memory location in RAM.

The output data port corresponds to the read/write address passed in. During a write operation, the old data at the write address is sent out as the output.

```
H = hdlram('RAMType','Single port','WriteOutputValue','Old data')
```

```
H =
```

```
System: hdlram
```

```
Properties:
```

```
RAMType: 'Single port'
```

```
WriteOutputValue: 'Old data'
```

### Create Simple Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

The output data port corresponds to the read address. If a read operation is performed at the same address as the write operation, old data at that address is read out as the output.

```
H = hdlram('RAMType','Simple dual port')
```

```
H =
```

```
System: hdlram
```

```
Properties:
```

```
RAMType: 'Simple dual port'
```

### Create Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

There are two output ports, a write output data port and a read output data port. The write output data port sends out the new data at the write address. The read output data port sends out the old data at the read address.

```
H = hdlram('RAMType','Dual port','WriteOutputValue','New data')
```

```
H =

System: hdlram

Properties:
 RAMType: 'Dual port'
 WriteOutputValue: 'New data'
```

## Read/Write Single-Port RAM

Create System object that can write to a single port RAM, and read the newly written value out.

Construct single-port RAM System object.

```
hRAM = hdlram('RAMType','Single port','WriteOutputValue','New data');
```

Preallocate memory.

```
dataLength = 100;
[dataIn dataOut] = deal(zeros(1,dataLength));
```

Write randomly generated data to the System object, and then read data back out again.

```
for ii = 1:dataLength
 dataIn(ii) = randi([0 63],1,1,'uint8');
 addressIn = uint8(ii-1);
 writeEnable = true;
 dataOut(ii) = step(hRAM,dataIn(ii),addressIn,writeEnable);
end ;
```

## Related Examples

- “Create System Objects”
- “Set Up System Objects”
- “Process Data Using System Objects”
- “Tuning System object™ Properties in MATLAB”
- “Find Help and Examples for System Objects”

<b>Purpose</b>	Construct hdlram System object with same property values
<b>Syntax</b>	<code>C = clone(H)</code>
<b>Description</b>	<code>C = clone(H)</code> creates another instance of the System object, H, with the same property values. The clone method creates a new, unlocked object with uninitialized states.
<b>Input Arguments</b>	<b>H</b> Instance of hdlram
<b>Output Arguments</b>	<b>C</b> New hdlram System object with the same property values as the original System object. The clone method creates a new, unlocked object with uninitialized states.
<b>See Also</b>	<code>hdlram</code>   <code>hdlram.isLocked</code>   <code>hdlram.release</code>   <code>hdlram.step</code>

# hdlram.IsLocked

---

<b>Purpose</b>	Locked status for input attributes and nontunable properties
<b>Syntax</b>	<code>L = isLocked(OBJ)</code>
<b>Description</b>	<code>L = isLocked(OBJ)</code> returns a logical value, L, which indicates whether input attributes and nontunable properties are locked for the System object, OBJ. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After the System object is locked, the <code>isLocked</code> method returns a true value.
<b>Input Arguments</b>	<b>OBJ</b> Instance of <code>hdlram</code>
<b>Output Arguments</b>	<b>L</b> Logical value. Either 1 (true) or 0 (false).
<b>See Also</b>	<code>hdlram</code>   <code>hdlram.clone</code>   <code>hdlram.release</code>   <code>hdlram.step</code>



<b>Purpose</b>	Allow changes to property values and input characteristics
<b>Syntax</b>	<code>release(OBJ)</code>
<b>Description</b>	<code>release(OBJ)</code> releases system resources (such as memory, file handles, and hardware connections) of System object, OBJ. Releasing these resources allows all of the System object properties and input characteristics to be changed. After you call the release method on a System object, any subsequent calls to <code>step</code> or <code>release</code> that you make are not supported for code generation.
<b>Input Arguments</b>	<b>OBJ</b> Instance of <code>hdlram</code>
<b>See Also</b>	<code>hdlram</code>   <code>hdlram.clone</code>   <code>hdlram.isLocked</code>   <code>hdlram.step</code>

# hdlram.step

---

## Purpose

Read or write input value to memory location

## Syntax

```
DATAOUT = step(H,WRITEDATA,READWRITEADDRESS,WRITEENABLE)
READDATAOUT = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)
[WRITEDATAOUT,READDATAOUT] = step(H,WRITEDATA,WRITEADDRESS,
 WRITEENABLE,READADDRESS)
```

## Description

`DATAOUT = step(H,WRITEDATA,READWRITEADDRESS,WRITEENABLE)` allows you to read the value in memory location `READWRITEADDRESS` when `WRITEENABLE` is false. It also allows you to write the value `WRITEDATA` into the memory location `READWRITEADDRESS` when `WRITEENABLE` is true. `DATAOUT` is the new or old data at `READWRITEADDRESS` when `WRITEENABLE` is true, or the data at `READWRITEADDRESS` when `WRITEENABLE` is false. This step syntax is appropriate for a single-port RAM System object.

`READDATAOUT = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)` allows you to write the value `WRITEDATA` into memory location `WRITEADDRESS` when `WRITEENABLE` is true. `READDATAOUT` is the old data at the address location `READADDRESS`. This step syntax is appropriate for a simple dual-port RAM System object.

`[WRITEDATAOUT,READDATAOUT] = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)` allows you to write the value `WRITEDATA` into the memory location `WRITEADDRESS` when `WRITEENABLE` is true. `WRITEDATAOUT` is the new or old data at memory location `WRITEADDRESS`. `READDATAOUT` is the old data at the address location `READADDRESS`. This step syntax is appropriate for a dual-port RAM System object.

## hdlram Input Requirements

Input	Requirement
WRITEDATA	Must be scalar. This value can be double, single, integer, or a fixed-point (fi) object, and can be real or complex.
WRITEENABLE	Must be a scalar, logical value.
WRITEADDRESS and READADDRESS	Must be real and unsigned. This value can be either fixed-point (fi) objects or integers.

## Examples

### Read/Write Single-Port RAM

Create System object that can write to a single port RAM and read the newly written value out.

Construct single-port RAM System object.

```
hRAM = hdlram('RAMType','Single port','WriteOutputValue','New data');
```

Preallocate memory.

```
dataLength = 100;
[dataIn dataOut] = deal(zeros(1,dataLength));
```

Write randomly generated data to the System object, and then read data back out again.

```
for ii = 1:dataLength
 dataIn(ii) = randi([0 63],1,1,'uint8');
 addressIn = uint8(ii-1);
 writeEnable = true;
 dataOut(ii) = step(hRAM,dataIn(ii),addressIn,writeEnable);
end ;
```

## See Also

hdlram | hdlram.clone | hdlram.isLocked | hdlram.release

# hex

---

**Purpose** 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 string. `hex(a)` is equivalent to `a.hex`.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

### 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
```

### 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
```

## 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 | storedInteger | oct

# hex2num

---

**Purpose** 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 string `h` to numeric matrix `x`. The attributes of the numbers in `x` are specified by quantizer object `q`. When `h` is a cell array containing hexadecimal strings, `hex2num` returns `x` as a cell array of the same dimension containing numbers. For fixed-point hexadecimal strings, `hex2num` uses two's complement representation. For floating-point strings, 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 strings `h1, h2,...` to numeric matrices `x1, x2,...`

`hex2num` and `num2hex` are inverses of one another, with the distinction that `num2hex` returns the hexadecimal strings 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`



**Purpose** Create histogram plot

**Description** Refer to the MATLAB `hist` reference page for more information.

# histc

---

**Purpose** Histogram count

**Description** Refer to the MATLAB `histc` reference page for more information.

**Purpose** 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,...)`.

---

**See Also** `vertcat`

# imag

---

**Purpose**            Imaginary part of complex number

**Description**      Refer to the MATLAB imag reference page for more information.

**Purpose** 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  $b$  has the same sign as the constant vector  $a$ . Therefore, the largest gain due to the vector  $a$  is  $a * \text{sign}(a')$ , which is equal to  $\text{sum}(\text{abs}(a))$ .

The overall number of integer bits necessary to guarantee that no overflow occurs in the inner product is computed by:

```
n = ceil(log2(sum(abs(a)))) + number of integer bits in b + 1 sign bit
```

## innerprodintbits

---

The extra sign bit is only added if both  $a$  and  $b$  are signed and  $b$  attains its minimum. This prevents overflow in the event of  $(-1)^*(-1)$ .

**Purpose** 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
```

**See Also** `storedInteger` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

# int16

---

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`



**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

# int64

---

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64`

**Purpose** 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`.

**See Also** `eps` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

# intmin

---

**Purpose**            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`

**Purpose** Inverse permute dimensions of multidimensional array

**Description** Refer to the MATLAB `ipermute` reference page for more information.

# isboolean

---

**Purpose** 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`

**Purpose** Determine whether `fi` object is column vector

**Syntax** `y = iscolumn(a)`

**Description** `y = iscolumn(a)` returns 1 if the `fi` object `a` is a column vector, and 0 otherwise.

**See Also** `isrow`

# isdouble

---

**Purpose** 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` | `isscalingbias` | `isscalingunspecified` | `issingle`



**Purpose** Determine whether array is empty

**Description** Refer to the MATLAB `isempty` reference page for more information.

# isequal

---

**Purpose** 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.

`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.

**See Also** `eq` | `fi` | `fimath` | `ispropequal` | `numerictype` | `quantizer`

**Purpose** 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.

**Input Arguments** **T1, T2 - Inputs to be compared**  
`numerictype` objects  
Inputs to be compared, specified as `numerictype` objects.

**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
```

# isequivalent

---

```
FractionLength: 12
```

```
isequivalent(T1,T2)
```

```
ans =
```

```
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.

## See Also

[isequal](#) | [ispropequal](#) | [eq](#)

<b>Purpose</b>	Determine whether variable is <code>fi</code> object
<b>Syntax</b>	<code>y = isfi(a)</code>
<b>Description</b>	<code>y = isfi(a)</code> returns 1 if <code>a</code> is a <code>fi</code> object, and 0 otherwise.
<b>See Also</b>	<code>fi</code>   <code>isfimath</code>   <code>isfipref</code>   <code>isnumericitype</code>   <code>isquantizer</code>

# isfimath

---

<b>Purpose</b>	Determine whether variable is <code>fimath</code> object
<b>Syntax</b>	<code>y = isfimath(F)</code>
<b>Description</b>	<code>y = isfimath(F)</code> returns 1 if <code>F</code> is a <code>fimath</code> object, and 0 otherwise.
<b>See Also</b>	<code>fimath</code>   <code>isfi</code>   <code>isfipref</code>   <code>isnumericitype</code>   <code>isquantizer</code>

**Purpose** 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`.

**See Also** `fimath` | `isfi` | `isfipref` | `isnumericitype` | `isquantizer` | `isfimathlocal` | `removefimath` | `sfi` | `ufi`

# isfinite

---

**Purpose** Determine whether array elements are finite

**Description** Refer to the MATLAB `isfinite` reference page for more information.



<b>Purpose</b>	Determine whether input is <code>fipref</code> object
<b>Syntax</b>	<code>y = isfipref(P)</code>
<b>Description</b>	<code>y = isfipref(P)</code> returns 1 if <code>P</code> is a <code>fipref</code> object, and 0 otherwise.
<b>See Also</b>	<code>fipref</code>   <code>isfi</code>   <code>isfimath</code>   <code>isnumericitype</code>   <code>isquantizer</code>

# isfixed

---

**Purpose** 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
```

**Purpose** 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` or `double`, and 0 otherwise.

`y = isfloat(T)` returns 1 when the `DataType` property of `numericType` object `T` is `single` or `double`, 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
```

# isinf

---

**Purpose** Determine whether array elements are infinite

**Description** Refer to the MATLAB `isinf` reference page for more information.

**Purpose** Determine whether array elements are NaN

**Description** Refer to the MATLAB `isnan` reference page for more information.

# isnumeric

---

**Purpose** Determine whether input is numeric array

**Description** Refer to the MATLAB `isnumeric` reference page for more information.

**Purpose** 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.

**See Also** `isfi` | `isfimath` | `isfipref` | `isquantizer` | `numerictype`

# isobject

---

**Purpose** Determine whether input is MATLAB object

**Description** Refer to the MATLAB `isobject` reference page for more information.



**Purpose** 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`

# isquantizer

---

**Purpose** 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** `quantizer` | `isfi` | `isfimath` | `isfipref` | `isnumericitype`

**Purpose** Determine whether array elements are real

**Description** Refer to the MATLAB `isreal` reference page for more information.

# isrow

---

**Purpose** Determine whether `fi` object is row vector

**Syntax** `y = isrow(a)`

**Description** `y = isrow(a)` returns 1 if the `fi` object `a` is a row vector, and 0 otherwise.

**See Also** `iscolumn`

**Purpose** Determine whether input is scalar

**Description** Refer to the MATLAB `isscalar` reference page for more information.

# isscaleddouble

---

**Purpose** 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 numeric type object `T` is `ScaledDouble`, and 0 otherwise.

**See Also** `isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

**Purpose** 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` | `numericType` | `isscaleddouble` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

# isscalingbinarypoint

---

**Purpose** 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`



**Purpose**

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 `numericType` 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
```

# isscalingunspecified

---

**Purpose** 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`

**Purpose** 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.

# issingle

---

**Purpose**

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 numeric type object `T` is `single`, and 0 otherwise.

**See Also**

```
isboolean | isdouble | isfixed | isfloat | isscaleddouble |
isscaledtype | isscalingbinarypoint | isscalingslopebias |
isscalingunspecified
```

**Purpose** Determine whether `numerictype` object has nontrivial slope and bias

**Syntax** `y = isslopebiasscaled(T)`

**Description** `y = isslopebiasscaled(T)` returns 1 when `numerictype` 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`

# isvector

---

**Purpose** Determine whether input is vector

**Description** Refer to the MATLAB `isvector` reference page for more information.

---

<b>Purpose</b>	Determine whether real-world value of <code>fi</code> object is less than or equal to another
<b>Syntax</b>	<code>c = le(a,b)</code> <code>a &lt;= b</code>
<b>Description</b>	<code>c = le(a,b)</code> is called for the syntax <code>a &lt;= b</code> when <code>a</code> or <code>b</code> is a <code>fi</code> object. <code>a</code> and <code>b</code> must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.  <code>a &lt;= b</code> does an element-by-element comparison between <code>a</code> and <code>b</code> 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.
<b>See Also</b>	<code>eq</code>   <code>ge</code>   <code>gt</code>   <code>lt</code>   <code>ne</code>

# length

---

**Purpose**      Vector length

**Description**      Refer to the MATLAB `length` reference page for more information.



**Purpose** Create line object

**Description** Refer to the MATLAB `line` reference page for more information.

# logical

---

**Purpose** Convert numeric values to logical

**Description** Refer to the MATLAB `logical` reference page for more information.

**Purpose** Create log-log scale plot

**Description** Refer to the MATLAB loglog reference page for more information.

# logreport

---

**Purpose** 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`

**Purpose** 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  $L = \text{lowerbound}(a)$  and  $U = \text{upperbound}(a)$ , then  $[L, U] = \text{range}(a)$ .

**See Also** eps | intmax | intmin | lsb | range | realmax | realmin | upperbound

# lsb

---

**Purpose**            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
```

**See Also**            `eps` | `intmax` | `intmin` | `lowerbound` | `quantize` | `range` | `realmax`  
                      | `realmin` | `upperbound`

---

<b>Purpose</b>	Determine whether real-world value of one <code>fi</code> object is less than another
<b>Syntax</b>	<code>c = lt(a,b)</code> <code>a &lt; b</code>
<b>Description</b>	<code>c = lt(a,b)</code> is called for the syntax <code>a &lt; b</code> when <code>a</code> or <code>b</code> is a <code>fi</code> object. <code>a</code> and <code>b</code> must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.  <code>a &lt; b</code> does an element-by-element comparison between <code>a</code> and <code>b</code> 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.
<b>See Also</b>	<code>eq</code>   <code>ge</code>   <code>gt</code>   <code>le</code>   <code>ne</code>

# max

---

**Purpose** Largest element in array of `fi` objects

**Syntax**

```
max(a)
max(a,b)
[y,v] = max(a)
[y,v] = max(a,[],dim)
```

**Description**

- For vectors, `max(a)` is the largest element in `a`.
- For matrices, `max(a)` is a row vector containing the maximum element from each column.
- For N-D arrays, `max(a)` operates along the first nonsingleton dimension.

`max(a,b)` returns an array the same size as `a` and `b` with the largest elements taken from `a` or `b`. Either one can be a scalar.

`[y,v] = max(a)` returns the indices of the maximum values in vector `v`. If the values along the first nonsingleton dimension contain more than one maximal element, the index of the first one is returned.

`[y,v] = max(a,[],dim)` operates along the dimension `dim`.

When complex, the magnitude `max(abs(a))` is used, and the angle `angle(a)` is ignored. NaNs are ignored when computing the maximum.

**See Also** `mean` | `median` | `min` | `sort`



**Purpose** 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](#)

# mean

---

**Purpose** 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 `Tx` represents the `numericType` properties of the fixed-point input array *a*:

```
c = Tx.divide(sum(a,dim), SizeA)
```

## See Also

`max` | `median` | `min`

# median

---

**Purpose** 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)
```

## See Also

[max](#) | [mean](#) | [min](#)

# mesh

---

**Purpose** Create mesh plot

**Description** Refer to the MATLAB mesh reference page for more information.



**Purpose** Create mesh plot with contour plot

**Description** Refer to the MATLAB meshc reference page for more information.

# meshz

---

**Purpose** Create mesh plot with curtain plot

**Description** Refer to the MATLAB meshz reference page for more information.

**Purpose** Smallest element in array of `fi` objects

**Syntax**

```
min(a)
min(a,b)
[y,v] = min(a)
[y,v] = min(a,[],dim)
```

**Description**

- For vectors, `min(a)` is the smallest element in `a`.
- For matrices, `min(a)` is a row vector containing the minimum element from each column.
- For N-D arrays, `min(a)` operates along the first nonsingleton dimension.

`min(a,b)` returns an array the same size as `a` and `b` with the smallest elements taken from `a` or `b`. Either one can be a scalar.

`[y,v] = min(a)` returns the indices of the minimum values in vector `v`. If the values along the first nonsingleton dimension contain more than one minimal element, the index of the first one is returned.

`[y,v] = min(a,[],dim)` operates along the dimension `dim`.

When complex, the magnitude `min(abs(a))` is used, and the angle `angle(a)` is ignored. NaNs are ignored when computing the minimum.

**See Also** `max` | `mean` | `median` | `sort`

# minlog

---

**Purpose** 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`

# minus

---

**Purpose** 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.

---

**See Also** `mtimes` | `plus` | `times` | `uminus`

---

<b>Purpose</b>	Modulus after division for <code>fi</code> objects
<b>Syntax</b>	$M = \text{mod}(X, Y)$
<b>Description</b>	<p><math>M = \text{mod}(X, Y)</math> if <math>Y \neq 0</math>, returns <math>X - n * Y</math>, where <math>n = \text{floor}(X ./ Y)</math>. The inputs <math>X</math> and <math>Y</math> 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 fixed-point inputs, the word length of the internally computed aggregate fixed-point output data type cannot exceed 32 bits. For floating-point input arguments, the output data type is the same as the inputs.</p> <p>The <code>mod</code> function ignores and discards any <code>fimath</code> attached to the inputs. The output is always associated with the default <code>fimath</code>.</p>
	<hr/> <b>Note</b> The combination of fixed-point and floating-point inputs is not currently supported. <hr/>
<b>Input Arguments</b>	<p><b>X</b> Integer, fixed-point, or floating-point array, or real scalar.</p> <p><b>Y</b> Array of the same size as <math>X</math>, or real scalar.</p>
<b>Output Arguments</b>	<p><b>M</b> Result of modulus operation. If both inputs <math>X</math> and <math>Y</math> are floating-point, then the data type of <math>M</math> is the same as the inputs. If either input <math>X</math> or <math>Y</math> is fixed-point, then the data type of <math>M</math> is the aggregate numerictype. This value equals that of <code>fixed.aggregateType(X, Y)</code>.</p>

## 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`

# mpower

---

**Purpose** Fixed-point matrix power (^)

**Syntax** `c = mpower(a,k)`  
`c = a^k`

**Description** `c = 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.

**Tips** For more information about the `mpower` function, see the MATLAB `mpower` reference page.

**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
```

**See Also** `mpower` | `power`

**Purpose** 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.

## See Also

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

**Purpose** Forward slash (/) or right-matrix division

**Syntax** `c = mrdivide(a,b)`  
`c = a/b`

**Description** `c = mrdivide(a,b)` and `c = a/b` perform right-matrix division.

When one or both of the inputs is a `fi` object, the denominator input, `b`, must be a scalar and the output `fi` object `c` is equivalent to `c = rdivide(a,b)` or `c = a./b` (right-array division).

The numerator input `a` can be complex, but the denominator input `b` must always be real-valued. When the numerator input `a` is complex, the real and imaginary parts of `a` are independently divided by `b`.

For information on the data type rules used by the `mrdivide` function, see the `rdivide` reference page.

**Examples** In this example, you use the forward slash (/) 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))
b = fi(3, 1, 12, 8)
c = a/b
```

The `mrdivide` function outputs a signed 3-by-3 array of `fi` objects, each of which has a word length of 16 bits and a fraction length of 3 bits.

`a =`

```
 8 1 6
 3 5 7
 4 9 2
```

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

# mrdivide

---

b =

3

DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 12  
FractionLength: 8

c =

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

## See Also

[add](#) | [divide](#) | [fi](#) | [fimath](#) | [numerictype](#) | [rdivide](#) | [sub](#) | [sum](#)

**Purpose** 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.

---

**See Also** `plus` | `minus` | `times` | `uminus`

# ndgrid

---

**Purpose**           Generate arrays for N-D functions and interpolation

**Description**       Refer to the MATLAB `ndgrid` reference page for more information.



**Purpose**      Number of array dimensions

**Description**      Refer to the MATLAB `ndims` reference page for more information.

**Purpose** 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.

**See Also** `eq` | `ge` | `gt` | `le` | `lt`

---

<b>Purpose</b>	Round toward nearest integer with ties rounding toward positive infinity
<b>Syntax</b>	<code>y = nearest(a)</code>
<b>Description</b>	<p><code>y = nearest(a)</code> rounds <code>fi</code> object <code>a</code> 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 <code>fi</code> object <code>y</code>.</p> <p><code>y</code> and <code>a</code> have the same <code>fi</code> object and <code>DataType</code> property.</p> <p>When the <code>DataType</code> property of <code>a</code> is <code>single</code>, <code>double</code>, or <code>boolean</code>, the <code>numericType</code> of <code>y</code> is the same as that of <code>a</code>.</p> <p>When the fraction length of <code>a</code> is zero or negative, <code>a</code> is already an integer, and the <code>numericType</code> of <code>y</code> is the same as that of <code>a</code>.</p> <p>When the fraction length of <code>a</code> is positive, the fraction length of <code>y</code> is 0, its sign is the same as that of <code>a</code>, and its word length is the difference between the word length and the fraction length of <code>a</code>, plus one bit. If <code>a</code> is signed, then the minimum word length of <code>y</code> is 2. If <code>a</code> is unsigned, then the minimum word length of <code>y</code> is 1.</p> <p>For complex <code>fi</code> objects, the imaginary and real parts are rounded independently.</p> <p><code>nearest</code> does not support <code>fi</code> 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.</p>

## 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 numeric type 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

### See Also

`ceil` | `convergent` | `fix` | `floor` | `round`

# noperations

---

**Purpose**                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`

**Purpose** Find logical NOT of array or scalar input

**Description** Refer to the MATLAB not reference page for more information.

# noverflows

---

**Purpose**            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 `fipref` 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`



**Purpose** Determine fixed-point data type

**Syntax**

```
nts
nts({'block',PORT})
nts({line-handle})
nts({gs1})
```

**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 523. 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({gs1})` 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.

## 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.

## Input Arguments

### **block**

Full path to the specified block.

### **line-handle**

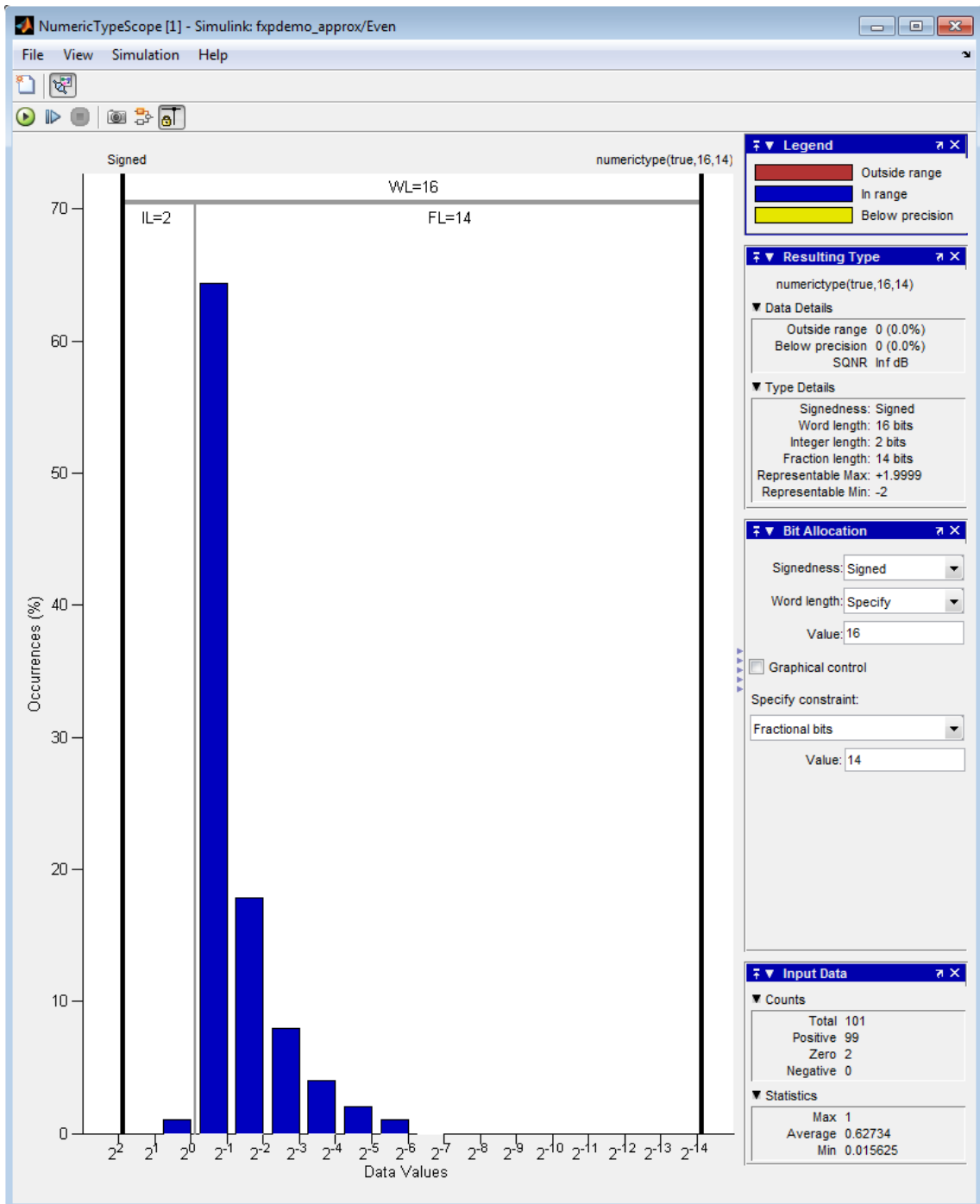
Handle of the Simulink signal that you want to view with the scope. To get the handle of the currently selected signal, at the MATLAB command line, enter `nts({gs1})`.

### **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 displayed at the top of the window and the dialog panels 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 522.

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.

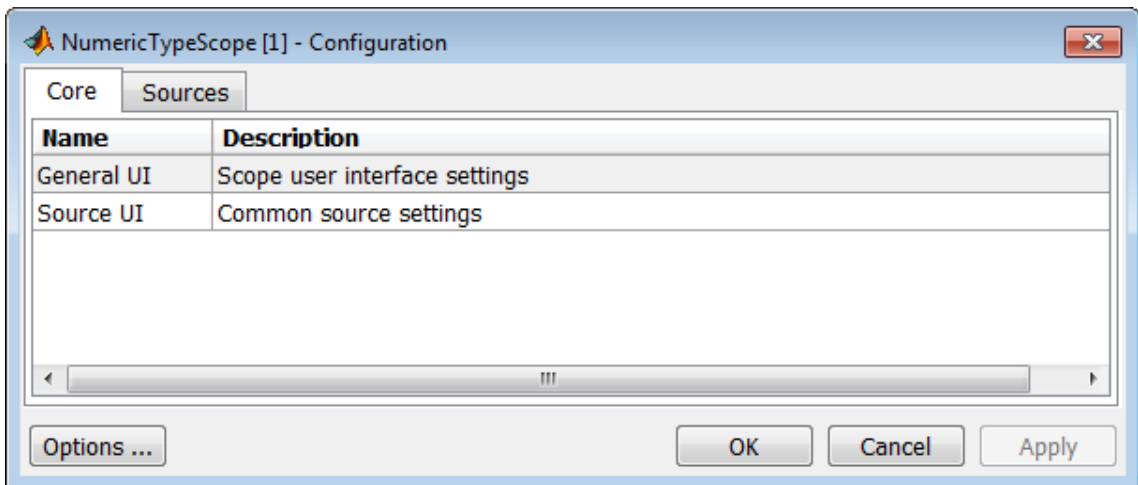
## Dialog Boxes and Panels

- “Configuration Dialog Box” on page 2-519
- “Dialog Panels” on page 2-523

### 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 520 and Sources Pane on page 522.

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.

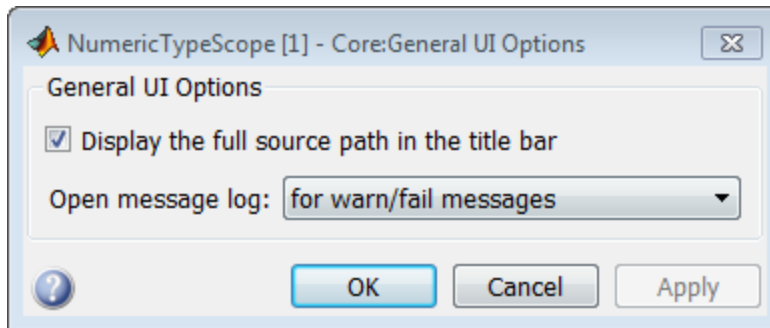
---

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 Options** dialog box, select **General UI** and then click **Options**.



- **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.
- **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:

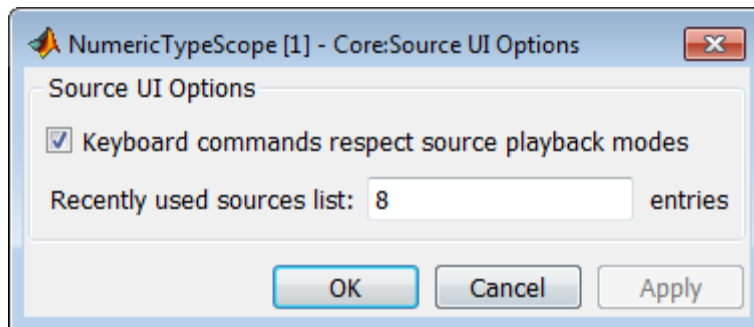
- for any new messages
- for warn/fail messages
- only for fail messages
- manually

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.

To open the **Core:Source UI Options** dialog box, select **General UI** and then click **Options**.



- **Keyboard commands respect source playback modes**—Has no effect. The following table shows the keyboard shortcut mapping. You cannot disable this mapping.

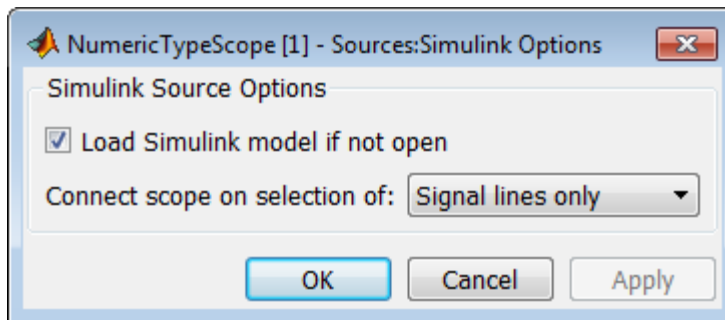
Action	Keyboard Shortcut
Open new NumericTypeScope	Insert
Change configuration	N
Display keyboard help	K
Play simulation	P
Pause simulation	Space
Stop simulation	S
Step forward	Right arrow, Page down

- **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 Options** dialog box, select the **Sources** tab and then click **Options**.



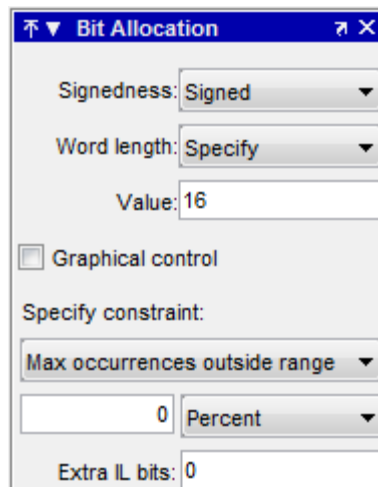


- **Load Simulink model if not open**—When selected, if you specify a signal in a Simulink model that is not currently open, the scope opens the model.
- **Connect scope on selection of**—Connects the scope only when you select signal lines or when you select signal lines or blocks. If you select **Signal 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.



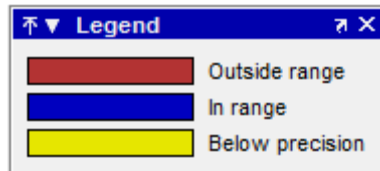
You can:

- 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.

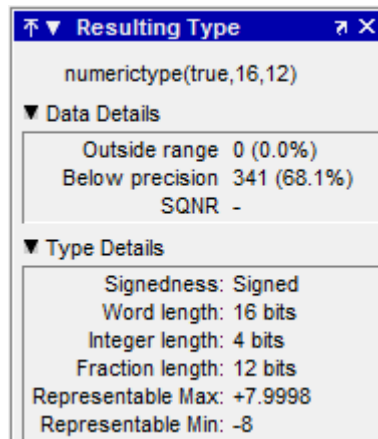
### Legend

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.

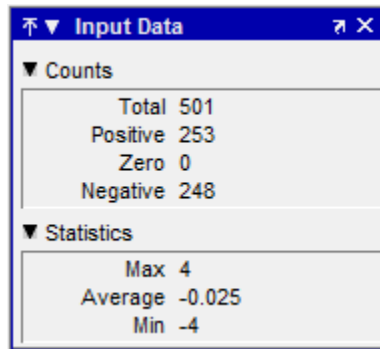


The **Data Details** section displays the percentage of values that fall outside range or below precision with the `numerictype` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data 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`.



## 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
```

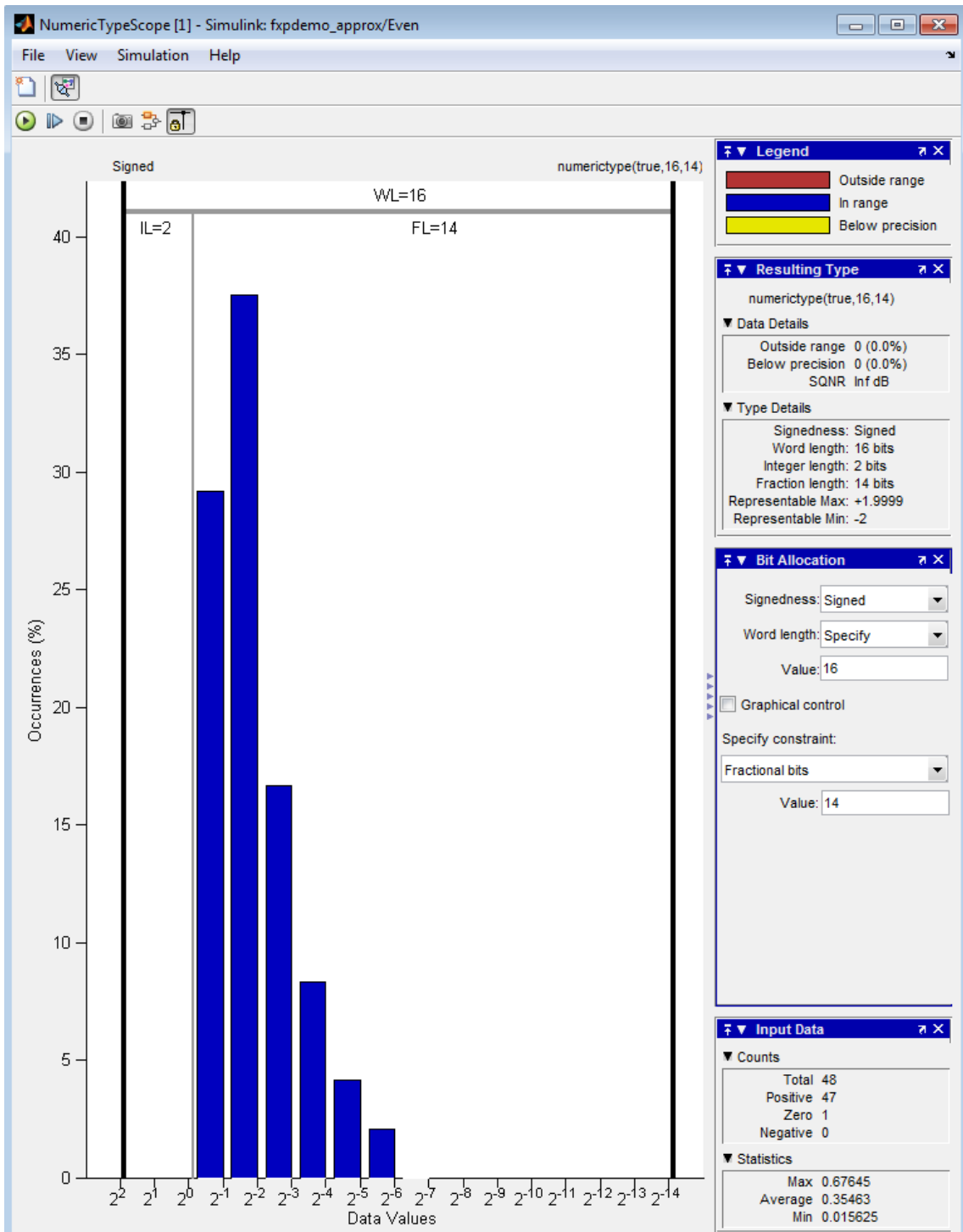
Open a NumericTypeScope.

```
nts
```

In the fxpdemo\_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.



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})
```

### **See Also**

`hist` | `log2` | `numerictypescope`

**Purpose** Convert number to binary string using quantizer object

**Syntax** `y = num2bin(q,x)`

**Description** `y = num2bin(q,x)` converts numeric array `x` into binary strings 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 strings 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`

# num2hex

---

**Purpose** Convert number to hexadecimal equivalent using `quantizer` object

**Syntax** `y = num2hex(q,x)`

**Description** `y = num2hex(q,x)` converts numeric array `x` into hexadecimal strings 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 strings 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](#)

# num2int

---

**Purpose** 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`

**Purpose** Number of data elements in an array

---

**Note** `numberofelements` will be removed in a future release. Use `numel` instead.

---

**Syntax** `numberofelements(a)`

**Description** `numberofelements(a)` returns the number of data elements in an array. Using `numberofelements` in your MATLAB code returns the same result for built-in types and `fi` objects. Use `numberofelements` to write data-type independent MATLAB code for array handling.

**See Also** `numel` | `nargin` | `nargout` | `prod` | `size` | `subsref` | `subsasgn`

# numel

---

**Purpose** 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.

**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

## 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 =
```

```
 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 =
```

# numel

---

24

## **See Also**

numel

**Purpose** 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.

# numerictype

---

- `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 numerictype 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 numerictype 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`.

---

# numerictype

---

## 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(false, 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`.

### See Also

`fi` | `fimath` | `fipref` | `quantizer`

### Related Examples

- “`numerictype` Objects Usage to Share Data Type and Scaling Settings of `fi` objects”

### Concepts

- “`numerictype` Object Properties”

**Purpose** 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 numeric type object that satisfies the specified criteria. See the section on Bit Allocation in “Dialog Panels” on page 2-550.

*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.

# NumericTypeScope

---

## Identifying Values Outside Range and Below Precision

`reset(H)` clears all stored information from the `NumericTypeScope` object `H`. Resetting the object clears the information displayed in the scope window.

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` 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 2-553.

See also Legend in “Dialog Panels” on page 2-550.

See the “Examples” on page 2-557 section to learn more about using the `NumericTypeScope` to select data types.

## Dialog Boxes and Toolbar

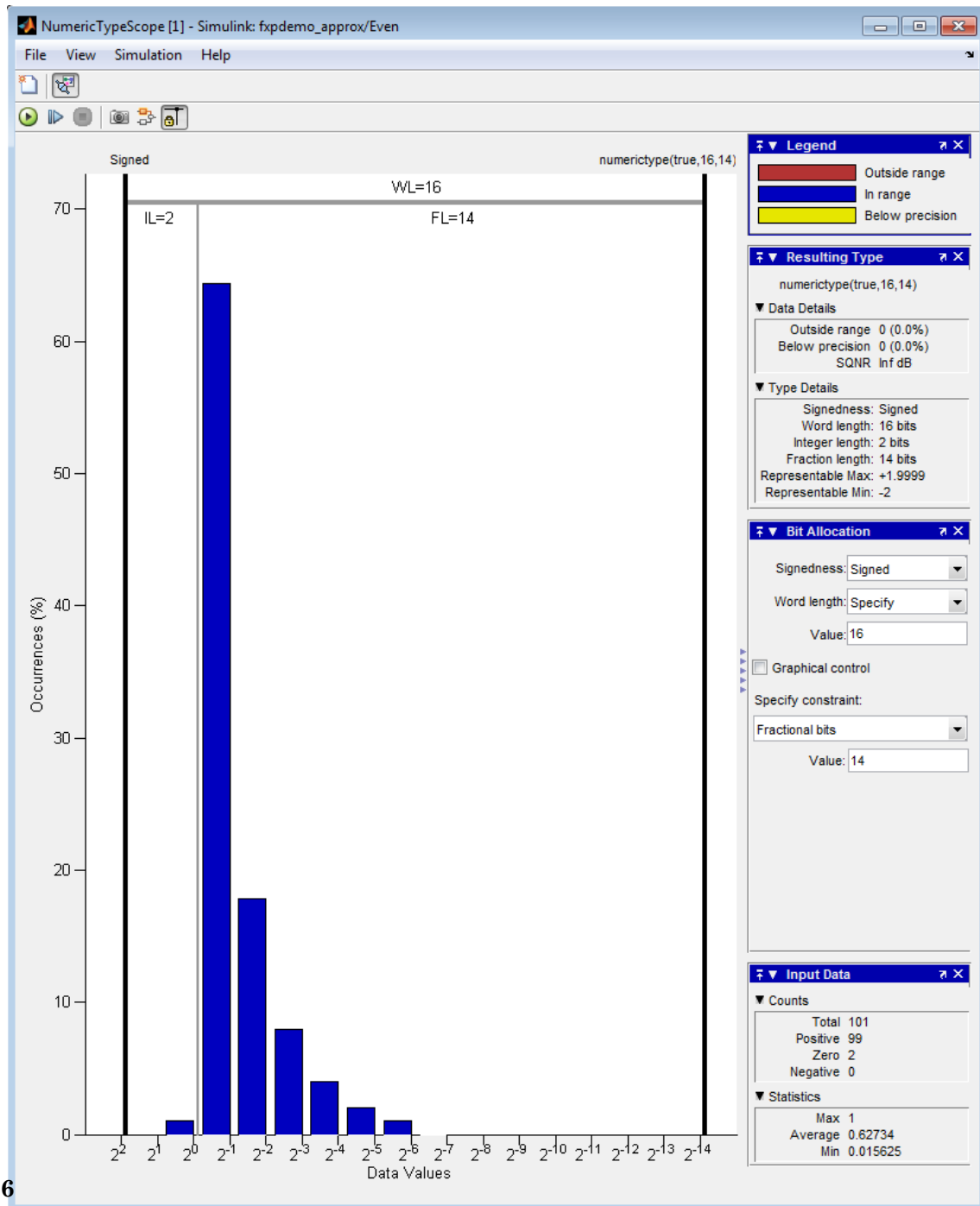
- “The `NumericTypeScope` Window” on page 2-545
- “Configuration Dialog Box” on page 2-547
- “Dialog Panels” on page 2-550

- “Vertical Units” on page 2-553
- “Bring All NumericType Scope Windows Forward” on page 2-555
- “Toolbar (Mac Only)” on page 2-556

## **The NumericTypeScope Window**

The NumericTypeScope opens with the default toolbars displayed at the top of the window and the dialog panels to the right.

# NumericTypeScope

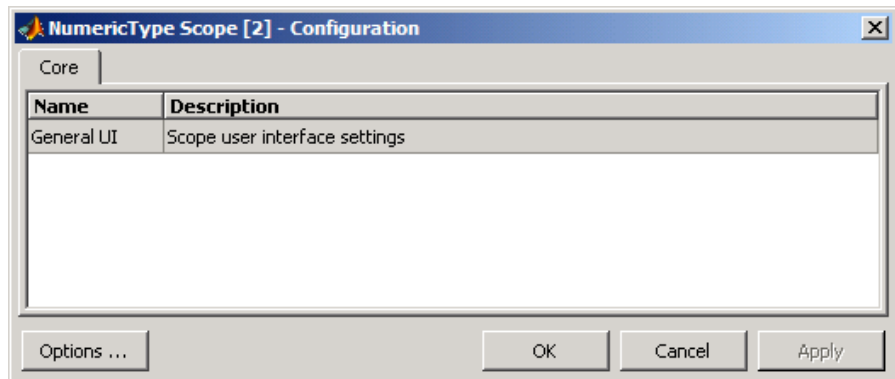




## Configuration Dialog Box

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.

# NumericTypeScope

---

---

## Caution

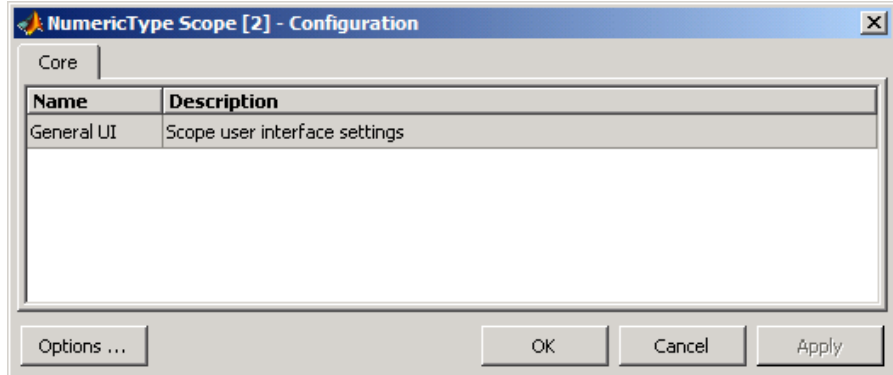
Before saving your own set of configuration settings in the matlab/toolbox/fixedpoint/fixedpoint folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

---

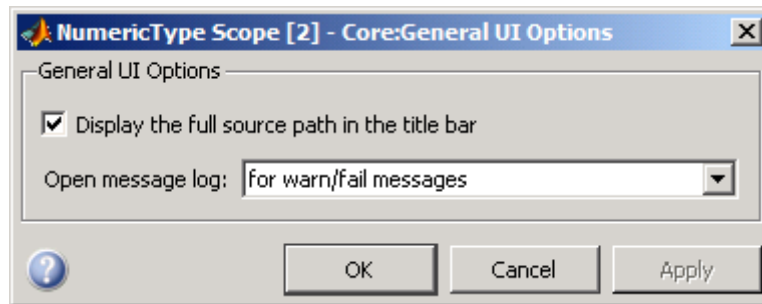
To save your configuration settings for future use, save them in the matlab/toolbox/fixedpoint/fixedpoint folder with the file name NumericTypeScopeComponent.cfg. You can re-save your configuration settings at anytime, but remember to do so in the specified folder using the specified file name.

## 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.

# NumericTypeScope

---

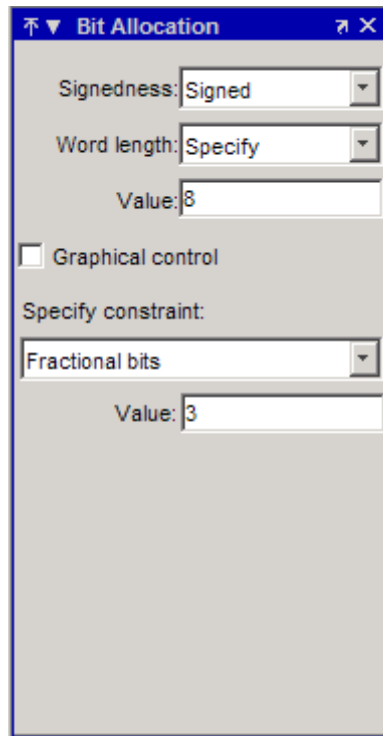
The scope uses Configuration messages to indicate when new configuration files are loaded, and Extension messages to indicate when components are registered. Category defaults to All.

## Dialog Panels

- Bit Allocation on page 550
- Legend on page 552
- Resulting Type on page 552
- Input Data on page 553

## Bit Allocation

The scope Bit Allocation dialog panel, as shown in the following figure, offers you several options for specifying data type criteria.



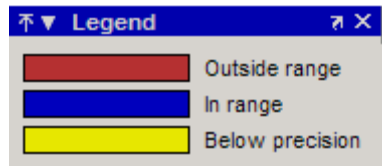
You can use this panel to specify a known word length and the desired maximum occurrences outside range. You can also use the panel to specify the desired number of occurrences outside range and the smallest value to be represented by the suggested data type. For streaming data, the suggested numerictype 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 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 data type.

# NumericTypeScope

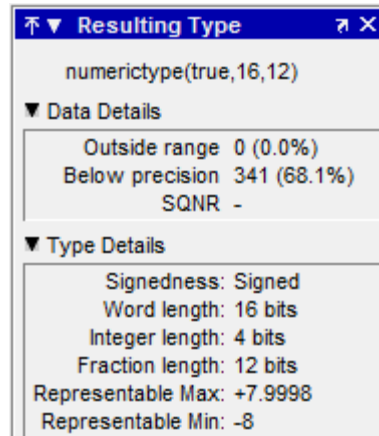
## Legend

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.

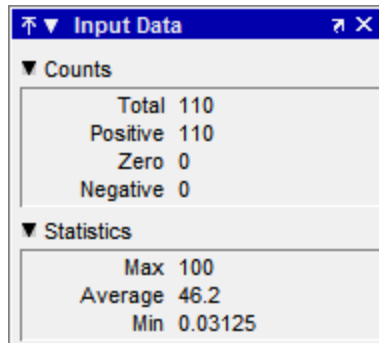


The Data Details section displays the percentage of values that fall outside range or below precision with the `numerictype` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data to 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". The "Counts" section lists: Total 110, Positive 110, Zero 0, and Negative 0. The "Statistics" section lists: Max 100, Average 46.2, and Min 0.03125.

Counts	
Total	110
Positive	110
Zero	0
Negative	0

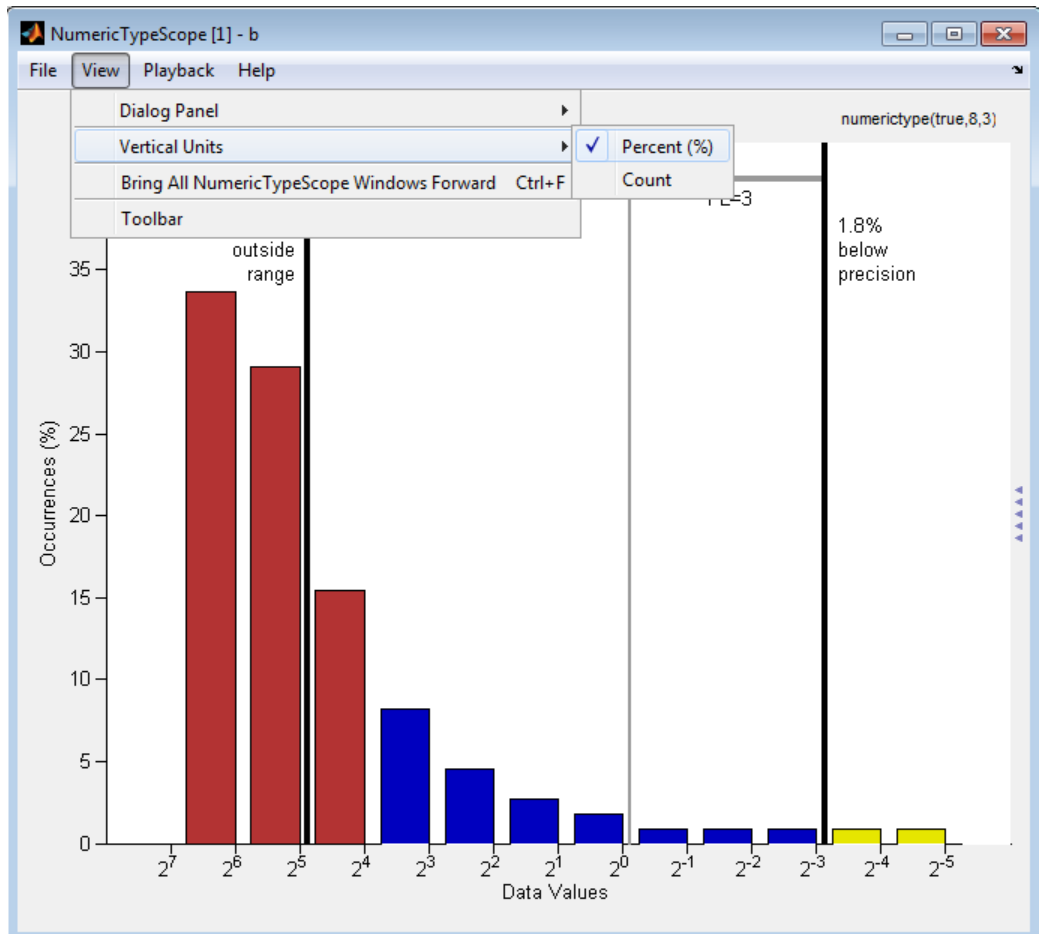
  

Statistics	
Max	100
Average	46.2
Min	0.03125

## Vertical Units

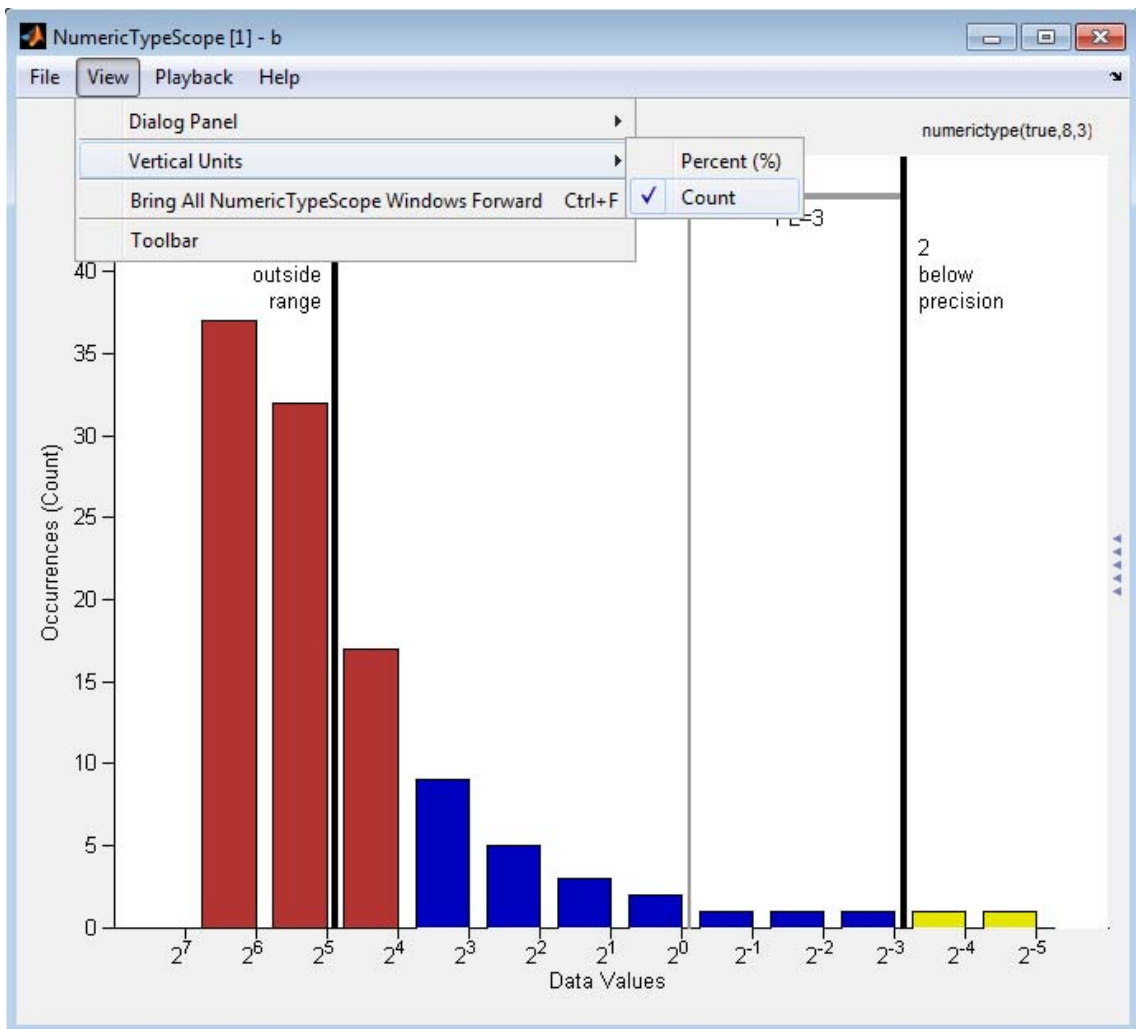
Use the Vertical Units selection to display values that are outside range or below precision as a percentage or as an actual count. For example, the following image shows the values that are outside range or below precision as a percentage of the total values.

# NumericTypeScope



This next example shows the values that are outside range or below precision as an actual count.





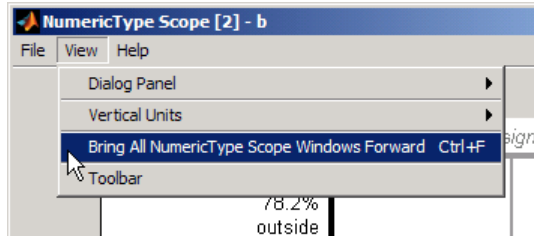
## Bring All NumericType Scope Windows Forward

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

# NumericTypeScope

---

**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)
```

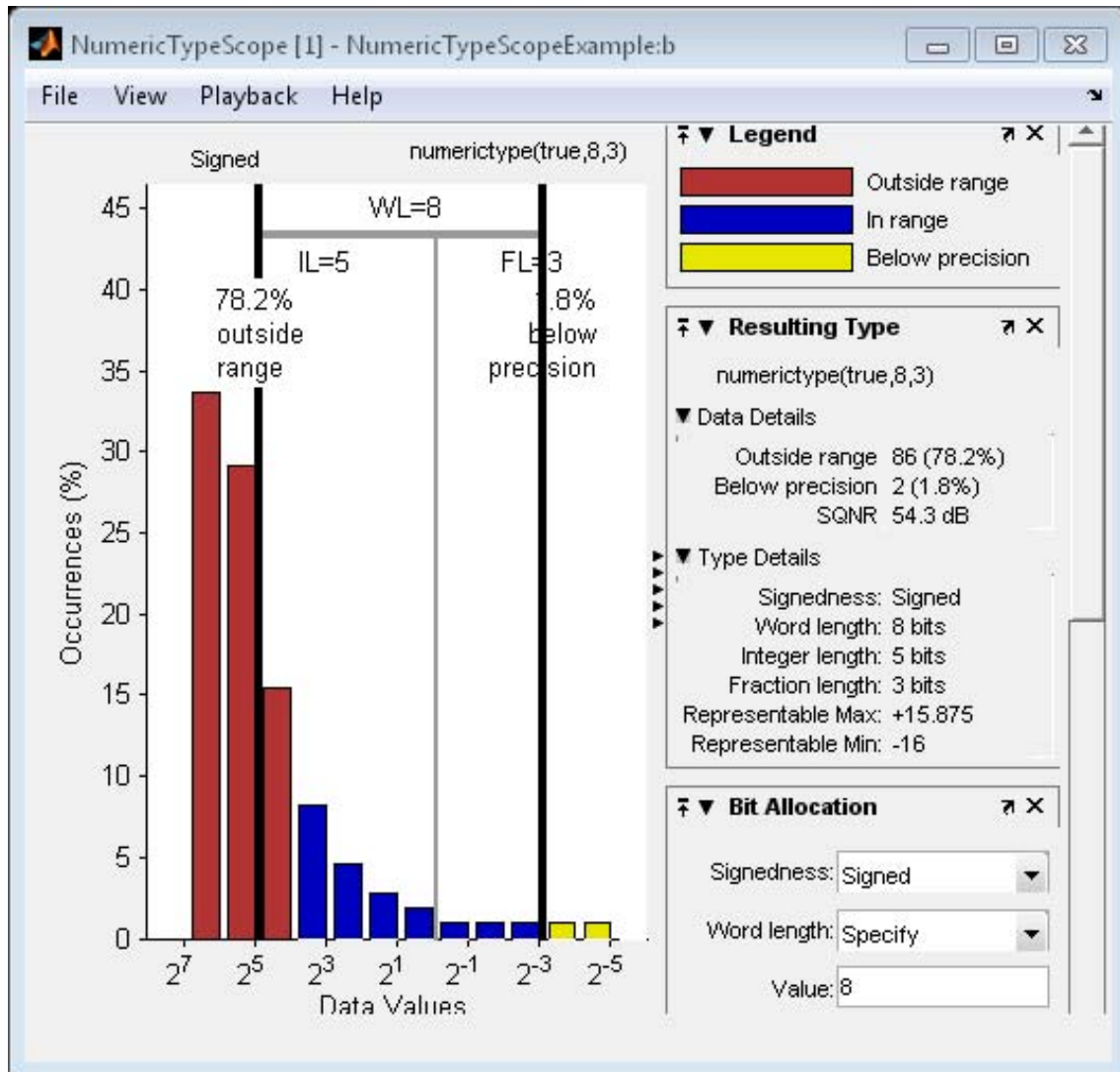
## **Examples**

### **View the Dynamic Range of a fi Object**

Set the fi object `DataTypeOverride` to `Scaled Doubles`, and then view its dynamic range.

```
fp = fipref;
initialDTOSetting = 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 = initialDTOSetting;
```

# NumericTypeScope



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 `numerictype(1,8,3)`. The `numerictype(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 `numerictype(0,13,5)`.

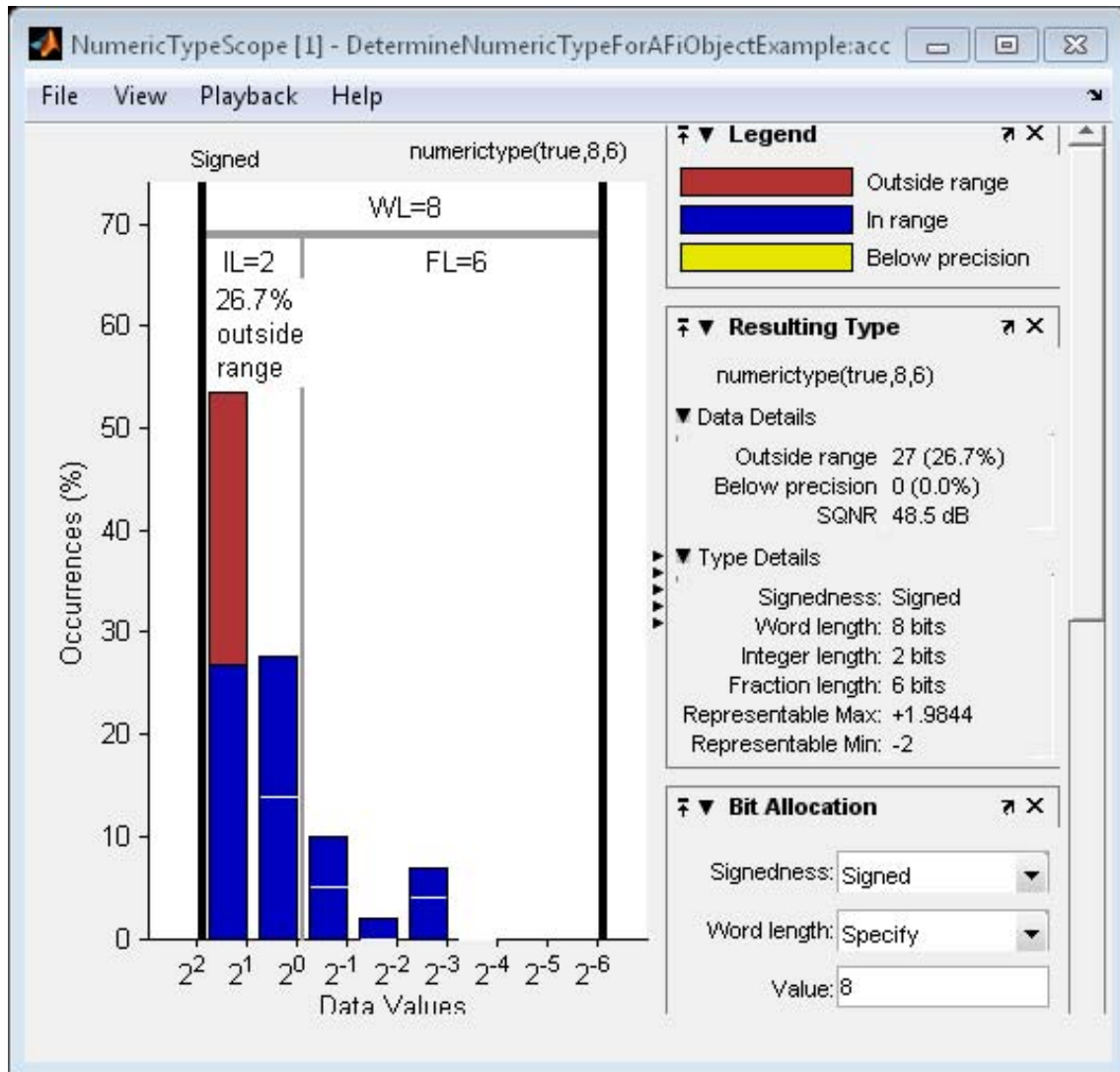
## Determine Numeric Type For a fi Object

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 `numerictype` object with a `DataTypeMode` of Scaled double: binary point scaling. You can then use that `numerictype` 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
```

# NumericTypeScope



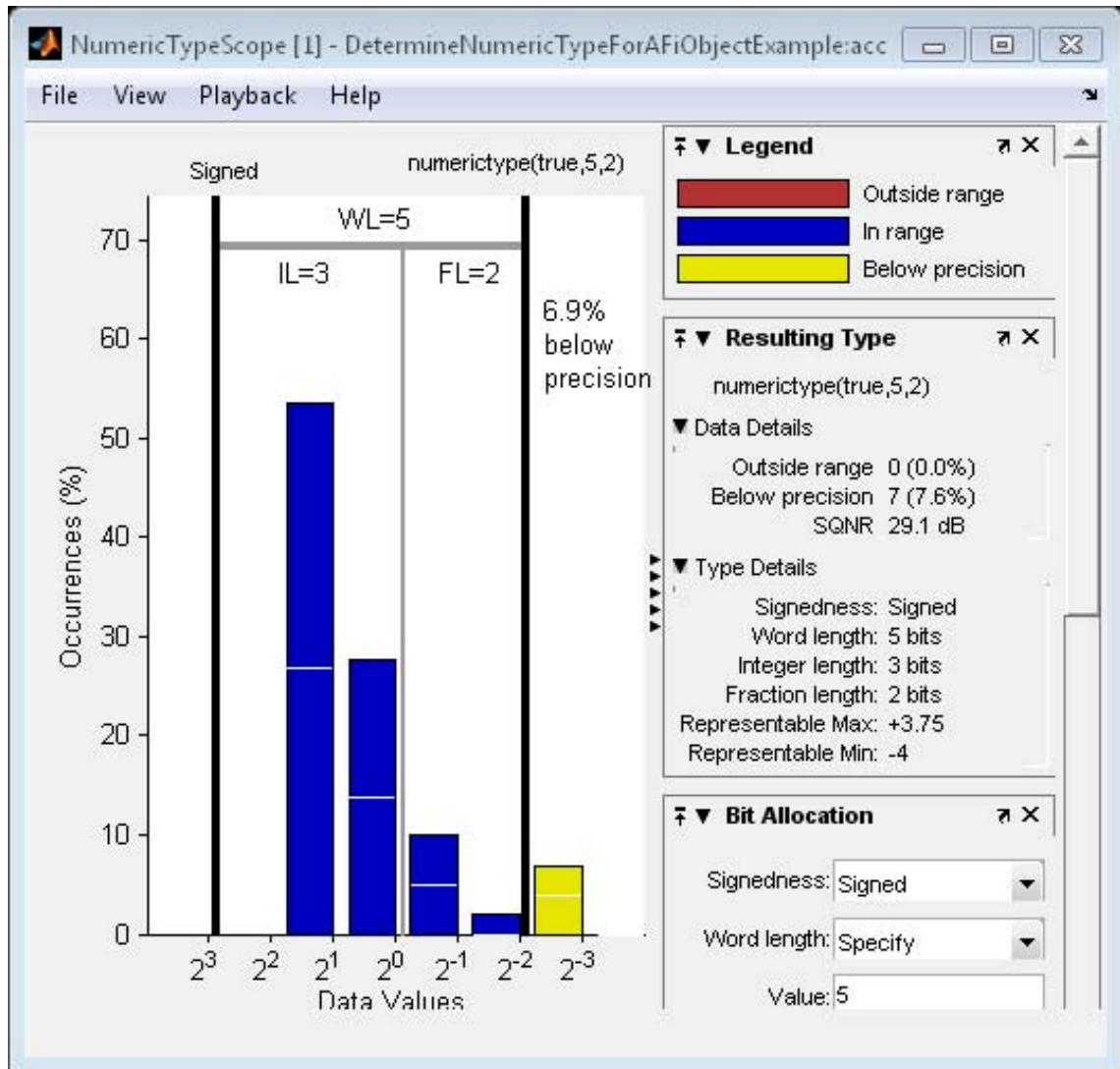
This dynamic range analysis shows that you can represent the entire range of data in the accumulator with 5 bits; two to the left of the binary

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
```

# NumericTypeScope



**See Also** [hist](#) | [log2](#)



**Purpose**            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 `fipref` 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`

# oct

---

**Purpose** 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 as a string. `oct(a)` is equivalent to `a.oct`.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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`

**Purpose**

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)`.

**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.

## Examples

### 2-D Array of Ones With Fixed-Point Attributes

Create a 2-by-3 array of ones with specified numeric type and `fi` 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 numeric type properties as `p`.

```
X = ones(2,3,'like',p)
```

```
X =
```

```

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 =
```

```
 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 =
```

```
 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 numerictype properties as `p`.

```
X = ones(4, 'like', p)
```

```
X =
```

```

 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

```

### 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 = [1.0000000000000000 -0.942809041582063 0.3333333333333333];

% 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
```

## See Also

[zeros](#) | [cast](#) | [ones](#)

## Related Examples

- “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros”

## Concepts

- “Manual Fixed-Point Conversion Workflow”
- “Manual Fixed-Point Conversion Best Practices”

**Purpose** Find logical OR of array or scalar inputs

**Description** Refer to the MATLAB or reference page for more information.

# patch

---

**Purpose** Create patch graphics object

**Description** Refer to the MATLAB patch reference page for more information.

**Purpose** Create pseudocolor plot

**Description** Refer to the MATLAB `pcolor` reference page for more information.

# permute

---

**Purpose** Rearrange dimensions of multidimensional array

**Description** Refer to the MATLAB permute reference page for more information.

**Purpose** Create linear 2-D plot

**Description** Refer to the MATLAB `plot` reference page for more information.

# plot3

---

**Purpose** Create 3-D line plot

**Description** Refer to the MATLAB `plot3` reference page for more information.



**Purpose** Draw scatter plots

**Description** Refer to the MATLAB `plotmatrix` reference page for more information.

# plotyy

---

**Purpose** Create graph with y-axes on right and left sides

**Description** Refer to the MATLAB `plotyy` reference page for more information.

---

**Purpose** 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.

---

**See Also** `minus` | `mtimes` | `times` | `uminus`

# polar

---

**Purpose** Plot polar coordinates

**Description** Refer to the MATLAB `polar` reference page for more information.

---

<b>Purpose</b>	Efficient fixed-point multiplication by $2^K$
<b>Syntax</b>	<code>b = pow2(a,K)</code>
<b>Description</b>	<code>b = pow2(a,K)</code> returns the value of <code>a</code> shifted by <code>K</code> bits where <code>K</code> is an integer and <code>a</code> and <code>b</code> are <code>fi</code> objects. The output <code>b</code> always has the same word length and fraction length as the input <code>a</code> .

---

**Note** In fixed-point arithmetic, shifting by `K` bits is equivalent to, and more efficient than, computing  $b = a \cdot 2^k$ .

---

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 `RoundMode` 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 = pow2(a)` when `a` is a `fi` object.

## Examples

### 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 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 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
```

### See Also

[bitshift](#) | [bitsl1](#) | [bitsra](#) | [bitsr1](#)



<b>Purpose</b>	Fixed-point array power ( <code>.</code> <sup><code>^</code></sup> )
<b>Syntax</b>	<code>c = power(a,k)</code> <code>c = a.^k</code>
<b>Description</b>	<p><code>c = power(a,k)</code> and <code>c = a.^k</code> compute element-by-element power. The exponent <math>k</math> requires a positive, real-valued integer value.</p> <p>The fixed-point output array <code>c</code> has the same local fimath as the input <code>a</code>. If <code>a</code> has no local fimath, the output <code>c</code> also has no local fimath. The array power operation is performed using default fimath settings.</p>
<b>Tips</b>	For more information about the <code>power</code> function, see the MATLAB <code>power</code> reference page.
<b>Examples</b>	<p>Compute the power of a 2-dimensional array for exponent values 0, 1, 2, and 3.</p> <pre>x = fi([0 1 2; 3 4 5], 1, 32);  px0 = x.^0 px1 = x.^1 px2 = x.^2 px3 = x.^3</pre>
<b>See Also</b>	<code>power</code>   <code>mpower</code>

**Purpose** 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.

**Purpose** 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,f1)
yBP = quantize(x,s,wl,f1,rm)
yBP = quantize(x,s,wl,f1,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:

# quantize

---

- 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`.

## 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

## 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 =
```

```
3.1410
```

```
 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);
```

# quantize

---

Define the input.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```
x_SB =
```

```
 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 =
```

```
 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 numeric type 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 =
```

```
 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 numeric type, `ntSB`, to quantize the input, `x_SB`, to a slope-bias data type.

```
ySB2 = quantize(x_SB,ntSB,'Ceiling','Wrap')
```

```
ySB2 =
```

```
 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 =
```

```
-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 =
```

```
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 =

 -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 =

 -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 =

 Columns 1 through 6

```

# quantize

---

```
-13.4000 -11.9999 -8.0000 -4.0001 -0.0002 4.0001
Columns 7 through 9
8.0000 12.0000 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 =
```

```
-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.

## See Also

`fi` | `fimath` | `fixed.Quantizer` | `numericType`

## Related Examples

- “Compute Quantization Error”

**Purpose** 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)
```

# quantize method

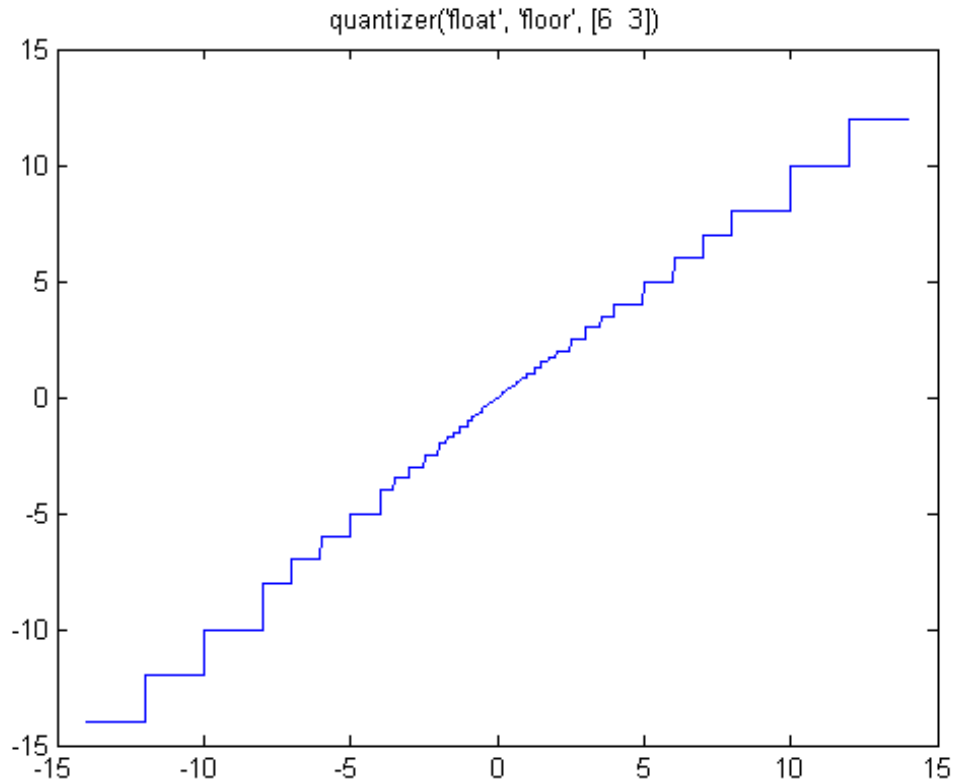
---

```
ans =
```

```
 -14 14
```

```
y=quantize(q, u);
plot(u, y); title(tostring(q))
```

```
Warning: 68 overflow(s) occurred in the fi quantize operation.
```



## Fixed-Point

The following example demonstrates using `quantize` to quantize data.

```
u=linspace(-15, 15, 1000);
q=quantizer([6 2], 'wrap');
range(q)
```

```
ans =
```

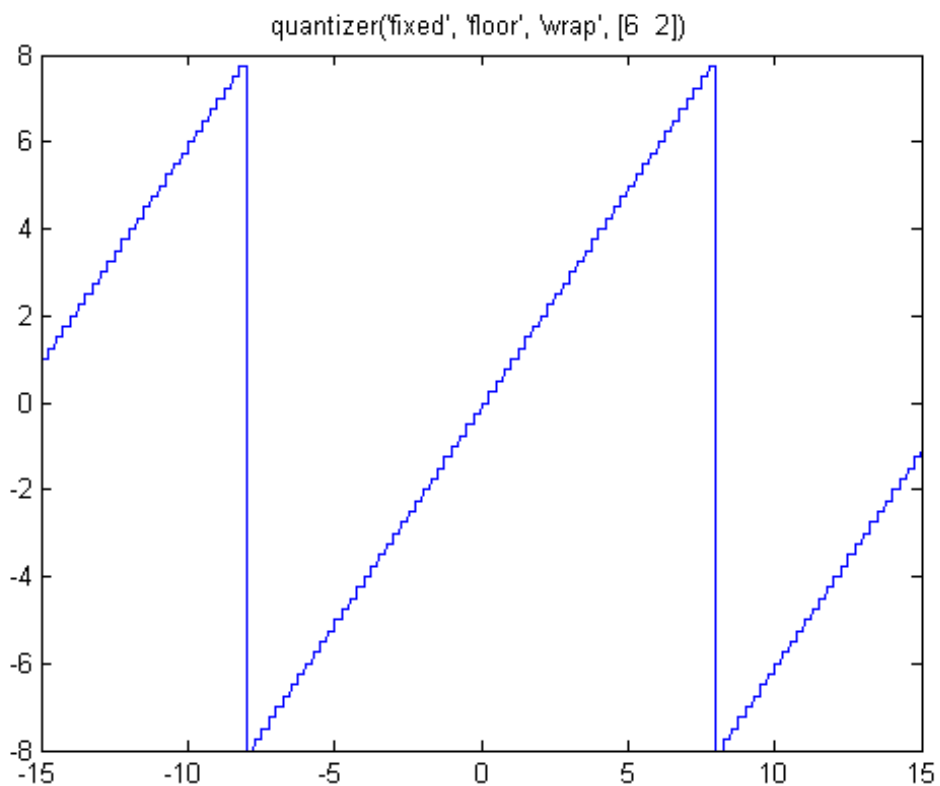
# quantize method

---

-8.0000 7.7500

```
y=quantize(q, u);
plot(u, y); title(tostring(q))
```

Warning: 468 overflow(s) occurred in the fi quantize operation.





### **See Also**

assignmentquantizer | quantizer | set | unitquantize |  
unitquantizer

# quantizer

---

**Purpose** Construct quantizer object

**Syntax**

```
q = quantizer
q = quantizer('PropertyName1',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('PropertyName1',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 strings `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 1-7.

<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.

# quantizer

Property Name	Property Value	Description
overflowmode (fixed-point only)	'saturate'	Saturate on overflow.
	'wrap'	Wrap on overflow.
format	[wordlength fractionlength]	Format for fixed or ufixed mode.
	[wordlength exponentlength]	Format for float mode.

The default property values for a quantizer object are

```
mode = '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);
```

# quantizer

---

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', 'roundingmode', 'ceil', ...
'overflowmode', 'saturate', 'format', [5 4]);
```

## See Also

[assignmentquantizer](#) | [fi](#) | [fimath](#) | [fipref](#) | [numerictype](#) | [quantize](#) | [set](#) | [unitquantize](#) | [unitquantizer](#)

**Purpose** Create quiver or velocity plot

**Description** Refer to the MATLAB `quiver` reference page for more information.

# quiver3

---

**Purpose** Create 3-D quiver or velocity plot

**Description** Refer to the MATLAB `quiver3` reference page for more information.



**Purpose**

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

# randquant

---

-[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](#)

**Purpose** 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.

## See Also

`eps` | `exponentmax` | `exponentmin` | `fractionlength` | `intmax` | `intmin` | `lowerbound` | `lsb` | `max` | `min` | `realmax` | `realmin` | `upperbound`

**Purpose** Right-array division (./)

**Syntax**  
`c = rdivide(a,b)`  
`c = a./b`

**Description** `c = rdivide(a,b)` and `c = a./b` perform right-array division by dividing each element of `a` by the corresponding element of `b`. If inputs `a` and `b` are not the same size, one of them must be a scalar value.

The numerator input `a` can be complex, but the denominator `b` requires a real-valued input. If `a` is complex, the real and imaginary parts of `a` are independently divided by `b`.

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 <code>c = a./b</code> , the fraction length of output <code>c</code> equals the fraction length of <code>a</code> minus the fraction length of <code>b</code> .

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, <code>B = int8(2)</code> is treated as an <code>s8,0 fi</code> object.
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	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:  <b>1</b> ScaledDouble <b>2</b> Fixed-point <b>3</b> Built-in double <b>4</b> Built-in single  When both inputs are <code>fi</code> objects, the only data types that are allowed to mix are ScaledDouble and Fixed-point.

## Examples

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`.

```
a = fi(magic(3))
b = int8([3 3 4; 1 2 4 ; 3 1 2])
c = a./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 =

8	1	6
3	5	7
4	9	2

```

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

```

b =

3	3	4
1	2	4
3	1	2

c =

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

```

## See Also

`add` | `divide` | `fi` | `fimath` | `mrdivide` | `numericType` | `sub` | `sum`

# real

---

**Purpose** Real part of complex number

**Description** Refer to the MATLAB real reference page for more information.



<b>Purpose</b>	Largest positive fixed-point value or quantized number
<b>Syntax</b>	realmax(a) realmax(q)
<b>Description</b>	<p>realmax(a) is the largest real-world value that can be represented in the data type of fi object a. Anything larger overflows.</p> <p>realmax(q) is the largest quantized number that can be represented where q is a quantizer object. Anything larger overflows.</p>
<b>Examples</b>	<pre>q = quantizer('float',[6 3]); x = realmax(q)  x =      14</pre>
<b>Algorithms</b>	<p>If q is a floating-point quantizer object, the largest positive number, <math>x</math>, is</p> $x = 2^{E_{max}} \cdot (2 - eps(q))$ <p>If q is a signed fixed-point quantizer object, the largest positive number, <math>x</math>, is</p> $x = \frac{2^{w-1} - 1}{2^f}$ <p>If q is an unsigned fixed-point quantizer object (datamode = 'ufixed'), the largest positive number, <math>x</math>, is</p> $x = \frac{2^w - 1}{2^f}$

# realmax

---

## See Also

eps | exponentmax | exponentmin | fractionlength | intmax  
| intmin | lowerbound | lsb | quantizer | range | realmin |  
upperbound

---

<b>Purpose</b>	Smallest positive normalized fixed-point value or quantized number
<b>Syntax</b>	<code>x=realmin(a)</code> <code>x=realmin(q)</code>
<b>Description</b>	<p><code>x=realmin(a)</code> is the smallest positive real-world value that can be represented in the data type of <code>fi</code> object <code>a</code>. Anything smaller than <code>x</code> underflows or is an IEEE “denormal” number.</p> <p><code>x=realmin(q)</code> is the smallest positive normal quantized number where <code>q</code> is a quantizer object. Anything smaller than <code>x</code> underflows or is an IEEE “denormal” number.</p>
<b>Examples</b>	<pre>q = quantizer('float',[6 3]); x = realmin(q)  x =      0.2500</pre>
<b>Algorithms</b>	<p>If <code>q</code> is a floating-point quantizer object, <math>x = 2^{E_{min}}</math> where <math>E_{min} = \text{exponentmin}(q)</math> is the minimum exponent.</p> <p>If <code>q</code> is a signed or unsigned fixed-point quantizer object, <math>x = 2^{-f} = \varepsilon</math> where <math>f</math> is the fraction length.</p>
<b>See Also</b>	<code>eps</code>   <code>exponentmax</code>   <code>exponentmin</code>   <code>fractionlength</code>   <code>intmax</code>   <code>intmin</code>   <code>lowerbound</code>   <code>lsb</code>   <code>range</code>   <code>realmax</code>   <code>upperbound</code>

# reinterprecast

---

**Purpose** Convert fixed-point data types without changing underlying data

**Syntax** `c = reinterprecast(a, T)`

**Description** `c = reinterprecast(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 `reinterprecast` 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 `reinterprecast` 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], true, 8, 7)
T = numericType(false, 8, 0);
c = reinterprecast(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
```

## See Also

```
cast | fi | numerictype | typecast
```

# removefimath

---

**Purpose** 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.

**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`.

**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)
f = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
a = setfimath(a,f)
b = removefimath(a)
```

```
a =
```

```
3.1416
```

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

```
a =
```

```
3.1416
```

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

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

```
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',...
```

# removefimath

---

```
 '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;
}
```

## See Also

[fi](#) | [fimath](#) | [setfimath](#)



**Purpose** Replicate and tile array

**Description** Refer to the MATLAB repmat reference page for more information.

# rescale

---

**Purpose** 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
```

**See Also**

```
fi
```

# reset

---

**Purpose**            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`

**Purpose** 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:

```
RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

**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('RoundMode','Floor','OverflowMode','Wrap');
globalfimath(F);
F1 = fimath
a = fi(pi)
```

```
F1 =
```

```
RoundMode: floor
OverflowMode: wrap
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

```
a =
```

```
3.1416
```

# resetglobalfimath

---

```
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 =

```
RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

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`

# removeglobalfimathpref

---

**Purpose** 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** **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`



**Purpose** 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

# reshape

---

**Purpose** Reshape array

**Description** Refer to the MATLAB reshape reference page for more information.

**Purpose** Plot colormap

**Description** Refer to the MATLAB `rgbplot` reference page for more information.

# ribbon

---

**Purpose** Create ribbon plot

**Description** Refer to the MATLAB ribbon reference page for more information.

**Purpose** Create angle histogram

**Description** Refer to the MATLAB rose reference page for more information.

# round

---

**Purpose** 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 `fimath` 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. Compare to `quantize`.

## 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 numeric type 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
```

# round

---

WordLength: 8  
FractionLength: 12

$y = \text{round}(a)$

$y =$

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



---

<b>a</b>	<b>convergent(a)</b>	<b>nearest(a)</b>	<b>round(a)</b>
2.5	2	3	3
3.5	4	4	4

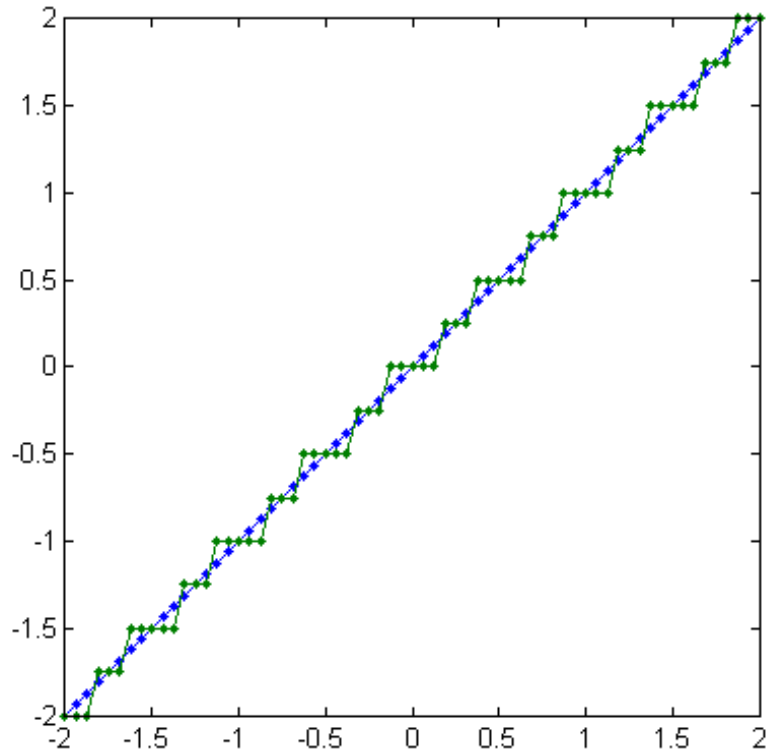
### Quantize an input

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;
```

# round

---



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.

## See Also

`ceil` | `convergent` | `fix` | `floor` | `nearest` | `quantize` | `quantizer`

<b>Purpose</b>	Save fi preferences for next MATLAB session
<b>Syntax</b>	savefipref
<b>Description</b>	savefipref saves the settings of the current fipref object for the next MATLAB session.
<b>See Also</b>	fipref

# scatter

---

**Purpose** Create scatter or bubble plot

**Description** Refer to the MATLAB scatter reference page for more information.

**Purpose** Create 3-D scatter or bubble plot

**Description** Refer to the MATLAB `scatter3` reference page for more information.

# sdec

---

**Purpose** Signed decimal representation of stored integer of `fi` object

**Syntax** `sdec(a)`

**Description** Fixed-point numbers can be represented as

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

or, equivalently as

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

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 as a string.

**Examples** The code

```
a = fi([-1 1],1,8,7);
sdec(a)
```

returns

```
-128 127
```

**See Also** `bin` | `dec` | `hex` | `storedInteger` | `oct`

**Purpose** Create semilogarithmic plot with logarithmic x-axis

**Description** Refer to the MATLAB `semilogx` reference page for more information.

# semilogy

---

**Purpose** Create semilogarithmic plot with logarithmic y-axis

**Description** Refer to the MATLAB `semilogy` reference page for more information.



**Purpose**

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`.

## set

---

`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`

<b>Purpose</b>	Attach fimath object to fi object
<b>Syntax</b>	<code>y = setfimath(x,f)</code>
<b>Description</b>	<p><code>y = setfimath(x,f)</code> returns a <code>fi</code> object, <code>y</code>, with <code>x</code>'s numeric type and value, and attached <code>fimath</code> object, <code>f</code>. This function and the related <code>removefimath</code> function are useful for preventing errors about embedded <code>fimath</code> of both operands needing to be equal.</p> <p>The <code>y = setfimath(x,f)</code> syntax does not modify the input, <code>x</code>. To modify <code>x</code>, use <code>x = setfimath(x,f)</code>. If you use <code>setfimath</code> in an expression, such as, <code>a*setfimath(b,f)</code>, the <code>fimath</code> object is used in the temporary variable, but <code>b</code> is not modified.</p>
<b>Input Arguments</b>	<p><b>x - Input data</b> <code>fi</code> object   built-in integer   double   single</p> <p>Input data, specified as a <code>fi</code> object or built-in integer value, from which to copy the data type and value to the output. <code>x</code> must be a <code>fi</code> object or an integer data type (<code>int8</code>, <code>int16</code>, <code>int32</code>, <code>int64</code>, <code>uint8</code>, <code>uint16</code>, <code>uint32</code>, or <code>uint64</code>). Otherwise, the <code>fimath</code> object is not applied. If <code>x</code> is not a <code>fi</code> object or integer data type, <code>y = x</code>.</p> <p><b>f - Input fimath object</b> <code>fimath</code> object</p> <p>Input <code>fimath</code> object, specified as an existing <code>fimath</code> object to attach to the output. An error occurs if <code>f</code> is not a <code>fimath</code> object.</p>
<b>Output Arguments</b>	<p><b>y - Output fi object</b> <code>fi</code> object</p> <p>Output <code>fi</code> object, returned as a <code>fi</code> object with the same data type and value as the <code>x</code> input. <code>y</code> also has attached <code>fimath</code> object, <code>f</code>. If the input, <code>x</code>, is not a <code>fi</code> object or integer data type, then <code>y = x</code>.</p>

## 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;
}
```

## See Also

`fi` | `fimath` | `removefimath`

**Purpose** 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,fixexponent,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,fixexponent,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixexponent`, and `bias`.

`fi` objects created by the `sfi` constructor function have the following general types of properties:

- “Data Properties” on page 2-345
- “fimath Properties” on page 2-657
- “numerictype Properties” on page 2-347

These properties are described in detail in “fi Object Properties” on page 1-2 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 1-2.

### 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
```

## See Also

```
fi | fimath | fipref | isfimathlocal | numerictype | quantizer |
ufi
```

**Purpose** 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)
```

# shiftdata

---

y =

1 2 3 4 5

## See Also

[permute](#) | [shiftdim](#) | [unshiftdata](#)



**Purpose** Shift dimensions

**Description** Refer to the MATLAB `shiftdim` reference page for more information.

# showfixptsimerrors

---

**Purpose** Show overflows from most recent fixed-point simulation

---

**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

**Purpose** Show logged maximum values, minimum values, and overflow data from fixed-point simulation

---

**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`

# showInstrumentationResults

---

**Purpose** 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.

- defaultDT *T*

Default data type to propose for double or single data type inputs, where *T* is either a numeric type object or one of these strings: 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.
- nocode

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 -printable option.
- optimizeWholeNumbers

Optimize the word length of variables whose simulation min/max logs indicate that they are always whole numbers.
- percentSafetyMargin *N*

Safety margin for simulation min/max, where *N* is a percent value.
- printable

Create and open a printable HTML report. The report opens in the system browser.

# showInstrumentationResults

---

-proposeFL	Propose fraction lengths for specified word lengths.
-proposeWL	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 the Code Generation Report. View the simulation minimum and maximum values, proposed fraction length, percent of current range, and whole number status by hovering 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
```


# showInstrumentationResults

```

23 n = length(x); t = log2(n);
24 x = fidemo.fi_bitreverse(x, n);
25
26 % Generate index variables
27 % the loop.
28 LL = int32(2.^(1:t));
29 rr = int32(n./LL);
30 LL2 = int32(LL./2);
31 for q=1:t
32 L = LL(q); r = rr(q); L2 = LL2(q);
33 for k=0:(r-1)
34 for j=0:(L2-1)
35 temp
36 x(k*L+j+L2+1)
37 x(k*L+j+1)
38 end
39 end
40 end
41

```










Information for the selected variable: computed

Size	128 x 1
Class	embedded.fi
Complex	Yes
DT Mode	ScaledDouble
Signedness	Signed
WL	16
FL	15
Proposed FL	14
Percent of Current Range	100
Always Whole Number	No
SimMin	-0.9998521328458923
SimMax	0.9988979427807565
Histogram	


```

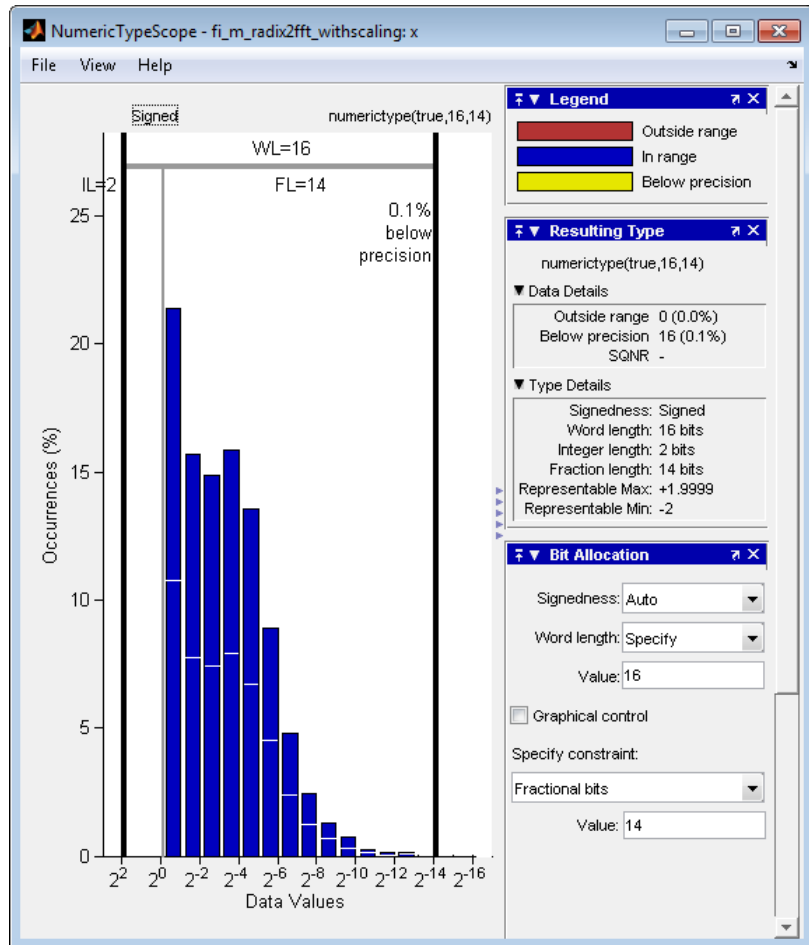
26 % Generate index variables as integer constants so they are not computed in
27 % the loop.
28 LL = int32(2.^(1:t));
29 rr = int32(n./LL);
30 LL2 = int32(LL./2);
31 for q=1:t
32 L = LL(q); r = rr(q); L2 = LL2(q);
33 for k=0:(r-1)
34 for j=0:(L2-1)
35 temp = v(L2-1+j+1) * x(k*L+j+L2+1);
36 x(k*L+j+L2+1) = bitcsc(x(k*L+j+1) - temp, 1);
37 x(k*L+j+1) = bitcsc(x(k*L+j+1) + temp, 1);
38 end
39 end
40 end
41

```

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Proposed FL	Percent of Current Range	Always Whole Number	SimMin	SimMax	Histogram
3	n	Local	1 x 1	double	No	-	-	-	-	-	-	Yes	128	128	
4	t	Local	1 x 1	double	No	-	-	-	-	-	-	Yes	7	7	
0	q	Local	1 x 1	double	No	-	-	-	-	-	-	Yes	1	7	
1	x	IO	128 x 1	embedded.fi	Yes	ScaledDouble	Signed	16	15	14	100	No	-0.9998521328458923	0.9988979427807565	
2	w	Input	127 x 1	embedded.fi	Yes	ScaledDouble	Signed	16	14	14	51	No	-1	1	
14	temp	Local	1 x 1	embedded.fi	Yes	ScaledDouble	Signed	33	29	31	13	No	-0.999895500048871	0.9991316043326937	
5	LL	Local	1 x 7	int32	No	-	-	-	-	-	-	Yes	2	128	
6	rr	Local	1 x 7	int32	No	-	-	-	-	-	-	Yes	1	64	
7	LL2	Local	1 x 7	int32	No	-	-	-	-	-	-	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.

# showInstrumentationResults

---

```
clearInstrumentationResults testfft_instrumented
```

- 7 Clear the MEX function, then delete temporary files.

```
clear testfft_instrumented;
tempdirObj.cleanUp;
```

## See Also

[fiaccel](#) | [clearInstrumentationResults](#) | [buildInstrumentedMex](#) | [NumericTypeScope](#) | [codegen](#) | [mex](#)

---

<b>Purpose</b>	Sine of fixed-point values
<b>Syntax</b>	<code>y = sin(theta)</code>
<b>Description</b>	<code>y = sin(theta)</code> returns the sine of fi input <code>theta</code> using a table-lookup algorithm.
<b>Input Arguments</b>	<p><b>theta</b></p> <p><code>theta</code> 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 <code>theta</code> are:</p> <ul style="list-style-type: none"><li>• fi single</li><li>• fi double</li><li>• fi fixed-point with binary point scaling</li><li>• fi scaled double with binary point scaling</li></ul>
<b>Output Arguments</b>	<p><b>y</b></p> <p><code>y</code> is the sine of <code>theta</code>. <code>y</code> is a signed, fixed-point number in the range [-1,1]. It has a 16-bit word length and 15-bit fraction length (<code>numericType(1,16,15)</code>) This sine calculation is accurate only to within the top 16 most-significant bits of the input.</p>
<b>Definitions</b>	<p><b>Sine</b></p> <p>The sine of angle <math>\Theta</math> is defined as</p> $\sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}$
<b>Examples</b>	<p>Calculate the sine of fixed-point input values.</p> <pre>theta = fi([-pi/2,-pi/3,-pi/4 0, pi/4,pi/3,pi/2])</pre>

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

## Algorithms

The `sin` function computes the sine of fixed-point input using an 8-bit lookup table as follows:

- 1 Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
- 2 Perform a modulo  $2\pi$ , so the input is in the range  $[0, 2\pi)$  radians.
- 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`.

### **See Also**

`sin` | `angle` | `cos` | `atan2` | `cordicsin` | `cordiccos`

### **Related Examples**

- Demo: Fixed-Point Sine and Cosine Calculation

# sign

---

**Purpose** 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.

**Purpose** Single-precision floating-point real-world value of `fi` object

**Syntax** `single(a)`

**Description** Fixed-point numbers can be represented as

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

or, equivalently as

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

`single(a)` returns the real-world value of a `fi` object in single-precision floating point.

**See Also** `double`

# size

---

**Purpose**      Array dimensions

**Description**      Refer to the MATLAB size reference page for more information.



**Purpose** Create volumetric slice plot

**Description** Refer to the MATLAB `slice` reference page for more information.

## sort

---

**Purpose** Sort elements of real-valued `fi` object in ascending or descending order

**Description** Refer to the MATLAB `sort` reference page for more information.

**Purpose** Visualize sparsity pattern

**Description** Refer to the MATLAB spy reference page for more information.

# sqrt

---

**Purpose** 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.

`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 2-687.

`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. 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 2-687.

`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.

<b>Data Type of Input fi Object a</b>	<b>Data Type of numerictype object T</b>	<b>Data Type of Output c</b>
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

# squeeze

---

**Purpose** Remove singleton dimensions

**Description** Refer to the MATLAB `squeeze` reference page for more information.

**Purpose** Create staircase graph

**Description** Refer to the MATLAB `stairs` reference page for more information.

# stem

---

**Purpose** Plot discrete sequence data

**Description** Refer to the MATLAB stem reference page for more information.



**Purpose** Plot 3-D discrete sequence data

**Description** Refer to the MATLAB `stem3` reference page for more information.

# storedInteger

---

**Purpose** 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

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

or, equivalently as

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

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)
```

## See Also

`int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `storedIntegerToDouble`

# storedIntegerToDouble

---

**Purpose** 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)
```

**See Also** `storedInteger` | `fi` | `class`

**Purpose** Create 3-D stream ribbon plot

**Description** Refer to the MATLAB `streamribbon` reference page for more information.

# streamslice

---

**Purpose** Draw streamlines in slice planes

**Description** Refer to the MATLAB `streamslice` reference page for more information.

**Purpose** Create 3-D stream tube plot

**Description** Refer to the MATLAB `streamtube` reference page for more information.

# stripscaling

---

**Purpose** 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 =

 0.1000000000000001
 s48,47
b = stripscaling(a)

b =

 14073748835533
 s48,0
bin(a)

ans =

0000110011001100110011001100110011001100110011001100110011001101

bin(b)

ans =

0000110011001100110011001100110011001100110011001100110011001101
```



Notice that the stored integer values of **a** and **b** are identical, while their real-world values are different.

# sub

---

**Purpose** 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.

## See Also

`add` | `divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide`

# subsasgn

---

**Purpose** 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` — String containing `'()''`, `'{}'`, or `'.'` specifying the subscript type
- `subs` — Cell array or string 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 the string `':'`.

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);
```

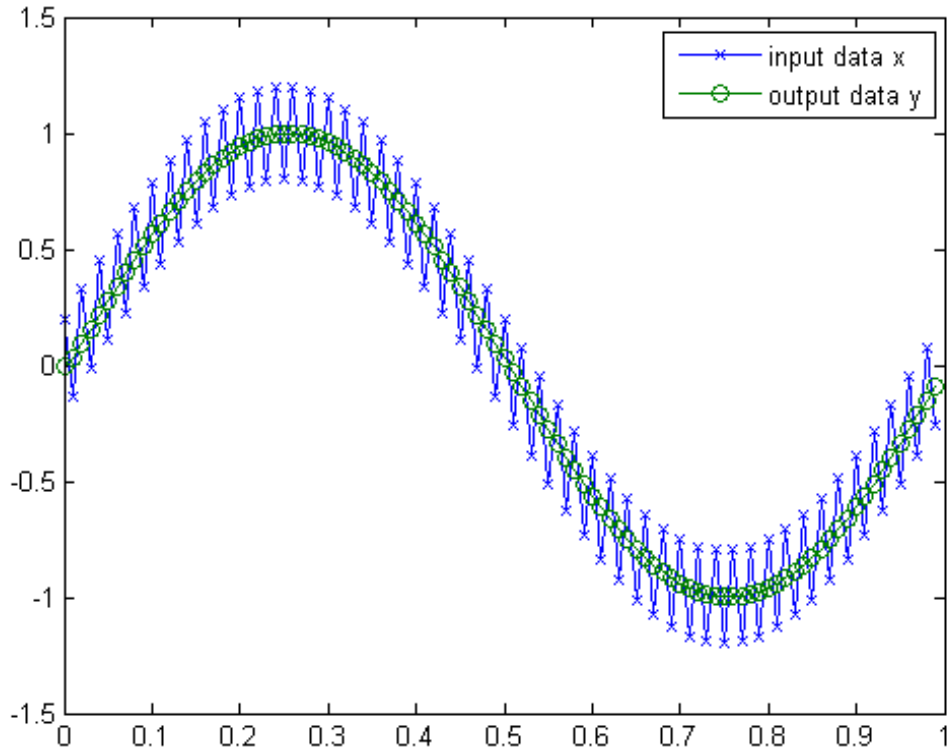
The following loop takes a running average of the input  $x$  using the coefficients in  $b$ . Notice that `acc` is assigned into `acc(1)=...` versus using `acc=...`, which would overwrite and change the data type of `acc`.

```
for k = 2:n
 acc(1) = b(1)*x(k);
 acc(1) = acc + b(2)*x(k-1);
 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')
```

# subsasgn



The log report shows the minimum and maximum logged values and ranges of the variables used. Because `acc` is assigned into, rather than over written, these logs reflect the accumulated minimum and maximum values.

```
logreport(x, y, b, acc)
```

	minlog	maxlog	lowerbound	upperbound
x	-1.200012	1.197998	-2	1.999939
y	-0.9990234	0.9990234	-2	1.999939



```
 b 0.5 0.5 -1 0.99999
acc -0.9990234 0.9989929 -512 5
```

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)
```

## See Also

```
subsref
```

## Related Examples

- “Cast fi Objects”

# subsref

---

**Purpose**           Subscribed reference

**Description**       Refer to the MATLAB `subsref` reference page for more information.

**Purpose**

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`.

# SUM

---

## 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.

## 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 =

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 =
```

```
 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 =
```

```
 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  $\text{ceil}(\log_2(\text{size}(A,1)))=2$ . 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 =
```

```
 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 =
```

```
 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 `ceil(log2(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 =
```

## sum

---

```
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 =
```

```
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`.

### See Also

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



**Purpose** Create 3-D shaded surface plot

**Description** Refer to the MATLAB surf reference page for more information.

# surf

---

**Purpose** Create 3-D shaded surface plot with contour plot

**Description** Refer to the MATLAB `surf` reference page for more information.

**Purpose** Create surface plot with colormap-based lighting

**Description** Refer to the MATLAB `surf1` reference page for more information.

# surfnorm

---

**Purpose** Compute and display 3-D surface normals

**Description** Refer to the MATLAB `surfnorm` reference page for more information.

**Purpose** Create text object in current axes

**Description** Refer to the MATLAB text reference page for more information.

# times

---

**Purpose** 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`.

## 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.

**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 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 =
```

```
 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 `fi` 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 =
```

```
 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 =
```

```
 2 4 7
 9 0 2
```

```
 DataTypeMode: Fixed-point: binary point scaling
 Signedness: Signed
 WordLength: 16
 FractionLength: 3
```

```
c=a.*b
```

```
c =
```

```
 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`.

**See Also**

`plus` | `minus` | `mtimes` | `uminus`

# toeplitz

---

**Purpose** 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

---

```
toeplitz(x,a)
```

```
ans =
```

	1	2	3
	2.7188	1	2
	3.1563	2.7188	1
s8,5			

**Purpose** Convert numerictype or quantizer object to string

**Syntax** `s = tostring(T)`  
`s = tostring(q)`

**Description** `s = tostring(T)` converts numerictype object T to a string s such that `eval(s)` would create a numerictype object with the same properties as T.

`s = tostring(q)` converts quantizer object q to a string s. After converting q to a string, the function `eval(s)` can use s to create a quantizer object with the same properties as q.

**Examples** This example uses the `tostring` function to convert a numerictype object T to a string s

```
T = numerictype(true,16,15);
s = tostring(T);
T1 = eval(s);
isequal(T,T1)
```

```
ans =
```

```
1
```

**See Also** `eval` | `numerictype` | `quantizer`

# transpose

---

**Purpose** Transpose operation

**Description** Refer to the MATLAB transpose reference page for more information.

**Purpose** Plot picture of tree

**Description** Refer to the MATLAB `treeplot` reference page for more information.

# tril

---

**Purpose** Lower triangular part of matrix

**Description** Refer to the MATLAB `tril` reference page for more information.



**Purpose** Create triangular mesh plot

**Description** Refer to the MATLAB `trimesh` reference page for more information.

# triplot

---

**Purpose** Create 2-D triangular plot

**Description** Refer to the MATLAB `triplot` reference page for more information.

**Purpose** Create triangular surface plot

**Description** Refer to the MATLAB `trisurf` reference page for more information.

# triu

---

**Purpose** Upper triangular part of matrix

**Description** Refer to the MATLAB `triu` reference page for more information.

**Purpose**

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 2-345
- “fimath Properties” on page 2-736
- “numericity Properties” on page 2-347

These properties are described in detail in “fi Object Properties” on page 1-2 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 1-2.

### **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
```

**See Also**

```
fi | fimath | fipref | isfimathlocal | numericitytype | quantizer |
sfi
```

# uint8

---

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint16` | `uint32` | `uint64`

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint32` | `uint64`

# uint32

---

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint64`

**Purpose** 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
```

**See Also** `storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

# uminus

---

**Purpose** 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,  $-(-1) = -1$  :

```
fioref('NumericTypeDisplay','short', ...
 'fimathDisplay','none');
format short g
a = fi(-1,true,8,7,'OverflowMode','wrap')

a =

 -1
 s8,7
-a

ans =

 -1
 s8,7
b = fi([-1-i -1-i],true,8,7,'OverflowMode','wrap')

b =

 -1 - 1i -1 - 1i
 s8,7
-b

ans =

 -1 - 1i -1 - 1i
```



b' s8,7

ans =

-1 - 1i  
-1 - 1i  
s8,7

When saturation occurs,  $-(-1) = 0.99\dots$  :

c = fi(-1,true,8,7,'OverflowMode','saturate')

c =

-1  
s8,7  
-c

ans =

0.99219  
s8,7  
d = fi([-1-i -1-i],true,8,7,'OverflowMode','saturate')

d =

-1 - 1i -1 - 1i  
s8,7  
-d

ans =

0.99219 + 0.99219i 0.99219 + 0.99219i  
s8,7  
d'

# uminus

---

ans =

```
-1 + 0.99219i
-1 + 0.99219i
s8,7
```

## See Also

plus | minus | mtimes | times

**Purpose** 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
```

# unitquantize

---

## See Also

`eps` | `quantize` | `quantizer` | `unitquantizer`

**Purpose** 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`

# unshiftdata

---

**Purpose** 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

```

# unshiftdata

---

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

`ipermute` | `shiftdata` | `shiftdim`



**Purpose**          Unary plus

**Description**      Refer to the MATLAB `uplus` reference page for more information.

# upperbound

---

**Purpose** 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)`.

**See Also** `eps` | `intmax` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin`

---

<b>Purpose</b>	Vertically concatenate multiple <code>fi</code> objects
<b>Syntax</b>	<code>c = vertcat(a,b,...)</code> <code>[a; b; ...]</code> <code>[a;b]</code>
<b>Description</b>	<p><code>c = vertcat(a,b,...)</code> is called for the syntax <code>[a; b; ...]</code> when any of <code>a</code>, <code>b</code>, <code>...</code>, is a <code>fi</code> object.</p> <p><code>[a;b]</code> is the vertical concatenation of matrices <code>a</code> and <code>b</code>. <code>a</code> and <code>b</code> 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.</p> <p>Horizontal and vertical concatenation can be combined, as in <code>[1 2;3 4]</code>.</p> <p><code>[a b; c]</code> is allowed if the number of rows of <code>a</code> equals the number of rows of <code>b</code>, and if the number of columns of <code>a</code> plus the number of columns of <code>b</code> equals the number of columns of <code>c</code>.</p> <p>The matrices in a concatenation expression can themselves be formed via a concatenation, as in <code>[a b;[c d]]</code>.</p> <hr/> <p><b>Note</b> The <code>fi</code>math and <code>numeric</code>type objects of a concatenated matrix of <code>fi</code> objects <code>c</code> are taken from the leftmost <code>fi</code> object in the list <code>(a,b,...)</code>.</p> <hr/>
<b>See Also</b>	<code>horzcat</code>

# voronoi

---

**Purpose** Create Voronoi diagram

**Description** Refer to the MATLAB `voronoi` reference page for more information.

**Purpose** Create n-D Voronoi diagram

**Description** Refer to the MATLAB `voronoin` reference page for more information.

# waterfall

---

**Purpose** Create waterfall plot

**Description** Refer to the MATLAB waterfall reference page for more information.

**Purpose** 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** `fi` | `fractionlength` | `exponentlength` | `numerictype` | `quantizer`

# xlim

---

**Purpose** Set or query x-axis limits

**Description** Refer to the MATLAB `xlim` reference page for more information.



**Purpose** Logical exclusive-OR

**Description** Refer to the MATLAB xor reference page for more information.

# ylim

---

**Purpose** Set or query y-axis limits

**Description** Refer to the MATLAB `ylim` reference page for more information.

**Purpose**

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 0 with the same numeric type, 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)`.

**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.

## Examples

### 2-D Array of Zeros With Fixed-Point Attributes

Create a 2-by-3 array of zeros 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 2-by-3 array of zeros that has the same numeric type properties as `p`.

```
X = zeros(2,3,'like',p)
```

```
X =
```

```
 0 0 0
 0 0 0
```

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

### Size Defined by Existing Array

Define a 3-by-2 array `A`.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz =
```

```
 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 numeric type properties as `p`.

```
X = zeros(sz, 'like', p)
```

```
X =
```

```
 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 `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 zeros that has the same numeric type properties as `p`.

```
X = zeros(4, 'like', p)
```

```
X =
```

```
 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 = [1.0000000000000000 -0.942809041582063 0.3333333333333333];

% 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
```

### See Also

[cast](#) | [ones](#) | [zeros](#)

### Related Examples

- “Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros”

### Concepts

- “Manual Fixed-Point Conversion Workflow”
- “Manual Fixed-Point Conversion Best Practices”

**Purpose** Set or query z-axis limits

**Description** Refer to the MATLAB `zlim` reference page for more information.



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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

*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

$$\textit{real-world value} = \textit{mantissa} \times 2^{\textit{exponent}}$$

2. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$\textit{exponent} = -1 \times \textit{fraction length}$$

*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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

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

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

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

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

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}$$

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.

*See also* binary word, bit, overflow

**incorrect range**

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.

*See also* range analysis

**integer**

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.

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

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

or

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

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}$$

*See also* bias, fixed-point representation, fractional slope, integer, real-world value, slope

### **integer length**

Number of bits to the left of the binary point in a fixed-point representation of a number.

*See also* binary point, bit, fixed-point representation, fraction length, integer

### **least significant bit (LSB)**

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

$$\text{weight of LSB} = 2^{-\text{fraction length}}$$

*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

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

*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**

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,

0000\_\_.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$  \_\_.

*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.

*See also* bit, data type, quantization

**radix point**

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.

*See also* binary point, bit, fraction, integer, sign bit

**range**

Span of numbers that a certain data type can represent.

*See also* data type, full range, precision, representable range

**range analysis**

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.

**real-world value**

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

$$\text{slope} = \text{fractional slope} \times 2^{\text{exponent}}$$

*See also* integer

**representable 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 full range.

**resolution**

*See* **precision**

**rounding**

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.



*See also* bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

**saturation**

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.

*See also* overflow, wrapping

**scaled double**

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.

**scaling**

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.
2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

*See also* bias, fixed-point representation, integer, slope

**shift**

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 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

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

*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:

$$\textit{real - world value} = \textit{stored integer}$$

In [Slope Bias] representation, fixed-point numbers can be represented as

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

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$\textit{real - world value} = \textit{stored integer} \times 2^{-\textit{fraction length}} = \textit{stored integer} \times 2^0$$

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 [http://www.mathworks.com/company/newsletters/news\\_notes/clevescorner/index.html](http://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.