Fixed-Point Toolbox

For Use with MATLAB®

■ Computation

■ Visualization

■ Programming

User's Guide

Version 1

The MathWorks

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Version 1.1 (Release 14SP1)
March 2005	Online only	Version 1.2 (Release 14SP2)
September 2005	Online only	Version 1.3 (Release 14SP3)
October 2005	Second printing	Version 1.3
March 2006	Online only	Version 1.4 (R2006a)

Contents

5 Working with fipref Objects

6 Working with numerictype Objects

7 Working with quantizer Objects

8 Interoperability with Other Products

9 Property Reference

10 Functions — By Category

Getting Started

What Is the Fixed-Point Toolbox?

The Fixed-Point Toolbox provides fixed-point data types in MATLAB® and enables algorithm development by providing fixed-point arithmetic. The Fixed-Point Toolbox enables you to create the following types of objects:

- `fi` — Defines a fixed-point numeric object in the MATLAB workspace. Each `fi` object is composed of value data, a `fimath` object, and a `numerictype` object.
- `fimath` — Governs how overloaded arithmetic operators work with `fi` objects
- `fipref` — Defines the display, logging, and data type override preferences of `fi` objects
- `numerictype` — Defines the data type and scaling attributes of `fi` objects
- `quantizer` — Quantizes data sets

Features

The Fixed-Point Toolbox provides you with

- The ability to define fixed-point data types, scaling, and rounding and overflow methods in the MATLAB workspace
- Bit-true real and complex simulation
- Basic fixed-point arithmetic
  - Arithmetic operators `+`, `-`, `*`, `.*` for binary point-only and real [Slope Bias] signals
  - Division using the `divide` function for binary point-only signals
- Arbitrary word length up to `intmax('uint16')` bits
- Logging of minimums, maximums, overflows, and underflows
- Data type override with singles, doubles, or scaled doubles
- Relational, logical, and bitwise operators
- Statistics functions such as `max` and `min`
- Conversions between binary, hex, double, and built-in integers

- Interoperability with Simulink®, Signal Processing Blockset, Embedded MATLAB, and Filter Design Toolbox
- Compatibility with the Simulink To Workspace and From Workspace blocks

Getting Help

This section tells you how to get help for the Fixed-Point Toolbox in this document and at the MATLAB command line.

Getting Help in This Document

The objects of the Fixed-Point Toolbox are discussed in the following chapters:

- Chapter 3, "Working with fi Objects"
- Chapter 4, "Working with fimath Objects"
- Chapter 5, "Working with fipref Objects"
- Chapter 6, "Working with numerictype Objects"
- Chapter 7, "Working with quantizer Objects"

To get in-depth information about the properties of these objects, refer to Chapter 9, "Property Reference".

To get in-depth information about the functions of these objects, refer to the Function Reference.

Getting Help at the MATLAB Command Line

To get command-line help for Fixed-Point Toolbox objects, type

```
help objectname
```

For example,

```
help fi
help fimath
help fipref
help numerictype
help quantizer
```

To invoke Help Browser documentation for Fixed-Point Toolbox functions from the MATLAB command line, type

```
doc fixedpoint/functionname
```

For example,

```
doc fixedpoint/int
doc fixedpoint/add
doc fixedpoint/savefipref
doc fixedpoint/quantize
```

Display Settings

In the Fixed-Point Toolbox, the display of `fi` objects is determined by the `fipref` object. Throughout this User's Guide, code examples of `fi` objects are usually shown as they appear when the `fipref` properties are set as follows:

- `NumberDisplay` — `'RealWorldValue'`
- `NumericTypeDisplay` — `'full'`
- `FimathDisplay` — `'none'`

For example,

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'full', 'FimathDisplay', 'none')

p =

         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'none'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'

a = fi(pi)

a =

    3.1416


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

In other cases, it makes sense to also show the `fimath` object display:

- `NumberDisplay` — `'RealWorldValue'`
- `NumericTypeDisplay` — `'full'`
- `FimathDisplay` — `'full'`

For example,

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'full', 'FimathDisplay', 'full')

p =

         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'

a = fi(pi)

a =

    3.1416


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

For more information, refer to Chapter 5, “Working with fipref Objects”.

Demos

You can access demos in the **Demos** tab of the **Help Navigator** window. The Fixed-Point Toolbox includes the following demos:

- Number Circle — Illustrates the definitions of unsigned and signed two's complement integer and fixed-point numbers
- `fi` Basics — Demonstrates the basic use of the fixed-point object `fi`
- `fi` Binary Point Scaling — Explains binary point-only scaling
- Fixed-Point Doubles Override, Min/Max Logging, and Scaling — Steps through the workflow of using doubles override and min/max logging in the Fixed-Point Toolbox to choose appropriate scaling for a fixed-point algorithm
- Fixed-Point C Development — Shows how to use the parameters from a fixed-point MATLAB program in a fixed-point C program
- Fixed-Point Algorithm Development — Presents the development and verification of a simple fixed-point algorithm
- Fixed-Point Fast Fourier Transform (FFT) — Provides an example of converting a textbook Fast Fourier Transform algorithm into fixed-point MATLAB code and then into fixed-point C code
- Analysis of a Fixed-Point State-Space System with Limit Cycles — Demonstrates a limit cycle detection routine applied to a state-space system
- Quantization Error — Demonstrates the statistics of the error when signals are quantized using various rounding methods

Fixed-Point Concepts

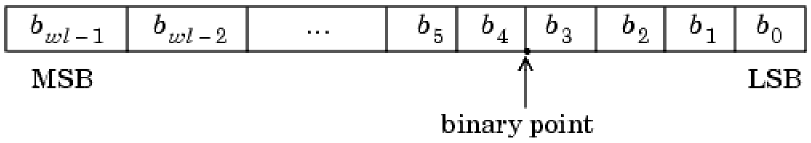
Fixed-Point Data Types

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

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

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

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



where

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

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

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

Two's complement is the most common representation of signed fixed-point numbers and is the only representation used by the Fixed-Point Toolbox. Refer to "Two's Complement" on page 2-9 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

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

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

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

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

or

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

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

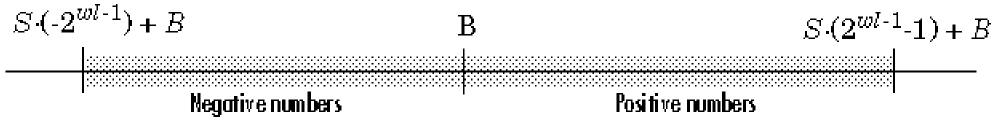
Note For examples of binary point-only scaling, see the Fixed-Point Toolbox demo "`fi` Binary Point Scaling."

Precision and Range

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

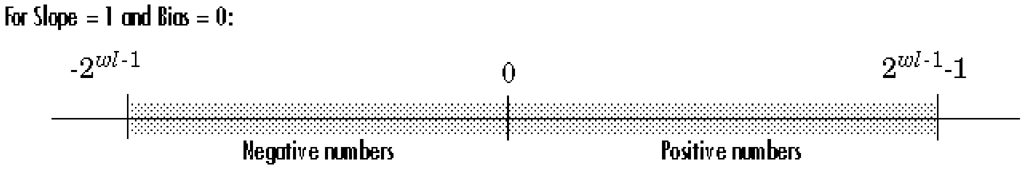
Range

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



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

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



Overflow Handling

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

The Fixed-Point Toolbox allows you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. Refer to “Modulo Arithmetic” on page 2-9 for more information.

When you create a `fi` object in the Fixed-Point Toolbox, any overflows are saturated. The `OverflowMode` property of the default `fimath` object is `saturate`. You can log overflows and underflows by setting the `LoggingMode` property of the `fipref` object to `on`. Refer to “LoggingMode” on page 9-14 for more information.

Precision

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

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

Rounding Methods

One of the limitations of representing numbers with finite precision is that not every number in the available range can be represented exactly. When the result of a fixed-point calculation is a number that cannot be represented exactly by the data type and scaling being used, precision is lost. A rounding method must be used to cast the result to a representable number. The Fixed-Point Toolbox currently supports the following rounding methods:

- `ceil` rounds to the closest representable number in the direction of positive infinity.
- `convergent` rounds to the closest representable integer. In the case of a tie, it rounds to the nearest even stored integer. This is the least biased rounding method provided by the Fixed-Point Toolbox.

- `fix` rounds to the closest representable integer in the direction of zero.
- `floor`, which is equivalent to two's complement truncation, rounds to the closest representable number in the direction of negative infinity.
- `nearest` rounds to the closest representable integer. In the case of a tie, it rounds to the closest representable integer in the direction of positive infinity. This is the default rounding method for `fi` object creation and `fi` arithmetic.
- `round` rounds to the closest representable integer. In the case of a tie, it rounds positive numbers to the closest representable integer in the direction of positive infinity, and it rounds negative numbers to the closest representable integer in the direction of negative infinity.

Arithmetic Operations

The following sections describe the arithmetic operations used by the Fixed-Point Toolbox:

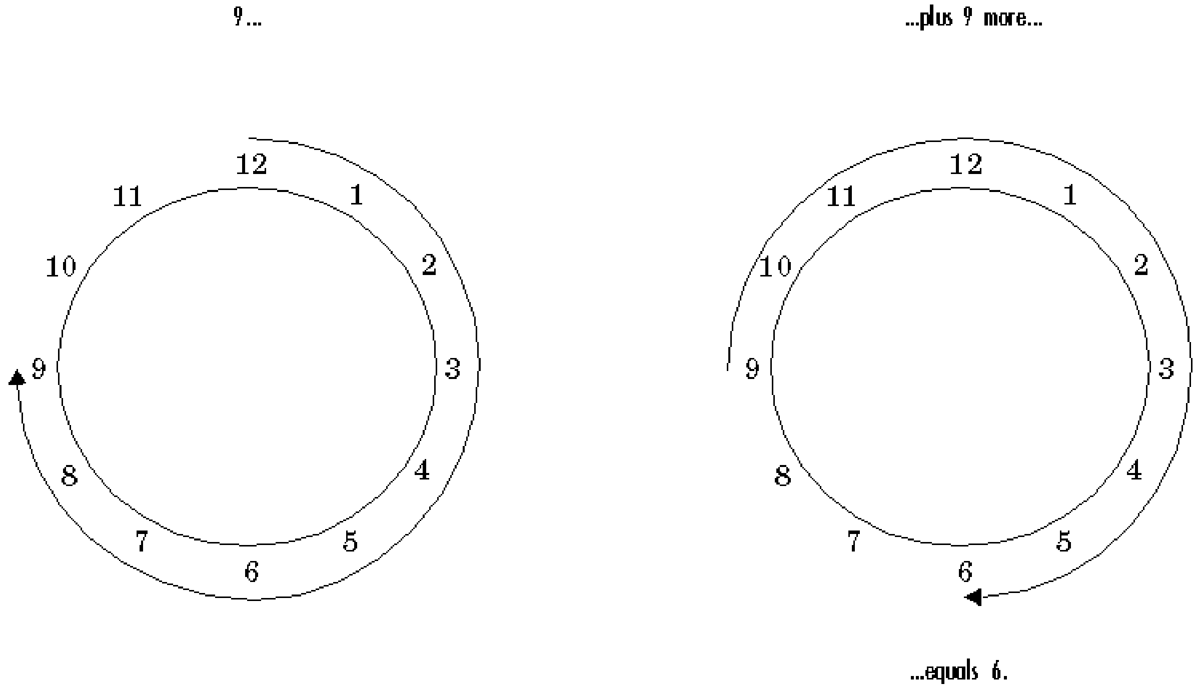
- “Modulo Arithmetic” on page 2-9
- “Two’s Complement” on page 2-9
- “Addition and Subtraction” on page 2-10
- “Multiplication” on page 2-11
- “Casts” on page 2-17

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

Modulo Arithmetic

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

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



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

Two's Complement

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

$$\begin{aligned} 01 &= (0 + 2^0) = 1 \\ 11 &= ((-2^1) + (2^0)) = (-2 + 1) = -1 \end{aligned}$$

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

1 Take the one's complement, or "flip the bits."

2 Add a 1 using binary math.

3 Discard any bits carried beyond the original word length.

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

$$11010 \longrightarrow 00101$$

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

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

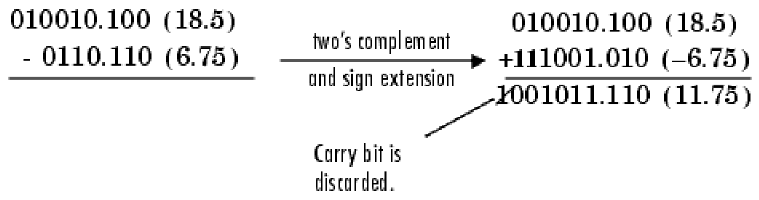
Addition and Subtraction

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

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

```
  010010.1     (18.5)
+ 0110.110     (6.75)
-----------
  011001.010   (25.25)
```

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



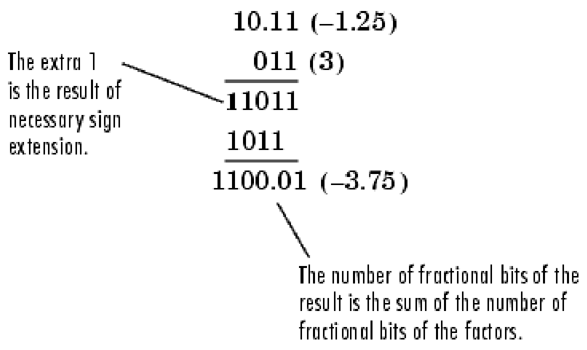
The default `fimath` object has a value of `1` (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

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

Multiplication

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

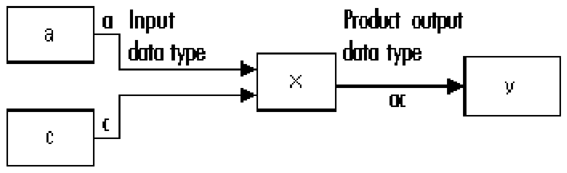
For example, consider the multiplication of 10.11 (-1.25) with 011 (3):


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

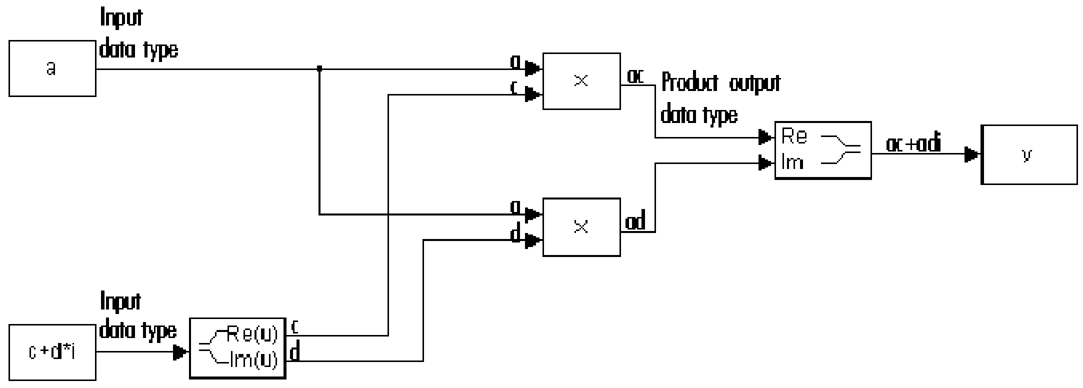
Multiplication Data Types

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

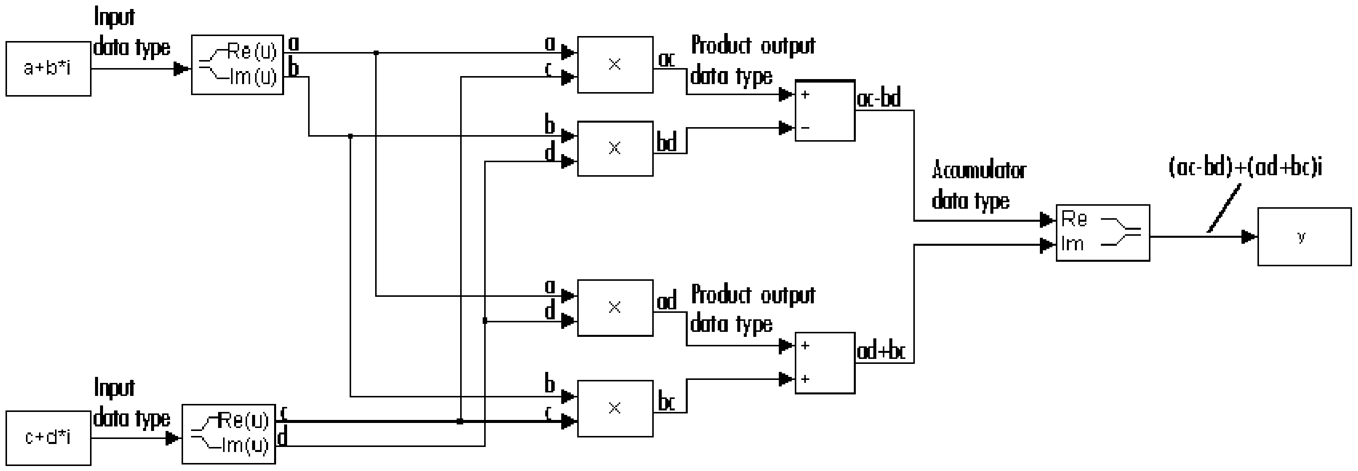
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in the Fixed-Point Toolbox. The output of this multiplication is in the product data type, which is governed by the `fimath` `ProductMode` property:



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



Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers in the Fixed-Point Toolbox. Note that the output of the multiplication is in the sum data type, which is governed by the `fimath` `SumMode` property. The product data type is determined by the `fimath` `ProductMode` property:



Multiplication with fimath

In the following examples, let

- `F = fimath('ProductMode','FullPrecision',...`

  `'SumMode','FullPrecision')`
- `T1 = numerictype('WordLength',24,'FractionLength',20)`
- `T2 = numerictype('WordLength',16,'FractionLength',10)`

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

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

x =

     5


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

 y = fi(10, T2, F)

y =

    10


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 16
        FractionLength: 10
 z = x*y

z =

    50


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

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

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

x =

     5


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

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

y =

  10.0000 + 2.0000i


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

z = x*y

z =

  50.0000 +10.0000i


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

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

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

x =

   5.0000 + 6.0000i


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

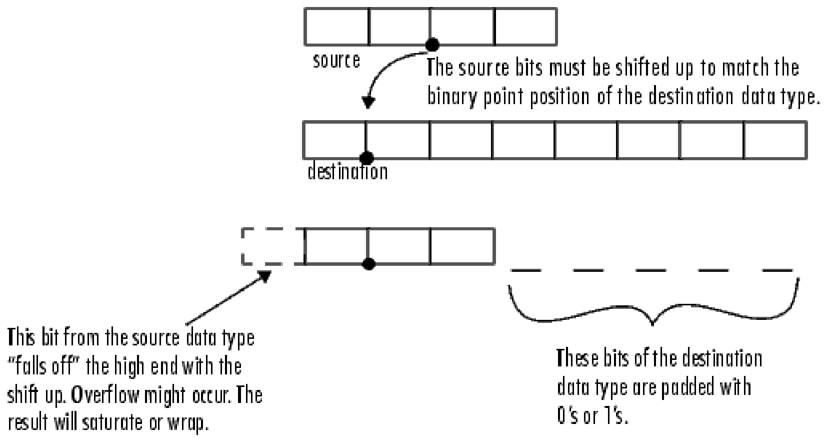
```
y = fi(10+2i,T2,F)

y =

  10.0000 + 2.0000i


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

z = x*y

z =

  38.0000 +70.0000i


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

Casts

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

Casting from a Shorter Data Type to a Longer Data Type

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


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

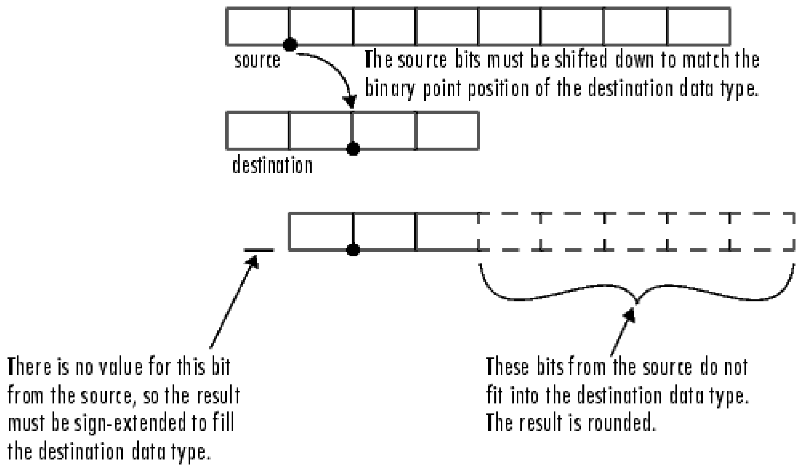
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either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

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

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

Casting from a Longer Data Type to a Shorter Data Type

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


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

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

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

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

fi Objects Compared to C Integer Data Types

The following sections compare the `fi` object with fixed-point data types and operations in C:

- "Integer Data Types" on page 2-20
- "Unary Conversions" on page 2-22
- "Binary Conversions" on page 2-23
- "Overflow Handling" on page 2-25

In these sections, the information on ANSI C is adapted from Samuel P. Harbison and Guy L. Steele Jr., *C: A reference manual*, 3rd ed., Prentice Hall, 1991.

Integer Data Types

This section compares the numerical range of `fi` integer data types to the minimum numerical ranges of ANSI C integer data types.

ANSI C Integer Data Types

The following table shows the minimum ranges of ANSI C integer data types. The integer ranges can be larger than or equal to those shown, but cannot be smaller. The range of a `long` must be larger than or equal to the range of an `int`, which must be larger than or equal to the range of a `short`.

Note that the minimum ANSI C ranges are large enough to accommodate one's complement or sign/magnitude representation, but not two's complement representation. In the one's complement and sign/magnitude representations, a signed integer with n bits has a range from $-2^{n-1}+1$ to $2^{n-1}-1$, inclusive. In both of these representations, an equal number of positive and negative numbers are represented, and zero is represented twice.

Integer Type	Minimum	Maximum
`signed char`	-127	127
`unsigned char`	0	255

Integer Type	Minimum	Maximum
`short int`	-32,767	32,767
`unsigned short`	0	65,535
`int`	-32,767	32,767
`unsigned int`	0	65,535
`long int`	-2,147,483,647	2,147,483,647
`unsigned long`	0	4,294,967,295

fi Integer Data Types

The following table lists the numerical ranges of the integer data types of the `fi` object, in particular those equivalent to the C integer data types. The ranges are large enough to accommodate the two's complement representation, which is the only signed binary encoding technique supported by the Fixed-Point Toolbox. In the two's complement representation, a signed integer with n bits has a range from -2^{n-1} to $2^{n-1}-1$, inclusive. An unsigned integer with n bits has a range from 0 to $2^{n}-1$, inclusive. The negative side of the range has one more value than the positive side, and zero is represented uniquely.

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

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
`fi(x,0,16,0)`	No	16	0	0	65,535	`unsigned short`
`fi(x,1,32,0)`	Yes	32	0	-2,147,483,648	2,147,483,647	`long int`
`fi(x,0,32,0)`	No	32	0	0	4,294,967,295	`unsigned long`

Unary Conversions

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

ANSI C Usual Unary Conversions

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

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

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

fi Usual Unary Conversions

The following table shows the `fi` unary conversions:

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

Binary Conversions

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

ANSI C Usual Binary Conversions

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

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

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

fi Usual Binary Conversions

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

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

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	uint8	• Signed = 0 • WordLength = 8 • FractionLength = 0
fi	int16	• Signed = 1 • WordLength = 16 • FractionLength = 0
fi	uint16	• Signed = 0 • WordLength = 16 • FractionLength = 0
fi	int32	• Signed = 1 • WordLength = 32 • FractionLength = 0
fi	uint32	• Signed = 0 • WordLength = 32 • FractionLength = 0

Overflow Handling

The following sections compare how overflows are handled in ANSI C and the Fixed-Point Toolbox.

ANSI C Overflow Handling

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

The results of unsigned integer overflows wrap in ANSI C.

fi Overflow Handling

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

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

fimath Object Properties Related to Overflow Handling	Property Value	Description
`OverflowMode`	`'saturate'`	Overflows are saturated to the maximum or minimum value in the range.
	`'wrap'`	Overflows wrap using modulo arithmetic if unsigned, two's complement wrap if signed.
`ProductMode`	`'FullPrecision'`	Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than `MaxProductWordLength`. The rules for computing the resulting product word and fraction lengths are given in "ProductMode" on page 9-7.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	`'KeepLSB'`	The least significant bits of the product are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations. The resulting word length is determined by the `ProductWordLength` property. If `ProductWordLength` is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If `ProductWordLength` is less than is necessary for the full-precision product, then overflow occurs. The rule for computing the resulting product fraction length is given in "ProductMode" on page 9-7.
	`'KeepMSB'`	The most significant bits of the product are kept. Overflow is prevented, but precision may be lost. The resulting word length is determined by the `ProductWordLength` property. If `ProductWordLength` is greater than is necessary for the full-precision product, then the result is stored in the most significant bits. If `ProductWordLength` is less than is necessary for the full-precision product, then rounding occurs. The rule for computing the resulting product fraction length is given in "ProductMode" on page 9-7.
	`'SpecifyPrecision'`	You can specify both the word length and the fraction length of the resulting product.

fimath Object Properties Related to Overflow Handling	Property Value	Description
`ProductWordLength`	Positive integer	The word length of product results when `ProductMode` is `'KeepLSB'`, `'KeepMSB'`, or `'SpecifyPrecision'`.
`MaxProductWordLength`	Positive integer	The maximum product word length allowed when `ProductMode` is `'FullPrecision'`. The default is 128 bits. The maximum is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.
`ProductFractionLength`	Integer	The fraction length of product results when `ProductMode` is `'Specify Precision'`.
`SumMode`	`'FullPrecision'`	Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than `MaxSumWordLength`. The rules for computing the resulting sum word and fraction lengths are given in “SumMode” on page 9-11.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	`'KeepLSB'`	The least significant bits of the sum are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations. The resulting word length is determined by the `SumWordLength` property. If `SumWordLength` is greater than is necessary for the full-precision sum, then the result is stored in the least significant bits. If `SumWordLength` is less than is necessary for the full-precision sum, then overflow occurs. The rule for computing the resulting sum fraction length is given in “SumMode” on page 9-11.
	`'KeepMSB'`	The most significant bits of the sum are kept. Overflow is prevented, but precision may be lost. The resulting word length is determined by the `SumWordLength` property. If `SumWordLength` is greater than is necessary for the full-precision sum, then the result is stored in the most significant bits. If `SumWordLength` is less than is necessary for the full-precision sum, then rounding occurs. The rule for computing the resulting sum fraction length is given in “SumMode” on page 9-11.
	`'SpecifyPrecision'`	You can specify both the word length and the fraction length of the resulting sum.

fimath Object Properties Related to Overflow Handling	Property Value	Description
`SumWordLength`	Positive integer	The word length of sum results when `SumMode` is `'KeepLSB'`, `'KeepMSB'`, or `'SpecifyPrecision'`.
`MaxSumWordLength`	Positive integer	The maximum sum word length allowed when `SumMode` is `'FullPrecision'`. The default is 128 bits. The maximum is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.
`SumFractionLength`	Integer	The fraction length of sum results when `SumMode` is `'SpecifyPrecision'`.

Working with fi Objects

Constructing fi Objects

You can create `fi` objects in the Fixed-Point Toolbox in one of two ways:

- You can use the `fi` constructor function to create a new object.
- You can use the `fi` constructor function to copy an existing `fi` object.

To get started, type

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

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

```
a =

     0


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

A signed `fi` object is created with a value of 0, word length of 16 bits, and fraction length of 15 bits.

Note For information on the display format of `fi` objects, refer to “Display Settings” on page 1-6.

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

- `fi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `fi(v,s)` returns a fixed-point object with value `v`, signedness `s`, 16-bit word length, and best-precision fraction length. `s` can be `0` (false) for unsigned or `1` (true) for signed.

- `fi(v,s,w)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, and best-precision fraction length.
- `fi(v,s,w,f)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, and fraction length `f`.
- `fi(v,s,w,slope,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slope`, and `bias`.
- `fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, slope adjustment `slopeadjustmentfactor`, exponent `fixedexponent`, and bias `bias`.
- `fi(v,T)` returns a fixed-point object with value `v` and `embedded.numerictype` `T`. Refer to Chapter 6, "Working with numerictype Objects" for more information on `numerictype` objects.
- `fi(a,F)` allows you to maintain the value and `numerictype` object of `fi` object `a`, while changing its `fimath` object to `F`. Refer to Chapter 4, "Working with fimath Objects" for more information on `fimath` objects.
- `fi(v,T,F)` returns a fixed-point object with value `v`, `embedded.numerictype` `T`, and `embedded.fimath` `F`.
- `fi(...'PropertyName',PropertyValue...)` and `fi('PropertyName',PropertyValue...)` allow you to set properties for a `fi` object using property name/property value pairs.

Examples of Constructing fi Objects

For example, the following creates a `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
                   Signed: true
               WordLength: 8
```

```
          FractionLength: 3
```

The value v can also be an array.

```
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
                 Signed: true
             WordLength: 16
         FractionLength: 12
```

If you omit the argument f, it is set automatically to the best precision possible.

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

a =

    3.1563


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

If you omit w and f, they are set automatically to 16 bits and the best precision possible, respectively.

```
a = fi(pi, 1)

a =
      3.1416


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

Constructing a fi Object with Property Name/Property Value Pairs

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

```
a = fi(pi, 'roundmode', 'floor', 'overflowmode', 'wrap')

a =

    3.1415

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

               RoundMode: floor
            OverflowMode: wrap
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                 SumMode: FullPrecision
        MaxSumWordLength: 128
           CastBeforeSum: true
```

Constructing a fi Object Using a numerictype Object

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

```
T = numerictype

T =
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 16
        FractionLength: 15

a = fi(pi, T)

a =

    1.0000


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

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

```
F = fimath

F =


             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
          MaxSumWordLength: 128
             CastBeforeSum: true

a = fi(pi, T, F)

a =

    1.0000


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

                 RoundMode: nearest
              OverflowMode: saturate
               ProductMode: FullPrecision
      MaxProductWordLength: 128
                   SumMode: FullPrecision
          MaxSumWordLength: 128
             CastBeforeSum: true
```

Determining Property Precedence

Note that the value of a property is taken from the last time it is set. For example, create a `numerictype` object with a value of `true` for the `'signed'` property:

```
T = numerictype('signed', true)

T =


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

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

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

a =

    1.0000

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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Contrast this with the following `fi` object in which the `numerictype` property is specified *before* the `signed` property, so the resulting fi object is unsigned:

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

b =

    3.1416

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

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

```
              SumMode: FullPrecision
     MaxSumWordLength: 128
        CastBeforeSum: true
```

Copying a fi Object

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

```
a = fi(pi)

a =

    3.1416

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

b =

    3.1416

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

fi Object Properties

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

- "Data Properties" on page 3-10
- "fimath Properties" on page 3-10
- "numerictype Properties" on page 3-11

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`, `uint8`, `uint16`, and `uint32` to get the stored integer value of a fi object in these formats
- `oct` — Stored integer value of a `fi` object in octal

fimath Properties

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

- `fimath` — `fimath` object 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
- `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
- `RoundMode` — 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

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 numeric type attributes of a `fi` object

The following `numerictype` properties are, by transitivity, also properties of a `fi` object. The properties of the `numerictype` object listed below are not writable once the `fi` object has been created. However, you can create a copy of a `fi` object with new values specified for the `numerictype` properties:

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

These properties are described in detail in Chapter 9, "Property Reference". There are two ways to specify properties for `fi` objects in the Fixed-Point Toolbox. Refer to the following sections:

- "Setting Fixed-Point Properties at Object Creation" on page 3-12
- "Using Direct Property Referencing with fi" on page 3-13

Setting Fixed-Point Properties at Object Creation

You can set properties of `fi` objects at the time of object creation by including properties after the arguments of the `fi` constructor function. For example, to set the overflow mode to `wrap` and the rounding mode to `convergent`,

```
a = fi(pi, 'OverflowMode', 'wrap', 'RoundMode', 'convergent')

a =

    3.1416


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

```
                Signed: true
            WordLength: 16
        FractionLength: 13

             RoundMode: convergent
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Using Direct Property Referencing with fi

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

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

```
a.DataTypeMode

ans =

Fixed-point: binary point scaling
```

To set the `OverflowMode` of a,

```
a.OverflowMode = 'wrap'

a =

    3.1416


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

             RoundMode: convergent
```

```
           OverflowMode: wrap
            ProductMode: FullPrecision
   MaxProductWordLength: 128
                SumMode: FullPrecision
       MaxSumWordLength: 128
          CastBeforeSum: true
```

fi Object Functions

The functions in the following table operate directly on `fi` objects.

`abs`	`all`	`and`	`any`	`area`
`bar`	`barh`	`bin`	`bitand`	`bitcmp`
`bitget`	`bitor`	`bitshift`	`bitxor`	`buffer`
`clabel`	`comet`	`comet3`	`compass`	`complex`
`coneplot`	`conj`	`contour`	`contour3`	`contourc`
`contourf`	`ctranspose`	`dec`	`diag`	`double`
`end`	`eps`	`eq`	`errorbar`	`etreeplot`
`ezcontour`	`ezcontourf`	`ezmesh`	`ezplot`	`ezplot3`
`ezpolar`	`ezsurf`	`ezsurfc`	`feather`	`fi`
`fimath`	`fplot`	`ge`	`get`	`gplot`
`gt`	`hankel`	`hex`	`hist`	`histc`
`horzcat`	`imag`	`innerprodintbits`	`inspect`	`int`
`int8`	`int16`	`int32`	`intmax`	`intmin`
`ipermute`	`iscolumn`	`isempty`	`isequal`	`isfi`
`isfinite`	`isinf`	`isnan`	`isnumeric`	`isobject`
`ispropequal`	`isreal`	`isrow`	`isscalar`	`issigned`
`isvector`	`le`	`length`	`line`	`logical`
`lowerbound`	`lsb`	`lt`	`max`	`mesh`
`meshc`	`meshz`	`min`	`minus`	`mtimes`
`ndims`	`ne`	`not`	`numberofelements`	`numerictype`
`oct`	`or`	`patch`	`pcolor`	`permute`
`plot`	`plot3`	`plotmatrix`	`plotyy`	`plus`
`polar`	`pow2`	`quantizer`	`quiver`	`quiver3`
`range`	`real`	`realmax`	`realmin`	`repmat`
`rescale`	`reshape`	`rgbplot`	`ribbon`	`rose`

scatter	scatter3	sdec	sign	single
size	slice	spy	stairs	stem
stem3	streamribbon	streamslice	streamtube	stripscaling
subsasgn	subsref	sum	surf	surfc
surfl	surfnorm	text	times	toeplitz
transpose	treeplot	tril	trimesh	triplot
trisurf	triu	uint8	uint16	uint32
uminus	uplus	upperbound	vertcat	voronoi
voronoin	waterfall	xlim	ylim	zlim

You can learn about the functions associated with `fi` objects in the Function Reference.

The following data-access functions can be also used to get the data in a `fi` object using dot notation.

- `bin`
- `data`
- `dec`
- `double`
- `hex`
- `int`
- `oct`

For example,

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

n =

  25736
```

```
a.int

ans =

  25736

h = hex(a)

h =

6488

a.hex

ans =

6488
```

Working with fimath Objects

Constructing fimath Objects

`fimath` objects define the arithmetic attributes of `fi` objects. You can create `fimath` objects in the Fixed-Point Toolbox in one of two ways:

- You can use the `fimath` constructor function to create a new object.
- You can use the `fimath` constructor function to copy an existing `fimath` object.

To get started, type

```
F = fimath
```

to create a default `fimath` object.

```
F = fimath

F =


            RoundMode: nearest
         OverflowMode: saturate
          ProductMode: FullPrecision
 MaxProductWordLength: 128
              SumMode: FullPrecision
     MaxSumWordLength: 128
        CastBeforeSum: true
```

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

```
F = fimath;
G = F;
isequal(F,G)

ans =

     1
```

The syntax

```
F = fimath(...'PropertyName',PropertyValue...)
```

allows you to set properties for a `fimath` object at object creation with property name/property value pairs. Refer to “Setting fimath Properties at Object Creation” on page 4-5.

fimath Object Properties

The following properties of `fimath` objects are always writable:

- `CastBeforeSum` — Whether both operands are cast to the sum data type before addition
- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow-handling 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
- `RoundMode` — 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` — Word length, in bits, of the sum data type

These properties are described in detail in Chapter 9, “Property Reference”. There are three ways to specify properties for `fimath` objects in the Fixed-Point Toolbox. Refer to the following sections:

- “Setting fimath Properties at Object Creation” on page 4-5
- “Using Direct Property Referencing with fimath” on page 4-6
- “Setting fimath Properties in the Model Explorer” on page 4-7

Setting fimath Properties at Object Creation

You can set properties of `fimath` objects at the time of object creation by including properties after the arguments of the `fimath` constructor function.

For example, to set the overflow mode to `saturate` and the rounding mode to `convergent`,

```
F = fimath('OverflowMode','saturate','RoundMode','convergent')

F =


            RoundMode: convergent
         OverflowMode: saturate
          ProductMode: FullPrecision
 MaxProductWordLength: 128
              SumMode: FullPrecision
     MaxSumWordLength: 128
        CastBeforeSum: true
```

Using Direct Property Referencing with fimath

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

For example, to get the `RoundMode` of `F`,

```
F.RoundMode

ans =

convergent
```

To set the `OverflowMode` of `F`,

```
F.OverflowMode = 'wrap'

F =


            RoundMode: convergent
         OverflowMode: wrap
          ProductMode: FullPrecision
 MaxProductWordLength: 128
```

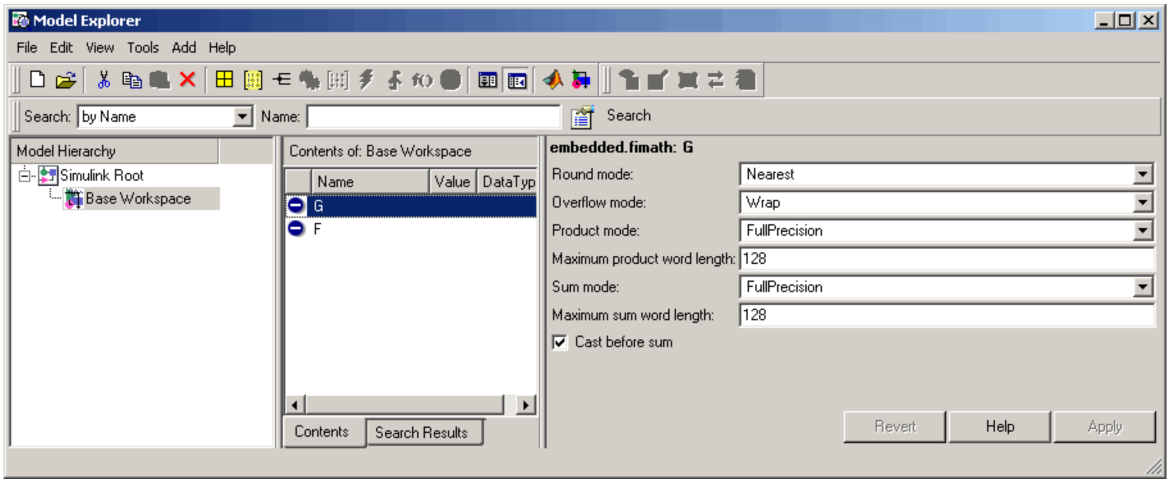
```
            SumMode: FullPrecision
   MaxSumWordLength: 128
      CastBeforeSum: true
```

Setting fimath Properties in the Model Explorer

You can view and change the properties for any `fimath` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View** > **Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The figure below shows the Model Explorer when you define the following `fimath` objects in the MATLAB workspace:

```
F = fimath

F =


             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

G = fimath('OverflowMode','wrap')

G =


             RoundMode: nearest
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```



embedded.fimath: G

Round mode: Nearest

Overflow mode: Wrap

Product mode: FullPrecision

Maximum product word length: 128

Sum mode: FullPrecision

Maximum sum word length: 128

[✓] Cast before sum

Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a `fimath` object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

Using fimath Objects to Perform Fixed-Point Arithmetic

The `fimath` object encapsulates the math properties of the Fixed-Point Toolbox, and is itself a property of the `fi` object.

Every `fi` object has a `fimath` object as a property.

```
a = fi(pi)

a =

    3.1416


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

a.fimath

ans =


             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

To perform arithmetic with `+`, `-`, `.*`, or `*`, two `fi` operands must have the same `fimath` properties.

```
a = fi(pi);
b = fi(8);
isequal(a.fimath, b.fimath)

ans =

     1

a + b

ans =

   11.1416


             DataTypeMode: Fixed-point: binary point scaling
                   Signed: true
               WordLength: 19
           FractionLength: 13

                RoundMode: nearest
             OverflowMode: saturate
              ProductMode: FullPrecision
     MaxProductWordLength: 128
                  SumMode: FullPrecision
         MaxSumWordLength: 128
            CastBeforeSum: true
```

To perform arithmetic with `+`, `-`, `.*`, or `*`, two `fi` operands must also have the same data type. For example, you can perform addition on two `fi` objects with data type `double`, but not on an object with data type `double` and one with data type `single`:

```
a = fi(3, 'DataType', 'double')

a =
      3

              DataTypeMode: double

b = fi(27, 'DataType', 'double')

b =

    27

              DataTypeMode: double

a + b

ans =

    30

              DataTypeMode: double

c = fi(12, 'DataType', 'single')

c =

    12

              DataTypeMode: single

a + c
??? Math operations are not allowed on FI objects with
  different data types.
```

Fixed-point `fi` object operands do not have to have the same scaling. Math is permitted between fixed-point and scaled doubles `fi` objects. In this sense, the scaled double data type acts as a fixed-point data type:

```
a = fi(pi)

a =
    3.1416

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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

b = fi(magic(2), 'DataTypeMode', 'Scaled double: binary point scaling')

b =

     1     3
     4     2


          DataTypeMode: Scaled double: binary point scaling
                Signed: true
            WordLength: 16
        FractionLength: 12

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

a + b

ans =
    4.1416    6.1416
    7.1416    5.1416


          DataTypeMode: Scaled double: binary point scaling
                Signed: true
            WordLength: 18
        FractionLength: 13

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Use the `divide` function to perform division with doubles, singles, or binary point-only scaling `fi` objects.

[Slope Bias] Arithmetic

Fixed-point arithmetic using the `fimath` object is supported for all binary point-only signals. Arithmetic is also supported for [Slope Bias] signals with the following restrictions:

- [Slope Bias] signals must be real.
- The `fimath` object `SumMode` and `ProductMode` properties must be set to `'SpecifyPrecision'` for sum and multiply operations, respectively.
- The `fimath` object `CastBeforeSum` property must be set to `'true'`.
- The `divide` function is not supported for [Slope Bias] signals.

```
f = fimath('SumMode', 'SpecifyPrecision', 'SumFractionLength', 16)

f =


                 RoundMode: nearest
              OverflowMode: saturate
```

```
                 ProductMode: FullPrecision
        MaxProductWordLength: 128
                     SumMode: SpecifyPrecision
               SumWordLength: 32
           SumFractionLength: 16
               CastBeforeSum: true

a = fi(pi, 'fimath', f)

a =

    3.1416

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

                   RoundMode: nearest
                OverflowMode: saturate
                 ProductMode: FullPrecision
        MaxProductWordLength: 128
                     SumMode: SpecifyPrecision
               SumWordLength: 32
           SumFractionLength: 16
               CastBeforeSum: true

b = fi(22, true, 16, 2^-8, 3, 'fimath', f)

b =

    22

                DataTypeMode: Fixed-point: slope and bias scaling
                      Signed: true
                  WordLength: 16
                       Slope: 0.00390625
                        Bias: 3

                   RoundMode: nearest
                 OverflowMode: saturate
                  ProductMode: FullPrecision
         MaxProductWordLength: 128
                      SumMode: SpecifyPrecision
                SumWordLength: 32
            SumFractionLength: 16
                CastBeforeSum: true

a + b

ans =

   25.1416

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

                    RoundMode: nearest
                 OverflowMode: saturate
                  ProductMode: FullPrecision
         MaxProductWordLength: 128
                      SumMode: SpecifyPrecision
                SumWordLength: 32
            SumFractionLength: 16
                CastBeforeSum: true
```

Setting the `SumMode` and `ProductMode` properties to `SpecifyPrecision` are mutually exclusive except when performing the `*` operation between matrices. In this case, both the `SumMode` and `ProductMode` properties must be set to `SpecifyPrecision` for [Slope Bias] signals, because both sum and multiply operations are performed while calculating the result.

Using fimath to Share Arithmetic Rules

You can use a `fimath` object to define common arithmetic rules that you would like to use for many `fi` objects. You can then create multiple `fi` objects, using the same `fimath` object for each. To do so, you also need to create a `numerictype` object to define a common data type and scaling. Refer to Chapter 6, "Working with numerictype Objects" for more information on `numerictype` objects. The following example shows the creation of a `numerictype` object and `fimath` object, which are then used to create two `fi` objects with the same `numerictype` and `fimath` attributes:

```
T = numerictype('WordLength', 32, 'FractionLength', 30)

T =


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 30

F = fimath('RoundMode', 'floor', 'OverflowMode', 'wrap')

F =


             RoundMode: floor
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

a = fi(pi, T, F)

a =

   -0.8584
```

```
          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 30

             RoundMode: floor
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

b = fi(pi/2, T, F)

b =

    1.5708


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 30

             RoundMode: floor
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Using fimath ProductMode and SumMode

The following example shows the differences among the `FullPrecision`, `KeepLSB`, `KeepMSB`, and `SpecifyPrecision` settings of the `ProductMode` and `SumMode` properties. To follow along, first set the following preferences:

```
p = fipref;
p.NumericTypeDisplay = 'short';
p.FimathDisplay = 'none';
p.LoggingMode = 'on';
F = fimath('OverflowMode','wrap','RoundMode','floor',...
  'CastBeforeSum',false);
warning off
format compact
```

Next define `fi` objects `a` and `b`. Both have signed 8–bit data types. The fraction length is automatically chosen for each `fi` object to yield the best possible precision:

```
a = fi(pi, true, 8)
a =
    3.1563
      s8,5
b = fi(exp(1), true, 8)
b =
    2.7188
      s8,5
```

FullPrecision

Now set `ProductMode` and `SumMode` for `a` and `b` to `FullPrecision` and look at some results:

```
F.ProductMode = 'FullPrecision';
F.SumMode = 'FullPrecision';
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5
```

```
b
b =
    2.7188   %010.10111
      s8,5
a*b
ans =
    8.5811   %001000.1001010011
      s16,10
a+b
ans =
    5.8750   %0101.11100
      s9,5
```

In `FullPrecision` mode, the product word length grows to the sum of the word lengths of the operands. In this case, each operand has 8 bits, so the product word length is 16 bits. The product fraction length is the sum of the fraction lengths of the operands, in this case 5 + 5 = 10 bits.

The sum word length grows by one most significant bit to accommodate the possibility of a carry bit. The sum fraction length is aligned with the fraction lengths of the operands, and all fractional bits are kept for full precision. In this case, both operands have 5 fractional bits, so the sum has 5 fractional bits.

KeepLSB

Now set `ProductMode` and `SumMode` for `a` and `b` to `KeepLSB` and look at some results:

```
F.ProductMode = 'KeepLSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepLSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5
```

```
b
b =
    2.7188   %010.10111
      s8,5
a*b
ans =
    0.5811   %00.1001010011
      s12,10
a+b
ans =
    5.8750   %0000101.11100
      s12,5
```

In KeepLSB mode, you specify the word lengths and the least significant bits of results are automatically kept. This mode models the behavior of integer operations in the C language.

The product fraction length is the sum of the fraction lengths of the operands. In this case, each operand has 5 fractional bits, so the product fraction length is 10 bits. In this mode, all 10 fractional bits are kept. Overflow occurs because the full-precision result requires 6 integer bits, and only 2 integer bits remain in the product.

The sum fraction length is aligned with the fraction lengths of the operands, and in this model all least significant bits are kept. In this case, both operands had 5 fractional bits, so the sum has 5 fractional bits. The full-precision result requires 4 integer bits, and 7 integer bits remain in the sum, so no overflow occurs in the sum.

KeepMSB

Now set ProductMode and SumMode for a and b to KeepMSB and look at some results:

```
F.ProductMode = 'KeepMSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepMSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
```

```
a
a =
    3.1563   %011.00101
      s8,5
b
b =
    2.7188   %010.10111
      s8,5
a*b
ans =
    8.5781   %001000.100101
      s12,6
a+b
ans =
    5.8750   %0101.11100000
      s12,8
```

In `KeepMSB` mode, you specify the word lengths and the most significant bits of sum and product results are automatically kept. This mode models the behavior of many DSP devices where the product and sum are kept in double-wide registers, and the programmer chooses to transfer the most significant bits from the registers to memory after each operation.

The full-precision product requires 6 integer bits, and the fraction length of the product is adjusted to accommodate all 6 integer bits in this mode. No overflow occurs. However, the full-precision product requires 10 fractional bits, and only 6 are available. Therefore, precision is lost.

The full-precision sum requires 4 integer bits, and the fraction length of the sum is adjusted to accommodate all 4 integer bits in this mode. The full-precision sum requires only 5 fractional bits; in this case there are 8, so there is no loss of precision.

SpecifyPrecision

Now set `ProductMode` and `SumMode` for `a` and `b` to `SpecifyPrecision` and look at some results:

```
F.ProductMode = 'SpecifyPrecision';
F.ProductWordLength = 8;
```

```
F.ProductFractionLength = 7;
F.SumMode = 'SpecifyPrecision';
F.SumWordLength = 8;
F.SumFractionLength = 7;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563   %011.00101
      s8,5
b
b =
    2.7188   %010.10111
      s8,5
a*b
ans =
    0.5781   %0.1001010
      s8,7
a+b
ans =
   -0.1250   %1.1110000
      s8,7
```

In `SpecifyPrecision` mode, you must specify both word length and fraction length for sums and products. This example unwisely uses fractional formats for the products and sums, with 8–bit word lengths and 7–bit fraction lengths.

The full-precision product requires 6 integer bits, and the example specifies only 1, so the product overflows. The full-precision product requires 10 fractional bits, and the example only specifies 7, so there is precision loss in the product.

The full-precision sum requires 2 integer bits, and the example specifies only 1, so the sum overflows. The full-precision sum requires 5 fractional bits, and the example specifies 7, so there is no loss of precision in the sum.

fimath Object Functions

The following functions operate directly on `fimath` objects:

- `add`
- `disp`
- `fimath`
- `isequal`
- `isfimath`
- `mpy`
- `sub`

You can learn about the functions associated with `fimath` objects in the Function Reference in the Fixed-Point Toolbox online documentation.

Working with fipref Objects

Constructing fipref Objects

The `fipref` object defines the display and logging attributes for all `fi` objects. You can use the `fipref` constructor function to create a new object.

To get started, type

```
P = fipref
```

to create a default `fipref` object.

```
P =
          NumberDisplay: 'RealWorldValue'
     NumericTypeDisplay: 'full'
          FimathDisplay: 'full'
            LoggingMode: 'Off'
       DataTypeOverride: 'ForceOff'
```

The syntax

```
P = fipref(...'PropertyName','PropertyValue'...)
```

allows you to set properties for a `fipref` object at object creation with property name/property value pairs.

Your `fipref` settings persist throughout your MATLAB session. Use `reset(fipref)` to return to the default settings during your session. Use `savefipref` to save your display preferences for subsequent MATLAB sessions.

fipref Object Properties

The following properties of `fipref` objects are always writable:

- `FimathDisplay` — Display options for the `fimath` attributes of a `fi` object
- `DataTypeOverride` — Data type override options
- `LoggingMode` — Logging options for operations performed on `fi` objects
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object
- `NumberDisplay` — Display options for the value of a `fi` object

These properties are described in detail in Chapter 9, “Property Reference”. There are two ways to specify properties for `fipref` objects in the Fixed-Point Toolbox. Refer to the following sections:

- “Setting fipref Properties at Object Creation” on page 5-3
- “Using Direct Property Referencing with fipref” on page 5-3

Setting fipref Properties at Object Creation

You can set properties of `fipref` objects at the time of object creation by including properties after the arguments of the `fipref` constructor function. For example, to set `NumberDisplay` to `bin` and `NumericTypeDisplay` to `short`,

```
P = fipref('NumberDisplay', 'bin', 'NumericTypeDisplay', 'short')

P =
         NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'
```

Using Direct Property Referencing with fipref

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

For example, to get the NumberDisplay of P,

```
P.NumberDisplay

ans =

bin
```

To set the NumericTypeDisplay of P,

```
P.NumericTypeDisplay = 'full'

P =
         NumberDisplay: 'bin'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'
```

Using fipref Objects to Set Display Preferences

You use the `fipref` object to dictate three aspects of the display of `fi` objects: how the value of a `fi` object is displayed, how the `fimath` properties are displayed, and how the `numerictype` properties are displayed.

For example, the following shows the default `fipref` display for a `fi` object:

```
a = fi(pi)

a =

    3.1416


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Now, change the `fipref` display properties:

```
P = fipref;
P.NumberDisplay = 'bin';
P.NumericTypeDisplay = 'short';
P.FimathDisplay = 'none'

P =
         NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
         FimathDisplay: 'none'
           LoggingMode: 'Off'
```

```
      DataTypeOverride: 'ForceOff'
a

a =
0110010010001000
      s16,13
```

Using fipref Objects to Set Logging Preferences

When the `LoggingMode` property of the `fipref` object is set to `on`, overflows and underflows are logged as warnings. When `LoggingMode` is `on`, you can also have minimum and maximum values and the number of overflows, underflows, and quantization errors returned to you using functions. Refer to the following sections:

- "Logging Overflows and Underflows as Warnings" on page 5-7
- "Accessing Logged Information with Functions" on page 5-10

Logging Overflows and Underflows as Warnings

Overflows and underflows are logged as warnings for all assignment, plus, minus, and multiplication operations when the `fipref` `LoggingMode` property is set to `on`. For example, try the following:

1. Create a signed `fi` object that is a vector of values from 1 to 5, with 8-bit word length and 6-bit fraction length.

   ```
   a = fi(1:5,1,8,6);
   ```

2. Define the `fimath` object associated with `a`, and indicate that you will specify the sum and product word and fraction lengths.

   ```
   F = a.fimath;
   F.SumMode = 'SpecifyPrecision';
   F.ProductMode = 'SpecifyPrecision';
   a.fimath = F;
   ```

3. Define the `fipref` object and turn on overflow and underflow logging.

   ```
   P = fipref;
   P.LoggingMode = 'on';
   ```

4. Suppress the `numerictype` and `fimath` displays.

   ```
   P.NumericTypeDisplay = 'none';
   P.FimathDisplay = 'none';
   ```

5 Specify the sum and product word and fraction lengths.

```
a.SumWordLength = 16;
a.SumFractionLength = 15;
a.ProductWordLength = 16;
a.ProductFractionLength = 15;
```

6 Warnings are displayed for overflows and underflows in assignment operations. For example, try:

```
a(1) = pi
Warning: 1 overflow occurred in the fi assignment operation.

a =

    1.9844    1.9844    1.9844    1.9844    1.9844
a(1) = double(eps(a))/10
Warning: 1 underflow occurred in the fi assignment operation.

a =

         0    1.9844    1.9844    1.9844    1.9844
```

7 Warnings are displayed for overflows and underflows in addition and subtraction operations. For example, try:

```
a+a
Warning: 12 overflows occurred in the fi + operation.

ans =

         0    1.0000    1.0000    1.0000    1.0000
a-a
Warning: 8 overflows occurred in the fi - operation.

ans =

     0     0     0     0     0
```

8 Warnings are displayed for overflows and underflows in multiplication operations. For example, try:

```
a.*a
Warning: 4 product overflows occurred in the fi .* operation.

ans =

         0    1.0000    1.0000    1.0000    1.0000

a*a'
Warning: 4 product overflows occurred in the fi * operation.
Warning: 3 sum overflows occurred in the fi * operation.

ans =

    1.0000
```

The final example above is a complex multiplication that requires both multiplication and addition operations. The warnings inform you of overflows and underflows in both.

Because overflows and underflows are logged as warnings, you can use the `dbstop` MATLAB function with the syntax

```
dbstop if warning
```

to find the exact lines in an M-file that are causing overflows or underflows.

Use

```
dbstop if warning fi:underflow
```

to stop only on lines that cause an underflow. Use

```
dbstop if warning fi:overflow
```

to stop only on lines that cause an overflow.

Accessing Logged Information with Functions

When the `fipref` `LoggingMode` property is set to `on`, you can use the following functions to return logged information about assignment and creation operations to the MATLAB command line:

- `maxlog` — Returns the maximum real-world value
- `minlog` — Returns the minimum value
- `noverflows` — Returns the number of overflows
- `nunderflows` — Returns the number of underflows

`LoggingMode` must be set to `on` before you perform any operation in order to log information about it. To clear the log, use the function `resetlog`.

For example, consider the following. First turn logging on, then perform operations, and then finally get information about the operations:

```
fipref('LoggingMode','on');
x = fi([-1.5 eps 0.5], true, 16, 15);
x(1) = 3.0;
maxlog(x)

ans =

     3

minlog(x)

ans =

   -1.5000

noverflows(x)

ans =

            2

nunderflows(x)
```

```
ans =

            1
```

Next, reset the log and request the same information again. Note that the functions return empty [], because logging has been reset since the operations were run:

```
resetlog(x)
maxlog(x)

ans =

     []

minlog(x)

ans =

     []

noverflows(x)

ans =

     []

nunderflows(x)

ans =

     []
```

Using fipref Objects to Set Data Type Override Preferences

Use the `fipref` `DataTypeOverride` property to override `fi` objects with singles, doubles, or scaled doubles. Data type override only occurs when the `fi` constructor function is called. Objects that are created while data type override is on have the overridden data type. They maintain that data type when data type override is later turned off. To obtain an object with a data type that is not the override data type, you must create an object when data type override is off:

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

p =

         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'TrueDoubles'

a = fi(pi)

a =

    3.1416

          DataTypeMode: double

p = fipref('DataTypeOverride', 'ForceOff')

p =

         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'

a
```

```
a =

    3.1416

          DataTypeMode: double

b = fi(pi)

b =

    3.1416

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

Tip To reset the `fipref` object to its default values use `reset(fipref)` or `reset(p)`, where `p` is a `fipref` object. This is useful to ensure that data type override and logging are off.

Using Data Type Override to Help Set Fixed-Point Scaling

Choosing the scaling for the fixed-point variables in your algorithms can be difficult. In the Fixed-Point Toolbox, you can use a combination of data type override and min/max logging to help you discover the numerical ranges that your fixed-point data types need to cover. These ranges dictate the appropriate scalings for your fixed-point data types. In general, the procedure is

1. Implement your algorithm using fixed-point `fi` objects, using initial "best guesses" for word lengths and scalings.

2. Set the `fipref` `DataTypeOverride` property to `ScaledDoubles`, `TrueSingles`, or `TrueDoubles`.

3. Set the `fipref` `LoggingMode` property to `on`.

4 Use the `maxlog` and `minlog` functions to log the maximum and minimum values achieved by the variables in your algorithm in floating-point mode.

5 Set the `fipref` `DataTypeOverride` property to `ForceOff`.

6 Use the information obtained in step 4 to set the fixed-point scaling for each variable in your algorithm such that the full numerical range of each variable is representable by its data type and scaling.

A detailed example of this process is shown in the Fixed-Point Toolbox "Fixed-Point Data Type Override, Min/Max Logging, and Scaling" demo.

fipref Object Functions

The following functions operate directly on `fipref` objects:

- `disp`
- `fipref`
- `reset`
- `savefipref`

You can learn about the functions associated with `fipref` objects in the Function Reference.

Working with numerictype Objects

Constructing numerictype Objects

`numerictype` objects define the data type and scaling attributes of `fi` objects. You can create `numerictype` objects in the Fixed-Point Toolbox in one of two ways:

- You can use the `numerictype` constructor function to create a new object.
- You can use the `numerictype` constructor function to copy an existing `numerictype` object.

To get started, type

```
T = numerictype
```

to create a default `numerictype` object.

```
T =


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

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`, signedness `s`, and 16-bit word length.
- `T = numerictype(s,w)` creates a `numerictype` object with `Fixed-point: unspecified scaling`, signedness `s`, and word length `w`.
- `T = numerictype(s,w,f)` creates a `numerictype` object with `Fixed-point:  binary point scaling`, signedness `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`, signedness `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`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.
- `T = numerictype(property1,value1, ...)` allows you to set properties for a `numerictype` object using property name/property value pairs.
- `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`.

Examples of Constructing numerictype Objects

For example, the following 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
                Signed: true
            WordLength: 32
        FractionLength: 30
```

If you omit the argument `f`, scaling is unspecified.

```
T = numerictype(1, 32)

T =


          DataTypeMode: Fixed-point: unspecified scaling
                Signed: true
            WordLength: 32
```

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
                Signed: true
            WordLength: 16
```

Constructing a numerictype Object with Property Name/Property Value Pairs

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', 'WordLength', 32, 'Slope', ...
2^-2, 'Bias', 4)

T =


          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 32
                 Slope: 0.25
                  Bias: 4
```

Copying a numerictype Object

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

```
T = numerictype;
U = T;
isequal(T,U)

ans =

     1
```

numerictype Object Properties

All the properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object are not writable once the `fi` object has been created:

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

These properties are described in detail in Chapter 9, “Property Reference”. There are two ways to specify properties for `numerictype` objects in the Fixed-Point Toolbox. Refer to the following sections:

- “Setting numerictype Properties at Object Creation” on page 6-6
- “Using Direct Property Referencing with numerictype Objects” on page 6-7
- “Setting numerictype Properties in the Model Explorer” on page 6-8

Setting numerictype Properties at Object Creation

You can set properties of `numerictype` objects at the time of object creation by including properties after the arguments of the `numerictype` constructor function.

For example, to set the word length to 32 bits and the fraction length to 30 bits,

```
T = numerictype('WordLength', 32, 'FractionLength', 30)

T =


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 30
```

Using Direct Property Referencing with numerictype Objects

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

For example, to get the word length of `T`,

```
T.WordLength

ans =

32
```

To set the fraction length of `T`,

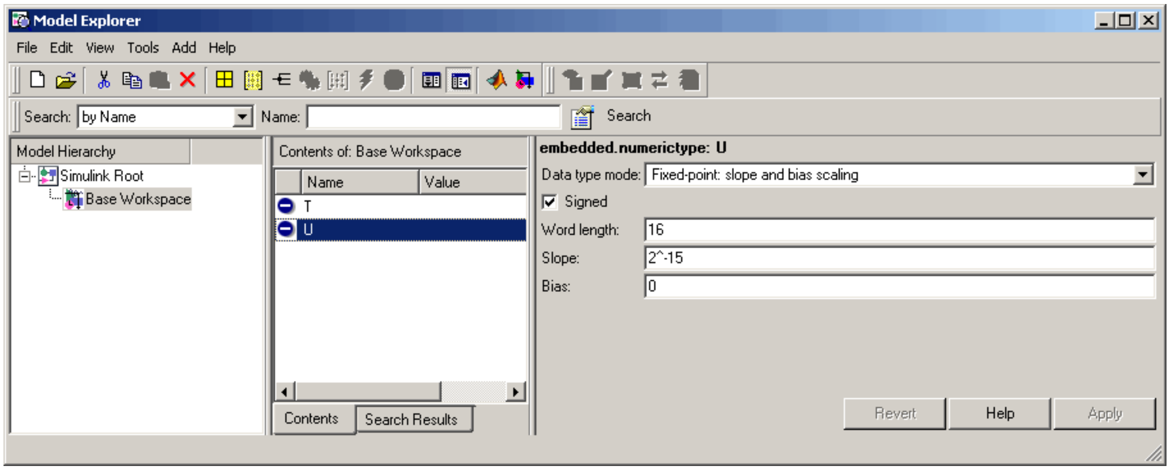
```
T.FractionLength = 31

T =


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 31
```

Setting numerictype Properties in the Model Explorer

You can view and change the properties for any `numerictype` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View** > **Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The figure below shows the Model Explorer when you define the following `numerictype` objects in the MATLAB workspace:

```
T = numerictype

T =


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

U = numerictype('DataTypeMode', 'Fixed-point: slope and bias')

U =


          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 16
                 Slope: 2^-15
                  Bias: 0
```



embedded.numerictype: U

Data type mode: Fixed-point: slope and bias scaling

☑ Signed

Word length: 16

Slope: 2^-15

Bias: 0

Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a `numerictype` object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

The numerictype Structure

The `numerictype` object contains all the data type and scaling attributes of a `fi` object. The object acts the same way as any MATLAB structure, except that it only lets you set valid values for defined fields. The following table shows the possible settings of each field of the structure that are valid for `fi` objects.

DataTypeMode	DataType	Scaling	Signed	WordLength	FractionLength	Slope	Bias
Fully specified fixed-point data types							
`Fixed-point: binary point scaling`	`Fixed`	`BinaryPoint`	`1/0` `true/false`	positive integer from 1 to 65,536	positive or negative integer	`1`	`0`
`Fixed-point: slope and bias scaling`	`Fixed`	`SlopeBias`	`1/0` `true/false`	positive integer from 1 to 65,536	`N/A`	any floating-point number	any floating-point number
Partially specified fixed-point data type							
`Fixed-point: unspecified scaling`	`Fixed`	`Unspecified`	`1/0` `true/false`	positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`
Fully specified scaled double data types							
`Scaled double: binary point scaling`	`ScaledDouble`	`BinaryPoint`	`1/0` `true/false`	positive integer from 1 to 65,536	positive or negative integer	`1`	`0`

DataTypeMode	DataType	Scaling	Signed	WordLength	FractionLength	Slope	Bias
Scaled double: slope and bias scaling	ScaledDouble	SlopeBias	1/0 true/false	positive integer from 1 to 65,536	N/A	any floating-point number	any floating-point number
Partially specified scaled double data type							
Scaled double: unspecified scaling	ScaledDouble	Unspecified	1/0 true/false	positive integer from 1 to 65,536	N/A	N/A	N/A
Built-in data types							
double	double	N/A	1 true	64	0	1	0
single	single	N/A	1 true	32	0	1	0
boolean	boolean	N/A	0 false	1	0	1	0

You cannot change the `numerictype` properties of a `fi` object after `fi` object creation.

Properties That Affect the Slope

The **Slope** field of the `numerictype` structure is related to the `SlopeAdjustmentFactor` and `FixedExponent` properties by

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

The `FixedExponent` and `FractionLength` properties are related by

$$fixed\ exponent = -fraction\ length$$

If you set the `SlopeAdjustmentFactor`, `FixedExponent`, or `FractionLength` property, the **Slope** field is modified.

Stored Integer Value and Real World Value

The `numerictype` `StoredIntegerValue` and `RealWorldValue` properties are related according to

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

which is equivalent to

$$\begin{aligned}&\textit{real-world value} = \textit{stored integer value}\\&\times (\textit{slope adjustment factor} \times 2^{\textit{fixed exponent}}) + \textit{bias}\end{aligned}$$

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

Using numerictype Objects to Share Data Type and Scaling Settings

You can use a `numerictype` object to define common data type and scaling rules that you would like to use for many `fi` objects. You can then create multiple `fi` objects, using the same `numerictype` object for each. The following example shows the creation of a `numerictype` object, which is then used to create two `fi` objects with the same `numerictype` attributes:

```
format long g
T = numerictype('WordLength',32,'FractionLength',28)

T =


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 28

a = fi(pi,T)

a =

           3.1415926553309


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 28

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

```
b = fi(pi/2, T)

b =

           1.5707963258028


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 28

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

The following example shows the creation of a `numerictype` object with [Slope Bias] scaling, which is then used to create two `fi` objects with the same `numerictype` attributes:

```
T = numerictype('scaling','slopebias','slope', 2^2, 'bias', 0)

T =


          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0
```

```
c = fi(pi, T)

c =

     4

          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

d = fi(pi/2, T)

d =

     0

          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

numerictype Object Functions

The following functions operate directly on `numerictype` objects:

- `divide`
- `isequal`
- `isnumerictype`

You can learn about the functions associated with `numerictype` objects in the Function Reference.

Working with quantizer Objects

Constructing quantizer Objects

You can use `quantizer` objects to quantize data sets. You can create `quantizer` objects in the Fixed-Point Toolbox in one of two ways:

- You can use the `quantizer` constructor function to create a new object.
- You can use the `quantizer` constructor function to copy a `quantizer` object.

To create a `quantizer` object with default properties, type

```
q = quantizer

q =

        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]

             Max = reset
             Min = reset
      NOverflows = 0
     NUnderflows = 0
     NOperations = 0
```

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

```
q = quantizer;
r = q;
isequal(q,r)

ans =

     1
```

A listing of all the properties of the `quantizer` object `q` you just created is displayed along with the associated property values. All property values are set to defaults when you construct a `quantizer` object this way. See “quantizer Object Properties” on page 7-3 for more details.

quantizer Object Properties

The following properties of `quantizer` objects are always writable:

- `DataMode` — Type of arithmetic used in quantization
- `Format` — Data format of a `quantizer` object
- `OverflowMode` — Overflow-handling mode
- `RoundMode` — Rounding mode

See Chapter 9, “Property Reference” for more details about these properties, including their possible values.

For example, to create a fixed-point `quantizer` object with

- The `Format` property value set to `[16,14]`
- The `OverflowMode` property value set to `'saturate'`
- The `RoundMode` property value set to `'ceil'`

type

```
q =
quantizer('datamode','fixed','format',[16,14],'overflowmode',...
    'saturate','roundmode','ceil')
```

You do not have to include `quantizer` object property names when you set `quantizer` object property values.

For example, you can create `quantizer` object `q` from the previous example by typing

```
q = quantizer('fixed',[16,14],'saturate','ceil')
```

Note You do not have to include default property values when you construct a `quantizer` object. In this example, you could leave out `'fixed'` and `'saturate'`.

Quantizing Data with quantizer Objects

You construct a `quantizer` object to specify the quantization parameters to use when you quantize data sets. You can use the `quantize` function to quantize data according to a `quantizer` object's specifications.

Once you quantize data with a `quantizer` object, its state values might change.

The following example shows

- How you use `quantize` to quantize data
- How quantization affects `quantizer` object states
- How you reset `quantizer` object states to their default values using `reset`

1 Construct an example data set and a `quantizer` object.

```
randn('state',0);
x = randn(100,4);
q = quantizer([16,14]);
```

2 Retrieve the values of the `max` and `noverflows` states.

```
q.max

ans =
reset

q.noverflows

ans =
     0
```

3 Quantize the data set according to the `quantizer` object's specifications.

```
y = quantize(q,x);
```

4 Check the values of `max` and `noverflows`.

```
q.max

ans =
2.3726

q.noverflows

ans =
     15
```

5 Reset the `quantizer` states and check them.

```
reset(q)
q.max

ans =
reset

q.noverflows

ans =
     0
```

Transformations for Quantized Data

You can convert data values from numeric to hexadecimal or binary according to a `quantizer` object's specifications.

Use

- `num2bin` to convert data to binary
- `num2hex` to convert data to hexadecimal
- `hex2num` to convert hexadecimal data to numeric
- `bin2num` to convert binary data to numeric

For example,

```
q = quantizer([3 2]);
      x = [0.75   -0.25
           0.50   -0.50
           0.25   -0.75
           0      -1   ];
      b = num2bin(q,x)

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

produces all two's complement fractional representations of 3-bit fixed-point numbers.

quantizer Object Functions

The functions in the table below operate directly on quantizer objects.

bin2num	copyobj	denormalmax	denormalmin	disp
eps	exponentbias	exponentlength	exponentmax	exponentmin
fractionlength	get	hex2num	isequal	length
max	min	noperations	noverflows	num2bin
num2hex	num2int	nunderflows	quantize	quantizer
randquant	range	realmax	realmin	reset
round	set	tostring	wordlength	

You can learn about the functions associated with quantizer objects in the Function Reference.

Interoperability with Other Products

Using fi Objects with Simulink

This section describes the ways you can use Fixed-Point Toolbox `fi` objects with Simulink models.

Reading Fixed-Point Data from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model via the From Workspace block. To do so, the data must be in structure format with a `fi` object in the `values` field. In array format, the From Workspace block only accepts real, double-precision data.

To read in `fi` data, the **Interpolate data** parameter of the From Workspace block must not be selected, and the **Form output after final data value by** parameter must be set to anything other than `Extrapolation`.

Writing Fixed-Point Data to the Workspace

You can write fixed-point output from a model to the MATLAB workspace via the To Workspace block in either array or structure format. Fixed-point data written by a To Workspace block to the workspace in structure format can be read back into a Simulink model in structure format by a From Workspace block.

Note To write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

For example, you can use the following code to create a structure in the MATLAB workspace with a `fi` object in the `values` field. You can then use the From Workspace block to bring the data into a Simulink model.

```
a = fi([sin(0:10)' sin(10:-1:0)'])

a =

         0   -0.5440
    0.8415    0.4121
    0.9093    0.9893
    0.1411    0.6570
   -0.7568   -0.2794
   -0.9589   -0.9589
   -0.2794   -0.7568
    0.6570    0.1411
    0.9893    0.9093
    0.4121    0.8415
   -0.5440         0


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

                RoundMode: nearest
             OverflowMode: saturate
              ProductMode: FullPrecision
     MaxProductWordLength: 128
                  SumMode: FullPrecision
         MaxSumWordLength: 128
            CastBeforeSum: true

s.signals.values = a

s =

    signals: [1x1 struct]
```

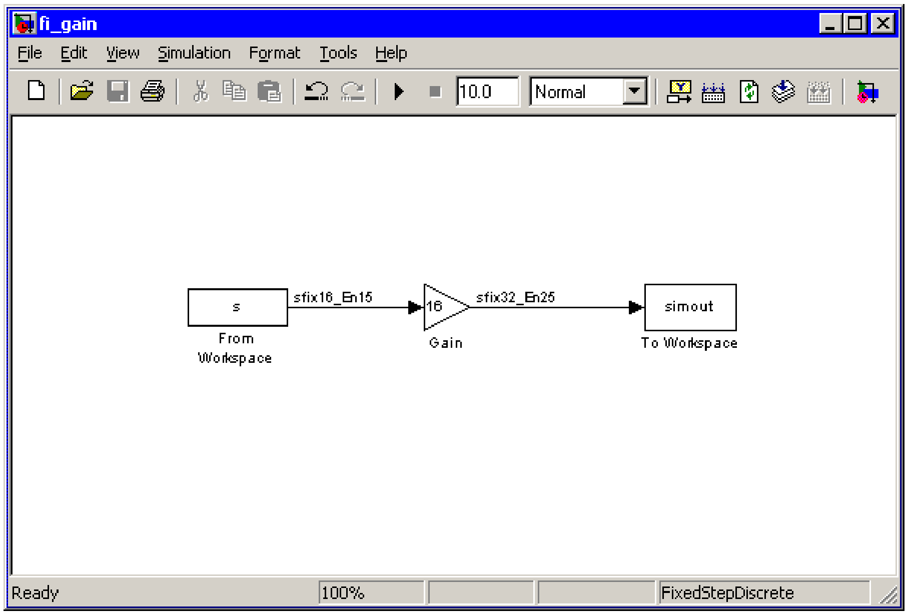
```
s.signals.dimensions = 2

s =

    signals: [1x1 struct]

s.time = [0:10]'

s =

    signals: [1x1 struct]
       time: [11x1 double]
```

The From Workspace block in the following model has the `fi` structure `s` in the **Data** parameter.

Remember, to write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

In the model, the following parameters in the **Solver** pane of the **Configuration Parameters** dialog have the indicated settings:

- **Start time** — `0.0`
- **Stop time** — `10.0`
- **Type** — `Fixed-step`
- **Solver** — `discrete (no continuous states)`
- **Fixed step size (fundamental sample time)** — `1.0`



The To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` structure.

```
simout.signals.values

ans =

          0   -8.7041
    13.4634    6.5938
    14.5488   15.8296
     2.2578   10.5117
   -12.1089   -4.4707
   -15.3428  -15.3428
    -4.4707  -12.1089
    10.5117    2.2578
    15.8296   14.5488
     6.5938   13.4634
    -8.7041         0
```

```
      DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 32
        FractionLength: 25

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Setting the Value and Data Type of Block Parameters

You can use expressions from the Fixed-Point Toolbox to specify the value and data type of block parameters in Simulink. Refer to “Block Support for Data and Numeric Signal Types” in the Using Simulink documentation for more information.

Logging Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as `fi` objects. To enable signal logging for a signal, select the **Log signal data** option in the signal’s **Signal Properties** dialog box. For more information, refer to “Logging Signals” in the Using Simulink documentation.

When you log signals from a referenced model or Stateflow® chart in your model, the word lengths of `fi` objects may be larger than you expect. The word lengths of fixed-point signals in referenced models and Stateflow charts are logged as the next largest data storage container size.

Accessing Fixed-Point Block Data During Simulation

Simulink provides an application program interface (API) that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to develop MATLAB programs capable of accessing block data while a simulation is running or to access the data from the MATLAB command line. Fixed-point signal information is returned to you via this API as `fi` objects. For more information on the API, refer to "Accessing Block Data During Simulation" in the Using Simulink documentation.

Using fi Objects with Signal Processing Blockset

Fixed-Point Toolbox `fi` objects can be used to pass fixed-point data between the MATLAB workspace and models using Signal Processing Blockset blocks.

Reading Fixed-Point Signals from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model using the Signal From Workspace and Triggered Signal From Workspace blocks from the Signal Processing Blockset. Enter the name of the defined `fi` variable in the **Signal** parameter of the Signal From Workspace or Triggered Signal From Workspace block.

Writing Fixed-Point Signals to the Workspace

Fixed-point output from a model can be written to the MATLAB workspace via the Signal To Workspace or Triggered To Workspace block from the Signal Processing Blockset. The fixed-point data is always written as a 2-D or 3-D array.

Note To write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace or Triggered To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

For example, you can use the following code to create a fi object in the MATLAB workspace. You can then use the Signal From Workspace block to bring the data into a Simulink model.

```
a = fi([sin(0:10)' sin(10:-1:0)'])

a =

         0   -0.5440
    0.8415    0.4121
    0.9093    0.9893
    0.1411    0.6570
   -0.7568   -0.2794
   -0.9589   -0.9589
   -0.2794   -0.7568
    0.6570    0.1411
    0.9893    0.9093
    0.4121    0.8415
   -0.5440         0


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

                RoundMode: nearest
             OverflowMode: saturate
              ProductMode: FullPrecision
     MaxProductWordLength: 128
                  SumMode: FullPrecision
         MaxSumWordLength: 128
            CastBeforeSum: true
```

The Signal From Workspace block in the following model has these settings:

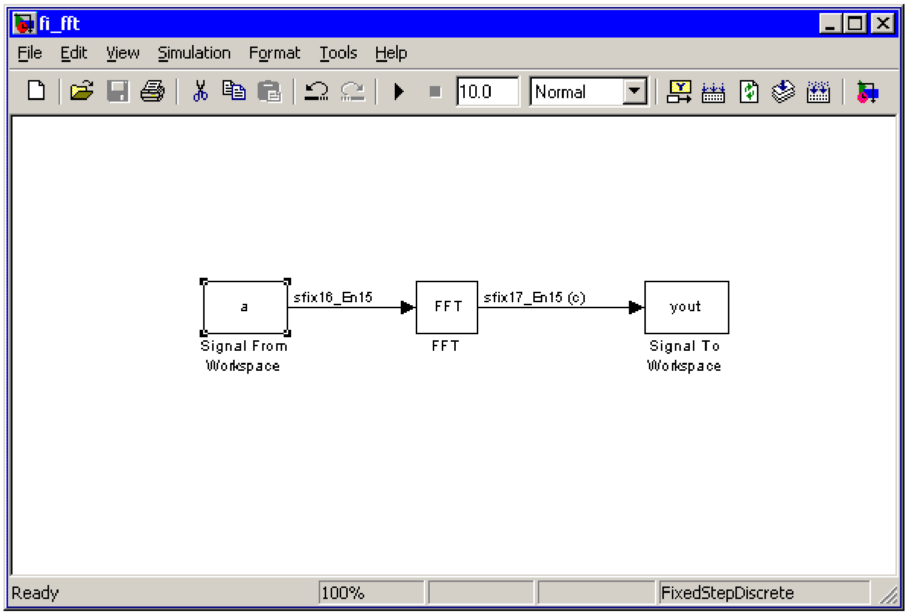
- **Signal** — a
- **Sample time** — 1
- **Samples per frame** — 2

- **Form output after final data value by** — `Setting to zero`

The following parameters in the **Solver** pane of the **Configuration Parameters** dialog have these settings:

- **Start time** — `0.0`
- **Stop time** — `10.0`
- **Type** — `Fixed-step`
- **Solver** — `discrete (no continuous states)`
- **Fixed step size (fundamental sample time)** — `1.0`

Remember, to write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.



The Signal To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` object.

```
yout =


(:,:,1) =

    0.8415   -0.1319
   -0.8415   -0.9561


(:,:,2) =

    1.0504    1.6463
    0.7682    0.3324


(:,:,3) =

   -1.7157   -1.2383
    0.2021    0.6795


(:,:,4) =

    0.3776   -0.6157
   -0.9364   -0.8979


(:,:,5) =

    1.4015    1.7508
    0.5772    0.0678


(:,:,6) =

   -0.5440         0
   -0.5440         0
```

```
              DataTypeMode: Fixed-point: binary point scaling
                    Signed: true
                WordLength: 17
            FractionLength: 15

                 RoundMode: nearest
              OverflowMode: saturate
               ProductMode: FullPrecision
      MaxProductWordLength: 128
                   SumMode: FullPrecision
          MaxSumWordLength: 128
             CastBeforeSum: true
```

Using the Fixed-Point Toolbox with Embedded MATLAB

The Embedded MATLAB Function block lets you compose a MATLAB language function in a Simulink model that generates embeddable code. When you simulate the model or generate code for a target environment, a function in an Embedded MATLAB Function block generates efficient C code. This code meets the strict memory and data type requirements of embedded target environments. In this way, Embedded MATLAB Function blocks bring the power of MATLAB for the embedded environment into Simulink.

For more information on using Embedded MATLAB, refer to the following sections in the Simulink documentation:

- Embedded MATLAB Function block reference page
- “Using the Embedded MATLAB Function Block”
- “Embedded MATLAB Function Block Reference”

Supported Functions and Limitations of Fixed-Point Embedded MATLAB

You can use a significant subset of Fixed-Point Toolbox functions with Embedded MATLAB. The Fixed-Point Toolbox functions supported for use with Embedded MATLAB are listed in the table below. The following general limitations always apply to the use of the Fixed-Point Toolbox with Embedded MATLAB:

- Dot notation is not supported.
- Word lengths larger than 32 bits are not supported.
- It is illegal to change the `fimath` or `numerictype` of a given variable once it has been created.
- The `double`, `single`, `boolean`, and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- `convergent` rounding is not supported.
- The `false` value of the `CastBeforeSum` property is not supported.
- The `numel` function works the same as MATLAB `numel` for `fi` objects in Embedded MATLAB, rather than returning `1` as in the Fixed-Point Toolbox.

To learn about the general limitations on the use of Embedded MATLAB that also apply to use with the Fixed-Point Toolbox, refer to "Unsupported MATLAB Features and Limitations" in the Simulink documentation.

Note To simulate models using fixed-point data types in Simulink, you must have a Simulink Fixed Point license.

Fixed-Point Toolbox Functions Supported for Use with Embedded MATLAB

Function	Remarks/Limitations
`abs`	—
`all`	—
`any`	—
`bitand`	—
`bitcmp`	—
`bitget`	—
`bitor`	—
`bitset`	—
`bitshift`	—
`bitxor`	—
`complex`	—
`conj`	—
`ctranspose`	—
`disp`	—
`divide`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object • Complex and imaginary divisors are not supported
`double`	—
`end`	—

Fixed-Point Toolbox Functions Supported for Use with Embedded MATLAB (Continued)

Function	Remarks/Limitations
`eps`	—
`eq`	• Not supported for fixed-point signals with different biases
`fi`	• Use to create a fixed-point constant or variable in Embedded MATLAB • The syntax `fi('PropertyName',PropertyValue...)` is not supported. To use property name/property value pairs, you must first specify the value `v` of the `fi` object as in `fi(v,'PropertyName',PropertyValue...)` • Works for constant input values only; that is, the value of the input must be known at compile time • `numerictype` object information must be available for nonfixed-point Simulink inputs
`fimath`	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned the `fimath` object defined in the Embedded MATLAB Function dialog in the Model Explorer • Use to create `fimath` objects in Embedded MATLAB code
`ge`	• Not supported for fixed-point signals with different biases
`gt`	• Not supported for fixed-point signals with different biases
`horzcat`	—
`imag`	—
`int8`, `int16`, `int32`	—
`iscolumn`	—
`isempty`	—
`isfi`	—
`isfimath`	—
`isfinite`	—
`isinf`	—

Fixed-Point Toolbox Functions Supported for Use with Embedded MATLAB (Continued)

Function	Remarks/Limitations
`isnan`	—
`isnumeric`	—
`isnumerictype`	—
`isreal`	—
`isrow`	—
`isscalar`	—
`issigned`	—
`isvector`	—
`le`	• Not supported for fixed-point signals with different biases
`length`	—
`logical`	—
`lowerbound`	—
`lsb`	—
`lt`	• Not supported for fixed-point signals with different biases
`max`	• Supported for 1-D and 2-D arrays only
`min`	• Supported for 1-D and 2-D arrays only
`minus`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object
`mtimes`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object
`ndims`	—
`ne`	• Not supported for fixed-point signals with different biases
`numberofelements`	• `numberofelements` and `numel` both work the same as MATLAB `numel` for `fi` objects in Embedded MATLAB

Fixed-Point Toolbox Functions Supported for Use with Embedded MATLAB (Continued)

Function	Remarks/Limitations
`numerictype`	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned a `numerictype` object that is populated with the signal's data type and scaling information • Returns the data type when the input is a nonfixed-point signal • Use to create `numerictype` objects in Embedded MATLAB code
`plus`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object
`pow2`	• For the syntax `pow2(a, K)`, `K` must be a constant; that is, its value must be known at compile time so that it can be cast to a `fi` object
`range`	—
`real`	—
`realmax`	—
`realmin`	—
`repmat`	—
`rescale`	—
`reshape`	• Supported for 1-D and 2-D arrays only
`sign`	—
`single`	—
`size`	—
`subsasgn`	—
`subsref`	—
`sum`	• Supported for 1-D and 2-D arrays only
`times`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object
`transpose`	—
`uint8`, `uint16`, `uint32`	—
`uminus`	—

Fixed-Point Toolbox Functions Supported for Use with Embedded MATLAB (Continued)

Function	Remarks/Limitations
`uplus`	—
`upperbound`	—
`vertcat`	—

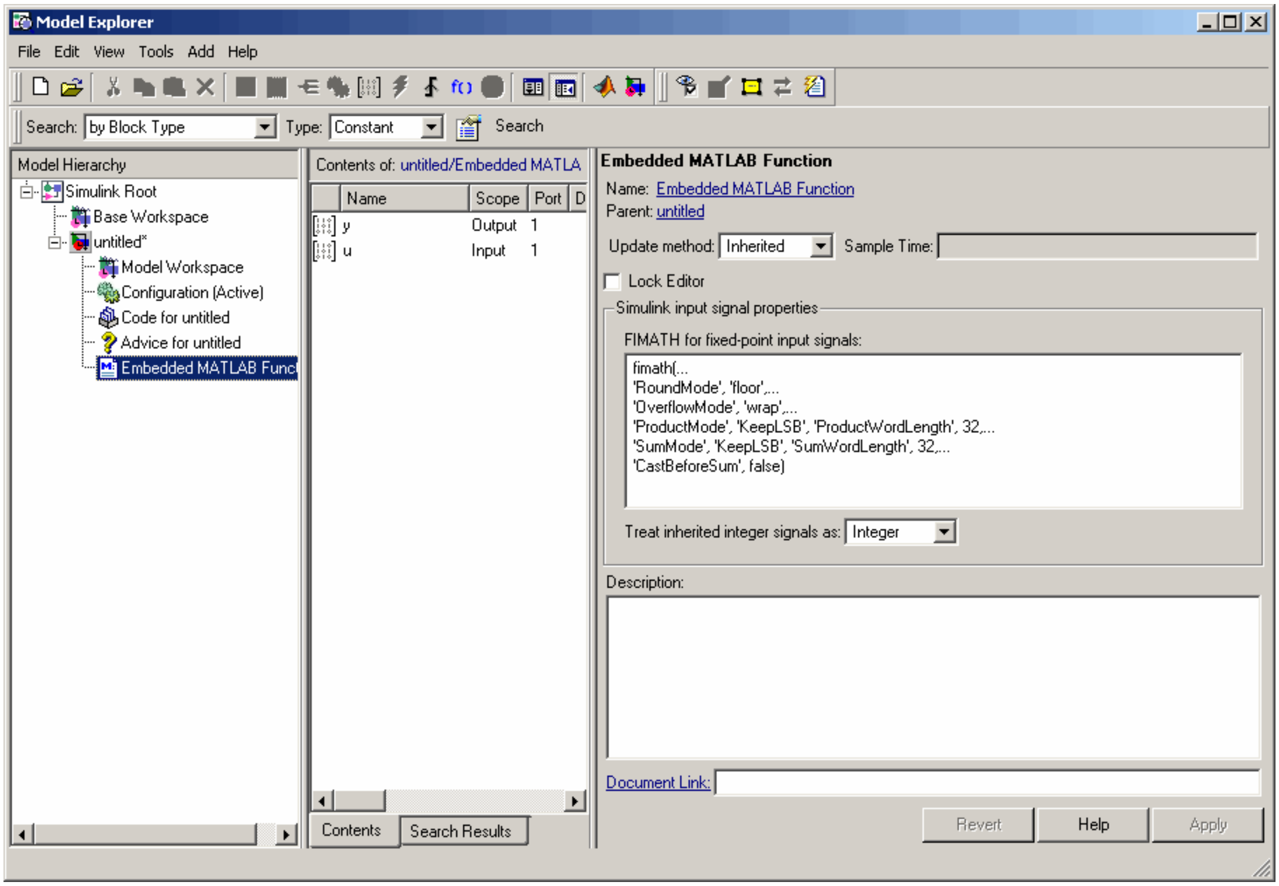
Using the Model Explorer with Fixed-Point Embedded MATLAB

You can specify parameters for an Embedded MATLAB Function block in a fixed-point model using the Model Explorer. Try the following:

1. Type `emlnew` at the MATLAB command line to open a new Simulink model populated with an Embedded MATLAB Function block.

2. Open the Model Explorer by selecting **View** > **Model Explorer** from your model.

3. Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer and select the **Embedded MATLAB Function** node. The Model Explorer now appears as follows:



Name	Scope	Port
y	Output	1
u	Input	1

Name: Embedded MATLAB Function
Parent: untitled
Update method: Inherited
Sample Time:
Lock Editor
Simulink input signal properties
FIMATH for fixed-point input signals:

```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'KeepLSB', 'ProductWordLength', 32,...
'SumMode', 'KeepLSB', 'SumWordLength', 32,...
'CastBeforeSum', false)
```

Treat inherited integer signals as: Integer
Description:

The parameters in the **Simulink input signal properties** group box in the **Dialog** pane apply to Embedded MATLAB Function blocks in models that use fixed-point data types.

FIMATH for fixed-point input signals

Define the `fimath` object to be associated with Simulink fixed-point or integer signals entering the Embedded MATLAB Function block as inputs. You can do this in either of two ways:

- Fully define the `fimath` object in the parameter value box using Fixed-Point Toolbox MATLAB code.

- Enter a variable name of a `fimath` object that is defined in the MATLAB or model workspace.

The default `fimath` object entered for this parameter emulates C-style math.

Treat inherited integer signals as
Choose whether to treat inherited integer signals as integers or fixed-point data.

- If you select `Integer`, Simulink integer inputs to the Embedded MATLAB Function block will be treated as MATLAB integers.
- If you select `Fixed-point`, Simulink integer inputs to the Embedded MATLAB Function block will be treated as Fixed-Point Toolbox `fi` objects.

Sharing Fixed-Point Embedded MATLAB Models

Sometimes you might need to share a fixed-point model using the Embedded MATLAB Function block with a coworker. When you do, make sure to move any variables you define in the MATLAB workspace, including `fimath` objects, to the model workspace. For example, try the following:

1. Type `emlnew` at the MATLAB command line to open a new Simulink model populated with an Embedded MATLAB Function block.
2. Define a `fimath` object in the MATLAB workspace that you want to use for any Simulink fixed-point signal entering the Embedded MATLAB Function block as an input:

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32)

F =


            RoundMode: floor
         OverflowMode: wrap
          ProductMode: KeepLSB
    ProductWordLength: 32
```

```
       SumMode: KeepLSB
 SumWordLength: 32
 CastBeforeSum: true
```

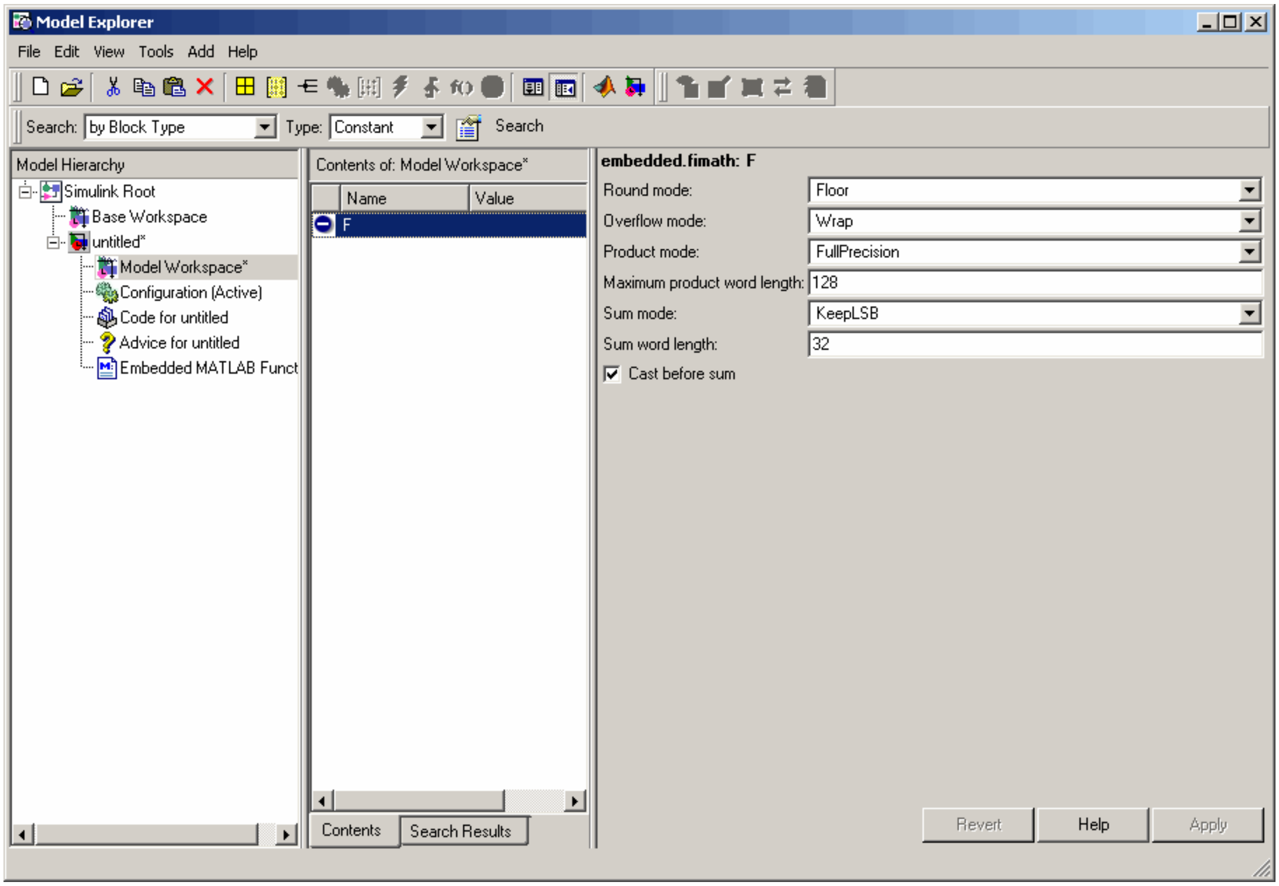
3 Open the Model Explorer by selecting **View** > **Model Explorer** from your model.

4 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer and select the **Embedded MATLAB Function** node.

5 Enter the variable `F` into the **FIMATH for fixed-point input signals** parameter on the **Dialog** pane and click **Apply**. You have now defined the `fimath` object for any Simulink fixed-point signal entering the Embedded MATLAB Function as an input.

6 Select the **Base Workspace** node in the **Model Hierarchy** pane. You can see the variable `F` that you have defined in the MATLAB workspace listed in the **Contents** pane. If you were to send this model to a coworker, they would have to define that same variable in their MATLAB workspace to get the same results as you with this model.

7 Cut the variable `F` from the base workspace and paste it into the model workspace listed under the node for your model, in this case **untitled***. The Model Explorer now looks like this:



embedded.fimath: F

Round mode:	Floor
Overflow mode:	Wrap
Product mode:	FullPrecision
Maximum product word length:	128
Sum mode:	KeepLSB
Sum word length:	32

[✓] Cast before sum

You can now e-mail your model to a coworker, and because the variables needed to run the model are included in the workspace of the model itself, your coworker can run the model and get the correct results without performing any extra steps.

Example: Implementing a Fixed-Point Direct Form FIR Using Embedded MATLAB

This example leads you through creating a fixed-point, low-pass, direct form FIR filter in Simulink using the Fixed-Point Toolbox and Embedded MATLAB in the following sections:

- “I. Program the Embedded MATLAB Block” on page 8-23
- “II. Prepare the Inputs” on page 8-24
- “III. Create the Model” on page 8-25
- “IV. Define the Input fimath Using the Model Explorer” on page 8-28
- “V. Run the Simulation” on page 8-30

I. Program the Embedded MATLAB Block

1 Place an Embedded MATLAB Function block in a new model. The block is located in the Simulink User-Defined Functions library.

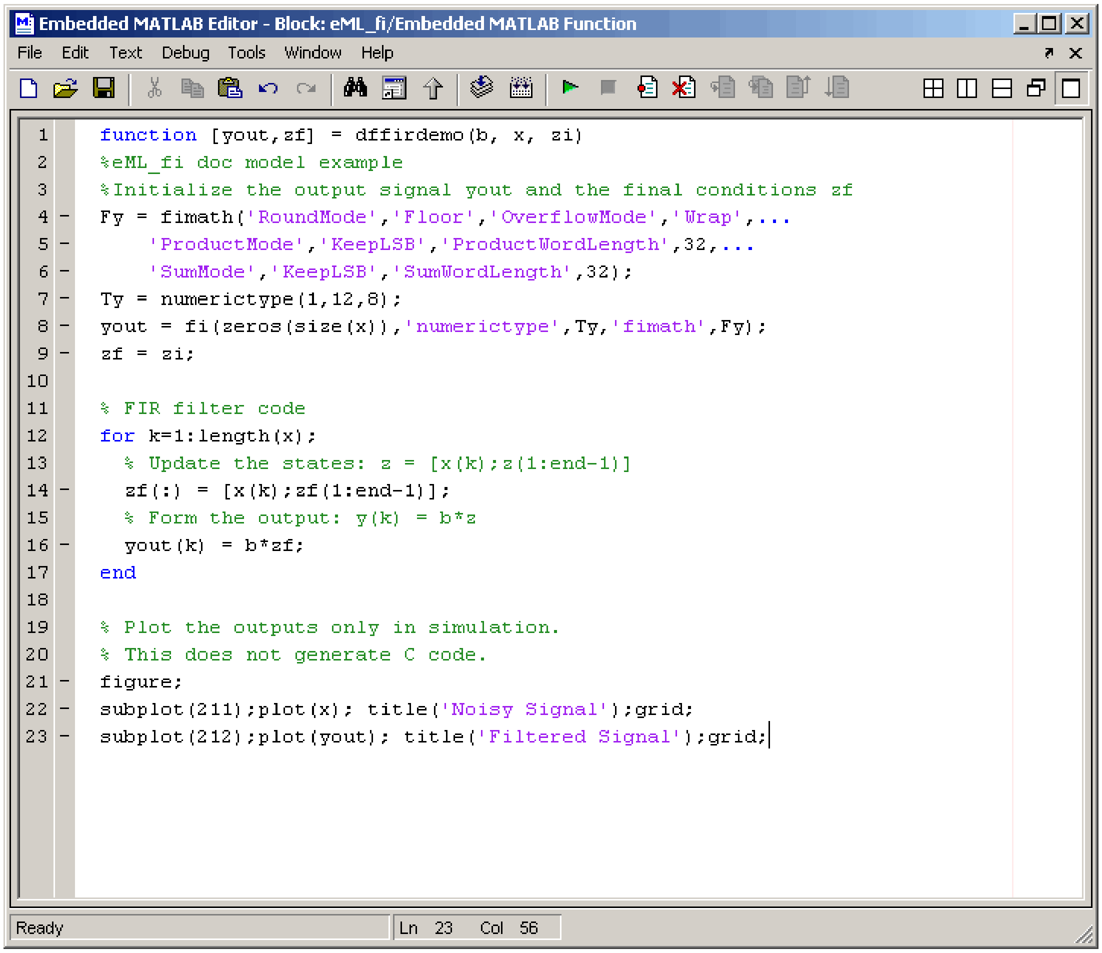
2 Save your model as `eML_fi.mdl`.

3 Double-click the Embedded MATLAB Function block in your model to open the Embedded MATLAB Editor. Type or copy and paste the following MATLAB code, including comments, into the Editor:

```
function [yout,zf] = dffirdemo(b, x, zi)
%eML_fi doc model example
%Initialize the output signal yout and the final conditions zf
Fy = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32);
Ty = numerictype(1,12,8);
yout = fi(zeros(size(x)),'numerictype',Ty,'fimath',Fy);
zf = zi;

% FIR filter code
for k=1:length(x);
  % Update the states: z = [x(k);z(1:end-1)]
  zf(:) = [x(k);zf(1:end-1)];
  % Form the output: y(k) = b*z
  yout(k) = b*zf;
end

% Plot the outputs only in simulation.
% This does not generate C code.
figure;
```

```
subplot(211);plot(x); title('Noisy Signal');grid;
subplot(212);plot(yout); title('Filtered Signal');grid;
```

The Editor should now appear as follows:


Embedded MATLAB Editor - Block: eML_fi/Embedded MATLAB Function
File Edit Text Debug Tools Window Help
Ready
Ln 23 Col 56

```
function [yout,zf] = dffirdemo(b, x, zi)
%eML_fi doc model example
%Initialize the output signal yout and the final conditions zf
Fy = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32);
Ty = numerictype(1,12,8);
yout = fi(zeros(size(x)),'numerictype',Ty,'fimath',Fy);
zf = zi;

% FIR filter code
for k=1:length(x);
  % Update the states: z = [x(k);z(1:end-1)]
  zf(:) = [x(k);zf(1:end-1)];
  % Form the output: y(k) = b*z
  yout(k) = b*zf;
end

% Plot the outputs only in simulation.
% This does not generate C code.
figure;
subplot(211);plot(x); title('Noisy Signal');grid;
subplot(212);plot(yout); title('Filtered Signal');grid;
```

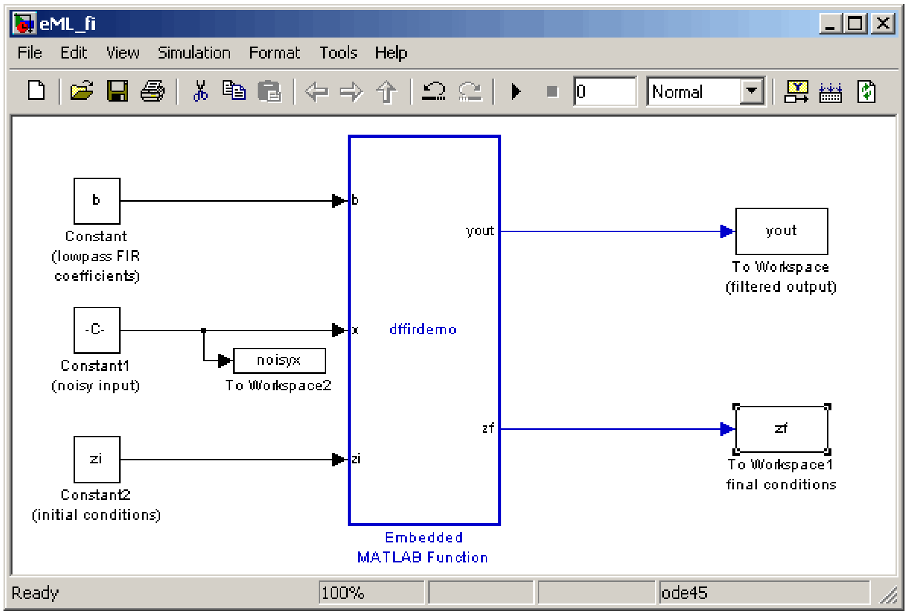
II. Prepare the Inputs

Define the filter coefficients *b*, noise *x*, and initial conditions *zi* by typing the following at the MATLAB command line:

```
b = fi_fir_coefficients;
load mtlb
x = mtlb;
n = length(x);
noise = sin(2*pi*2140*(0:n-1)'./Fs);
x = x + noise;
zi = zeros(length(b),1);
```

III. Create the Model

1 Add blocks to your model to create the system shown below.



2 Set the block parameters in the model to the following values:

Block	Parameter	Value
Constant	**Constant value**	`b`
	Interpret vector parameters as 1-D	unselected
	Sampling mode	`Sample based`
	Sample time	`inf`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-12`
Constant1	**Constant value**	`x+noise`
	Interpret vector parameters as 1-D	unselected
	Sampling mode	`Sample based`
	Sample time	`1`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-8`

Block	Parameter	Value
Constant2	**Constant value**	`zi`
	Interpret vector parameters as 1-D	unselected
	Sampling mode	`Sample based`
	Sample time	`inf`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-8`
To Workspace	**Variable name**	`yout`
	Limit data points to last	`inf`
	Decimation	`1`
	Sample time	`-1`
	Save format	`Array`
	Log fixed-point data as a fi object	selected

Block	Parameter	Value
To Workspace1	**Variable name**	`zf`
	Limit data points to last	`inf`
	Decimation	`1`
	Sample time	`1`
	Save format	`Array`
	Log fixed-point data as a fi object	selected
To Workspace2	**Variable name**	`noisyx`
	Limit data points to last	`inf`
	Decimation	`1`
	Sample time	`1`
	Save format	`Array`
	Log fixed-point data as a fi object	selected

IV. Define the Input fimath Using the Model Explorer

1 Define the `fimath` object to be used for the Embedded MATLAB block inputs in the MATLAB workspace. Note that it must have the same properties as the `fimath` object defined in your Embedded MATLAB code in order to perform arithmetic between the quantities:

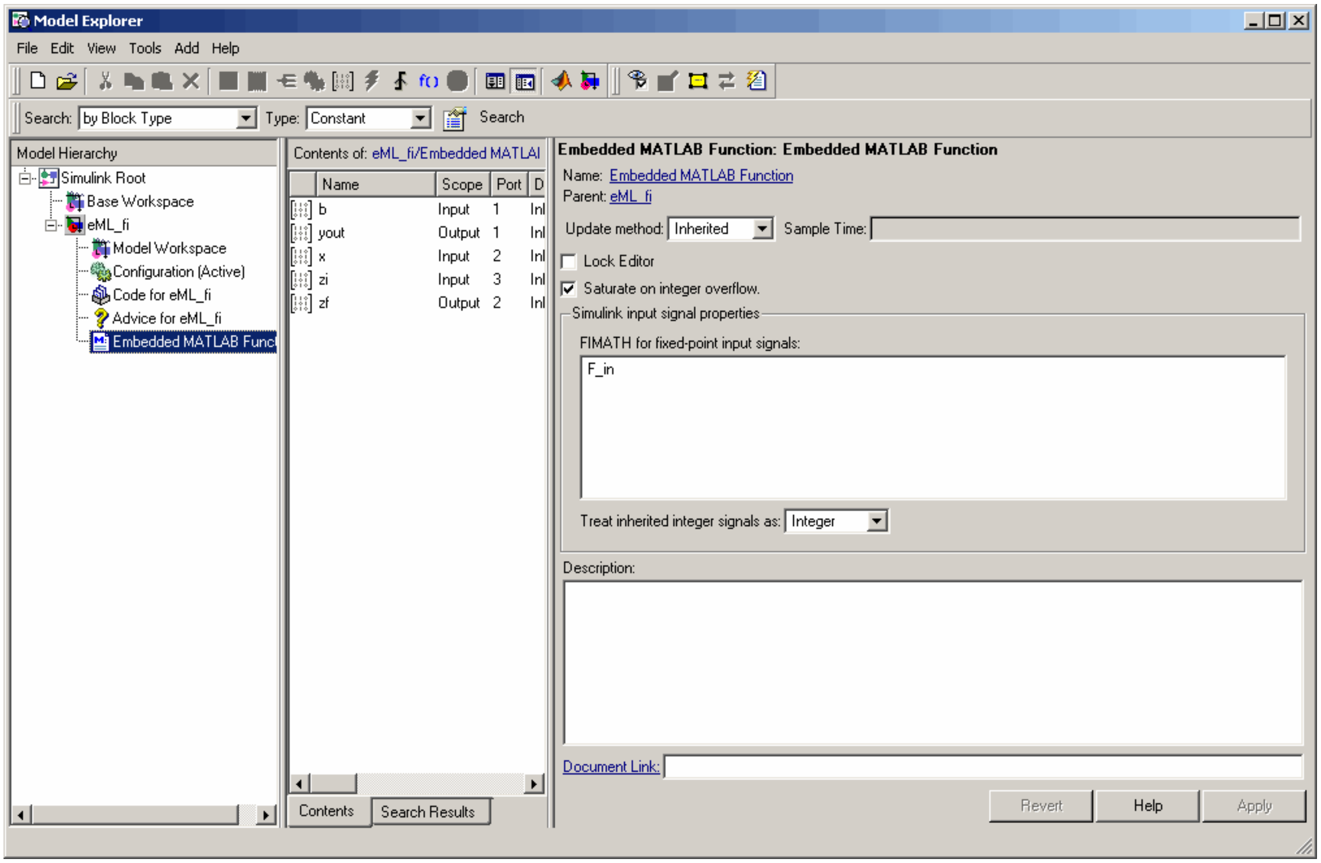
```
F_in = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32)

F_in =


             RoundMode: floor
          OverflowMode: wrap
          ProductMode: KeepLSB
    ProductWordLength: 32
              SumMode: KeepLSB
        SumWordLength: 32
        CastBeforeSum: true
```

2 Open the Model Explorer for the model by selecting **View** > **Model Explorer**.

3 Click the **Base Workspace** node in the **Model Hierarchy** pane of the Model Explorer. You see the `fimath` `F_in` you just defined listed in the **Contents** pane.

4 Click the **eML_fi** > **Embedded MATLAB Function** node in the **Model Hierarchy** pane. The dialog for the Embedded MATLAB Function block appears in the **Dialog** pane of the Model Explorer.

5 Enter `F_in` in the **FIMATH for fixed-point input signals** parameter on the Embedded MATLAB Function dialog in the **Dialog** pane of the Model Explorer and click **Apply**. This step sets the `fimath` object for the three inputs entering into the Embedded MATLAB Function block in your model. The Model Explorer now appears as follows:



Name	Scope	Port	D
b	Input	1	Inl
yout	Output	1	Inl
x	Input	2	Inl
zi	Input	3	Inl
zf	Output	2	Inl

Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function
Parent: eML_fi

Update method: Inherited Sample Time:

☐ Lock Editor
☑ Saturate on integer overflow.

Simulink input signal properties

FIMATH for fixed-point input signals:
F_in

Treat inherited integer signals as: Integer

Description:

Document Link:

V. Run the Simulation

1 You can now run the simulation by selecting your model and typing **Ctrl+T**. While the simulation is running, information will output to the MATLAB command line. You can look at the plots of the noisy signal and the filtered signal.

2 Now build your Embedded MATLAB code by selecting your model and typing **Ctrl+B**. While the code is building, information will output to the MATLAB command line. A directory called `eML_fi_grt_rtw` will be created in your current working directory.

3 Navigate to `eML_fi_grt_rtw` > `eML_fi.c`. In this file you can see the code that has been generated from your model. Search on the comment in your code

   ```
   %eML_fi doc model example
   ```

   This brings you to the beginning of the section of the code that is generated from your Embedded MATLAB Function block.

Using fi Objects with Filter Design Toolbox

When the `Arithmetic` property is set to `'fixed'`, you can use an existing `fi` object as the input, states, or coefficients of a `dfilt` object in the Filter Design Toolbox. Also, fixed-point filters in the Filter Design Toolbox return `fi` objects as outputs. Refer to the Filter Design Toolbox documentation for more information.

Property Reference

fi Object Properties

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

Note The `fimath` properties and `numerictype` properties are also properties of the `fi` object. Refer to “fimath Object Properties” on page 9-6 and “numerictype Object Properties” on page 9-17 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` object associated with a `fi` object. The default `fimath` object has the following settings:

```
              RoundMode: nearest
           OverflowMode: saturate
            ProductMode: FullPrecision
   MaxProductWordLength: 128
                SumMode: FullPrecision
       MaxSumWordLength: 128
          CastBeforeSum: true
```

To learn more about `fimath` properties, refer to "fimath Object Properties" on page 9-6.

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`, `uint8`, `uint16`, and `uint32` to get the stored integer value of a fi object in these formats.

NumericType

Structure containing all the data type and scaling attributes of a `fi` object. The `numerictype` object acts the same way as any MATLAB structure, except that it only lets you set valid values for defined fields. The following table shows the possible settings of each field of the structure that are valid for `fi` objects.

DataTypeMode	DataType	Scaling	Signed	WordLength	FractionLength	Slope	Bias
Fully specified fixed-point data types							
`Fixed-point: binary point scaling`	`Fixed`	`BinaryPoint`	`1/0` `true/false`	positive integer from 1 to 65,536	positive or negative integer	`1`	`0`
`Fixed-point: slope and bias scaling`	`Fixed`	`SlopeBias`	`1/0` `true/false`	positive integer from 1 to 65,536	`N/A`	any floating-point number	any floating-point number
Partially specified fixed-point data type							

DataTypeMode	Data-Type	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
`Fixed-point: unspecified scaling`	`Fixed`	`Unspecified`	`1/0` `true/ false`	positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`
Fully specified scaled double data types							
`Scaled double: binary point scaling`	`ScaledDouble`	`BinaryPoint`	`1/0` `true/ false`	positive integer from 1 to 65,536	positive or negative integer	`1`	`0`
`Scaled double: slope and bias scaling`	`ScaledDouble`	`SlopeBias`	`1/0` `true/ false`	positive integer from 1 to 65,536	`N/A`	any floating-point number	any floating-point number
Partially specified scaled double data type							
`Scaled double: unspecified scaling`	`ScaledDouble`	`Unspecified`	`1/0` `true/ false`	positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`
Built-in data types							
`double`	`double`	`N/A`	`1` `true`	`64`	`0`	`1`	`0`
`single`	`single`	`N/A`	`1` `true`	`32`	`0`	`1`	`0`
`boolean`	`boolean`	`N/A`	`0` `false`	`1`	`0`	`1`	`0`

You cannot change the `numerictype` properties of a `fi` object after `fi` object creation.

oct

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

fimath Object Properties

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

CastBeforeSum

Whether both operands are cast to the sum data type before addition. Possible values of this property are `1` (cast before sum) and `0` (do not cast before sum).

The default value of this property is `1` (true).

MaxProductWordLength

Maximum allowable word length for the product data type.

The default value of this property is `128`.

MaxSumWordLength

Maximum allowable word length for the sum data type.

The default value of this property is `128`.

OverflowMode

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

- `saturate` — Saturate to maximum or minimum value of the fixed-point range on overflow.
- `wrap` — Wrap on overflow. This mode is also known as two's complement overflow.

The default value of this property is `saturate`.

ProductBias

Bias of the product data type. This value can be any floating-point number. The product data type defines the data type of the result of a multiplication of two `fi` objects.

The default value of this property is `0`.

ProductFixedExponent

Fixed exponent of the product data type. This value can be any positive or negative integer. The product data type defines the data type of the result of a multiplication of two `fi` objects.

`ProductSlope = ProductSlopeAdjustmentFactor * 2 ^ ProductFixedExponent`
Changing one of these properties changes the others.

The `ProductFixedExponent` is the negative of the `ProductFractionLength`. Changing one property changes the other.

The default value of this property is `-30`.

ProductFractionLength

Fraction length, in bits, of the product data type. This value can be any positive or negative integer. The product data type defines the data type of the result of a multiplication of two `fi` objects.

The `ProductFractionLength` is the negative of the `ProductFixedExponent`. Changing one property changes the other.

The default value of this property is `30`.

ProductMode

Defines how the product data type is determined. In the following descriptions, let A and B be real operands, with [word length, fraction length] pairs [W_a F_a] and [W_b F_b], respectively. W_p is the product data type word length and F_p is the product data type fraction length.

- `FullPrecision` — The full precision of the result is kept. An error is generated if the calculated word length is greater than `MaxProductWordLength`.

  $$W_p = W_a + W_b$$
  $$F_p = F_a + F_b$$

- `KeepLSB` — Keep least significant bits. You specify the product data type word length, while the fraction length is set to maintain the least significant bits of the product. In this mode, full precision is kept, but overflow is possible. This behavior models the C language integer operations.

  $W_p =$ specified in the `ProductWordLength` property

  $F_p = F_a + F_b$

- `KeepMSB` — Keep most significant bits. You specify the product data type word length, while the fraction length is set to maintain the most significant bits of the product. In this mode, overflow is prevented, but precision may be lost.

  $W_p =$ specified in the `ProductWordLength` property

  $F_p = W_p -$ integer length

  where

  integer length $= (W_a + W_b) - (F_a - F_b)$

- `SpecifyPrecision` — You specify both the word length and fraction length of the product data type.

  $W_p =$ specified in the `ProductWordLength` property

  $F_p =$ specified in the `ProductFractionLength` Property

  For [Slope Bias] math, you specify both the slope and bias of the product data type.

S_p = specified in the `ProductSlope` property

B_p = specified in the `ProductBias` property

[Slope Bias] math is only defined for products when `ProductMode` is set to `SpecifyPrecision`.

The default value of this property is `FullPrecision`.

ProductSlope

Slope of the product data type. This value can be any floating-point number. The product data type defines the data type of the result of a multiplication of two `fi` objects.

`ProductSlope` = `ProductSlopeAdjustmentFactor` * 2 ^ `ProductFixedExponent`
Changing one of these properties changes the others.

The default value of this property is `9.3132e-010`.

ProductSlopeAdjustmentFactor

Slope adjustment factor of the product data type. This value can be any floating-point number greater than or equal to 1 and less than 2. The product data type defines the data type of the result of a multiplication of two `fi` objects.

`ProductSlope` = `ProductSlopeAdjustmentFactor` * 2 ^ `ProductFixedExponent`
Changing one of these properties changes the others.

The default value of this property is `1`.

ProductWordLength

Word length, in bits, of the product data type. This value must be a positive integer. The product data type defines the data type of the result of a multiplication of two `fi` objects.

The default value of this property is `32`.

RoundMode

The rounding mode. The value of the `RoundMode` property can be one of the following strings:

- `ceil` — Round toward positive infinity.
- `convergent` — Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased rounding method provided by the Fixed-Point Toolbox.
- `fix` — Round toward zero.
- `floor` — Round toward negative infinity.
- `nearest` — Round toward nearest. Ties round toward positive infinity.
- `round` — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The default value of this property is `nearest`.

SumBias

The bias of the sum data type. This value can be any floating-point number. The sum data type defines the data type of the result of a sum of two `fi` objects.

The default value of this property is `0`.

SumFixedExponent

The fixed exponent of the sum data type. This value can be any positive or negative integer. The sum data type defines the data type of the result of a sum of two `fi` objects

`SumSlope = SumSlopeAdjustmentFactor*2^SumFixedExponent.`
Changing one of these properties changes the others.

The `SumFixedExponent` is the negative of the `SumFractionLength`. Changing one property changes the other.

The default value of this property is `-30`.

SumFractionLength

The fraction length, in bits, of the sum data type. This value can be any positive or negative integer. The sum data type defines the data type of the result of a sum of two `fi` objects.

The `SumFractionLength` is the negative of the `SumFixedExponent`. Changing one property changes the other.

The default value of this property is`30` .

SumMode

Defines how the sum data type is determined. In the following descriptions, let A and B be real operands, with [word length, fraction length] pairs [W_a F_a] and [W_b F_b], respectively. W_s is the sum data type word length and F_s is the sum data type fraction length.

Note In the case where there are two operands, as in $A + B$, *NumberOfSummands* is `2`, and `ceil(log2(`*`NumberOfSummands`*`)) = 1`. In `sum(`*`A`*`)` where A is a matrix, the *NumberOfSummands* is `size(`*`A`*`,1)`. In `sum(`*`A`*`)` where A is a vector, the *NumberOfSummands* is `length(`*`A`*`)`.

- `FullPrecision` — The full precision of the result is kept. An error is generated if the calculated word length is greater than `MaxSumWordLength`.

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

  where

  $$\text{integer length} = \max(W_a - F_a, W_b - F_b) + \text{ceil}\left(\log 2(\mathit{NumberOfSummands})\right)$$

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

- `KeepLSB` — Keep least significant bits. You specify the sum data type word length, while the fraction length is set to maintain the least significant bits of the sum. In this mode, full precision is kept, but overflow is possible. This behavior models the C language integer operations.

W_s = specified in the `SumWordLength` property

$F_s = \max(F_a, F_b)$

- `KeepMSB` — Keep most significant bits. You specify the sum data type word length, while the fraction length is set to maintain the most significant bits of the sum and no more fractional bits than necessary. In this mode, overflow is prevented, but precision may be lost.

  W_s = specified in the `SumWordLength` property

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

  where

  $$\text{integer length} = \max\left(W_a - F_a, W_b - F_b\right) + \text{ceil}\left(\log 2\left(\mathit{NumberOfSummands}\right)\right)$$

- `SpecifyPrecision` — You specify both the word length and fraction length of the sum data type.

  W_s = specified in the `SumWordLength` property

  F_s = specified in the `SumFractionLength` property

  For [Slope Bias] math, you specify both the slope and bias of the sum data type.

  S_s = specified in the `SumSlope` property

  B_s = specified in the `SumBias` property

  [Slope Bias] math is only defined for sums when `SumMode` is set to `SpecifyPrecision`.

The default value of this property is `FullPrecision`.

SumSlope

The slope of the sum data type. This value can be any floating-point number. The sum data type defines the data type of the result of a sum of two `fi` objects.

`SumSlope = SumSlopeAdjustmentFactor * 2^SumFixedExponent`.
Changing one of these properties changes the others.

The default value of this property is `9.3132e-010`.

SumSlopeAdjustmentFactor

The slope adjustment factor of the sum data type. This value can be any floating-point number greater than or equal to 1 and less than 2. The sum data type defines the data type of the result of a sum of two `fi` objects.

`SumSlope = SumSlopeAdjustmentFactor*2^SumFixedExponent`. Changing one of these properties changes the others.

The default value of this property is `1`.

SumWordLength

The word length, in bits, of the sum data type. This value must be a positive integer. The sum data type defines the data type of the result of a sum of two `fi` objects.

The default value of this property is `32`.

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
- `True Singles` — Override with singles

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

The default value of this property is `ForceOff`.

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 of 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

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

numerictype Object Properties

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

Bias

Bias associated with a `fi` object. 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. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

DataType

Data type associated with a `fi` object. The possible value of this property are

- `boolean` — Built-in MATLAB `boolean` data type
- `double` — Built-in MATLAB `double` data type
- `Fixed` — Fixed-point or integer data type
- `ScaledDouble` — Scaled double data type
- `single` — Built-in MATLAB `single` data type

The default value of this property is `fixed`.

DataTypeMode

Data type and scaling associated with a `fi` object. The possible values of this property are

- `boolean` — Built-in `boolean`

- `double` — Built-in `double`
- `Fixed-point:  binary point scaling` — Fixed-point data type and scaling defined by the word length and fraction length
- `Fixed-point:  slope and bias scaling` — Fixed-point data type and scaling defined by the slope and bias
- `Fixed-point:  unspecified scaling` —- Fixed-point data type with unspecified scaling
- `Scaled double:  binary point scaling` — Double data type with fixed-point word length and fraction length information retained
- `Scaled double:  slope and bias scaling` — Double data type with fixed-point slope and bias information retained
- `Scaled double:  unspecified scaling` —- Double data type with unspecified fixed-point scaling
- `single` — Built-in `single`

The default value of this property is `Fixed-point:  binary point scaling`.

FixedExponent

Fixed-point exponent associated with a `fi` object. The exponent is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$\text{fixed exponent} = -\text{fraction length}$$

FractionLength

Value of the `FractionLength` property is the fraction length of the stored integer value of a `fi` object, in bits. The fraction length can be any integer value. If you do not specify the fraction length of a `fi` object, it is set to the best possible precision.

This property is automatically set by default to the best precision possible based on the value of the word length.

Scaling

Fixed-point scaling mode of a `fi` 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, in order to allow for the automatic assignment of a binary point best precision scaling.
- `Integer` — The `fi` object is an integer; the binary point is understood to be at the far right of the word, making the fraction length zero.

The default value of this property is `BinaryPoint`.

Signed

Whether a `fi` object is signed. The possible values of this property are

- `1` — signed
- `0` — unsigned
- `true` — signed
- `false` — unsigned

The default value of this property is `true` .

Slope

Slope associated with a `fi` 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. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractionalslope} \times 2^{\textit{fixed exponent}}$$

SlopeAdjustmentFactor

Slope adjustment associated with a `fi` 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. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractionalslope} \times 2^{\textit{fixed exponent}}$$

WordLength

Value of the `WordLength` property is the word length of the stored integer value of a fixed-point object, in bits. The word length can be any positive integer value.

The default value of this property is `16`.

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`	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. 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.
`float`	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. The word length can range from 2 to the limits of memory on your PC. The exponent length can range from 0 to 11.
`double`	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.
`single`	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.

OverflowMode

Overflow-handling mode. The value of the `OverflowMode` 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 `±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`.

RoundMode

Rounding mode. The value of the `RoundMode` property can be one of the following strings:

- `ceil` — 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.
- `fix` — 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 — By Category

fimath Object Functions (p. 10-17)	All functions that operate directly on `fimath` objects
fipref Object Functions (p. 10-18)	All functions that operate directly on `fipref` objects
numerictype Object Functions (p. 10-19)	All functions that operate directly on `numerictype` objects
quantizer Object Functions (p. 10-20)	All functions that operate directly on `quantizer` objects

Bitwise Functions

`bitand`	Bitwise `AND` of two `fi` objects
`bitcmp`	Bitwise complement of `fi` object
`bitget`	Bit at certain position
`bitor`	Bitwise `OR` of two `fi` objects
`bitset`	Set bit at certain position
`bitshift`	Shift bits specified number of places
`bitxor`	Bitwise exclusive `OR` of two `fi` objects

Constructor and Property Functions

`copyobj`	Make independent copy of `quantizer` object
`fi`	Construct `fi` object
`fimath`	Construct `fimath` object
`fipref`	Construct `fipref` object
`get`	Property values of `quantizer` object

`inspect`	Property Inspector
`numerictype`	Construct `numerictype` object
`quantizer`	Construct `quantizer` object
`reset`	Reset objects to initial conditions
`savefipref`	Save `fi` preferences for next MATLAB session
`set`	Set or display property values for `quantizer` objects
`stripscaling`	Stored integer of `fi` object
`tostring`	Convert `quantizer` object to string

Data Manipulation Functions

`denormalmax`	Largest denormalized quantized number for `quantizer` object
`denormalmin`	Smallest denormalized quantized number for `quantizer` object
`eps`	Quantized relative accuracy for `fi` or `quantizer` objects
`exponentbias`	Exponent bias for `quantizer` object
`exponentlength`	Exponent length of `quantizer` object
`exponentmax`	Maximum exponent for `quantizer` object
`exponentmin`	Minimum exponent for `quantizer` object
`fractionlength`	Fraction length of `quantizer` object
`intmax`	Largest positive stored integer value representable by `numerictype` of `fi` object

`intmin`	Smallest stored integer value representable by `numerictype` of `fi` object
`isequal`	Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isfi`	Determine whether variable is `fi` object
`isfimath`	Determine whether variable is `fimath` object
`isnumerictype`	Determine whether variable is `numerictype` object
`ispropequal`	Determine whether properties of two `fi` objects are equal
`issigned`	Determine whether `fi` object is signed
`lowerbound`	Lower bound of range of `fi` object
`lsb`	Scaling of least significant bit of `fi` object
`range`	Numerical range of `fi` or `quantizer` object
`realmax`	Largest positive fixed-point value or quantized number
`realmin`	Smallest positive normalized fixed-point value or quantized number
`rescale`	Change scaling of `fi` object
`upperbound`	Upper bound of range of `fi` object
`wordlength`	Word length of `quantizer` object

Data Type Functions

`double`	Double-precision floating-point real-world value of `fi` object
`int`	Smallest built-in integer in which stored integer value of `fi` object will fit
`int16`	Stored integer value of `fi` object as built-in `int16`
`int32`	Stored integer value of `fi` object as built-in `int32`
`int8`	Stored integer value of `fi` object as built-in `int8`
`logical`	Convert numeric values to logical
`single`	Single-precision floating-point real-world value of `fi` object
`uint16`	Stored integer value of `fi` object as built-in `uint16`
`uint32`	Stored integer value of `fi` object as built-in `uint32`
`uint8`	Stored integer value of `fi` object as built-in `uint8`

Data Quantizing Functions

`convergent`	Apply convergent rounding
`quantize`	Apply `quantizer` object to data

`randquant`	Generate uniformly distributed, quantized random number using `quantizer` object
`round`	Round input data using `quantizer` object without checking for overflow

Element-Wise Logical Operator Functions

`all`	Determine whether all array elements are nonzero
`and`	Find logical AND of array or scalar inputs
`any`	Determine whether any array elements are nonzero
`not`	Find logical NOT of array or scalar input
`or`	Find logical OR of array or scalar inputs

Math Operation Functions

`abs`	Absolute value of `fi` object
`add`	Add two objects using `fimath` object
`complex`	Construct complex `fi` object from real and imaginary parts
`conj`	Complex conjugate of `fi` object
`divide`	Divide two objects
`imag`	Imaginary part of complex number

`innerprodintbits`	Number of integer bits needed for fixed-point inner product
`minus`	Matrix difference between `fi` objects
`mpy`	Multiply two objects using `fimath` object
`mtimes`	Matrix product of `fi` objects
`plus`	Matrix sum of `fi` objects
`pow2`	Multiply by a power of 2
`real`	Real part of complex number
`sign`	Perform signum function on array
`sub`	Subtract two objects using `fimath` object
`sum`	Sum of array elements
`times`	Element-by-element multiplication of `fi` objects
`uminus`	Negate elements of `fi` object array
`uplus`	Unary plus

Matrix Manipulation Functions

`buffer`	Buffer signal vector into matrix of data frames
`ctranspose`	Complex conjugate transpose of `fi` object
`diag`	Diagonal matrices or diagonals of matrix
`disp`	Display object
`end`	Last index of array
`hankel`	Hankel matrix

`horzcat`	Horizontally concatenate multiple `fi` objects
`ipermute`	Inverse permute dimensions of multidimensional array
`iscolumn`	Determine whether `fi` object is column vector
`isempty`	Determine whether array is empty
`isfinite`	Determine whether array elements are finite
`isinf`	Determine whether array elements are infinite
`isnan`	Determine whether array elements are NaN
`isnumeric`	Determine whether input is numeric array
`isobject`	Determine whether input is MATLAB OOPS object
`isreal`	Determine whether array elements are real
`isrow`	Determine whether `fi` object is row vector
`isscalar`	Determine whether input is scalar
`isvector`	Determine whether input is vector
`length`	Vector length
`ndims`	Number of array dimensions
`permute`	Rearrange dimensions of multidimensional array
`repmat`	Replicate and tile array
`reshape`	Reshape array
`size`	Array dimensions
`squeeze`	Remove singleton dimensions

`toeplitz`	Create Toeplitz matrix
`transpose`	Transpose operation
`tril`	Lower triangular part of matrix
`vertcat`	Vertically concatenate multiple`fi` objects

Plotting Functions

`area`	Create filled area 2-D plot
`bar`	Create vertical bar graph
`barh`	Create horizontal bar graph
`clabel`	Create contour plot elevation labels
`comet`	Create 2-D comet plot
`comet3`	Create 3-D comet plot
`compass`	Plot arrows emanating from origin
`coneplot`	Plot velocity vectors as cones in 3-D vector field
`contour`	Create contour graph of matrix
`contour3`	Create 3-D contour plot
`contourc`	Create two-level contour plot computation
`contourf`	Create filled 2-D contour plot
`errorbar`	Plot error bars along curve
`etreeplot`	Plot elimination tree
`ezcontour`	Easy-to-use contour plotter
`ezcontourf`	Easy-to-use filled contour plotter
`ezmesh`	Easy-to-use 3-D mesh plotter
`ezplot`	Easy-to-use function plotter

`ezplot3`	Easy-to-use 3-D parametric curve plotter
`ezpolar`	Easy-to-use polar coordinate plotter
`ezsurf`	Easy-to-use 3-D colored surface plotter
`ezsurfc`	Easy-to-use combination surface/contour plotter
`feather`	Plot velocity vectors
`fplot`	Plot function between specified limits
`gplot`	Plot set of nodes using adjacency matrix
`hist`	Create histogram plot
`histc`	Histogram count
`line`	Create line object
`loglog`	Create log-log scale plot
`mesh`	Create mesh plot
`meshc`	Create mesh plot with contour plot
`meshz`	Create mesh plot with curtain plot
`patch`	Create patch graphics object
`pcolor`	Create pseudocolor plot
`plot`	Create linear 2-D plot
`plot3`	Create 3-D line plot
`plotmatrix`	Draw scatter plots
`plotyy`	Create graph with y-axes on right and left sides
`polar`	Plot polar coordinates
`quiver`	Create quiver or velocity plot
`quiver3`	Create 3-D quiver or velocity plot

`rgbplot`	Plot colormap
`ribbon`	Create ribbon plot
`rose`	Create angle histogram
`scatter`	Create scatter or bubble plot
`scatter3`	Create 3-D scatter or bubble plot
`semilogx`	Create semilogarithmic plot with logarithmic x-axis
`semilogy`	Create semilogarithmic plot with logarithmic y-axis
`slice`	Create volumetric slice plot
`spy`	Visualize sparsity pattern
`stairs`	Create stairstep graph
`stem`	Plot discrete sequence data
`stem3`	Plot 3-D discrete sequence data
`streamribbon`	Create 3-D stream ribbon plot
`streamslice`	Draw streamlines in slice planes
`streamtube`	Create 3-D stream tube plot
`surf`	Create 3-D shaded surface plot
`surfc`	Create 3-D shaded surface plot with contour plot
`surfl`	Create surface plot with colormap-based lighting
`surfnorm`	Compute and display 3-D surface normals
`text`	Create text object in current axes
`treeplot`	Plot picture of tree
`trimesh`	Create triangular mesh plot
`triplot`	Create 2-D triangular plot
`trisurf`	Create triangular surface plot

`triu`	Upper triangular part of matrix
`voronoi`	Create Voronoi diagram
`voronoin`	Create n-D Voronoi diagram
`waterfall`	Create waterfall plot
`xlim`	Set or query x-axis limits
`ylim`	Set or query y-axis limits

Radix Conversion Functions

`bin`	Binary representation of stored integer of `fi` object
`bin2num`	Convert two's complement binary string to number using `quantizer` object
`dec`	Unsigned decimal representation of stored integer of `fi` object
`hex`	Hexadecimal representation of stored integer of `fi` object
`hex2num`	Convert hexadecimal string to number using `quantizer` object
`num2bin`	Convert number to binary string using `quantizer` object
`num2hex`	Convert number to hexadecimal equivalent using `quantizer` object
`num2int`	Convert number to signed integer
`oct`	Octal representation of stored integer of `fi` object
`sdec`	Signed decimal representation of stored integer of `fi` object

Relational Operator Functions

`eq`	Determine whether real-world values of two `fi` objects are equal
`ge`	Determine whether real-world value of one `fi` object is greater than or equal to another
`gt`	Determine whether real-world value of one `fi` object is greater than another
`le`	Determine whether real-world value of `fi` object is less than or equal to another
`lt`	Determine whether real-world value of one `fi` object is less than another
`ne`	Determine whether real-world values of two `fi` objects are not equal

Statistics Functions

`max`	Largest element in array of `fi` objects
`maxlog`	Largest real-world value of `fi` object or maximum value of `quantizer` object before quantization
`min`	Smallest element in array of `fi` objects
`minlog`	Smallest real-world value of `fi` object or minimum value of `quantizer` object before quantization
`noperations`	Number of operations
`noverflows`	Number of overflows

`numberofelements`	Number of data elements in `fi` array
`nunderflows`	Number of underflows
`resetlog`	Clear log for `fi` or `quantizer` object

Subscripted Assignment and Reference Functions

`subsasgn`	Subscripted assignment
`subsref`	Subscripted reference

fi Object Functions

abs	all	and	any	area
bar	barh	bin	bitand	bitcmp
bitget	bitor	bitshift	bitxor	buffer
clabel	comet	comet3	compass	complex
coneplot	conj	contour	contour3	contourc
contourf	ctranspose	dec	diag	double
end	eps	eq	errorbar	etreeplot
ezcontour	ezcontourf	ezmesh	ezplot	ezplot3
ezpolar	ezsurf	ezsurfc	feather	fi
fimath	fplot	ge	get	gplot
gt	hankel	hex	hist	histc
horzcat	imag	innerprodintbits	inspect	int
int8	int16	int32	intmax	intmin
ipermute	iscolumn	isempty	isequal	isfi
isfinite	isinf	isnan	isnumeric	isobject
ispropequal	isreal	isrow	isscalar	issigned
isvector	le	length	line	logical
lowerbound	lsb	lt	max	mesh
meshc	meshz	min	minus	mtimes
ndims	ne	not	numberofelements	numerictype
oct	or	patch	pcolor	permute
plot	plot3	plotmatrix	plotyy	plus
polar	pow2	quantizer	quiver	quiver3
range	real	realmax	realmin	repmat
rescale	reshape	rgbplot	ribbon	rose
scatter	scatter3	sdec	sign	single

size	slice	spy	stairs	stem
stem3	streamribbon	streamslice	streamtube	stripscaling
subsasgn	subsref	sum	surf	surfc
surfl	surfnorm	text	times	toeplitz
transpose	treeplot	tril	trimesh	triplot
trisurf	triu	uint8	uint16	uint32
uminus	uplus	upperbound	vertcat	voronoi
voronoin	waterfall	xlim	ylim	zlim

fimath Object Functions

- `add`
- `disp`
- `fimath`
- `isequal`
- `isfimath`
- `mpy`
- `sub`

fipref Object Functions

- `disp`
- `fipref`
- `reset`
- `savefipref`

numerictype Object Functions

- `divide`
- `isequal`
- `isnumerictype`

quantizer Object Functions

`bin2num`	`copyobj`	`denormalmax`	`denormalmin`	`disp`
`eps`	`exponentbias`	`exponentlength`	`exponentmax`	`exponentmin`
`fractionlength`	`get`	`hex2num`	`isequal`	`length`
`max`	`min`	`noperations`	`noverflows`	`num2bin`
`num2hex`	`num2int`	`nunderflows`	`quantize`	`quantizer`
`randquant`	`range`	`realmax`	`realmin`	`reset`
`round`	`set`	`tostring`	`wordlength`	

Functions — Alphabetical List

Purpose

Absolute value of fi object

Syntax

```
abs(a)
```

Description

abs(a) returns the absolute value of fi object a.

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.

abs does not support complex inputs.

Examples

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
                Signed: true
            WordLength: 16
        FractionLength: 8

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

```
abs(a)

ans =

  127.9961

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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
a.OverflowMode = 'wrap'

a =

  -128

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

             RoundMode: nearest
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
```

```
         CastBeforeSum: true

abs(a)

ans =

  -128

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

             RoundMode: nearest
          OverflowMode: wrap
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

Purpose

Add two objects using `fimath` object

Syntax

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

Description

`c = F.add(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` objects of `a` and `b` are different.

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

If either `a` or `b` is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length.

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 = F.add(a,b)

c =

    5.8599


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: SpecifyPrecision
          SumWordLength: 32
      SumFractionLength: 16
          CastBeforeSum: true
```

Algorithm

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

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

except that the `fimath` properties of `a` and `b` are not modified when you use the functional form.

See Also

`divide`, `fi`, `fimath`, `mpy`, `numerictype`, `sub`, `sum`

Purpose Determine whether all array elements are nonzero

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

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.

Purpose Create filled area 2-D plot

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

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

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`bin(a)` returns the stored integer of `fi` object `a` in unsigned binary format as a string.

Examples

The following code

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

returns

```
10000000   01111111
```

See Also

`dec`, `hex`, `int`, `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`

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 fimath and the numerictype objects of a and b 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

Bitwise complement of `fi` object

Syntax

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

Description

`c = bitcmp(a)` returns the bitwise complement of `fi` object `a` as an n-bit nonnegative integer. If `a` has a signed `numerictype`, then the bit representation of the stored integer is in two's complement representation.

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

See Also

`bitand`, `bitget`, `bitor`, `bitset`, `bitxor`

Purpose

Bit at certain position

Syntax

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

Description

`c = bitget(a, bit)` returns the value of the bit at position `bit` in `a`. `a` must be a nonnegative integer, and `bit` must be a number between `1` and the number of bits in the floating-point integer representation of `a`. If `a` has a signed `numerictype`, then the bit representation of the stored integer is in two's complement representation.

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

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 `fimath` and the `numerictype` objects of `a` and `b` 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.

See Also

`bitand`, `bitcmp`, `bitget`, `bitset`, `bitxor`

Purpose

Set bit at certain position

Syntax

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

Description

`c = bitset(a, bit)` sets bit position `bit` in `a` to `1` (on).

`c = bitset(a, bit, v)` sets bit position `bit` in `a` to `v`. `v` must be `0` (off) or `1` (on). Any value `v` other than `0` is automatically set to `1`.

`a` must be a nonnegative integer, and `bit` must be a number between `1` and the number of bits in the floating-point integer representation of `a`. If `a` has a signed `numerictype`, then the bit representation of the stored integer is in two's complement representation.

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

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.

`fi` object `a` can be any fixed-point numeric type. The `OverflowMode` and `RoundMode` properties are obeyed.

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

Example

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

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

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

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

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

```
13. 0110000000000000
14. 0111111111111111
15. 0111111111111111
16. 0111111111111111
```

Now change `OverflowMode` 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,'OverflowMode','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`, `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 fimath and the numerictype objects of a and b 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.

See Also bitand, bitcmp, bitget, bitor, bitset

buffer

Purpose Buffer signal vector into matrix of data frames

Description Refer to the Signal Processing Toolbox `buffer` reference page for more information.

Purpose Create contour plot elevation labels

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

Purpose Create 2-D comet plot

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

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.

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 `numerictype` and `fimath` objects of the leftmost input that is a `fi` object are applied to the output `c`.

See Also

`imag`, `real`

coneplot

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

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

Purpose

Complex conjugate of `fi` object

Syntax

`conj(a)`

Description

`conj(a)` is the complex conjugate of `fi` object `a`.

When `a` is complex,

$$\mathrm{conj}(a) = \mathrm{real}(a) - i \times \mathrm{imag}(a)$$

The `numerictype` and `fimath` objects of 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.

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.

Purpose Create filled 2-D contour plot

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

Purpose

Apply convergent rounding

Syntax

`convergent(x)`

Description

`convergent(x)` rounds the elements of `x` to the nearest integer, except in a tie, then rounds to the nearest even integer.

Examples

MATLAB `round` and `convergent` differ in the way they treat values whose fractional part is 0.5. In `round`, every tie is rounded up in absolute value. `convergent` rounds ties to the nearest even integer.

```
x=[-3.5:3.5]';
[x convergent(x) round(x)]
ans =

   -3.5000   -4.0000   -4.0000
   -2.5000   -2.0000   -3.0000
   -1.5000   -2.0000   -2.0000
   -0.5000         0   -1.0000
    0.5000         0    1.0000
    1.5000    2.0000    2.0000
    2.5000    2.0000    3.0000
    3.5000    4.0000    4.0000
```

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('CoefficientFormat',[8 7]);
  q1 = copyobj(q);
```

See Also

`quantizer`, `get`, `set`

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`

Purpose

Unsigned decimal representation of stored integer of `fi` object

Syntax

`dec(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`dec(a)` returns the stored integer of `fi` object `a` in unsigned decimal format as a string.

Examples

The code

```
a = fi([-1 1],1,8,7);
dec(a)
```

returns

```
128   127
```

See Also

`bin`, `hex`, `int`, `oct`, `sdec`

denormalmax

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
```

Algorithm

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`

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]);
denormalmin(q)

ans =

    0.0625
```

Algorithm

When `q` is a floating-point `quantizer` object,

$$x = 2^{Emin - f}$$

where $E_{\min}$ is equal to `exponentmin(q)`.

When `q` is a fixed-point `quantizer` object,

$$x = \mathrm{eps(q)} = 2^{-f}$$

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.

Purpose

Display object

Description

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

Purpose

Divide two objects

Syntax

`c = T.divide(a,b)`

Description

`c = T.divide(a,b)` performs division on the elements of `a` by the elements of `b`. The result `c` has the `numerictype` object `T`.

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

If either `a` or `b` is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length.

If `a` and `b` are both MATLAB built-in doubles, then `c` is the double-precision quotient `a./b`, and `numerictype T` is ignored.

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

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 =

1100110011001100110011001100110011001100110011001101000000000000
0000000000000000
(bin)
```

```
      u80,83
   110011001100110011001100110011001100110011001100110011001100110011001100
   1100110011001100
```

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.

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 = T.divide(a,b);
   c.bin

   ans =

   110011001100110011001100110011001100110011001100110011001100110011001100
   1100110011001100
```

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 = T.divide(a,b);
```

```
c.bin

ans =

1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100
```

See Also `add`, `fi`, `fimath`, `mpy`, `numerictype`, `sub`, `sum`

Purpose

Double-precision floating-point real-world value of `fi` object

Syntax

```
double(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

`double(a)` returns the real-world value of a `fi` object in double-precision floating point.

See Also

`single`

end

Purpose Last index of array

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

Purpose

Quantized relative accuracy for `fi` or `quantizer` objects

Syntax

```
eps(obj)
```

Description

`eps(obj)` returns the value of the least significant bit of the value of the `fi` object or `quantizer` object `obj`. The result of this function is equivalent to that given by the Fixed-Point Toolbox `lsb` function.

See Also

`lsb`

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 Plot error bars along curve

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

Purpose

Plot elimination tree

Description

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

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
```

Algorithm

For floating-point `quantizer` objects,

$$b = 2^{e-1} - 1$$

where `e` = `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`

exponentlength

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
```

Algorithm 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`

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');
exponentmax(q)

ans =

        1023
```

Algorithm

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
```

Algorithm

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`

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.

Purpose Easy-to-use 3-D mesh plotter

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

ezplot

Purpose Easy-to-use function plotter

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

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

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

ezpolar

Purpose Easy-to-use polar coordinate plotter

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

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

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

ezsurfc

Purpose Easy-to-use combination surface/contour plotter

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

Purpose Plot velocity vectors

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

Purpose

Construct `fi` object

Syntax

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

Description

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

- `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`, signedness `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`, signedness `s`, word length `w`, and best-precision fraction length.
- `a = fi(v,s,w,f)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, and fraction length `f`.
- `a = fi(v,s,w,slope,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slope`, and `bias`.
- `a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.
- `a = fi(v,T)` returns a fixed-point object with value `v` and `embedded.numerictype` `T`. Refer to for more information on `numerictype` objects.

- `fi(a,F)` allows you to maintain the value and `numerictype` object of `fi` object `a`, while changing its `fimath` object to `F`. Refer to for more information on `fimath` objects.
- `a = fi(v,T,F)` returns a fixed-point object with value `v`, `embedded.numerictype` `T`, and embedded.`fimath` `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.

Note These properties are described in detail in "fi Object Properties" on page 9-2 in the Properties Reference.

- "Data Properties" on page 11-67
- "fimath Properties" on page 11-68
- "numerictype Properties" on page 11-69

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`,

uint8, uint16, and uint32 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 9-2.

fimath Properties

When you create a fi object, a fimath object is also automatically created as a property of the fi object.

- fimath — fimath object 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
- 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
- `RoundMode` — 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” on page 9-6.

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 numeric type attributes of a `fi` object

The following `numerictype` properties are, by transitivity, also properties of a `fi` object. The properties of the `numerictype` object listed below are not writable once the `fi` object has been created. 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
- `Slope` — Slope associated with a `fi` object
- `WordLength` — Word length of the stored integer value of a `fi` object in bits

These properties are described in detail in "numerictype Object Properties" on page 9-17.

Examples

Note For information about the display format of fi objects, refer to "Display Settings" on page 1-6.

Example 1

For example, the following creates a `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
                Signed: true
            WordLength: 8
        FractionLength: 3
```

Example 2

The value v can also be an array.

```
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
                Signed: true
            WordLength: 16
        FractionLength: 12
```

Example 3

If you omit the argument f, it is set automatically to the best precision possible.

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

a =

    3.1563

          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            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 = fi(pi, 1)
```

```
a =

    3.1416

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

Example 5

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

```
a = fi(pi, 'roundmode', 'floor', 'overflowmode', 'wrap')

a =

    3.1415

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

See Also

`fimath`, `fipref`, `numerictype`, `quantizer`, “fi Object Properties” on page 9-2

Purpose

Construct `fimath` object

Syntax

```
F = fimath
F = fimath(...'PropertyName',PropertyValue...)
```

Description

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

- `F = fimath` creates a default `fimath` object.
- `F = fimath(...'PropertyName',PropertyValue...)` allows you to set the attributes of a `fimath` object using property name/property value pairs.

The properties of the `fimath` object are listed below. These properties are described in detail in “fimath Object Properties” on page 9-6 in the Properties Reference.

- `CastBeforeSum` — Whether both operands are cast to the sum data type before addition
- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow-handling 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
- `RoundMode` — 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` — Word length, in bits, of the sum data type

Examples

Example 1

Type

```
F = fimath
```

to create a default `fimath` object.

```
F = fimath

F =

               RoundMode: nearest
            OverflowMode: saturate
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                 SumMode: FullPrecision
        MaxSumWordLength: 128
           CastBeforeSum: true
```

Example 2

You can set properties of `fimath` objects at the time of object creation by including properties after the arguments of the `fimath` constructor function. For example, to set the overflow mode to `saturate` and the rounding mode to `convergent`,

```
F = fimath('OverflowMode','saturate','RoundMode','convergent')

F =

                RoundMode: convergent
             OverflowMode: saturate
              ProductMode: FullPrecision
     MaxProductWordLength: 128
                  SumMode: FullPrecision
         MaxSumWordLength: 128
            CastBeforeSum: true
```

See Also

`fi`, `fipref`, `numerictype`, `quantizer`, “fimath Object Properties” on page 9-6

fipref

Purpose

Construct `fipref` object

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 9-14.

- `FimathDisplay` — Display options for the `fimath` attributes of a `fi` object
- `DataTypeOverride` — Data type override options
- `LoggingMode` — Logging options for operations performed on `fi` objects
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object
- `NumberDisplay` — Display options for the value of a `fi` object

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.

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 `AttributesDisplay` to `short`,

```
P =

         NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'
```

See Also

`fi`, `fimath`, `numerictype`, `quantizer`, `savefipref`, “fipref Object Properties” on page 9-14

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`.

Algorithm

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 `quantizer` object

Syntax

```
get(q,pn,pv)
value = get(q, 'propertyname')
structure = get(q)
```

Description

`get(q,pn,pv)` displays the property names and property values associated with `quantizer` object q.

`pn` is the name of a property of the object `obj`, and `pv` is the value associated with `pn`.

`value = get(q, 'propertyname')` returns the property value associated with the property named in the string `'propertyname'` for the `quantizer` object `q`. 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(q)` returns a structure containing the properties and states of `quantizer` object `q`.

See Also

`quantizer`, `set`

gplot

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`

hankel

Purpose Hankel matrix

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

Purpose

Hexadecimal representation of stored integer of `fi` object

Syntax

`hexadecimal(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`hexadecimal(a)` returns the stored integer of `fi` object `a` in hexadecimal format as a string.

Examples

The following code

```
a = fi([-1 1],1,8,7);
hex(a)
```

returns

```
80   7f
```

See Also

`bin`, `dec`, `int`, `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` objects 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
```

Algorithm

In general, an inner product grows `log2(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*sign(a')`, which is equal to `sum(abs(a))`.

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

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

inspect

Purpose Property Inspector

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

Purpose

Smallest built-in integer in which stored integer value of `fi` object will fit

Syntax

`int(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`int(a)` returns the smallest built-in integer of the data type in which the stored integer value of `fi` object `a` will fit.

The following table gives the return type of the `int` function.

Word Length	**Return Type for Signed fi**	**Return Type for Unsigned fi**
word length <= 8 bits	`int8`	`uint8`
8 bits < word length <= 16 bits	`int16`	`uint16`
16 bits < word length <= 32 bits	`int32`	`uint32`
32 < word length	`double`	`double`

Note When the word length is greater than 52 bits, the return value can have quantization error. For bit-true integer representation of very large word lengths, use `bin`, `oct`, `dec`, `hex`, or `sdec`.

See Also `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`

Purpose

Stored integer value of `fi` object as built-in `int8`

Syntax

```
int8(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`int8(a)` returns the stored integer value of `fi` object `a` as a built-in `int8`. If the stored integer word length is too big for an `int8`, or if the stored integer is unsigned, the returned value saturates to an `int8`.

See Also

`int`, `int16`, `int32`, `uint8`, `uint16`, `uint32`

Purpose Stored integer value of `fi` object as built-in `int16`

Syntax `int16(a)`

Description Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`int16(a)` returns the stored integer value of `fi` object `a` as a built-in `int16`. If the stored integer word length is too big for an `int16`, or if the stored integer is unsigned, the returned value saturates to an `int16`.

See Also `int`, `int8`, `int32`, `uint8`, `uint16`, `uint32`

Purpose

Stored integer value of `fi` object as built-in `int32`

Syntax

```
int32(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`int32(a)` returns the stored integer value of `fi` object `a` as a built-in `int32`. If the stored integer word length is too big for an `int32`, or if the stored integer is unsigned, the returned value saturates to an `int32`.

See Also

`int`, `int8`, `int16`, `uint8`, `uint16`, `uint32`

intmax

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

`intmin`, `lsb`, `stripscaling`

Purpose

Smallest stored integer value representable by `numerictype` of `fi` object

Syntax

```
x = intmin(a)
```

Description

`x = intmin(a)` returns the smallest stored integer value representable by the `numerictype` of `a`.

Examples

```
a = fi(pi, true, 16, 12);
x = intmin(a)

x =

      -32768

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

See Also

`intmax`, `lsb`, `stripscaling`

ipermute

Purpose Inverse permute dimensions of multidimensional array

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

Purpose Determine whether `fi` object is column vector

Syntax `iscolumn(a)`

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

See Also `isrow`

isempty

Purpose Determine whether array is empty

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

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

```
isequal(a,b,...)
isequal(F,G,...)
isequal(T,U,...)
isequal(q,r,...)
```

Description

`isequal(a,b,...)` returns `1` if all the `fi` object inputs have the same real-world value. Otherwise, the function returns `0`.

`isequal(F,G,...)` returns `1` if all the `fimath` object inputs have the same properties. Otherwise, the function returns `0`.

`isequal(T,U,...)` returns `1` if all the `numerictype` object inputs have the same properties. Otherwise, the function returns `0`.

`isequal(q,r,...)` returns `1` if all the `quantizer` object inputs have the same properties. Otherwise, the function returns `0`.

See Also

`eq`, `ispropequal`

isfi

Purpose

Determine whether variable is `fi` object

Syntax

`isfi(a)`

Description

`isfi(a)` returns `1` if `a` is a `fi` object, and `0` otherwise.

See Also

`fi`, `isfimath`, `isnumerictype`

Purpose

Determine whether variable is `fimath` object

Syntax

```
isfimath(F)
```

Description

`isfimath(F)` returns `1` if `F` is a `fimath` object, and `0` otherwise.

See Also

`fimath`, `isfi`, `isnumerictype`

isfinite

Purpose Determine whether array elements are finite

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

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.

Purpose Determine whether input is numeric array

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

isnumerictype

Purpose Determine whether variable is `numerictype` object

Syntax `isnumerictype(T)`

Description `isnumerictype(T)` returns `1` if `a` is a `numerictype` object, and `0` otherwise.

See Also `isfi`, `isfimath`, `numerictype`

Purpose Determine whether input is MATLAB OOPS object

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

ispropequal

Purpose

Determine whether properties of two `fi` objects are equal

Syntax

```
ispropequal(a,b,...)
```

Description

`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`

Purpose Determine whether array elements are real

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

Purpose

Determine whether `fi` object is row vector

Syntax

```
isrow(a)
```

Description

`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.

Purpose

Determine whether `fi` object is signed

Syntax

```
issigned(a)
```

Description

`issigned(a)` returns `1` if the `fi` object `a` is signed, and `0` if it is unsigned.

Purpose Determine whether input is vector

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

Purpose

Determine whether real-world value of `fi` object is less than or equal to another

Syntax

```
c = le(a,b)
a <= b
```

Description

`c = le(a,b)` is called for the syntax `'a <= b'` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a <= b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to `1` where the relation is true, and `0` where the relation is false.

See Also

`eq`, `ge`, `gt`, `lt`, `ne`

Purpose Vector length

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

line

Purpose Create line object

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

Purpose Convert numeric values to logical

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

loglog

Purpose Create log-log scale plot

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

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=lowerbound(a)` and `U=upperbound(a)`, then `[L,U]=range(a)`.

See Also `range`, `upperbound`

Purpose Scaling of least significant bit of `fi` object

Syntax `lsb(a)`

Description `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.

See Also `eps`

Purpose

Determine whether real-world value of one `fi` object is less than another

Syntax

```
c = lt(a,b)
a < b
```

Description

`c = lt(a,b)` is called for the syntax `'a < b'` when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a < b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to `1` where the relation is true, and `0` where the relation is false.

See Also

`eq`, `ge`, `gt`, `le`, `ne`

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

`min`

Purpose

Largest real-world value of `fi` object or maximum value of `quantizer` object before quantization

Syntax

```
maxlog(a)
maxlog(q)
```

Description

`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` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`maxlog(q)` is the maximum value before quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the maximum value encountered over successive calls to `quantize` and is reset with `resetlog(q)`. `maxlog(q)` is equivalent to `get(q,'maxlog')` and `q.maxlog`.

Examples

```
P = fipref('LoggingMode','on');
x = fi([-1.5 eps 0.5], true, 16, 15);
x(1) = 3.0;
maxlog(x)

ans =

     3
```

See Also

`fipref`, `minlog`, `noverflows`, `nunderflows`, `resetlog`

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`

Purpose

Smallest real-world value of `fi` object or minimum value of `quantizer` object before quantization

Syntax

```
minlog(a)
minlog(q)
```

Description

`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` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`minlog(q)` is the minimum value before quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the minimum value encountered over successive calls to `quantize` and is reset with `resetlog(q)`. `minlog(q)` is equivalent to `get(q,'minlog')` and `q.minlog`.

Examples

```
P = fipref('LoggingMode','on');
x = fi([-1.5 eps 0.5], true, 16, 15);
x(1) = 3.0;
minlog(x)

ans =

     -1.5
```

See Also

`fipref`, `maxlog`, `noverflows`, `nunderflows`, `resetlog`

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 (a 1-by-1 matrix). A scalar can be subtracted from anything.

`minus` does not support `fi` objects of data type `Boolean`.

See Also

`mtimes`, `plus`, `times`, `uminus`

Purpose

Multiply two objects using `fimath` object

Syntax

```
c = F.mpy(a,b)
```

Description

`c = F.mpy(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` objects of `a` and `b` are different.

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

If either `a` or `b` is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length.

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 = F.mpy(a, b)

c =

    8.5397


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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 40
 ProductFractionLength: 30
```

```
                  SumMode: FullPrecision
         MaxSumWordLength: 128
            CastBeforeSum: true
```

Algorithm

`c = F.mpy(a,b)` is equivalent to

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

except that the `fimath` properties of `a` and `b` are not modified when you use the functional form.

See Also

`add`, `divide`, `fi`, `fimath`, `numerictype`, `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`. Any scalar (a 1-by-1 matrix) can multiply anything. 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`.

See Also

`plus`, `minus`, `times`, `uminus`

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`

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`

not

Purpose Find logical NOT of array or scalar input

Description Refer to the MATLAB `not` reference page for more information.

Purpose

Number of overflows

Syntax

```
noverflows(a)
noverflows(q)
```

Description

`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.

`noverflows(q)` returns the accumulated number of overflows resulting from quantization operations performed by a `quantizer` object `q`.

See Also

`maxlog`, `minlog`, `nunderflows`, `resetlog`

num2bin

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`

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

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
```

Algorithm

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`

numberofelements

Purpose

Number of data elements in `fi` array

Syntax

```
numberofelements(a)
```

Description

`numberofelements(a)` returns the number of data elements in a `fi` array. `numberofelements(a) == prod(size(a))`.

Note that `fi` is a MATLAB object, and therefore `numel(a)` returns `1` when `a` is a `fi` object. Refer to the information about classes in the MATLAB `numel` reference page.

See Also

`max`, `min`, `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`, signedness `s`, and 16-bit word length.
- `T = numerictype(s,w)` creates a `numerictype` object with `Fixed-point:  unspecified scaling`, signedness `s`, and word length `w`.
- `T = numerictype(s,w,f)` creates a `numerictype` object with `Fixed-point:  binary point scaling`, signedness `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`, signedness `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`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

- `T = numerictype(property1,value1, ...)` allows you to set properties for a `numerictype` object using property name/property value pairs.
- `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" on page 9-17.

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

Examples

Example 1

Type

```
T = numerictype
```

to create a default `numerictype` object.

```
T =
              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15
```

Example 2

The following 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
                Signed: true
            WordLength: 32
        FractionLength: 30
```

Example 3

If you omit the argument `f`, the scaling is unspecified.

```
T = numerictype(1, 32)

T =

          DataTypeMode: Fixed-point: unspecified scaling
                Signed: true
            WordLength: 32
```

Example 4

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
                Signed: true
            WordLength: 16
```

Example 5

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', 'WordLength', 32, 'Slope', ...
2^-2, 'Bias', 4)

T =

          DataTypeMode: Fixed-point: slope and bias scaling
                Signed: true
            WordLength: 32
                 Slope: 0.25
                  Bias: 4
```

See Also

`fi`, `fimath`, `fipref`, `quantizer`, “numerictype Object Properties” on page 9-17

Purpose

Number of underflows

Syntax

```
nunderflows(a)
nunderflows(q)
```

Description

`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.

`nunderflows(q)` returns the accumulated number of underflows resulting from quantization operations performed by a `quantizer` object `q`.

See Also

`maxlog`, `minlog`, `noverflows`, `resetlog`

Purpose

Octal representation of stored integer of `fi` object

Syntax

`oct(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`oct(a)` returns the stored integer of `fi` object `a` in octal format as a string.

Examples

The following code

```
a = fi([-1 1],1,8,7);
oct(a)
```

returns

```
200   177
```

See Also

`bin`, `dec`, `hex`, `int`

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.

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 (a 1-by-1 matrix). A scalar can be added to anything.

`plus` does not support `fi` objects of data type `Boolean`.

See Also

`minus`, `mtimes`, `times`, `uminus`

polar

Purpose Plot polar coordinates

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

Purpose

Multiply by a power of 2

Syntax

```
b = pow2(a, K)
```

Description

`b = pow2(a, K)` returns

$$b = a \times 2^K$$

where `K` is an integer and `a` and `b` are fi objects. If `K` is a non-integer, it will be rounded to `floor` before the calculation is performed. The scaling of `a` must be equivalent to binary point-only scaling; in other words, it must have a fractional slope of 1 and a bias of 0.

The syntax `b = pow2(a)` is not supported when `a` is a `fi` object.

`a` can be real or complex. If `a` is complex, `pow2` operates on both the real and complex portions of `a`.

`pow2` does not support `fi` objects of data type `Boolean`.

Examples

The following example shows the use of `pow2` with a complex `fi` object:

```
format long g
P = fipref('NumericTypeDisplay', 'short', 'FimathDisplay',...
'none');
a = fi(57 - 2i, 1, 16, 8)

a =

                        57 -                       2i
      s16,8

pow2(a, 2)

ans =

               127.99609375 -                      8i
      s16,8
```

See Also `bitshift`

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. Nonnumeric elements or fields of `x` are left unchanged and `quantize` does not issue warnings for nonnumeric values.

`[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

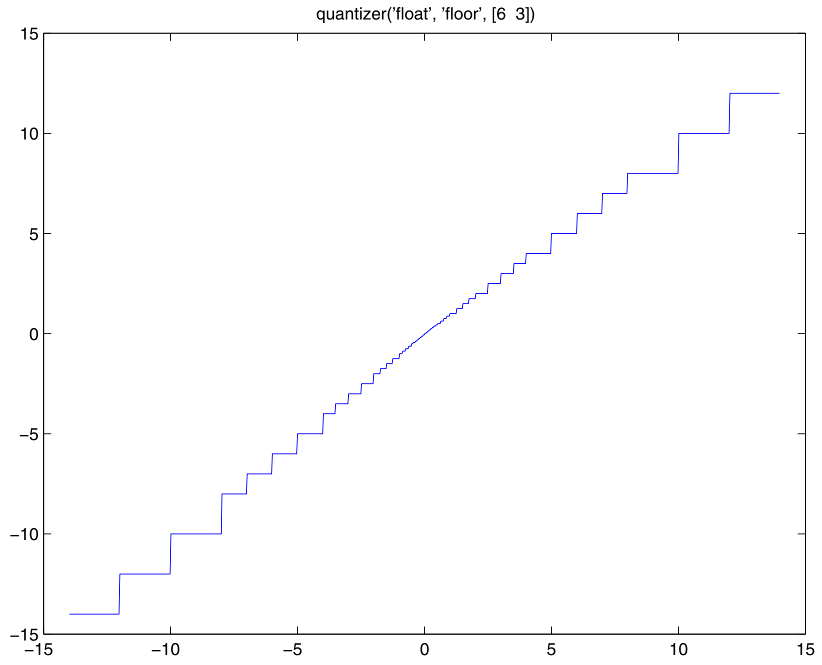
The following examples demonstrate using `quantize` to quantize data.

Example 1 - Custom Precision Floating-Point

The code listed here produces the plot shown in the following figure.

```
u=linspace(-15,15,1000);
q=quantizer([6 3],'float');
range(q)
```

```
ans =

   -14    14
y=quantize(q,u);
plot(u,y);title(tostring(q))
Warning: 68 overflows.
```

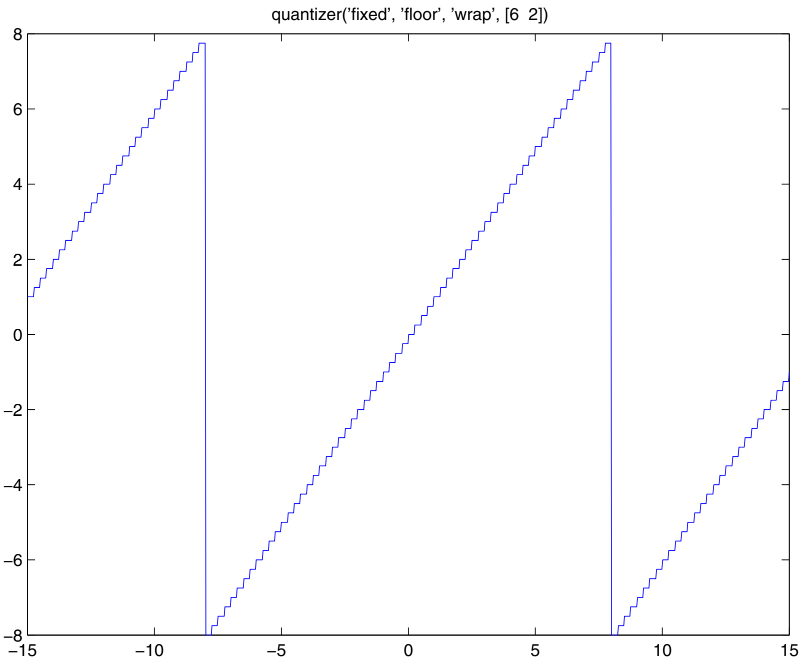


Example 2 - Fixed-Point

The code listed here produces the plot shown in the following figure.

```
u=linspace(-15,15,1000);
q=quantizer([6 2],'wrap');
```

```
range(q)

ans =

   -8.0000    7.7500
y=quantize(q,u);
plot(u,y);title(tostring(q))
Warning: 468 overflows.
```



See Also

`quantizer`, `set`

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.

`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 9-21.

Property Name	Property Value	Description
`mode`	`'double'`	Double-precision mode. Override all other parameters.
	`'float'`	Custom-precision floating-point mode.

Property Name	Property Value	Description
	`'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'`	Convergent rounding.
	`'fix'`	Round toward zero.
	`'floor'`	Round toward negative infinity.
	`'nearest'`	Round toward nearest.
`overflowmode (fixed-point only)`	`'saturate'`	Saturate on overflow.
	`'wrap'`	Wrap on overflow.
`format`	`[wordlength exponentlength]`	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', 'ceil', 'saturate', [5 4])
```

Using a structure struct to set quantizer object properties,

```
struct.mode = 'fixed';
struct.roundmode = 'ceil';
struct.overflowmode = 'saturate';
struct.format = [5 4];
q = quantizer(struct);
```

Using property name and property value cell arrays `pn` and `pv` to set `quantizer` object properties,

```
pn = {'mode',  'roundmode', 'overflowmode', 'format'};
pv = {'fixed', 'ceil', 'saturate', [5 4]};
q = quantizer(pn, pv)
```

Using property name/property value pairs to configure a `quantizer` object,

```
q = quantizer( 'mode', fixed','roundmode','ceil',...
'overflowmode', 'saturate', 'format', [5 4]);
```

See Also

`fi`, `fimath`, `fipref`, `numerictype`, `quantize`, `set`, “quantizer Object Properties” on page 9-21

Purpose Create quiver or velocity plot

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

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

```
-[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 `rand` in most respects, including the generator used, but it does not support the `'state'` and `'seed'` options available in `rand`.

Examples

```
q=quantizer([4 3]);
rand('state',0)
randquant(q,3)

ans =

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

See Also

`quantizer`, `range`, `realmax`

Purpose

Numerical range of `fi` or `quantizer` object

Syntax

```
range(a)
[min, max]= range(a)
r = range(q)
[min, max] = 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, max]= 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 = quantize(q,x)` returns y in the range $a \le y \le b$.

`[min, max] = 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,max] = range(q)

min =

    -2


max =

    1.7500
```

Algorithm

If `q` is a floating-point `quantizer` object, a = `-realmax`(q), b = `realmax`(q).

If `q` is a signed fixed-point `quantizer` object (`datamode = 'fixed'`),

$$a = -\mathrm{realmax}(q) - \mathrm{eps}(q) = \frac{-2^{w-1}}{2^f}$$

$$b = \mathrm{realmax}(q) = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point `quantizer` object (`datamode = 'ufixed'`),

$$a = 0$$

$$b = \mathrm{realmax}(q) = \frac{2^{w} - 1}{2^f}$$

See `realmax` for more information.

See Also

`exponentmin`, `fractionlength`, `max`, `min`, `realmax`, `realmin`

real

Purpose Real part of complex number

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

Purpose

Largest positive fixed-point value or quantized number

Syntax

```
realmax(a)
realmax(q)
```

Description

`realmax(a)` is the largest real-world value that can be represented in the data type of `fi` object `a`. Anything larger overflows.

`realmax(q)` is the largest quantized number that can be represented where `q` is a `quantizer` object. Anything larger overflows.

Examples

```
q = quantizer('float',[6 3]);
x = realmax(q)

x =

    14
```

Algorithm

If `q` is a floating-point `quantizer` object, the largest positive number, x, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point `quantizer` object, the largest positive number, x, is

$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point `quantizer` object (`datamode = 'ufixed'`), the largest positive number, x, is

$$x = \frac{2^w - 1}{2^f}$$

See Also `quantizer`, `realmin`, `exponentmin`, `fractionlength`

Purpose

Smallest positive normalized fixed-point value or quantized number

Syntax

```
realmin(a)
realmin(q)
```

Description

`realmin(a)` is the smallest real-world value that can be represented in the data type of `fi` object `a`. Anything smaller underflows.

`realmin(q)` is the smallest positive normal quantized number where `q` is a `quantizer` object. Anything smaller than `x` underflows or is an IEEE “denormal” number.

Examples

```
  q = quantizer('float',[6 3]);
  realmin(q)

  ans =

      0.2500
```

Algorithm

If `q` is a floating-point `quantizer` object, $x = 2^{E_{min}}$ where

$E_{min} = \text{exponentmin}(q)$ is the minimum exponent.

If `q` is a signed or unsigned fixed-point `quantizer` object, $x = 2^{-f} = \varepsilon$ where f is the fraction length.

See Also

`exponentmin`, `fractionlength`, `realmax`

Purpose Replicate and tile array

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

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 = a.int;
stored_integer_b = b.int;
isequal(stored_integer_a, stored_integer_b)

ans =

1
```

See Also `fi`

Purpose

Reset objects to initial conditions

Syntax

```
reset(obj)
```

Description

`reset(obj)` resets `fi`, `fimath`, `fipref`, or `quantizer` object `obj` to its initial conditions.

See Also

`resetlog`

resetlog

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`

Purpose Reshape array

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

rgbplot

Purpose Plot colormap

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

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.

Purpose

Round input data using `quantizer` object without checking for overflow

Syntax

```
round(q,x)
```

Description

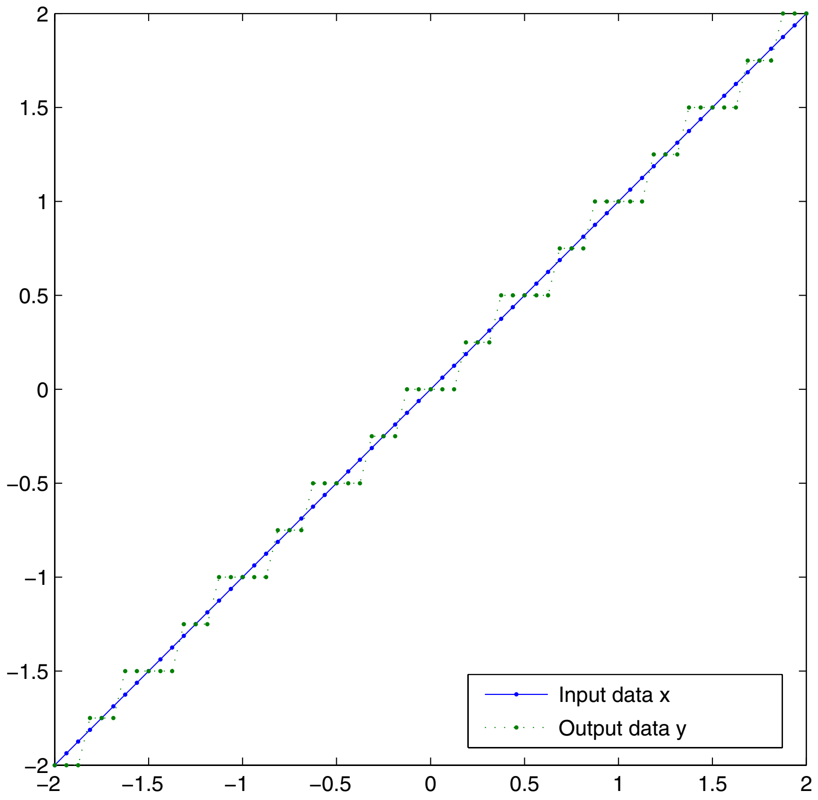
`round(q,x)` uses the `RoundMode` and `FractionLength` settings of `q` to round the numeric data `x`, but does not check for overflows during the operation. Compare to `quantize`.

Examples

Create a `quantizer` object and use it to quantize input data. The `quantizer` object applies its properties to the input data to return quantized output.

```
q = quantizer('fixed', 'convergent', 'wrap', [3 2]);
x = (-2:eps(q)/4:2)';
y = round(q,x);
plot(x,[x,y],'.-'); axis square;
```

Applying `quantizer` object `q` to the data results in the staircase shape output plot shown here. Where the input data is linear, output `y` shows distinct quantization levels.



See Also `quantize`, `quantizer`

Purpose

Save `fi` preferences for next MATLAB session

Syntax

```
savefipref
```

Description

`savefipref` saves the settings of the current `fipref` object for the next MATLAB session.

See Also

`fipref`

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.

Purpose

Signed decimal representation of stored integer of `fi` object

Syntax

`sdec(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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`, `int`, `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 that you can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`q.PropertyName = Value` uses dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of `quantizer` object `q`.

`s = set(q)` returns a structure containing the possible values for the properties of `quantizer` object `q`.

See Also

`get`

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,

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

`single(a)` returns the real-world value of a `fi` object in single-precision floating point.

See Also `double`

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.

spy

Purpose Visualize sparsity pattern

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

Purpose Remove singleton dimensions

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

Purpose Create stairstep graph

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

Purpose

Plot discrete sequence data

Description

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

stem3

Purpose Plot 3-D discrete sequence data

Description Refer to the MATLAB `stem3` reference page for more information.

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.

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 zero bias and the same word length and sign as `a`.

Purpose

Subtract two objects using `fimath` object

Syntax

```
c = F.sub(a,b)
```

Description

`c = F.sub(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` objects of `a` and `b` are different.

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

If either `a` or `b` is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length.

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 = F.sub(a, b)

c =

    0.4233


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

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

```
              SumMode: SpecifyPrecision
        SumWordLength: 32
    SumFractionLength: 16
        CastBeforeSum: true
```

Algorithm

`c = F.sub(a,b)` is equivalent to

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

except that the `fimath` properties of `a` and `b` are not modified when you use the functional form.

See Also

`add`, `divide`, `fi`, `fimath`, `mpy`, `numerictype`

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.

`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 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 `':'`.

Examples

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`, and `a` assumes the value, `numerictype` object, and `fimath` object of `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
                Signed: true
            WordLength: 8
        FractionLength: 7

b = fi(pi/4, 1, 16, 15)

b =

    0.7854

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

```
a(:) = b

a =

    0.7891

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

In this kind of assignment operation, the `fimath` objects of `a` and `b` can be different. A common use for this is when casting the result of an accumulation to an output data type, where the two have different rounding and overflow modes. Another common use is in a series of multiply/accumulate operations. For example,

```
for k = 1:n
 acc(1) = acc + b * x(k)
end
```

See Also

`subsref`

subsref

Purpose Subscripted reference

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

Purpose

Sum of array elements

Syntax

```
b = sum(a)
b = sum(a, dim)
```

Description

`b = 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 nonsingleton dimension as vectors, returning an array of row vectors.

`b = sum(a, dim)` sums along the dimension `dim` of `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(length(a)))`.

`sum` does not support `fi` objects of data type `Boolean`.

See Also

`add`, `divide`, `fi`, `fimath`, `mpy`, `numerictype`, `sub`

surf

Purpose Create 3-D shaded surface plot

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

Purpose

Create 3-D shaded surface plot with contour plot

Description

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

surfl

Purpose Create surface plot with colormap-based lighting

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

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.

Purpose

Element-by-element multiplication of `fi` objects

Syntax

```
times(a,b)
```

Description

`times(a,b)` is called for the syntax `'a .* b'` when `a` or `b` is an object.

`a.*b` denotes element-by-element multiplication. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be multiplied into anything.

`times` does not support `fi` objects of data type `Boolean`.

See Also

`plus`, `minus`, `mtimes`, `uminus`

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 `numerictype` and `fimath` objects of the leftmost input that is a `fi` object are applied to the output `t`.

Purpose Convert `quantizer` object to string

Syntax

```
s = tostring(q)
```

Description `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`.

See Also `quantizer`

Purpose Transpose operation

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

Purpose

Plot picture of tree

Description

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

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.

Purpose Upper triangular part of matrix

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

Purpose

Stored integer value of `fi` object as built-in `uint8`

Syntax

`uint8(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`uint8(a)` returns the stored integer value of `fi` object `a` as a built-in `uint8`. If the stored integer word length is too big for a `uint8`, or if the stored integer is signed, the returned value saturates to a `uint8`.

See Also

`int`, `int8`, `int16`, `int32`, `uint16`, `uint32`

Purpose

Stored integer value of `fi` object as built-in `uint16`

Syntax

```
uint16(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`uint16(a)` returns the stored integer value of `fi` object `a` as a built-in `uint16`. If the stored integer word length is too big for a `uint16`, or if the stored integer is signed, the returned value saturates to a `uint16`.

See Also

`int`, `int8`, `int16`, `int32`, `uint8`, `uint32`

Purpose

Stored integer value of `fi` object as built-in `uint32`

Syntax

`uint32(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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

`uint32(a)` returns the stored integer value of `fi` object `a` as a built-in `uint32`. If the stored integer word length is too big for a `uint32`, or if the stored integer is signed, the returned value saturates to a `uint32`.

See Also

`int`, `int8`, `int16`, `int32`, `uint8`, `uint16`

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`.

See Also `plus`, `minus`, `mtimes`, `times`

Purpose Unary plus

Description Refer to the MATLAB `arithmetic operators` 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 `lowerbound`, `range`

Purpose

Vertically concatenate multiple`fi` objects

Syntax

```
c = vertcat(a,b,...)
[a; b; ...]
[a;b]
```

Description

`c = vertcat(a,b,...)` is called for the syntax `[a; b; ...]` when any of `a`, `b`, `...` , is a `fi` object.

`[a;b]` is the vertical concatenation of matrices `a` and `b`. `a` and `b` must have the same number of columns. Any number of matrices can be concatenated within one pair of brackets. N-D arrays are vertically concatenated along the first dimension. The remaining dimensions must match.

Horizontal and vertical concatenation can be combined, as in `[1 2;3 4]`.

`[a b; c]` is allowed if the number of rows of `a` equals the number of rows of `b`, and if the number of columns of `a` plus the number of columns of `b` equals the number of columns of `c`.

The matrices in a concatenation expression can themselves be formed via a concatenation, as in `[a b;[c d]]`.

Note The `fimath` and `numerictype` objects of a concatenated matrix of `fi` objects `c` are taken from the leftmost `fi` object in the list `(a,b,...)`.

See Also

`horzcat`

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.

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

Set or query y-axis limits

Description

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

Purpose

e Set or query z-axis limits

Description

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

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe products from The MathWorks 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

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

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 Toolbox.

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, word length

data type override

Parameter in the Fixed-Point Settings interface that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

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

2. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$slope = fractional\,slope \times 2^{exponent}$$

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$exponent = -1 \times fraction\ length$$

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

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

where the slope can be expressed as

$$slope = fractional\,slope \times 2^{exponent}$$

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

$$\textit{real-world value} = \textit{mantissa} \times 2^{\textit{exponent}}$$

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

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

where the slope can be expressed as

$$slope = fractional\,slope \times 2^{exponent}$$

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

See also bias, exponent, fixed-point representation, integer, slope

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

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

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

or

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

where the slope can be expressed as

$$slope = fractionalslope \times 2^{exponent}$$

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

$$weight\ of\ LSB = 2^{-fraction\ length}$$

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

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

See also exponent, floating-point representation

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 Toolbox.

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, precision

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

See also integer

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 Toolbox 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, 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, 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

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

where the slope can be expressed as

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

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:

$$\textit{real-world value} = \textit{integer}$$

In [Slope Bias] representation, fixed-point numbers can be represented as

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

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$\textit{real-world value} = \textit{integer} \times 2^{-\textit{fraction length}} = \textit{integer} \times 2^{0}$$

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

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 Toolbox.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation

Index