Fixed-Point Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 1

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

Fixed-Point Toolbox User's Guide

Printing History: June 2004 First printing New for Version 1.0 (Release 14)

Contents

1 Getting Started

What Is the Fixed-Point Toolbox? **1-2**
Features **1-2**

Getting Help **1-3**
Getting Help in this Document **1-3**
Getting Help at the MATLAB Command Line **1-3**

Display Settings **1-5**

Demos **1-7**

2 Fixed-Point Concepts

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

Scaling **2-4**

Precision and Range **2-5**
Range **2-5**
Precision **2-6**

Arithmetic Operations **2-8**
Modulo Arithmetic **2-8**
Two’s Complement **2-9**
Addition and Subtraction **2-10**
Multiplication **2-11**
Casts **2-17**

fi Objects Compared to C Integer Data Types **2-20**
Integer Data Types **2-20**
Unary Conversions **2-22**
Binary Conversions **2-23**
Overflow Handling **2-25**

3 Working with fi Objects

Constructing fi Objects **3-2**
Examples of Constructing fi Objects **3-3**

fi Object Properties **3-9**
Data Properties **3-9**
fimath Properties **3-9**
numerictype Properties **3-10**
Setting Fixed-Point Properties at Object Creation **3-11**
Using Direct Property Referencing with fi **3-11**

fi Object Functions **3-13**

4 Working with fimath Objects

Constructing fimath Objects **4-2**

fimath Object Properties **4-4**
Setting fimath Properties at Object Creation **4-4**
Using Direct Property Referencing with fimath **4-5**

Using fimath Objects to Perform Fixed-Point Arithmetic **4-6**

Using fimath to Share Arithmetic Rules **4-8**

fimath Object Functions **4-10**

5 Working with fipref Objects

Constructing fipref Objects **5-2**

fipref Object Properties **5-3**
Setting fipref Properties at Object Creation **5-3**
Using Direct Property Referencing with fipref **5-3**

Using fipref Objects to Set Display Preferences **5-5**

fipref Object Functions **5-7**

6 Working with numerictype Objects

Constructing numerictype Objects **6-2**

numerictype Object Properties **6-4**
Setting numerictype Properties at Object Creation **6-4**
Using Direct Property Referencing with numerictype objects . **6-5**

The numerictype Structure **6-6**
Properties That Affect the Slope **6-7**
Stored Integer Value and Real World Value **6-7**

Using numerictype Objects to Share Data Type and Scaling Settings **6-8**

numerictype Object Functions **6-11**

7 Working with quantizer Objects

Constructing quantizer Objects **7-2**

quantizer Object Properties **7-4**
Settable quantizer Object Properties **7-4**
Read-Only quantizer Object Properties **7-5**

Quantizing Data with quantizer Objects **7-6**

Transformations for Quantized Data **7-8**

quantizer Object Functions **7-9**

8 Interoperability with Other Products

Using fi Objects with Simulink **8-2**
Reading Fixed-Point Data from the Workspace **8-2**
Writing Fixed-Point Data to the Workspace **8-2**
Logging Fixed-Point Signals **8-5**
Accessing Fixed-Point Block Data During Simulation **8-6**

Using fi Objects with Signal Processing Blockset **8-7**
Reading Fixed-Point Signals from the Workspace **8-7**
Writing Fixed-Point Signals to the Workspace **8-7**

Using fi Objects with Filter Design Toolbox **8-11**

9 Property Reference

fi Object Properties **9-2**
bin **9-2**
data **9-2**
dec **9-2**
double **9-2**
fimath **9-2**
hex **9-3**
int **9-3**
NumericType **9-3**
oct **9-4**

fimath Object Properties **9-5**
CastBeforeSum **9-5**
MaxProductWordLength **9-5**
MaxSumWordLength **9-5**
OverflowMode **9-5**
ProductFractionLength **9-5**
ProductMode **9-6**
ProductWordLength **9-7**
RoundMode **9-7**
SumFractionLength **9-7**
SumMode **9-7**
SumWordLength **9-9**

fipref Object Properties **9-10**
FimathDisplay **9-10**
NumericTypeDisplay **9-10**
NumberDisplay **9-10**

numerictype Object Properties **9-11**
Bias **9-11**
DataType **9-11**
DataTypeMode **9-11**
FixedExponent **9-12**
FractionLength **9-12**
Scaling **9-12**
Signed **9-13**
Slope **9-13**
SlopeAdjustmentFactor **9-13**
WordLength **9-13**

quantizer Object Properties **9-14**
DataMode **9-14**
Format **9-14**
Max **9-15**
Min **9-15**
NOperations **9-16**
NOverflows **9-16**
NUnderflows **9-16**
OverflowMode **9-16**
RoundMode **9-17**

10 Function Reference

Functions — Categorical List **10-2**
Bitwise Functions **10-2**
Constructor and Property Functions **10-2**
Data Manipulation Functions **10-3**
Data Type Functions **10-4**
Data Quantizing Functions **10-5**
Math Operation Functions **10-5**
Matrix Manipulation Functions **10-6**
Numerical Type Functions **10-6**
One-Dimensional Plotting Functions **10-6**
Radix Conversion Functions **10-6**

Relational Operator Functions **10-7**
Statistics Functions **10-7**
Subscripted Assignment and Reference Functions **10-7**

fi Object Functions 10-8

fimath Object Functions 10-9

fipref Object Functions 10-10

numerictype Object Functions 10-11

quantizer Object Functions 10-12

Functions — Alphabetical List 9-13

Glossary

Index

1

Getting Started

What Is the Fixed-Point Toolbox? (p. 1-2)	Describes the Fixed-Point Toolbox and its major features
Getting Help (p. 1-3)	Tells you how to get help on Fixed-Point Toolbox objects, properties, and functions
Display Settings (p. 1-5)	Describes the `fi` object display settings used in the code examples in this User's Guide
Demos (p. 1-7)	Lists the Fixed-Point Toolbox Demos

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 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 with binary point-only signals
  - Arithmetic operators `+`, `-`, `*`, `.*`
  - Division using the `divide` function
- Arbitrary word length up to `intmax('uint16')`
- Relational, logical, and bitwise operators
- Data visualization via the `plot` function
- Statistics functions such as `max` and `min`
- Conversions between binary, hex, double, and built-in integers
- Interoperability with Simulink®, Signal Processing Blockset, 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" in the online or PDF documentation.

To get in-depth information about the functions of these objects, refer to Chapter 10, "Function Reference" in the online or PDF documentation.

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'

a = fi(pi)

a =

    3.1416


              DataType: Fixed
               Scaling: BinaryPoint
                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'

a = fi(pi)

a =

    3.1416


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 13

             RoundMode: round
          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**. The Fixed-Point Toolbox includes the following demos:

- `fi` Basics — Demonstrates the basic use of the fixed-point object `fi`
- Fixed-Point Algorithm Development — Shows the development and verification of a simple 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
- Number Circle — Illustrates the definitions of unsigned and signed two's complement integer and fixed-point numbers
- Quantization Error — Demonstrates the statistics of the error when signals are quantized using various rounding methods
- Analysis of a Fixed-Point State-Space System with Limit Cycles — Demonstrates a limit cycle detection routine applied to a state-space system

2

Fixed-Point Concepts

Fixed-Point Data Types (p. 2-2)	Defines fixed-point data types
Scaling (p. 2-4)	Discusses the types of scaling used in the Fixed-Point Toolbox; binary point-only and [Slope Bias]
Precision and Range (p. 2-5)	Discusses the concepts of limited precision and range, and discusses overflow handling and rounding methods
Arithmetic Operations (p. 2-8)	Introduces the concepts behind arithmetic operations in the Fixed-Point Toolbox
fi Objects Compared to C Integer Data Types (p. 2-20)	Compares ANSI C integer data type ranges, conversions, and exception handling with those of `fi` objects

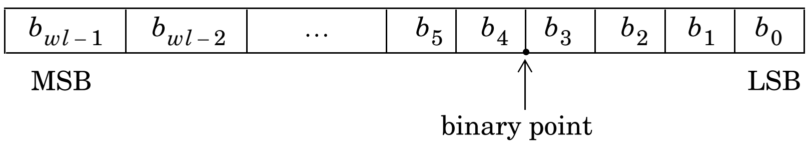
Fixed-Point Data Types

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

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

where the slope can be expressed as

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

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In 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:

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

or

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

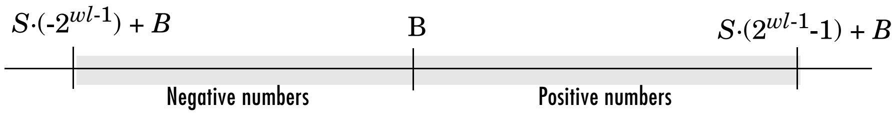
The Fixed-Point Toolbox supports both binary point-only scaling and [Slope Bias] scaling.

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 will occur.

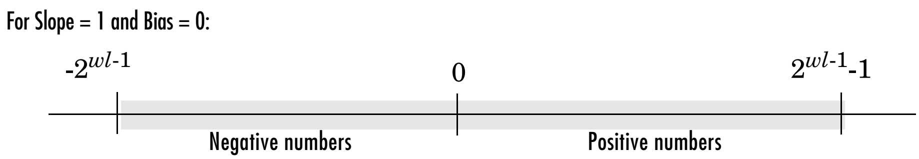
Range

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 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 2-8 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`.

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:

- `floor`, which is equivalent to truncation, rounds to the closest representable number in the direction of negative infinity.
- `ceil` rounds to the closest representable number in the direction of positive infinity.
- `fix` rounds to the closest representable integer in the direction of zero.
- `convergent` rounds to the closest representable integer. In the case of a tie, it rounds to the nearest even integer.
- `round` 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.

Arithmetic Operations

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

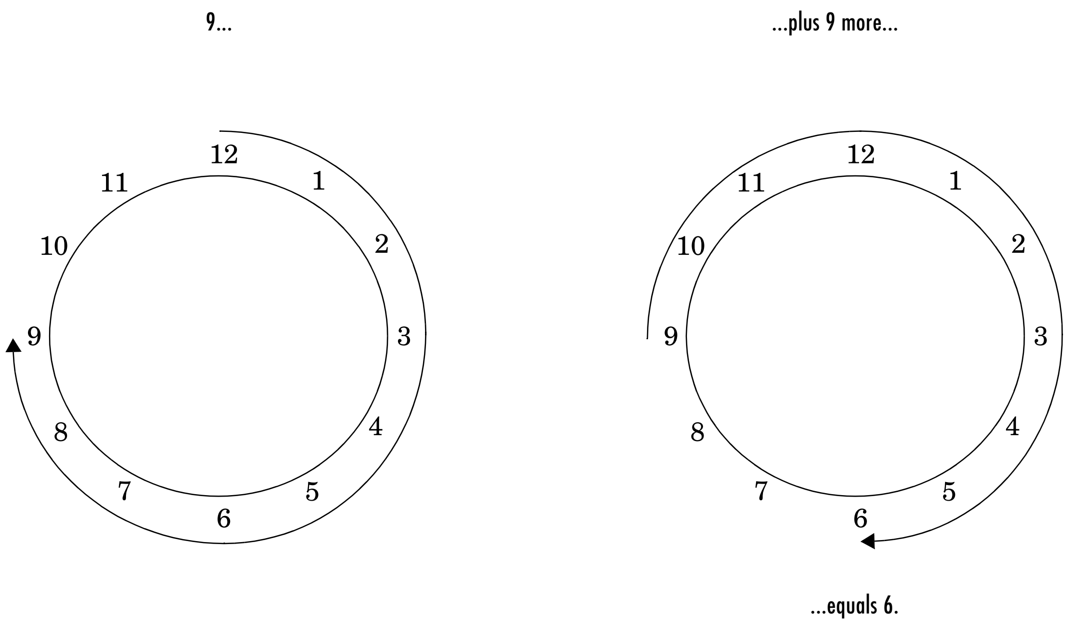
- "Modulo Arithmetic" on page 2-8
- "Two's Complement" on page 2-9
- "Addition and Subtraction" on page 2-10
- "Multiplication" on page 2-11
- "Casts" on page 2-17

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,

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

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

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

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

2 Add a 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:

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

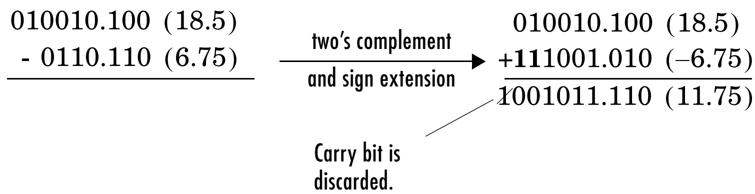
Addition and Subtraction

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

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

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

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



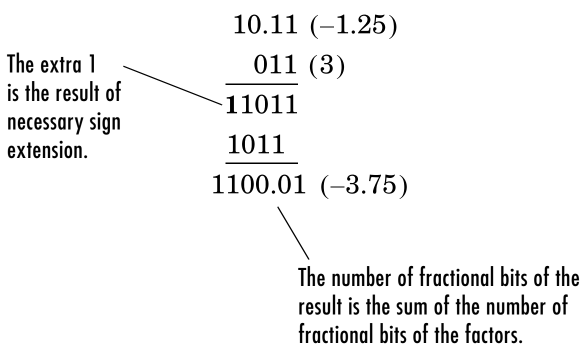
The default `fimath` object has a value of `1` (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

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.

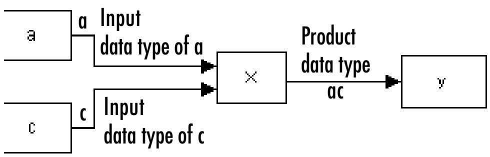
For example, consider the multiplication of 10.11 (-1.25) with 011 (3):



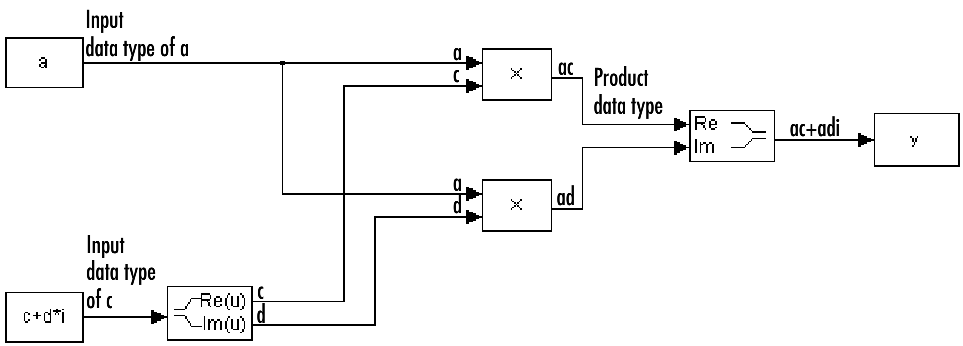
Multiplication Data Types

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.

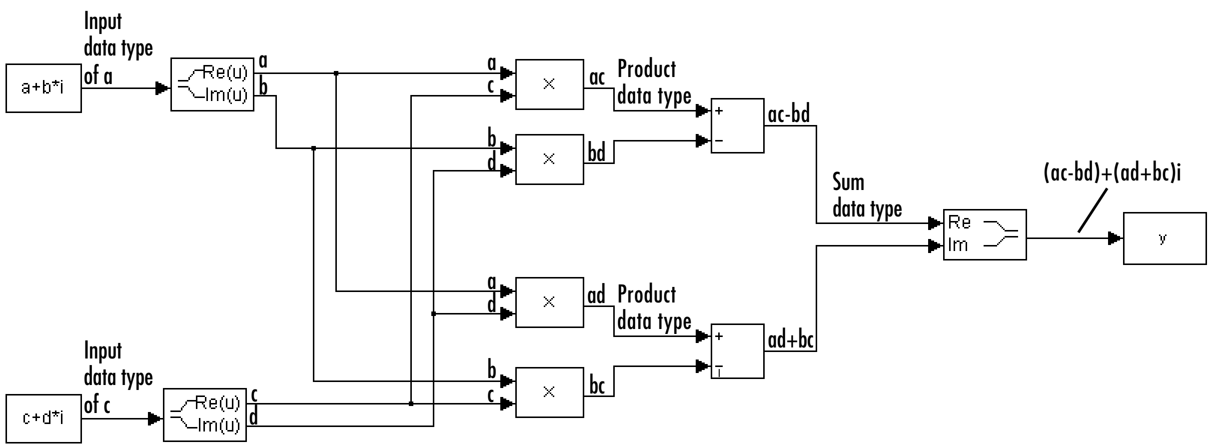
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in the Fixed-Point Toolbox. The output of this multiplication is in the product data type, which is governed by the `fimath` `ProductMode` property:



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


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 24
        FractionLength: 20


y = fi(10, T2, F)

y =

    10


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 10


z = x*y

z =

    50


              DataType: Fixed
               Scaling: BinaryPoint
                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


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 24
       FractionLength: 20


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

y =

  10.0000 + 2.0000i


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 16
       FractionLength: 10


z = x*y

z =

  50.0000 +10.0000i
```

```
            DataType: Fixed
             Scaling: BinaryPoint
              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:

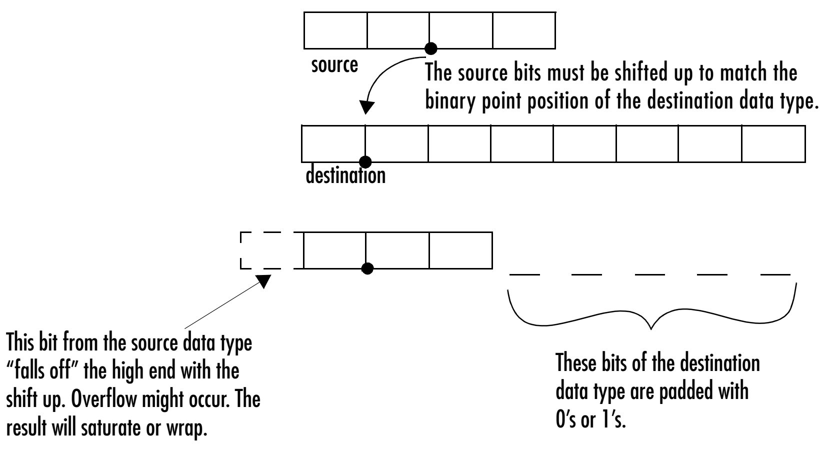
```
x = fi(5+6i,T1,F)

x =

   5.0000 + 6.0000i


            DataType: Fixed
             Scaling: BinaryPoint
              Signed: true
          WordLength: 24
      FractionLength: 20


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

y =

  10.0000 + 2.0000i


            DataType: Fixed
             Scaling: BinaryPoint
              Signed: true
          WordLength: 16
      FractionLength: 10


z = x*y
```

```
z =

  38.0000 +70.0000i


            DataType: Fixed
             Scaling: BinaryPoint
              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:

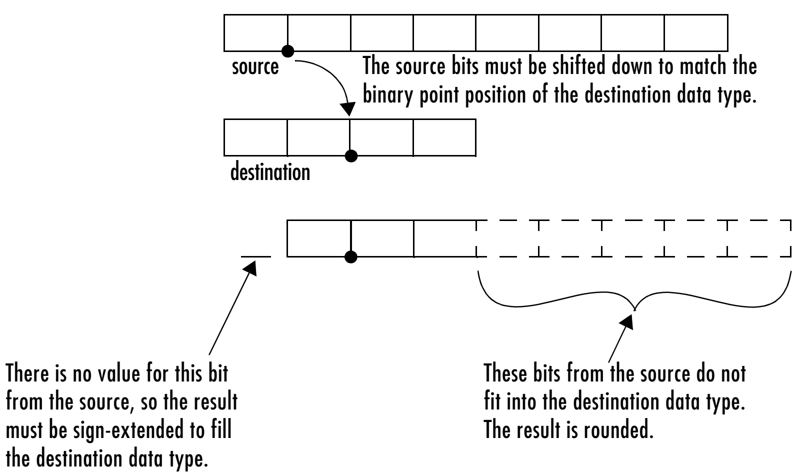


As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either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

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

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

Casting from a Longer Data Type to a Shorter Data Type. Consider the cast of a nonzero number, represented by an 8-bit data type with seven fractional bits, to a 4-bit data type with two fractional bits:



As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so 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
`short int`	-32,767	32,767

Integer Type	Minimum	Maximum
`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` in the online or PDF documentation.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepLSB'	The least significant bits of the product are kept. The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If ProductWordLength is less than is necessary for the full-precision product, then overflow occurs. The rule for computing the resulting product fraction length is given in ProductMode in the online or PDF documentation.
	'KeepMSB'	The most significant bits of the product are kept. The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the most significant bits. If ProductWordLength is less than is necessary for the full-precision product, then rounding occurs. The rule for computing the resulting product fraction length is given in ProductMode in the online or PDF documentation.
	'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 in the online or PDF documentation.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepLSB'	The least significant bits of the sum are kept. The resulting word length is determined by the SumWordLength property. If SumWordLength is greater than is necessary for the full-precision sum, then the result is stored in the least significant bits. If SumWordLength is less than is necessary for the full-precision sum, then overflow occurs. The rule for computing the resulting sum fraction length is given in SumMode in the online or PDF documentation.
	'KeepMSB'	The most significant bits of the sum are kept. 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 in the online or PDF documentation.
	'SpecifyPrecision'	You can specify both the word length and the fraction length of the resulting sum.
SumWordLength	Positive integer	The word length of sum results when SumMode is 'KeepLSB', 'KeepMSB', or 'SpecifyPrecision'.

fimath Object Properties Related to Overflow Handling	Property Value	Description
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'.

3

Working with fi Objects

Constructing fi Objects (p. 3-2)	Teaches you how to create `fi` objects
fi Object Properties (p. 3-9)	Tells you how to find more information about the properties associated with `fi` objects, and shows you how to set these properties
fi Object Functions (p. 3-13)	Introduces the functions in the toolbox that operate directly on `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


          DataType: Fixed
           Scaling: BinaryPoint
            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" in Chapter 1.

The fi constructor function can be used 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(v,T,F)` returns a fixed-point object with value `v`, `embedded.numerictype T`, and `embedded.fimath F`. Refer to Chapter 4, “Working with fimath Objects,” for more information on `fimath` objects.
- `fi(...'PropertyName',PropertyValue...)` and `fi('PropertyName',PropertyValue...)` allow you to set fixed-point objects 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


              DataType: Fixed
               Scaling: BinaryPoint
                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


              DataType: Fixed
               Scaling: BinaryPoint
                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


              DataType: Fixed
               Scaling: BinaryPoint
                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


              DataType: Fixed
               Scaling: BinaryPoint
                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


          DataType: Fixed
           Scaling: BinaryPoint
            Signed: true
        WordLength: 16
    FractionLength: 13
```

Constructing a fi Object Using a numerictype Object

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

```
T = numerictype

T =


          DataType: Fixed
           Scaling: BinaryPoint
            Signed: true
        WordLength: 16
    FractionLength: 15

a = fi(pi, T)

a =

    1.0000
```

```
              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15


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

a = fi(pi, T, F)

a =

    1.0000


              DataType: Fixed
               Scaling: BinaryPoint
```

```
              Signed: true
          WordLength: 16
      FractionLength: 15


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

Copying a fi Object

To copy a `fi` object, use the `fi` constructor function:

```
a = fi(pi)

a =

    3.1416


            DataType: Fixed
             Scaling: BinaryPoint
              Signed: true
          WordLength: 16
      FractionLength: 13


b = fi(a)

b =

    3.1416


            DataType: Fixed
             Scaling: BinaryPoint
              Signed: true
```

```
     WordLength: 16
 FractionLength: 13
```

fi Object Properties

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

- "Data Properties" on page 3-9
- "fimath Properties" on page 3-9
- "numerictype Properties" on page 3-10

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
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined

- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode
- `SumFractionLength` — Fraction length, in bits, of the sum data type
- `SumMode` — Defines how the sum data type is determined
- `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
- `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 Chapter 9, "Property Reference" in the online or PDF documentation. 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 3-11
- "Using Direct Property Referencing with fi" on page 3-11

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


              DataType: Fixed
               Scaling: BinaryPoint
                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 =

    3.1250


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 8
       FractionLength: 3


            RoundMode: floor
         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.

bin	bitand	bitcmp	bitget	bitor	bitxor
complex	conj	ctranspose	dec	disp	double
eps	eq	fi	ge	get	gt
hex	horzcat	imag	int	int8	int16
int32	iscolumn	isempty	isequal	isfi	ispropequal
isreal	isrow	isscalar	issigned	isvector	le
length	loglog	lsb	lt	max	min
minus	mtimes	ndims	ne	oct	plot
plus	range	real	realmax	realmin	repmat
rescale	reset	reshape	semilogx	semilogy	single
size	squeeze	stripscaling	subsasgn	subsref	times
transpose	uint8	uint16	uint32	uminus	vertcat

You can learn about the functions associated with fi objects in Chapter 10, "Function Reference" in the online or PDF documentation.

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
```

4

Working with fimath Objects

Constructing fimath Objects (p. 4-2)	Teaches you how to create `fimath` objects
fimath Object Properties (p. 4-4)	Tells you how to find more information about the properties associated with `fimath` objects, and shows you how to set these properties
Using fimath Objects to Perform Fixed-Point Arithmetic (p. 4-6)	Gives examples of using `fimath` objects to control the results of fixed-point arithmetic with `fi` objects
Using fimath to Share Arithmetic Rules (p. 4-8)	Gives an example of using a `fimath` object to share modular arithmetic information among multiple `fi` objects
fimath Object Functions (p. 4-10)	Introduces the functions in the toolbox that operate directly on `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: round
         OverflowMode: saturate
          ProductMode: FullPrecision
 MaxProductWordLength: 128
              SumMode: FullPrecision
     MaxSumWordLength: 128
        CastBeforeSum: true
```

To copy a fimath object, use the fimath constructor function:

```
F = fimath;
G = fimath(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 4-4.

fimath Object Properties

All the properties of `fimath` objects are 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
- `ProductFractionLength` — Fraction length, in bits, of the product data type
- `ProductMode` — Defines how the product data type is determined
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode
- `SumFractionLength` — Fraction length, in bits, of the sum data type
- `SumMode` — Defines how the sum data type is determined
- `SumWordLength` — Word length, in bits, of the sum data type

These properties are described in detail in Chapter 9, "Property Reference" in the online or PDF documentation. There are two 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 4-4
- "Using Direct Property Referencing with fimath" on page 4-5

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
```

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


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 13


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

a.fimath

ans =


             RoundMode: round
          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


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 19
        FractionLength: 13


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

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 =


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15

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
```

```
              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15

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

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

b =

   -0.4292

              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15

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

fimath Object Functions

The following functions operate directly on `fimath` objects.

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

You can learn about the functions associated with `fimath` objects in Chapter 10, “Function Reference” in the online or PDF documentation.

5

Working with fipref Objects

Constructing fipref Objects (p. 5-2)	Teaches you how to create `fipref` objects
fipref Object Properties (p. 5-3)	Tells you how to find more information about the properties associated with `fipref` objects, and shows you how to set these properties
Using fipref Objects to Set Display Preferences (p. 5-5)	Gives examples of using `fipref` objects to set display preferences for `fi` objects
fipref Object Functions (p. 5-7)	Introduces the functions in the toolbox that operate directly on `fipref` objects

Constructing fipref Objects

fipref objects define the display attributes for 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'
```

The syntax

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

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

fipref Object Properties

All the properties of `fipref` objects are writable.

- `FimathDisplay` — Display options for the `fimath` attributes of a `fi` object
- `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” in the online or PDF documentation. 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'
```

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'
```

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


               DataType: Fixed
                Scaling: BinaryPoint
                 Signed: true
             WordLength: 16
         FractionLength: 13


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

Now, change the `fipref` properties:

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

P =

          NumberDisplay: 'bin'
     NumericTypeDisplay: 'short'
```

```
         FimathDisplay: 'none'

a

a =

0110010010001000
(two's complement bin)
      S16Q13
```

fipref Object Functions

The following functions operate directly on `fipref` objects.

- `fipref`
- `savefipref`

You can learn about the functions associated with `fipref` objects in Chapter 10, “Function Reference” in the online or PDF documentation.

6

Working with numerictype Objects

Constructing numerictype Objects (p. 6-2)	Teaches you how to create `numerictype` objects
numerictype Object Properties (p. 6-4)	Tells you how to find more information about the properties associated with `numerictype` objects, and shows you how to set these properties
The numerictype Structure (p. 6-6)	Presents the `numerictype` object as a MATLAB structure, and gives the valid fields and settings for those fields
Using numerictype Objects to Share Data Type and Scaling Settings (p. 6-8)	Gives an example of using a `numerictype` object to share modular data type and scaling information among multiple `fi` objects
numerictype Object Functions (p. 6-11)	Introduces the functions in the toolbox that operate directly on `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 =


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15
```

To copy a numerictype object, use the numerictype constructor function:

```
U = numerictype(T)

U =


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15
```

The syntax

```
T = numerictype(...'PropertyName',PropertyValue...)
```

allows you to set properties for a `numerictype` object at object creation with property name/property value pairs. Refer to “Setting numerictype Properties at Object Creation” on page 6-4.

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" in the online or PDF documentation. 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 6-4
- "Using Direct Property Referencing with numerictype objects" on page 6-5

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 =


              DataType: Fixed
               Scaling: BinaryPoint
                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 =


            DataType: Fixed
             Scaling: BinaryPoint
              Signed: true
          WordLength: 32
      FractionLength: 31
```

The numerictype Structure

The `numerictype` object contains all the data type and scaling attributes of a `fi` object. The object acts the same 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 is valid for `fi` objects.

DataTypeMode	DataType	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
Fully specified fixed-point data types							
Fixed-point: binary point scaling	fixed	BinaryPoint	1/0	w	f	1	0
Fixed-point: slope and bias scaling	fixed	SlopeBias	1/0	w	N/A	s	b
Partially specified fixed-point data type							
Fixed-point: unspecified scaling	fixed	Unspecified	1/0	w	N/A	N/A	N/A
Built-in data types							
int8	fixed	BinaryPoint	1	8	0	1	0
int16	fixed	BinaryPoint	1	16	0	1	0
int32	fixed	BinaryPoint	1	32	0	1	0
uint8	fixed	BinaryPoint	0	8	0	1	0
uint16	fixed	BinaryPoint	0	16	0	1	0
uint32	fixed	BinaryPoint	0	32	0	1	0

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

Properties That Affect the Slope

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

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

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

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

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

Stored Integer Value and Real World Value

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

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

which is equivalent to

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

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

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 =


                DataType: Fixed
                 Scaling: BinaryPoint
                  Signed: true
              WordLength: 32
          FractionLength: 28

a = fi(pi,T)

a =

           3.1415926553309


                DataType: Fixed
                 Scaling: BinaryPoint
                  Signed: true
              WordLength: 32
          FractionLength: 28


               RoundMode: round
            OverflowMode: saturate
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                 SumMode: FullPrecision
```

```
      MaxSumWordLength: 128
         CastBeforeSum: true

b = fi(pi/2, T)

b =

           1.5707963258028


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 32
        FractionLength: 28


             RoundMode: round
          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 =


              DataType: Fixed
               Scaling: SlopeBias
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0
c = fi(pi, T)
```

```
c =

     4


              DataType: Fixed
               Scaling: SlopeBias
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

             RoundMode: round
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
d = fi(pi/2, T)

d =

     0


              DataType: Fixed
               Scaling: SlopeBias
                Signed: true
            WordLength: 16
                 Slope: 2^2
                  Bias: 0

             RoundMode: round
          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 Chapter 10, "Function Reference" in the online or PDF documentation.

Working with quantizer Objects

Constructing quantizer Objects (p. 7-2)	Explains how to create `quantizer` objects
quantizer Object Properties (p. 7-4)	Outlines the properties of the `quantizer` objects
Quantizing Data with quantizer Objects (p. 7-6)	Discusses using `quantizer` objects to quantize data — how and what quantizing data does
Transformations for Quantized Data (p. 7-8)	Offers a brief explanation of transforming quantized data between representations
quantizer Object Functions (p. 7-9)	Introduces the functions in the toolbox that operate directly on `quantizer` objects

Constructing quantizer Objects

You can use `quantizer` objects to quantize data sets before you pass them to `fi` objects. 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, use the `quantizer` constructor function:

```
r = quantizer(q)

r =

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

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

```
NOperations = 0
```

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 7-4 for more details.

quantizer Object Properties

You can set the values of some quantizer object properties. However, some properties have read-only values. The following sections cover settable and read-only properties:

- "Settable quantizer Object Properties" on page 7-4
- "Read-Only quantizer Object Properties" on page 7-5

Settable quantizer Object Properties

You can set the following four quantizer object properties:

- 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," in the online or PDF documentation 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'`.

Read-Only quantizer Object Properties

`quantizer` objects have five read-only properties:

- `Max` — Maximum value data has before a `quantizer` object is applied, that is, before quantization using `quantize`
- `Min` — Minimum value data has before a `quantizer` object is applied, that is, before quantization using `quantize`
- `NOperations` — Number of quantization operations that occur during quantization when you use a `quantizer` object
- `NOverflows` — Number of overflows that occur during quantization using `quantize`
- `NUnderflows` — Number of underflows that occur during quantization using `quantize`

These properties log quantization information each time you use `quantize` to quantize data with a `quantizer` object. The associated property values change each time you use `quantize` with a given `quantizer` object. You can reset these values to the default value using `reset`.

For an example, see "Quantizing Data with quantizer Objects" on page 7-6.

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 data-related, read-only property values might change.

The following example shows

- How you use `quantize` to quantize data
- How quantization affects read-only properties
- How you reset read-only properties 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` properties.

```
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 `quantizer` object property values.

```
q.max

ans =
2.3726
```

```
q.noverflows

ans =
     15
```

5 Reset the read-only properties 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 Chapter 10, "Function Reference" in the online or PDF documentation.

8

Interoperability with Other Products

Using fi Objects with Simulink (p. 8-2)	Describes how to pass fixed-point data back and forth between the MATLAB workspace and Simulink models using Simulink blocks
Using fi Objects with Signal Processing Blockset (p. 8-7)	Describes how to pass fixed-point data back and forth between the MATLAB workspace and Simulink models using Signal Processing Blockset blocks
Using fi Objects with Filter Design Toolbox (p. 8-11)	Provides a brief description of how to use `fi` objects to supply fixed-point information to `dfilt` objects in the Filter Design Toolbox

Using fi Objects with Simulink

Fixed-Point Toolbox `fi` objects can be used to pass fixed-point data back and forth between the MATLAB workspace and 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 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.

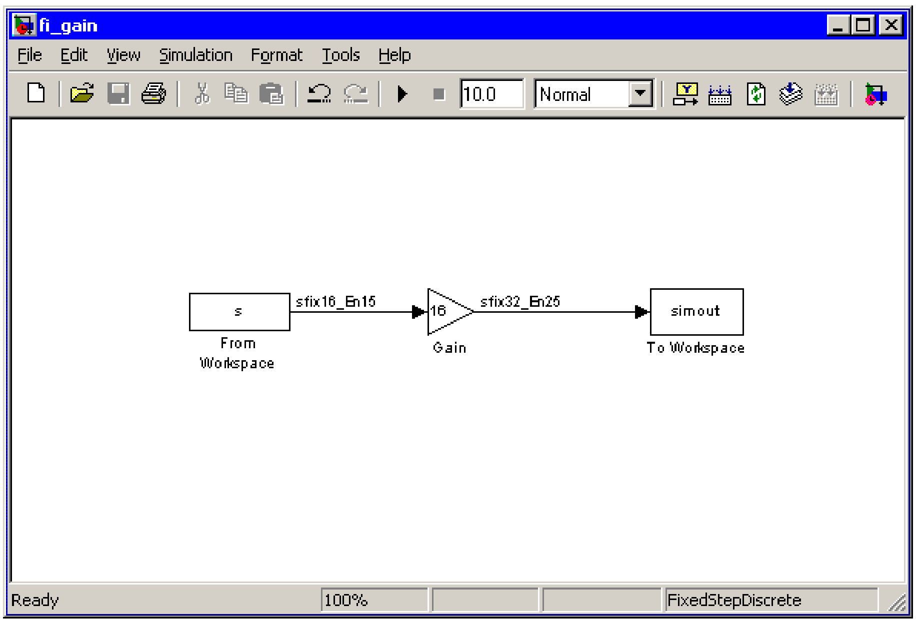
```
a = fi([sin(0:10)' sin(10:-1:0)'])

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


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 16
       FractionLength: 15


            RoundMode: round
         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. 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


              DataType: Fixed
               Scaling: SlopeBias
                Signed: true
            WordLength: 32
                 Slope: 2^-25
                  Bias: 0


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

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 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 programming interface (API) that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to develop MATLAB programs capable of accessing block data while a simulation is running or to access the data from the MATLAB command line. Fixed-point signal information is returned to you via this API as `fi` objects. For more information 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 back and forth 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 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
```

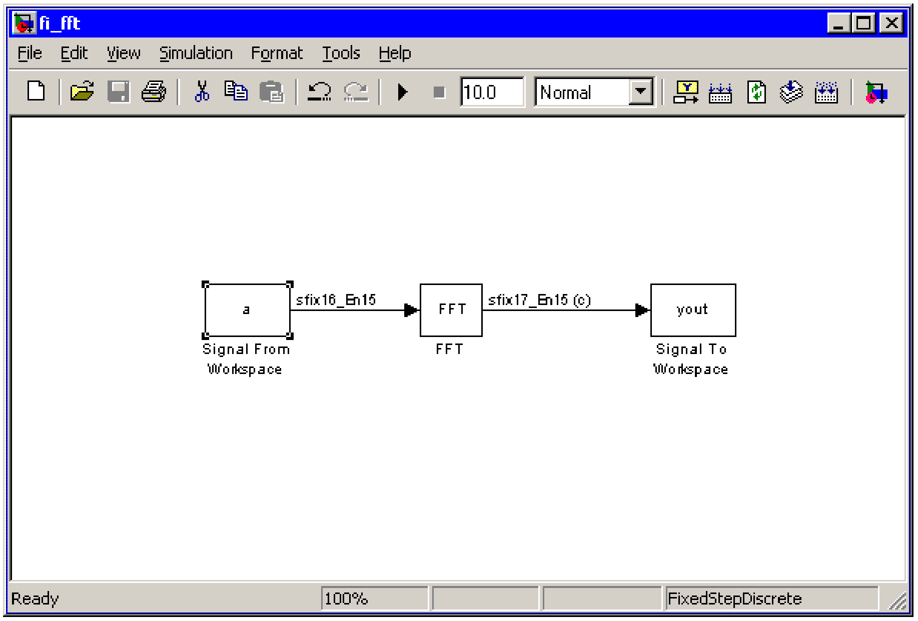
```
-0.9589   -0.9589
-0.2794   -0.7568
 0.6570    0.1411
 0.9893    0.9093
 0.4121    0.8415
-0.5440         0


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 16
       FractionLength: 15


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

The Signal From Workspace block in the following model has the following 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 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 Signal To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` object.

```
yout

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

             DataType: Fixed
              Scaling: SlopeBias
               Signed: true
           WordLength: 17
                Slope: 2^-15
                 Bias: 0


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

Using fi Objects with Filter Design Toolbox

When you set the `Arithmetic` property of `dfilts` in the Filter Design Toolbox to `fixed`, you can provide fixed-point information for `dfilt` inputs, states, and coefficients with `fi` objects using the `InheritSettings` property. Refer to the Filter Design Toolbox documentation for more information.

9

Property Reference

fi Object Properties (p. 9-2)	Defines the `fi` object properties
fimath Object Properties (p. 9-5)	Defines the `fimath` object properties
fipref Object Properties (p. 9-10)	Defines the `fipref` object properties
numerictype Object Properties (p. 9-11)	Defines the `numerictype` object properties
quantizer Object Properties (p. 9-14)	Defines the `quantizer` object properties

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 9-5 and “numerictype Object Properties” on page 9-11 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: round
         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 9-5.

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 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 is valid for `fi` objects.

DataTypeMode	DataType	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
Fully specified fixed-point data types							
Fixed-point: binary point scaling	fixed	BinaryPoint	1/0	w	f	1	0
Fixed-point: slope and bias scaling	fixed	SlopeBias	1/0	w	N/A	s	b
Partially specified fixed-point data type							
Fixed-point: unspecified scaling	fixed	Unspecified	1/0	w	N/A	N/A	N/A
Built-in data types							
int8	fixed	BinaryPoint	1	8	0	1	0

DataTypeMode	DataType	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
int16	fixed	BinaryPoint	1	16	0	1	0
int32	fixed	BinaryPoint	1	32	0	1	0
uint8	fixed	BinaryPoint	0	8	0	1	0
uint16	fixed	BinaryPoint	0	16	0	1	0
uint32	fixed	BinaryPoint	0	32	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`.

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

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.

  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.

  W_p = specified in the `ProductWordLength` property

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

  where

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

- `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

The default value of this property is FullPrecision.

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 toward nearest. Ties round to even numbers.
- fix — Round toward zero.
- floor — Round toward negative infinity.
- round — Round toward nearest. Ties round to the number toward positive infinity.

The default value of this property is round.

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

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.

- 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

  $$\text{integer length} = \max(W_a - F_a, W_b - F_b) + 1$$

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

- 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.

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

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

- 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.

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

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

  where

  $$\text{integer length} = \max(W_a - F_a, W_b - F_b) + 1$$

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

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

  $$F_s = \text{specified in the ProductWordLength property}$$

The default value of this property is FullPrecision.

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.

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`.

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

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 as a `double`
- `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`.

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

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

where the slope can be expressed as

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

DataType

Data type associated with a `fi` object. The only possible value of this property is `Fixed` — Fixed-point or integer data type.

DataTypeMode

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

- `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` — 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
- `int8` — Built-in signed 8-bit integer
- `int16` — Built-in signed 16-bit integer
- `int32` — Built-in signed 32-bit integer
- `uint8` — Built-in unsigned 8-bit integer
- `uint16` — Built-in unsigned 16-bit integer
- `uint32` — Built-in unsigned 32-bit integer

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

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

where the slope can be expressed as

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

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

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

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 default value of this property is `1` (signed).

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

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

where the slope can be expressed as

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

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

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

where the slope can be expressed as

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

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.

Max

Maximum value data has before a `quantizer` object is applied to it, that is, before quantization using `quantize`. The value of `Max` accumulates if you use the same `quantizer` object to quantize several data sets. You can reset the value using `reset`.

The `Max` property is read only.

Min

Minimum value data has before a `quantizer` object is applied to it, that is, before quantization using `quantize`. The value of `Min` accumulates if you use the same quantizer object to quantize several data sets. You can reset the value using reset.

The Min property is read only.

NOperations

Number of quantization operations that occur during quantization when you use a quantizer object. This value accumulates when you use the same quantizer object to process several data sets. You reset the value using reset.

The default value of this property is 0.

The NOperations property is read only.

NOverflows

Number of overflows that occur during quantization using quantize. This value accumulates if you use the same quantizer object to quantize several data sets. You can reset the value using reset.

The default value of this property is 0.

The NOverflows property is read only.

NUnderflows

Number of underflows that occur during quantization using quantize. This value accumulates when you use the same quantizer object to quantize several data sets. You can reset the value using reset.

The default value of this property is 0.

The NUnderflows property is read only.

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.
- `round` — 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`.

Function Reference

Functions — Categorical List (p. 10-2)	Tables of Fixed-Point Toolbox functions by category
fi Object Functions (p. 10-8)	Lists the functions that operate directly on `fi` objects
fimath Object Functions (p. 10-9)	Lists the functions that operate directly on `fimath` objects
fipref Object Functions (p. 10-10)	Lists the functions that operate directly on `fipref` objects
numerictype Object Functions (p. 10-11)	Lists the functions that operate directly on `fipref` objects
quantizer Object Functions (p. 10-12)	Lists the functions that operate directly on `quantizer` objects
Functions — Alphabetical List (p. 10-13)	An Alphabetical List of Fixed-Point Toolbox functions

Functions — Categorical List

- "Bitwise Functions" on page 10-2
- "Constructor and Property Functions" on page 10-2
- "Data Manipulation Functions" on page 10-3
- "Data Type Functions" on page 10-4
- "Data Quantizing Functions" on page 10-5
- "Math Operation Functions" on page 10-5
- "Matrix Manipulation Functions" on page 10-6
- "Numerical Type Functions" on page 10-6
- "One-Dimensional Plotting Functions" on page 10-6
- "Radix Conversion Functions" on page 10-6
- "Relational Operator Functions" on page 10-7
- "Statistics Functions" on page 10-7
- "Subscripted Assignment and Reference Functions" on page 10-7

Bitwise Functions

`bitand`	Return the bitwise `AND` of two `fi` objects
`bitcmp`	Return the bitwise complement of a `fi` object
`bitget`	Return the bit at a certain position
`bitor`	Return the bitwise `OR` of two `fi` objects
`bitset`	Set the bit at a certain position
`bitxor`	Return the bitwise exclusive `OR` of two `fi` objects

Constructor and Property Functions

`copyobj`	Make an independent copy of a `quantizer` object
`disp`	Display an object
`fi`	Construct a `fi` object
`fimath`	Construct a `fimath` object

`fipref`	Construct a `fipref` object
`get`	Return the property values of a `quantizer` object
`numerictype`	Construct a `numerictype` object
`quantizer`	Construct a `quantizer` object
`reset`	Reset one or more objects to their initial conditions
`savefipref`	Save display preferences for the next MATLAB session
`set`	Set or display property values for `quantizer` objects
`stripscaling`	Return the stored integer of a `fi` object
`tostring`	Convert a `quantizer` object to a string

Data Manipulation Functions

`denormalmax`	Return the largest denormalized quantized number for a `quantizer` object
`denormalmin`	Return the smallest denormalized quantized number for a `quantizer` object
`eps`	Return the quantized relative accuracy for `fi` objects or `quantizer` objects
`exponentbias`	Return the exponent bias for a `quantizer` object
`exponentlength`	Return the exponent length of a `quantizer` object
`exponentmax`	Return the maximum exponent for a `quantizer` object
`exponentmin`	Return the minimum exponent for a `quantizer` object
`fractionlength`	Return the fraction length of a `quantizer` object
`iscolumn`	Determine whether a `fi` object is a column vector
`isequal`	Determine whether the real-world values of two `fi` objects are equal, or determine whether the properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isempty`	Determine whether a `fi` object array is empty
`isfi`	Determine whether a variable is a `fi` object
`isfimath`	Determine whether a variable is a `fimath` object
`isnumerictype`	Determine whether a variable is a `numerictype` object

`ispropequal`	Determine whether the properties of two `fi` objects are equal
`isreal`	Test `fi` objects for purely real values
`isrow`	Determine whether a `fi` object is a row vector
`isscalar`	Determine whether an array is a scalar
`issigned`	Determine whether a `fi` object is signed
`isvector`	Determine whether a `fi` object is a vector
`length`	Return the length of a `fi` object
`lsb`	Return the scaling of the least significant bit of a `fi` object
`ndims`	Return the number of dimensions of a `fi` object
`range`	Return the numerical range of a `fi` object or `quantizer` object
`realmax`	Return the largest positive fixed-point value or quantized number
`realmin`	Return the smallest positive normalized fixed-point value or quantized number
`repmat`	Replicate and tile a `fi` object
`rescale`	Change the scaling of a `fi` object
`reshape`	Change the size of a `fi` object
`size`	Return the size of the value of a `fi` object
`squeeze`	Remove the singleton dimensions of a `fi` object
`wordlength`	Return the word length of a `quantizer` object

Data Type Functions

`double`	Return the double-precision floating-point real-world value of a `fi` object
`int`	Return the smallest built-in integer in which the stored integer value of a `fi` object will fit
`int8`	Return the stored integer value of a `fi` object as a built-in `int8`
`int16`	Return the stored integer value of a `fi` object as a built-in `int16`
`int32`	Return the stored integer value of a `fi` object as a built-in `int32`
`single`	Return the single-precision floating-point real-world value of a `fi` object

`uint8`	Return the stored integer value of a `fi` object as a built-in `uint8`
`uint16`	Return the stored integer value of a `fi` object as a built-in `uint16`
`uint32`	Return the stored integer value of a `fi` object as a built-in `uint32`
`intmax`	Return the largest positive stored integer value representable by the `numerictype` of a `fi` object

Data Quantizing Functions

`convergent`	Apply convergent rounding
`quantize`	Apply a `quantizer` object to data
`randquant`	Generate a uniformly distributed, quantized random number using a `quantizer` object
`round`	Round input data using a `quantizer` object without checking for overflow

Math Operation Functions

`add`	Add two objects using a `fimath` object
`conj`	Return the complex conjugate of a `fi` object
`divide`	Divide two objects using a `fimath` object
`minus`	Return the matrix difference between `fi` objects
`mpy`	Multiply two objects using a `fimath` object
`mtimes`	Return the matrix product of `fi` objects
`plus`	Return the matrix sum of `fi` objects
`sub`	Subtract two objects using a `fimath` object
`times`	Return the result of element-by-element multiplication of `fi` objects
`uminus`	Negate the elements of a `fi` object array

Matrix Manipulation Functions

`ctranspose`	Return the complex conjugate transpose of a `fi` object
`horzcat`	Horizontally concatenate two or more `fi` objects
`transpose`	Return the nonconjugate transpose of a `fi` object
`vertcat`	Vertically concatenate two or more `fi` objects

Numerical Type Functions

`complex`	Construct a complex `fi` object from real and imaginary parts
`imag`	Return the imaginary part of a `fi` object
`real`	Return the real part of a `fi` object

One-Dimensional Plotting Functions

`loglog`	Plot the real-world values of `fi` objects on logarithmic axes
`plot`	Plot the real-world values of two `fi` objects against each other
`semilogx`	Plot the real-world values of `fi` objects on a logarithmically scaled *x*-axis and a linearly scaled *y*-axis
`semilogy`	Plot the real-world values of `fi` objects on a linearly scaled *x*-axis and a logarithmically scaled *y*-axis

Radix Conversion Functions

`bin`	Return the binary representation of the stored integer of a `fi` object as a string
`bin2num`	Convert a two's complement binary string to a number using a `quantizer` object
`dec`	Return the unsigned decimal representation of the stored integer of a `fi` object as a string
`hex`	Return the hexadecimal representation of the stored integer of a `fi` object as a string
`hex2num`	Convert hexadecimal string to a number using a `quantizer` object
`num2bin`	Convert a number to a binary string using a `quantizer` object

`num2hex`	Convert a number to its hexadecimal equivalent using a `quantizer` object
`num2int`	Convert a number to a signed integer using a `quantizer` object
`oct`	Return the octal representation of the stored integer of a `fi` object as a string

Relational Operator Functions

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

Statistics Functions

`max`	Return the largest element in an array of `fi` objects or the maximum value of a `quantizer` object object before quantization
`min`	Return the smallest element in an array of `fi` objects or the minimum value of a `quantizer` object object before quantization
`noperations`	Return the number of quantization operations performed by a `quantizer` object
`noverflows`	Return the number of overflows from quantization operations performed by a `quantizer` object
`nunderflows`	Return the number of underflows from quantization operations performed by a `quantizer` object

Subscripted Assignment and Reference Functions

`subsasgn`	Subscripted assignment
`subsref`	Subscripted reference

fi Object Functions

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

bin	bitand	bitcmp	bitget	bitor	bitxor
complex	conj	ctranspose	dec	disp	double
eps	eq	fi	ge	get	gt
hex	horzcat	imag	int	int8	int16
int32	iscolumn	isempty	isequal	isfi	ispropequal
isreal	isrow	isscalar	issigned	isvector	le
length	loglog	lsb	lt	max	min
minus	mtimes	ndims	ne	oct	plot
plus	range	real	realmax	realmin	repmat
rescale	reset	reshape	semilogx	semilogy	single
size	squeeze	stripscaling	subsasgn	subsref	times
transpose	uint8	uint16	uint32	uminus	vertcat

fimath Object Functions

The following functions operate directly on `fimath` objects.

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

fipref Object Functions

The following functions operate directly on `fipref` objects.

- `fipref`
- `savefipref`

numerictype Object Functions

The following functions operate directly on `numerictype` objects.

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

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	

Functions — Alphabetical List

The following pages contain the reference pages for the Fixed-Point Toolbox functions in alphabetical order.

Purpose Add two objects using a `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 `numerictype` object, 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


              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 32
        FractionLength: 16


             RoundMode: round
          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`

bin

Purpose

Return the binary representation of the stored integer of a `fi` object as a string

Syntax

`bin(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.

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

Examples

Example 1

The following code

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

returns

```
10000000   01111111
```

See Also

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

Purpose

Convert a two's complement binary string to a number using a `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 Return the bitwise AND of two `fi` objects

Syntax

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

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

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

bitcmp

Purpose Return the bitwise complement of a `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.

See Also `bitand`, `bitget`, `bitor`, `bitset`, `bitxor`

Purpose Return the bit at a 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.

See Also `bitand`, `bitcmp`, `bitor`, `bitset`, `bitxor`

bitor

Purpose Return the 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 `numerictype` 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.

See Also `bitand`, `bitcmp`, `bitget`, `bitset`, `bitxor`

Purpose

Set the bit at a 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 either `0` (off) or `1` (on).

`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.

See Also

`bitand`, `bitcmp`, `bitget`, `bitor`, `bitxor`

Purpose Return the 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 `numerictype` 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.

See Also `bitand`, `bitcmp`, `bitget`, `bitor`, `bitset`

Purpose

Construct a complex `fi` object from real and imaginary parts

Syntax

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

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.

See Also

`imag`, `real`

conj

Purpose Return the complex conjugate of a `fi` object

Syntax

```
conj(a)
```

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

When `a` is complex,

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

See Also `complex`, `imag`, `real`

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
```

copyobj

Purpose Make an independent copy of a `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 Return the complex conjugate transpose of a `fi` object

Syntax `ctranspose(a)`

Description `ctranspose(a)` returns the complex conjugate transpose of `fi` object `a`. It is also called for the syntax `a'`.

See Also `transpose`

dec

Purpose Return the unsigned decimal representation of the stored integer of a `fi` object as a string

Syntax `dec(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.

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

Examples

Example 1

The code

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

returns

```
128   127
```

See Also `bin`, `hex`, `int`, `oct`

Purpose Return the largest denormalized quantized number for a `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`

denormalmin

Purpose Return the smallest denormalized quantized number for a `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 Display an object

Syntax

```
disp(obj)
```

Description Similar to omitting the closing semicolon from an expression on the command line, except that `disp` does not display the variable name. `disp` lists the property names and property values for a `fi`, `fimath`, `fipref`, or `quantizer` object.

Purpose

Divide two objects using a numerictype object

Syntax

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

Description

c = T.divide(a,b) performs division on the elements of a by the elements of b using 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 numerictype object, 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.

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 =

11001100110011001100110011001100110011001100110011010000000000000
000000000000000
(bin)
      u80,83
```

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

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 =


1100110011001100110011001100110011001100110011001100110011001100
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`

Purpose Return the double-precision floating-point real-world value of a `fi` object

Syntax

```
double(a)
(d1,d2,d3,...) = double(a1,a2,a3,...)
```

Description Fixed-point numbers can be represented as

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

or, equivalently,

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

`double(a)` returns the real-world value of a `fi` object in double-precision floating point.

See Also `single`

Purpose Return the quantized relative accuracy for `fi` objects 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 the 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`

exponentbias

Purpose Return the exponent bias for a `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`

Purpose Return the exponent length of a `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`

exponentmax

Purpose Return the maximum exponent for a `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 Return the minimum exponent for a `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 Construct a `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(..., property1, value1, ...)
a = fi(property1, value1, ....)
```

Description 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`, `slopeadjustmentfactor`, `fixedexponent`, and `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(v,T,F)` returns a fixed-point object with value `v`, `embedded.numerictype` `T`, and embedded.`fimath` `F`. Refer to Chapter 4, "Working with fimath Objects," for more information on `fimath` objects.

- `fi(...'PropertyName',PropertyValue...)` and `fi('PropertyName',PropertyValue...)` allow you to set fixed-point objects for a `fi` object by property name/property value pairs.

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

- "Data Properties" on page 10-45
- "Fimath Properties" on page 10-45
- "Numerictype Properties" on page 10-46

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
- `ProductFractionLength` — Fraction length, in bits, of the product data type

- `ProductMode` — Defines how the product data type is determined
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode
- `SumFractionLength` — Fraction length, in bits, of the sum data type
- `SumMode` — Defines how the sum data type is determined
- `SumWordLength` — 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
- `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 “fi Object Properties” on page 9-2 in the Properties Reference.

Examples

Note For information on the display format of fi objects, refer to "Display Settings" in Chapter 1.

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


          DataType: Fixed
           Scaling: BinaryPoint
            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


          DataType: Fixed
           Scaling: BinaryPoint
            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


          DataType: Fixed
           Scaling: BinaryPoint
            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


          DataType: Fixed
           Scaling: BinaryPoint
            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

        DataType: Fixed
         Scaling: BinaryPoint
          Signed: true
      WordLength: 16
  FractionLength: 13
```

See Also

fimath, fipref, numerictype, quantizer

Purpose Construct a 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

- 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
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined
- ProductWordLength — Word length, in bits, of the product data type
- RoundMode — Rounding mode
- SumFractionLength — Fraction length, in bits, of the sum data type
- SumMode — Defines how the sum data type is determined
- SumWordLength — Word length, in bits, of the sum data type

These properties are described in detail in "fimath Object Properties" on page 9-5 in the Properties Reference.

Examples

Example 1

Type

```
F = fimath
```

to create a default fimath object.

```
F = fimath
```

```
F =


             RoundMode: round
          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`

fipref

Purpose

Construct a `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 `fipref` object using property name/property value pairs.

The properties of the `fipref` object are

- `FimathDisplay` — Display options for the `fimath` attributes of a `fi` object
- `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 “fipref Object Properties” on page 9-10 in the Properties Reference.

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'
```

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 `qpoint`,

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

P =

          NumberDisplay: 'bin'
     NumericTypeDisplay: 'short'
          FimathDisplay: 'full'
```

See Also

`fi`, `fimath`, `numerictype`, `quantizer`, `savefipref`

fractionlength

Purpose Return the fraction length of a `quantizer` object

Syntax

```
fractionlength(q)
```

Description `fractionlength(q)` returns the fraction length of `quantizer` object `q`.

Examples For a floating-point `quantizer` object,

```
q = quantizer('float',[32 8]);
f = fractionlength(q)

f =

    23
```

where $f = 23 = 32 - 8 - 1$.

For a fixed-point `quantizer` object,

```
q = quantizer('fixed',[6 4])
f = fractionlength(q)

q =

          DataMode = fixed
         RoundMode = floor
      OverflowMode = saturate
            Format = [6  4]

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

f =

     4
```

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 the 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 Return the property values of a 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 `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`

Purpose

Determine whether the value of one `fi` object is greater than another

Syntax

```
c = gt(a,b)
a > b
```

Description

`c = gt(a,b)` is called for the syntax '`a > b`' when `a` or `b` is a `fi` object. `a` and `b` must have the same dimensions unless one is a scalar. A scalar can be compared with another object of any size.

`a > b` does an element-by-element comparison between `a` and `b` and returns a matrix of the same size with elements set to `1` where the relation is true, and `0` where the relation is false.

See Also

`eq`, `ge`, `le`, `lt`, `ne`

Purpose Return the hexadecimal representation of the stored integer of a `fi` object as a string

Syntax `hexadecimal(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.

`hex(a)` returns the stored integer of `fi` object `a` in hexadecimal format as a string.

Examples

Example 1

The following code

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

returns

```
80   7f
```

See Also `bin`, `dec`, `int`, `oct`

hex2num

Purpose

Convert a hexadecimal string to a number using a `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 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

Horizontally concatenate two or more 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 Return the imaginary part of a `fi` object

Syntax `imag(a)`

Description `imag(a)` returns the imaginary part of a `fi` object.

See Also `complex`, `real`

Purpose

Return the smallest built-in integer in which the stored integer value of a `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`, or `hex`.

See Also

`int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`

int8

Purpose Return the stored integer value of a `fi` object as a 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 Return the stored integer value of a `fi` object as a built-in `int16`

Syntax `int16(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.

`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`

int32

Purpose Return the stored integer value of a `fi` object as a 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`

Purpose

Return the largest positive stored integer value representable by the `numerictype` of a `fi` object

Syntax

```
x = intmax(a)
```

Description

`x = intmax(a)` returns the largest positive value representable by the `numerictype` of `a`.

See Also

`lsb`, `stripscaling`

iscolumn

Purpose Determine whether a fi object is a 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`

Purpose Determine whether a `fi` object array is empty

Syntax `isempty(a)`

Description `isempty(a)` returns `1` if `a` is an empty array and `0` otherwise. An empty array has no elements; that is, `prod(size(a))==0`.

See Also `isscalar`, `isvector`

isequal

Purpose

Determine whether the real-world values of two `fi` objects are equal, or determine whether the 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`

Purpose Determine whether a variable is a `fi` object

Syntax `isfi(a)`

Description `isfi(a)` returns `1` if `a` is a `fi` object, and `0` otherwise.

See Also `fi`, `isfimath`, `isnumerictype`

isfimath

Purpose Determine whether a variable is a `fimath` object

Syntax

```
isfimath(F)
```

Description `isfimath(F)` returns `1` if `F` is a `fimath` object, and `0` otherwise.

See Also `fimath`, `isfi`, `isnumerictype`

Purpose Determine whether a variable is a `numerictype` object

Syntax `isnumerictype(T)`

Description `isnumerictype(T)` returns `1` if a is a `numerictype` object, and `0` otherwise.

See Also `isfi`, `isfimath`, `numerictype`

ispropequal

Purpose Determine whether the 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`.

See Also `fi`, `isequal`

Purpose Test `fi` objects for purely real values

Syntax

```
isreal(a)
```

Description `isreal(a)` returns `1` if `fi` object `a` does not have an imaginary part, and `0` otherwise.

isrow

Purpose Determine whether a `fi` object is a 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 a `fi` object array is a scalar

Syntax

```
isscalar(a)
```

Description `isscalar(a)` returns `1` if `a` is a 1-by-1 matrix, and `0` otherwise.

See Also `isempty`, `isvector`

issigned

Purpose Determine whether a `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 a `fi` object is a vector

Syntax `isvector(a)`

Description `isvector(a)` returns `1` if `a` is a 1-by-n or n-by-1 vector, where n >= 0, and `0` otherwise.

See Also `isempty`, `isscalar`

Purpose Determine whether the value of a `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 Return the length of a `fi` object

Syntax

```
length(a)
```

Description `length(a)` returns the length of `fi` object `a`. It is equivalent to `max(size(a))` for nonempty arrays and to `0` for empty arrays.

loglog

Purpose Plot the real-world values of `fi` objects on logarithmic axes

Syntax

```
loglog(a)
loglog(a,b)
```

Description The `loglog` function works the same as the `plot` function, except that the axes drawn by `loglog` are base-10 logarithmic.

See Also `plot`, `semilogx`, `semilogy`

Purpose Return the scaling of the least significant bit of a `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`

lt

Purpose Determine whether the value of a `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

Return the largest element in an array of `fi` objects or the maximum value of a `quantizer` object before quantization

Syntax

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

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,y)` returns an array the same size as `a` and `y` with the largest elements taken from `a` or `y`. 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.

`max(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 `reset(q)`. `max(q)` is equivalent to `get(q,'max')` and `q.max`.

Examples

```
q = quantizer;
warning on
y = quantize(q,-20:10);
max(q)
Warning: 29 overflows.
ans =

    10
```

See Also `min`, `quantize`

Purpose

Return the smallest element in an array of `fi` objects or the minimum value of a `quantizer` object before quantization

Syntax

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

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

`[y,v] = max(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] = 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 minimum.

`min(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 `reset(q)`. `min(q)` is equivalent to `get(q,'min')` and `q.min`.

See Also

`max`, `quantize`

minus

Purpose Return the 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.

See Also `mtimes`, `plus`, `times`, `uminus`

Purpose Multiply two objects using a 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 `numerictype` object, 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


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 40
       FractionLength: 30


            RoundMode: round
         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`

Purpose Return the 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`.

See Also `plus`, `minus`, `times`, `uminus`

ndims

Purpose Return the number of dimensions of a `fi` object

Syntax

```
ndims(a)
```

Description `ndims(a)` returns the number of dimensions of the `fi` object `a`. The number of dimensions in an array is always greater than or equal to 2. Trailing singleton dimensions are ignored. `ndims(a)` is equivalent to `length(size(a))`.

See Also `reshape`, `size`

Purpose

Determine whether the 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`

noperations

Purpose Return the number of quantization operations performed by a `quantizer` object

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 `reset(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, since `(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 `get`, `quantizer`, `reset`

Purpose Return the number of overflows from quantization operations performed by a `quantizer` object

Syntax `noverflows(q)`

Description `noverflows` returns the accumulated number of overflows resulting from quantization operations performed by a `quantizer` object.

See Also `get`, `max`, `range`, `reset`

Purpose

Convert a number to a binary string using a `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 a number to its hexadecimal equivalent using a `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 a number to a signed integer

Syntax

```
y = num2int(q,x)
[y1,y2,...] = 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`

numerictype

Purpose Construct a numerictype object

Syntax

```
T = numerictype
T = numerictype(...'PropertyName',PropertyValue...)
```

Description You can use the numerictype constructor function in the following ways:

- `T = numerictype` creates a default numerictype object.
- `T = numerictype(...'PropertyName',PropertyValue...)` allows you to set properties for a numerictype object at object creation with property name/property value pairs.

The properties of the numerictype object are

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

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 =


          DataType: Fixed
           Scaling: BinaryPoint
            Signed: true
        WordLength: 32
    FractionLength: 30
```

See Also

`fi`, `fimath`, `fipref`, `quantizer`

nunderflows

Purpose Return the number of underflows from quantization operations performed by a `quantizer` object

Syntax `nunderflows(q)`

Description `nunderflows` returns the accumulated number of underflows resulting from quantization operations performed by a `quantizer` object. An underflow is defined as a number that is nonzero before it is quantized, and zero after it is quantized.

See Also `denormalmin`, `eps`, `quantize`, `quantizer`, `reset`

Purpose Return the octal representation of the stored integer of a `fi` object as a string

Syntax

```
oct(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.

`oct(a)` returns the stored integer of `fi` object `a` in octal format as a string.

Examples

Example 1

The following code

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

returns

```
200   177
```

See Also `bin`, `dec`, `hex`, `int`

plot

Purpose

Plot the real-world values of two `fi` objects against each other

Syntax

```
plot(a)
plot(a,b)
plot(a,b,s)
plot(a1,b1,s1,a2,b2,s2,...)
```

Description

The plot function for `fi` objects works the same as the built-in `plot` function.

`plot(a)` plots the columns of `a` versus their index. If `a` is complex, `plot(a)` is equivalent to `plot(real(a),imag(a))`. In all other uses of `plot`, the imaginary part is ignored.

`plot(a,b)` plots vector `b` versus vector `a`. If `a` or `b` is a matrix, then the vector is plotted versus the rows or the columns of the matrix, depending on which matches the dimension of the vector. If `a` is a scalar and `b` is a vector, `length(b)` disconnected points are plotted.

You can plot with various line types, plot symbols, and colors using `plot(a,b,s)` where `s` is a character string composed of one element from any or all of the three columns in the following table.

Color	**Symbol**	**Line Type**
b blue	. point	- solid
g green	o circle	: dotted
r red	x x-mark	-. dashdot
c cyan	+ plus	-- dashed
m magenta	* star	
y yellow	s square	
k black	d diamond	
	v triangle (down)	
	^ triangle (up)	
	< triangle (left)	

Color	Symbol	Line Type
	> triangle (right)	
	p pentagram	
	h hexagram	

For example, `plot(a,b,'c+:')` plots a cyan dotted line with a plus symbol at each data point. `plot(a,b,'bd')` plots a blue diamond at each data point, but does not draw any line.

`plot(a1,b1,s1,a2,b2,s2,...)` combines the plots defined by the `(a,b,s)` triples. For example, `plot(a,b,'y-',a,b,'go')` plots the data twice, with a solid yellow line interpolating green circles at the data points.

See Also

`loglog`, `semilogx`, `semilogy`

plus

Purpose Return the 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.

See Also `minus`, `mtimes`, `times`, `uminus`

Purpose

Apply a 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 reset is made.

Examples

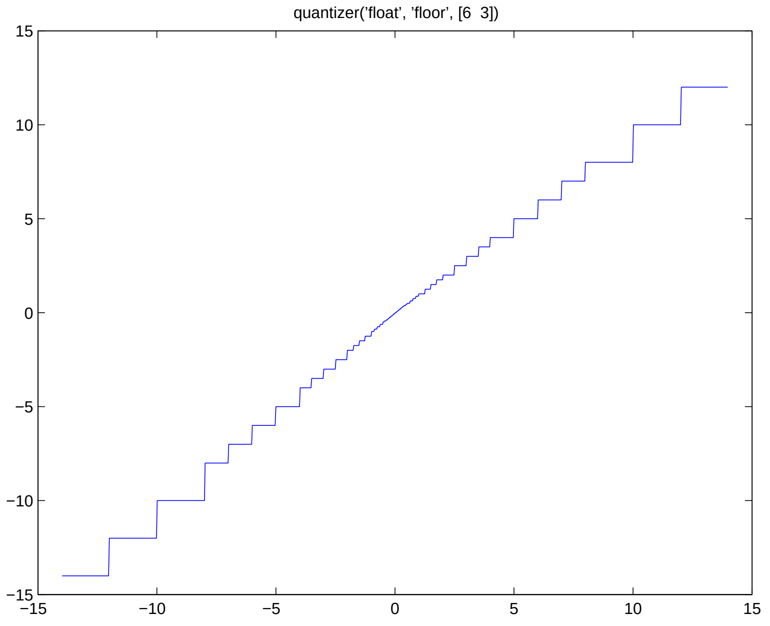
The following examples demonstrate using quantize to quantize data.

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.
```

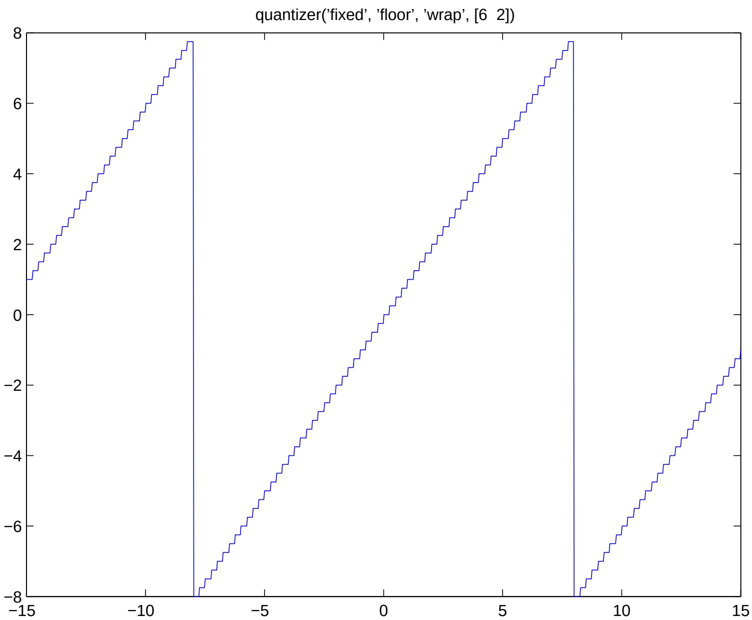


Example 2 - Fixed-Point

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 a `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`.

These are the `quantizer` object property values, sorted by associated property name:

Property Name	Property Value	Description
`mode`	`'double'`	Double-precision mode. Override all other parameters.
	`'float'`	Custom-precision floating-point mode.
	`'fixed'`	Signed fixed-point mode.
	`'single'`	Single-precision mode. Override all other parameters.

Property Name	Property Value	Description
	'ufixed'	Unsigned fixed-point mode.
roundmode	'ceil'	Round toward negative infinity.
	'convergent'	Convergent rounding.
	'fix'	Round toward zero.
	'floor'	Round toward positive infinity.
	'round'	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 properties: 'max', 'min', 'noverflows', 'nunderflows', and 'noperations'. They can be accessed through quantizer/get or q.max, q.min, 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 quantizer/reset method.

The following table lists the read-only `quantizer` object properties:

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`

Purpose

Generate a uniformly distributed, quantized random number using a `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 Return the numerical range of a `fi` object or `quantizer` object

Syntax

```
range(a)
[min, max] = range(a)
r = range(q)
[min, max] = range(q)
```

Description

`range(a)` returns the minimum and maximum possible values of `fi` object `a` in two-vector format. 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

Purpose Return the real part of a `fi` object

Syntax `real(a)`

Description `real(a)` returns the real part of a `fi` object.

See Also `complex`, `imag`

realmax

Purpose

Return the 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 Return the 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} = `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`

repmat

Purpose Replicate and tile a fi object

Syntax

```
repmat(a,m,n)
repmat(a,[m n])
repmat(a,[m n p ...])
```

Description

repmat(a,m,n) creates a large matrix consisting of an m-by-n tiling of copies of a. When a is a scalar, repmat(a,m,n) is commonly used to produce an m-by-n matrix filled with the value of a.

repmat(a,[m n]) is equivalent to repmat(a,m,n).

repmat(a,[m n p ...]) tiles the array a to produce an m-by-n-by-p-by-... block array. a can be n-D.

Purpose

Change the scaling of a `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 one or more objects to their initial conditions

Syntax

```
reset(obj)
reset(q1, q2, ...)
```

Description `reset(obj)` resets `fi`, `fimath`, `fipref`, or `quantizer` object `obj` to its initial conditions.

`reset(q1, q2,...)` resets the states of the `quantizer` objects `q1`, `q2`,.... to their initial conditions.

The states of a `quantizer` object are

- `max` — Maximum value before quantizing
- `min` — Minimum value before quantizing
- `noverflows` — Number of overflows
- `nunderflows` — Number of underflows
- `noperations` — Number of quantization operations performed

See Also `quantizer`, `set`

reshape

Purpose Change the size of a `fi` object

Syntax

```
reshape(a,m,n)
reshape(a,m,n,p,...)
reshape(a,...,[ ],...)
```

Description

`reshape(a,m,n)` returns the `m`-by-`n` matrix whose elements are taken columnwise from the `fi` object `a`. If `a` does not have `m`-by-`n` elements, an error is returned.

`reshape(a,m,n,p,...)` returns an `n`-D array with the same elements as `a`, but reshaped to have the size `m`-by-`n`-by-`p`-by-`...`. `m*n*p*...` must be the same as `prod(size(a))`.

`reshape(a,...,[ ],...)` calculates the length of the dimension represented by `[ ]`, such that the product of the dimensions equals `prod(size(a))`. `prod(size(a))` must be evenly divisible by the product of the known dimensions. You can use only one occurrence of `[ ]`.

See Also `ndims`, `size`

Purpose

Round input data using a `quantizer` object without checking for overflow

Syntax

```
round(q,x)
```

Description

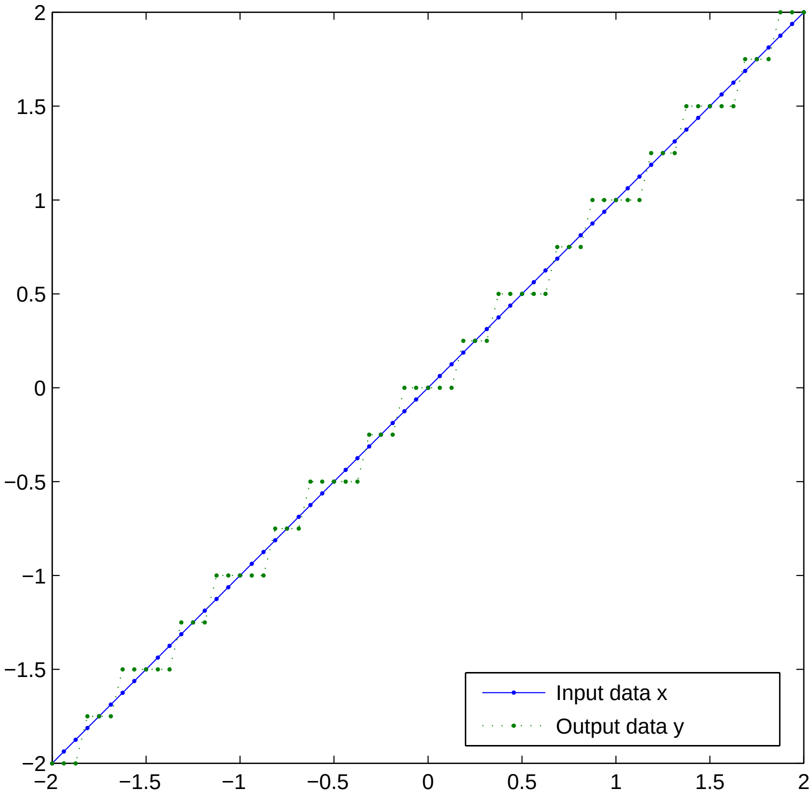
`round(q,x)` uses the `RoundMode` and `FractionLength` settings of `q` to round the numeric data `x`, but does not check for overflows during the operation. Compare to `quantize`.

Example

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 display preferences for the next MATLAB session

Syntax `savefipref`

Description `savefipref` saves the settings of the current `fipref` object for the next MATLAB session.

See Also `fipref`

semilogx

Purpose

Plot the real-world values of `fi` objects on a logarithmically scaled *x*-axis and a linearly scaled *y*-axis

Syntax

```
semilogx(a)
semilogx(a,b)
```

Description

The `semilogx` function works the same as the `plot` function, except that a base-10 logarithmic scale is used for the *x*-axis.

See Also

`loglog`, `plot`, `semilogy`

Purpose

Plot the real-world values of `fi` objects on a linearly scaled *x*-axis and a logarithmically scaled *y*-axis

Syntax

```
semilogy(a)
semilogy(a,b)
```

Description

The `semilogy` function works the same as the `plot` function, except that a base-10 logarithmic scale is used for the *y*-axis.

See Also

`loglog`, `plot`, `semilogx`

set

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
set(q)
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`.

The states are cleared when you set any value other than `WarnIfOverflow`.

See Also `get`

Purpose Return the single-precision floating-point real-world value of a `fi` object

Syntax

```
single(a)
(s1,s2,s3,...) = single(a1,a2,a3,...)
```

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.

`(s1,s2,s3,...) = single(a1,a2,a3,...)` converts `fi` objects `a1`, `a2`, `...` to single-precision floating-point `s1`, `s2`, `...`, respectively.

See Also `double`

size

Purpose

Return the size of the value of a `fi` object

Syntax

```
size(a)
[m,n] = size(a)
[m1,m2,m3,...,mn] = size(a)
m = size(a,dim)
```

Description

`size(a)` returns the two-element row vector `d = [m, n]` containing the number of rows and columns in `a`. For `n`-D arrays, `size(a)` ret urns a 1-by-n vector. Trailing singleton dimensions are ignored.

`[m,n] = size(a)` returns the number of rows and columns in `a` as separate output variables.

`[m1,m2,m3,...,mn] = size(a)` returns the sizes of the first `n` dimensions of `a`. If the number of output arguments n does not equal `ndims(a)`, then for

- `n > ndims(a)` — Ones are returned for `ndims(a)+1` through `n`.
- `n < ndims(a)` — `mn` contains the product of the sizes of the dimensions `n+1` through `ndims(a)`.

`m = size(a,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(a,1)` returns the number of rows of `a`.

See Also

`ndims`, `reshape`

Purpose Remove the singleton dimensions of a `fi` object

Syntax `squeeze(a)`

Description `squeeze(a)` returns an array with the same elements as a but with all the singleton dimensions removed. A singleton is a dimension such that `size(A,dim)==1`. 2-D arrays are unaffected by `squeeze` so that row vectors remain rows.

stripscaling

Purpose Return the stored integer of a `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 a `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 `numerictype` object, 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


             DataType: Fixed
              Scaling: BinaryPoint
               Signed: true
           WordLength: 32
       FractionLength: 16


            RoundMode: round
         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 `':'`.

See Also `subsref`

subsref

Purpose Subscripted reference

Syntax

```
a(I)
a(I,J)
a(I,:)
a(:,I)
a(I,J,K,...)
b = subsref(a,S)
```

Description

`a(I)` is an array formed from the elements of a specified by the subscript vector `I`. The resulting array is the same size as `I` except for the special case where a and `I` are both vectors. In this case, `a(I)` has the same number of elements as `I` but has the orientation of `a`.

`a(I,J)` is an array formed from the elements of the rectangular submatrix of a specified by the subscript vectors `I` and `J`. The resulting array has `length(I)` rows and `length(J)` columns.

A colon used as a subscript, as in `a(I,:)` or `a(:,I)` indicates the entire column or row.

For multidimensional arrays, `a(I,J,K,...)` is the subarray specified by the subscripts. The result is `length(I)`-by-`length(J)`-by-`length(K)`-....

`b = subsref(a,S)` is called for the syntax `a(I)`, `a{I}`, or `a.I` 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,:)` invokes `subsref(a,S)` 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 `':'`.

See Also `subsasgn`

Purpose Return the result of 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.

See Also `plus`, `minus`, `mtimes`, `uminus`

tostring

Purpose Convert a quantizer object to a 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`.

Examples When you use `tostring` with a `quantizer` object you see the following response:

```
q = quantizer

q =

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

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


s = tostring(q)

s =

quantizer('fixed', 'floor', 'saturate', [16  15])


eval(s)

ans =

        DataMode = fixed
       RoundMode = floor
```

```
    OverflowMode = saturate
          Format = [16  15]

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

Note that s is the same as q.

See Also quantizer

transpose

Purpose Return the nonconjugate transpose of a `fi` object

Syntax `transpose(a)`

Description `transpose(a)` returns the nonconjugate transpose of `fi` object `a`. It is also called for the syntax `a.'`.

See Also `ctranspose`

Purpose Return the stored integer value of a `fi` object as a built-in `uint8`

Syntax `uint8(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.

`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`

uint16

Purpose

Return the stored integer value of a `fi` object as a built-in `uint16`

Syntax

`uint16(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently,

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

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 Return the stored integer value of a `fi` object as a 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 the elements of a `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`.

See Also `plus`, `minus`, `mtimes`, `times`

Purpose

Vertically concatenate two or more fi objects

Syntax

```
c = vertcat(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

wordlength

Purpose Return the word length of a `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`

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

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

where the slope can be expressed as

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

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

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

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

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

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

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

or

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, 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

logging

Tool provided by the **Fixed-Point Settings** interface that outputs the minimum values, maximum values, and any overflows for all fixed-point blocks in any model that you run with a fixed-point license.

See also overflow

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

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

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 `round` 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

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

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

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:

$$\textit{real-world value} = integer$$

In [Slope Bias] representation, fixed-point numbers can be represented as

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

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$\textit{real-world value} = integer \times 2^{-\textit{fraction length}} = integer \times 2^{0}$$

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

A

add function 9-14
ANSI C
  compared with fi objects 2-20
arithmetic operations
  fixed-point 2-8

B

Bias property 9-11
bin function 9-16
bin property 9-2
bin2num function 9-17
binary conversions 2-23
bitand function 9-19
bitcmp function 9-20
bitget function 9-21
bitor function 9-22
bitset function 9-23
bitxor function 9-24

C

CastBeforeSum property 9-5
casts
  fixed-point 2-17
complex function 9-25
complex multiplication
  fixed-point 2-11
conj function 9-26
convergent function 9-27
copyobj function 9-28
ctranspose function 9-29

D

Data property 9-2
DataType property 9-11
DataTypeMode property 9-11
dec function 9-30
demos 1-7
denormalmax function 9-31
denormalmin function 9-32
disp function 9-33
display settings 1-5
div function 9-34
double function 9-37
double property 9-2

E

eps function 9-38
eq function 9-39
exponentbias function 9-40
exponentlength function 9-41
exponentmax function 9-42
exponentmin function 9-43

F

fi function 9-44
fi objects
  constructing 3-2
  properties
    bin 9-2
    Data 9-2
    double 9-2
    hex 9-3
    int 9-3
    NumericType 9-3
    oct 9-4

fimath function 9-50
fimath objects 2-13
constructing 4-2
properties
CastBeforeSum 9-5
MaxProductWordLength 9-5
MaxSumWordLength 9-5
OverflowMode 9-5
ProductFractionLength 9-5
ProductMode 9-6
ProductWordLength 9-7
RoundMode 9-7
SumFractionLength 9-7
SumMode 9-7
SumWordLength 9-9
fimath property 9-2
FimathDisplay property 9-10
fiobjects
properties
fimath 9-2
fipref function 9-52
fipref objects
constructing 5-2
properties
FimathDisplay 9-10
NumberDisplay 9-10
NumericTypeDisplay 9-10
FixedExponent property 9-12
fixed-point data
reading from workspace 8-2
writing to workspace 8-2
fixed-point data types
addition 2-10
arithmetic operations 2-8
casts 2-17
complex multiplication 2-11
modular arithmetic 2-8
multiplication 2-11
overflow handling 2-5
precision 2-5
range 2-5
rounding 2-6
saturation 2-5
scaling 2-4
subtraction 2-10
two's complement 2-9
wrapping 2-5
fixed-point run-time API 8-6
fixed-point signal logging 8-5
Format property 9-14
fractionlength function 9-54
FractionLength property 9-12
functions
add 9-14
bin 9-16
bin2num 9-17
bitand 9-19
bitcmp 9-20
bitget 9-21
bitor 9-22
bitset 9-23
bitxor 9-24
complex 9-25
conj 9-26
convergent 9-27
copyobj 9-28
ctranspose 9-29
dec 9-30
denormalmax 9-31
denormalmin 9-32
disp 9-33
div 9-34
double 9-37
eps 9-38

functions, continued
eq 9-39
exponentbias 9-40
exponentlength 9-41
exponentmax 9-42
exponentmin 9-43
fi 9-44
fimath 9-50
fipref 9-52
fractionlength 9-54
ge 9-56
get 9-57
gt 9-58
hex 9-59
hex2num 9-60
horzcat 9-61
imag 9-62
int 9-63
int16 9-65
int32 9-66
int8 9-64
intmax 9-67
iscolumn 9-68
isempty 9-69
isequal 9-70
isfi 9-71
isreal 9-75
isrow 9-76
isscalar 9-77
issigned 9-78
isvector 9-79
le 9-80
length 9-81
loglog 9-82
lsb 9-83
lt 9-84
max 9-85
min 9-87
minus 9-88
mpy 9-89
mtimes 9-91
ndims 9-92
ne 9-93
noperations 9-94
noverflows 9-95
num2bin 9-96
num2hex 9-97
num2int 9-99
numerictype 9-100
nunderflows 9-102
oct 9-103
plot 9-104
plus 9-106
quantize 9-107
quantizer 9-110
randquant 9-113
range 9-115
real 9-117
realmax 9-118
realmin 9-119
repmat 9-120
reset 9-123
reshape 9-124
round 9-125
savefipref 9-127
semilogx 9-128
semilogy 9-129
set 9-130
single 9-131
size 9-132
squeeze 9-133
stripscaling 9-134
sub 9-135
subsasgn 9-137

functions, continued
  subsref 9-138
  times 9-139
  tostring 9-140
  transpose 9-142
  uint16 9-144
  uint32 9-145
  uint8 9-143
  uminus 9-146
  vertcat 9-147
  wordlength 9-148

G
ge function 9-56
get function 9-57
gt function 9-58

H
help
  getting 1-3
hex function 9-59
hex property 9-3
hex2num function 9-60
horzcat function 9-61

I
imag function 9-62
int function 9-63
int property 9-3
int16 function 9-65
int32 function 9-66
int8 function 9-64
interoperability
  fi objects with Filter Design Toolbox 8-11
  fi objects with Signal Processing Blockset 8-7
  fi objects with Simulink 8-2
intmax function 9-67
iscolumn function 9-68
isempty function 9-69
isequal function 9-70
isfi function 9-71
isreal function 9-75
isrow function 9-76
isscalar function 9-77
issigned function 9-78
isvector function 9-79

L
le function 9-80
length function 9-81
loglog function 9-82
lsb function 9-83
lt function 9-84

M
max function 9-85
Max property 9-15
MaxProductWordLength property 9-5
MaxSumWordLength property 9-5
min function 9-87
Min property 9-15
minus function 9-88
Mode property 9-14
modular arithmetic 2-8
mpy function 9-89
mtimes function 9-91
multiplication
  fixed-point 2-11

N

ndims function 9-92
ne function 9-93
NOperations property 9-16
nopnerations function 9-94
noverflows function 9-95
NOverflows property 9-16
num2bin function 9-96
num2hex function 9-97
num2int function 9-99
NumberDisplay property 9-10
numerictype function 9-100
numerictype objects
  constructing 6-2
  properties
    Bias 9-11
    DataType 9-11
    DataTypeMode 9-11
    FixedExponent 9-12
    FractionLength 9-12
    Scaling 9-12
    Signed 9-13
    Slope 9-13
    SlopeAdjustmentFactor 9-13
    WordLength 9-13
NumericType property 9-3
NumericTypeDisplay property 9-10
nunderflows function 9-102
NUnderflows property 9-16

O

oct function 9-103
oct property 9-4
one's complement 2-10
overflow handling 2-5, 2-25
OverflowMode property 9-5, 9-16

P

padding 2-17
plot function 9-104
plus function 9-106
precision
  fixed-point data types 2-5
ProductFractionLength property 9-5
ProductMode property 9-6
ProductWordLength property 9-7
properties
  Bias, numerictype objects 9-11
  bin, fi objects 9-2
  CastBeforeSum, fimath objects 9-5
  Data, fi objects 9-2
  DataType, numerictype objects 9-11
  DataTypeMode, numerictype objects 9-11
  double, fi objects 9-2
  fimath, fi objects 9-2
  FimathDisplay, fipref objects 9-10
  FixedExponent, numerictype objects 9-12
  Format, quantizers 9-14
  FractionLength, numerictype objects 9-12
  hex, fi objects 9-3
  int, fi objects 9-3
  Max, quantizers 9-15
  MaxProductWordLength, fimathobjects 9-5
  MaxSumWordLength, fimath objects 9-5
  Min, quantizers 9-15
  Mode, quantizers 9-14
  NOperations, quantizers 9-16
  NOverflows, quantizers 9-16
  NumberDisplay, fipref objects 9-10
  NumericType, fi objects 9-3
  NumericTypeDisplay, fipref objects 9-10
  NUnderflows, quantizers 9-16
  oct, fi objects 9-4
  OverflowMode, fimath objects 9-5

properties, continued
OverflowMode, quantizers 9-16
ProductFractionLength, fimath objects 9-5
ProductMode, fimath objects 9-6
ProductWordLength, fimath objects 9-7
RoundMode, fimath objects 9-7
RoundMode, quantizers 9-17
Scaling, numerictype objects 9-12
Signed, numerictype objects 9-13
Slope, numerictype objects 9-13
SlopeAdjustmentFactor, numerictype objects 9-13
SumFractionLength, fimath objects 9-7
SumMode, fimath objects 9-7
SumWordLength, fimath objects 9-9
WordLength, numerictype objects 9-13
property values
quantizer objects 7-4

Q

quantize function 9-107
quantizer function 9-110
quantizer objects
constructing 7-2
property values 7-4
quantizers
properties
Format 9-14
Max 9-15
Min 9-15
Mode 9-14
NOperations 9-16
NOverflows 9-16
NUnderflows 9-16
OverflowMode 9-16
RoundMode 9-17

R

randquant function 9-113
range
fixed-point data types 2-5
range function 9-115
reading fixed-point data from workspace 8-2
real function 9-117
realmax function 9-118
realmin function 9-119
repmat function 9-120
reset function 9-123
reshape function 9-124
round function 9-125
rounding
fixed-point data types 2-6
RoundMode property 9-7, 9-17
run-time API
fixed-point data 8-6

S

saturation 2-5
savefipref function 9-127
scaling 2-4
Scaling property 9-12
semilogx function 9-128
semilogy function 9-129
set function 9-130
signal logging
fixed-point 8-5
Signed property 9-13
single function 9-131
size function 9-132
Slope property 9-13
SlopeAdjustmentFactor property 9-13
squeeze function 9-133
stripscaling function 9-134

`sub` function 9-135
`subsasgn` function 9-137
`subsref` function 9-138
`SumFractionLength` property 9-7
`SumMode` property 9-7
`SumWordLength` property 9-9

T
`times` function 9-139
`tostring` function 9-140
`transpose` function 9-142
two's complement 2-9

U
`uint16` function 9-144
`uint32` function 9-145
`uint8` function 9-143
`uminus` function 9-146
unary conversions 2-22

V
`vertcat` function 9-147

W
`wordlength` function 9-148
`WordLength` property 9-13
wrapping
    fixed-point data types 2-5
writing fixed-point data to workspace 8-2