Fixed-Point Toolbox™ 3
Reference

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Fixed-Point Toolbox™ Reference

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Version 1.1 (Release 14SP1)
March 2005	Online only	Version 1.2 (Release 14SP2)
September 2005	Online only	Version 1.3 (Release 14SP3)
October 2005	Second printing	Version 1.3
March 2006	Online only	Version 1.4 (R2006a)
September 2006	Online only	Version 1.5 (R2006b)
March 2007	Online only	Version 2.0 (R2007a)
September 2007	Online only	Revised for Version 2.1 (R2007b)
March 2008	Online only	Revised for Version 2.2 (R2008a)
October 2008	Online only	Revised for Version 2.3 (R2008b)
March 2009	Online only	Revised for Version 2.4 (R2009a)
September 2009	Online only	Revised for Version 3.0 (R2009b)
March 2010	Online only	Revised for Version 3.1 (R2010a)

Contents

1 Property Reference

2 Function Reference

1

Property Reference

fi Object Properties

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

Note The `fimath` properties and `numerictype` properties are also properties of the `fi` object. Refer to "fimath Object Properties" on page 1-4 and "numerictype Object Properties" on page 1-15 for more information.

---

bin

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

data

Numerical real-world value of a `fi` object.

dec

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

double

Real-world value of a `fi` object stored as a MATLAB® `double`.

fimath

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

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

```
        MaxSumWordLength: 128
```

To learn more about `fimath` objects and the global fimath, refer to "Working with fimath Objects". For more information about each of the `fimath` object properties, refer to "fimath Object Properties" on page 1-4.

hex

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

int

Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type. You can also use `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64` to get the stored integer value of a `fi` object in these formats.

NumericType

The `numerictype` object contains all the data type and scaling attributes of a fixed-point object. The `numerictype` object behaves like any MATLAB structure, except that it only lets you set valid values for defined fields. For a table of the possible settings of each field of the structure, see "Valid Values for numerictype Structure Properties" in the *Fixed-Point Toolbox™ User's Guide*.

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

---

oct

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

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 MATLAB factory default value of this property is 1 (true).

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

MaxProductWordLength

Maximum allowable word length for the product data type.

The MATLAB factory default value of this property is 128.

MaxSumWordLength

Maximum allowable word length for the sum data type.

The MATLAB factory 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 MATLAB factory default value of this property is saturate.

ProductBias

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

The MATLAB factory default value of this property is `0`.

ProductFixedExponent

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

$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$. Changing one of these properties changes the others.

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

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

ProductFractionLength

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

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

The MATLAB factory default value of this property is `30`.

ProductMode

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

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

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

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

  $$W_p = \text{specified in the } \texttt{ProductWordLength} \text{ property}$$
  $$F_p = F_a + F_b$$

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

  $$W_p = \text{specified in the } \texttt{ProductWordLength} \text{ property}$$
  $$F_p = W_p - \text{integer length}$$

  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.

  $$W_p = \text{specified in the } \texttt{ProductWordLength} \text{ property}$$
  $$F_p = \text{specified in the } \texttt{ProductFractionLength} \text{ property}$$

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

  $$S_p = \text{specified in the } \texttt{ProductSlope} \text{ property}$$
  $$B_p = \text{specified in the } \texttt{ProductBias} \text{ property}$$

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

The MATLAB factory default value of this property is FullPrecision.

ProductSlope

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

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}.$$

Changing one of these properties changes the others.

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

ProductSlopeAdjustmentFactor

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

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}.$$

Changing one of these properties changes the others.

The MATLAB factory default value of this property is 1.

ProductWordLength

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

The MATLAB factory 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 the nearest even stored integer. This is the least biased rounding method provided by Fixed-Point Toolbox software.
- fix — Round toward zero.
- floor — Round toward negative infinity.
- nearest — Round toward nearest. Ties round toward positive infinity.
- round — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The MATLAB factory default value of this property is nearest.

See "Rounding Methods" in the Fixed-Point Toolbox User's Guide for more information.

SumBias

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

The MATLAB factory default value of this property is 0.

SumFixedExponent

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

$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$. Changing one of these properties changes the others.

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

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

SumFractionLength

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

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

The MATLAB factory default value of this property is `30` .

SumMode

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

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

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

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

  where

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

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

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

  W_s = specified in the `SumWordLength` property

  $F_s = \max(F_a, F_b)$

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

  W_s = specified in the `SumWordLength` property

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

  where

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

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

  W_s = specified in the `SumWordLength` property

  F_s = specified in the `SumFractionLength` property

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

  S_s = specified in the `SumSlope` property

  B_s = specified in the `SumBias` property

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

The MATLAB factory default value of this property is `FullPrecision`.

SumSlope

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

$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$. Changing one of these properties changes the others.

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

SumSlopeAdjustmentFactor

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

$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$. Changing one of these properties changes the others.

The MATLAB factory default value of this property is `1`.

SumWordLength

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

The MATLAB factory default value of this property is `32`.

fipref Object Properties

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

DataTypeOverride

Data type override options for fi objects

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

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

The default value of this property is ForceOff.

FimathDisplay

Display options for the fimath attributes of a fi object

- full — Displays all of the fimath attributes of a fixed-point object
- none — None of the fimath attributes are displayed

The default value of this property is full.

LoggingMode

Logging options for operations performed on fi objects

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

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

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

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

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

The default value of this property of `off`.

NumericTypeDisplay

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

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

The default value of this property is `full`.

NumberDisplay

Display options for the value of a `fi` object

- `bin` — Displays the stored integer value in binary format
- `dec` — Displays the stored integer value in unsigned decimal format

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

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

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

numerictype Object Properties

This section describes the properties associated with `numerictype` objects.

Bias

The bias is part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

DataType

The possible value of the `DataType` property are:

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

The default value of this property is `Fixed`.

DataTypeMode

Data type and scaling associated with the object. The possible values of this property are:

- `boolean` — Built-in `boolean`
- `double` — Built-in `double`

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

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

FixedExponent

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

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

where the slope can be expressed as

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

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

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

FractionLength

Fraction length of the stored integer value of the object, in bits. The fraction length can be any integer value.

This property automatically defaults to the best precision possible based on the value of the word length and the real-world value of the `fi` object.

Scaling

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

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

The default value of this property is `BinaryPoint`.

Signed

Whether the object is signed. The possible values of this property are:

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

The default value of this property is `true`.

Note Although the `Signed` property is still supported, the `Signedness` property always appears in the `numerictype` object display. If you choose to change or set the signedness of your `numerictype` objects using the `Signed` property, MATLAB updates the corresponding value of the `Signedness` property.

Signedness

Whether the object is signed, unsigned, or has an unspecified sign. The possible values of this property are:

- `Signed` — signed
- `Unsigned` — unsigned
- `Auto` — unspecified sign

The default value of this property is `Signed`.

All `numerictype` object properties of a `fi` object must be specified at the time of `fi` object creation. If this property is set to `Auto` at the time of `fi` object creation, the property automatically defaults to `Signed`.

Slope

Slope associated with the object. The slope is part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

SlopeAdjustmentFactor

Slope adjustment associated with the object. The slope adjustment is equivalent to the fractional slope of a fixed-point number. The fractional slope is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

WordLength

Word length of the stored integer value of the object, in bits. The word length can be any positive integer value.

The default value of this property is 16.

quantizer Object Properties

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

DataMode

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

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

The default value of this property is `fixed`.

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

Format

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

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

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

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

OverflowMode

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

- saturate — Overflows saturate.

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

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

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

The default value of this property is `saturate`.

Note Floating-point numbers that extend beyond the dynamic range overflow to ±`inf`.

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

RoundMode

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

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

The default value of this property is `floor`.

2

Function Reference

Bitwise Operations

`bitand`	Bitwise `AND` of two `fi` objects
`bitandreduce`	Bitwise `AND` of consecutive range of bits
`bitcmp`	Bitwise complement of `fi` object
`bitconcat`	Concatenate bits of `fi` objects
`bitget`	Bit at certain position
`bitor`	Bitwise `OR` of two `fi` objects
`bitorreduce`	Bitwise `OR` of consecutive range of bits
`bitreplicate`	Replicate and concatenate bits of `fi` object
`bitrol`	Bitwise rotate left
`bitror`	Bitwise rotate right
`bitset`	Set bit at certain position
`bitshift`	Shift bits specified number of places
`bitsliceget`	Consecutive slice of bits

`bitsll`	Bit shift left logical
`bitsra`	Bit shift right arithmetic
`bitsrl`	Bit shift right logical
`bitxor`	Bitwise exclusive `OR` of two `fi` objects
`bitxorreduce`	Bitwise exclusive `OR` of consecutive range of bits
`getlsb`	Least significant bit
`getmsb`	Most significant bit

Constructors and Properties

`assignmentquantizer`	Assignment `quantizer` object of `fi` object
`copyobj`	Make independent copy of `quantizer` object
`fi`	Construct fixed-point numeric object
`fimath`	Construct `fimath` object
`fipref`	Construct `fipref` object
`get`	Property values of object
`globalfimath`	Configure global fimath and return handle object
`numerictype`	Construct `numerictype` object
`quantizer`	Construct `quantizer` object
`removedefaultfimathpref`	Remove global fimath preference
`removeglobalfimathpref`	Remove global fimath preference
`reset`	Reset objects to initial conditions
`resetdefaultfimath`	Set global fimath to MATLAB factory default

`resetglobalfimath`	Set global fimath to MATLAB factory default
`savedefaultfimathpref`	Save global fimath for next MATLAB session
`savefipref`	Save `fi` preferences for next MATLAB session
`saveglobalfimathpref`	Save global fimath for next MATLAB session
`set`	Set or display property values for `quantizer` objects
`setdefaultfimath`	Set MATLAB global fimath
`sfi`	Construct signed fixed-point numeric object
`tostring`	Convert `numerictype` or `quantizer` object to string
`ufi`	Construct unsigned fixed-point numeric object
`unitquantizer`	Constructor for `unitquantizer` object

Data Manipulation

`denormalmax`	Largest denormalized quantized number for `quantizer` object
`denormalmin`	Smallest denormalized quantized number for `quantizer` object
`eps`	Quantized relative accuracy for `fi` or `quantizer` objects
`exponentbias`	Exponent bias for `quantizer` object
`exponentlength`	Exponent length of `quantizer` object

`exponentmax`	Maximum exponent for `quantizer` object
`exponentmin`	Minimum exponent for `quantizer` object
`fractionlength`	Fraction length of `quantizer` object
`intmax`	Largest positive stored integer value representable by `numerictype` of `fi` object
`intmin`	Smallest stored integer value representable by `numerictype` of `fi` object
`isboolean`	Determine whether input is Boolean
`isdouble`	Determine whether input is double-precision data type
`isequal`	Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isfi`	Determine whether variable is `fi` object
`isfimath`	Determine whether variable is `fimath` object
`isfimathlocal`	Determine whether `fi` object has local fimath
`isfipref`	Determine whether input is `fipref` object
`isfixed`	Determine whether input is fixed-point data type
`isfloat`	Determine whether input is floating-point data type
`isnumerictype`	Determine whether input is `numerictype` object

`ispropequal`	Determine whether properties of two `fi` objects are equal
`isquantizer`	Determine whether input is `quantizer` object
`isscaleddouble`	Determine whether input is scaled double data type
`isscaledtype`	Determine whether input is fixed-point or scaled double data type
`issigned`	Determine whether `fi` object is signed
`issingle`	Determine whether input is single-precision data type
`isslopebiasscaled`	Determine whether `numerictype` object has nontrivial slope and bias
`lowerbound`	Lower bound of range of `fi` object
`lsb`	Scaling of least significant bit of `fi` object, or value of least significant bit of `quantizer` object
`range`	Numerical range of `fi` or `quantizer` object
`realmax`	Largest positive fixed-point value or quantized number
`realmin`	Smallest positive normalized fixed-point value or quantized number
`sort`	Sort elements of real-valued fi object in ascending or descending order
`upperbound`	Upper bound of range of `fi` object
`wordlength`	Word length of `quantizer` object

Data Type Operations

`double`	Double-precision floating-point real-world value of `fi` object
`int`	Smallest built-in integer fitting stored integer value of `fi` object
`int16`	Stored integer value of `fi` object as built-in `int16`
`int32`	Stored integer value of `fi` object as built-in `int32`
`int64`	Stored integer value of `fi` object as built-in `int64`
`int8`	Stored integer value of `fi` object as built-in `int8`
`logical`	Convert numeric values to logical
`reinterpretcast`	Convert fixed-point data types without changing underlying data
`rescale`	Change scaling of `fi` object
`single`	Single-precision floating-point real-world value of `fi` object
`stripscaling`	Stored integer of `fi` object
`uint16`	Stored integer value of `fi` object as built-in `uint16`
`uint32`	Stored integer value of `fi` object as built-in `uint32`
`uint64`	Stored integer value of `fi` object as built-in `uint64`
`uint8`	Stored integer value of `fi` object as built-in `uint8`

Data Type Tools

`NumericTypeScope`	Determine numeric type for data

Data Quantizing

`quantize`	Apply `quantizer` object to data
`randquant`	Generate uniformly distributed, quantized random number using `quantizer` object
`round`	Round `fi` object toward nearest integer or round input data using `quantizer` object
`unitquantize`	Quantize except numbers within `eps` of `+1`
`unitquantizer`	Constructor for `unitquantizer` object

Element-Wise Logical Operators

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

`or`	Find logical OR of array or scalar inputs
`xor`	Logical exclusive-OR

Math Operations

`abs`	Absolute value of `fi` object
`add`	Add two objects using `fimath` object
`ceil`	Round toward positive infinity
`complex`	Construct complex `fi` object from real and imaginary parts
`conj`	Complex conjugate of `fi` object
`conv`	Convolution and polynomial multiplication of `fi` objects
`convergent`	Round toward nearest integer with ties rounding to nearest even integer
`cordiccexp`	CORDIC-based approximation of complex exponential
`cordiccos`	CORDIC-based approximation of cosine
`cordicsin`	CORDIC-based approximation of sine
`cordicsincos`	CORDIC-based approximation of sine and cosine
`divide`	Divide two objects
`filter`	One-dimensional digital filter of fi objects
`fix`	Round toward zero
`floor`	Round toward negative infinity

imag	Imaginary part of complex number
innerprodintbits	Number of integer bits needed for fixed-point inner product
minus	Matrix difference between fi objects
mpower	Fixed-point matrix power (^)
mpy	Multiply two objects using fimath object
mrdivide	Forward slash (/) or right-matrix division
mtimes	Matrix product of fi objects
nearest	Round toward nearest integer with ties rounding toward positive infinity
plus	Matrix sum of fi objects
pow2	Efficient fixed-point multiplication by 2^K
power	Fixed-point array power (.^)
rdivide	Right-array division (./)
real	Real part of complex number
round	Round fi object toward nearest integer or round input data using quantizer object
sign	Perform signum function on array
sqrt	Square root of fi object
sub	Subtract two objects using fimath object
sum	Sum of array elements
times	Element-by-element multiplication of fi objects
uminus	Negate elements of fi object array
uplus	Unary plus

Matrix Manipulation

`buffer`	Buffer signal vector into matrix of data frames
`ctranspose`	Complex conjugate transpose of `fi` object
`diag`	Diagonal matrices or diagonals of matrix
`disp`	Display object
`end`	Last index of array
`flipdim`	Flip array along specified dimension
`fliplr`	Flip matrix left to right
`flipud`	Flip matrix up to down
`hankel`	Hankel matrix
`horzcat`	Horizontally concatenate multiple `fi` objects
`ipermute`	Inverse permute dimensions of multidimensional array
`iscolumn`	Determine whether `fi` object is column vector
`isempty`	Determine whether array is empty
`isfinite`	Determine whether array elements are finite
`isinf`	Determine whether array elements are infinite
`isnan`	Determine whether array elements are NaN
`isnumeric`	Determine whether input is numeric array
`isobject`	Determine whether input is MATLAB object

`isreal`	Determine whether array elements are real
`isrow`	Determine whether `fi` object is row vector
`isscalar`	Determine whether input is scalar
`isvector`	Determine whether input is vector
`length`	Vector length
`ndgrid`	Generate arrays for N-D functions and interpolation
`ndims`	Number of array dimensions
`permute`	Rearrange dimensions of multidimensional array
`repmat`	Replicate and tile array
`reshape`	Reshape array
`shiftdata`	Shift data to operate on specified dimension
`shiftdim`	Shift dimensions
`size`	Array dimensions
`sort`	Sort elements of real-valued fi object in ascending or descending order
`squeeze`	Remove singleton dimensions
`toeplitz`	Create Toeplitz matrix
`transpose`	Transpose operation
`tril`	Lower triangular part of matrix
`triu`	Upper triangular part of matrix
`unshiftdata`	Inverse of `shiftdata`
`vertcat`	Vertically concatenate multiple `fi` objects

Plots

`area`	Create filled area 2-D plot
`bar`	Create vertical bar graph
`barh`	Create horizontal bar graph
`clabel`	Create contour plot elevation labels
`comet`	Create 2-D comet plot
`comet3`	Create 3-D comet plot
`compass`	Plot arrows emanating from origin
`coneplot`	Plot velocity vectors as cones in 3-D vector field
`contour`	Create contour graph of matrix
`contour3`	Create 3-D contour plot
`contourc`	Create two-level contour plot computation
`contourf`	Create filled 2-D contour plot
`errorbar`	Plot error bars along curve
`etreeplot`	Plot elimination tree
`ezcontour`	Easy-to-use contour plotter
`ezcontourf`	Easy-to-use filled contour plotter
`ezmesh`	Easy-to-use 3-D mesh plotter
`ezplot`	Easy-to-use function plotter
`ezplot3`	Easy-to-use 3-D parametric curve plotter
`ezpolar`	Easy-to-use polar coordinate plotter
`ezsurf`	Easy-to-use 3-D colored surface plotter
`ezsurfc`	Easy-to-use combination surface/contour plotter

`feather`	Plot velocity vectors
`fplot`	Plot function between specified limits
`gplot`	Plot set of nodes using adjacency matrix
`hist`	Create histogram plot
`histc`	Histogram count
`line`	Create line object
`loglog`	Create log-log scale plot
`mesh`	Create mesh plot
`meshc`	Create mesh plot with contour plot
`meshz`	Create mesh plot with curtain plot
`patch`	Create patch graphics object
`pcolor`	Create pseudocolor plot
`plot`	Create linear 2-D plot
`plot3`	Create 3-D line plot
`plotmatrix`	Draw scatter plots
`plotyy`	Create graph with y-axes on right and left sides
`polar`	Plot polar coordinates
`quiver`	Create quiver or velocity plot
`quiver3`	Create 3-D quiver or velocity plot
`rgbplot`	Plot colormap
`ribbon`	Create ribbon plot
`rose`	Create angle histogram
`scatter`	Create scatter or bubble plot
`scatter3`	Create 3-D scatter or bubble plot

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

`ylim`	Set or query y-axis limits
`zlim`	Set or query z-axis limits

Radix Conversion

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

Relational Operators

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

Statistics

`errmean`	Mean of quantization error
`errpdf`	Probability density function of quantization error
`errvar`	Variance of quantization error
`logreport`	Quantization report
`max`	Largest element in array of `fi` objects
`maxlog`	Log maximums
`mean`	Average or mean value of fixed-point array
`median`	Median value of fixed-point array

`min`	Smallest element in array of `fi` objects
`minlog`	Log minimums
`noperations`	Number of operations
`noverflows`	Number of overflows
`numberofelements`	Number of data elements in `fi` array
`nunderflows`	Number of underflows
`resetlog`	Clear log for `fi` or `quantizer` object

Subscripted Assignment and Reference

`subsasgn`	Subscripted assignment
`subsref`	Subscripted reference

fi Object Operations

`abs`	Absolute value of `fi` object
`all`	Determine whether all array elements are nonzero
`and`	Find logical AND of array or scalar inputs
`any`	Determine whether any array elements are nonzero
`area`	Create filled area 2-D plot
`assignmentquantizer`	Assignment `quantizer` object of `fi` object
`bar`	Create vertical bar graph
`barh`	Create horizontal bar graph
`bin`	Binary representation of stored integer of `fi` object
`bitand`	Bitwise `AND` of two `fi` objects
`bitandreduce`	Bitwise `AND` of consecutive range of bits
`bitcmp`	Bitwise complement of `fi` object
`bitconcat`	Concatenate bits of `fi` objects
`bitget`	Bit at certain position
`bitor`	Bitwise `OR` of two `fi` objects
`bitorreduce`	Bitwise `OR` of consecutive range of bits
`bitreplicate`	Replicate and concatenate bits of `fi` object
`bitrol`	Bitwise rotate left
`bitror`	Bitwise rotate right
`bitset`	Set bit at certain position

`bitshift`	Shift bits specified number of places
`bitsliceget`	Consecutive slice of bits
`bitsll`	Bit shift left logical
`bitsra`	Bit shift right arithmetic
`bitsrl`	Bit shift right logical
`bitxor`	Bitwise exclusive `OR` of two `fi` objects
`bitxorreduce`	Bitwise exclusive `OR` of consecutive range of bits
`buffer`	Buffer signal vector into matrix of data frames
`ceil`	Round toward positive infinity
`clabel`	Create contour plot elevation labels
`comet`	Create 2-D comet plot
`comet3`	Create 3-D comet plot
`compass`	Plot arrows emanating from origin
`complex`	Construct complex `fi` object from real and imaginary parts
`coneplot`	Plot velocity vectors as cones in 3-D vector field
`conj`	Complex conjugate of `fi` object
`contour`	Create contour graph of matrix
`contour3`	Create 3-D contour plot
`contourc`	Create two-level contour plot computation
`contourf`	Create filled 2-D contour plot
`conv`	Convolution and polynomial multiplication of `fi` objects
`convergent`	Round toward nearest integer with ties rounding to nearest even integer

`cordiccexp`	CORDIC-based approximation of complex exponential
`cordiccos`	CORDIC-based approximation of cosine
`cordicsin`	CORDIC-based approximation of sine
`cordicsincos`	CORDIC-based approximation of sine and cosine
`ctranspose`	Complex conjugate transpose of `fi` object
`dec`	Unsigned decimal representation of stored integer of `fi` object
`diag`	Diagonal matrices or diagonals of matrix
`disp`	Display object
`double`	Double-precision floating-point real-world value of `fi` object
`end`	Last index of array
`eps`	Quantized relative accuracy for `fi` or `quantizer` objects
`eq`	Determine whether real-world values of two `fi` objects are equal
`errorbar`	Plot error bars along curve
`etreeplot`	Plot elimination tree
`ezcontour`	Easy-to-use contour plotter
`ezcontourf`	Easy-to-use filled contour plotter
`ezmesh`	Easy-to-use 3-D mesh plotter
`ezplot`	Easy-to-use function plotter
`ezplot3`	Easy-to-use 3-D parametric curve plotter
`ezpolar`	Easy-to-use polar coordinate plotter

`ezsurf`	Easy-to-use 3-D colored surface plotter
`ezsurfc`	Easy-to-use combination surface/contour plotter
`feather`	Plot velocity vectors
`fi`	Construct fixed-point numeric object
`filter`	One-dimensional digital filter of fi objects
`fimath`	Construct `fimath` object
`fix`	Round toward zero
`flipdim`	Flip array along specified dimension
`fliplr`	Flip matrix left to right
`flipud`	Flip matrix up to down
`floor`	Round toward negative infinity
`fplot`	Plot function between specified limits
`ge`	Determine whether real-world value of one `fi` object is greater than or equal to another
`get`	Property values of object
`getlsb`	Least significant bit
`getmsb`	Most significant bit
`gplot`	Plot set of nodes using adjacency matrix
`gt`	Determine whether real-world value of one `fi` object is greater than another
`hankel`	Hankel matrix
`hex`	Hexadecimal representation of stored integer of `fi` object

`hist`	Create histogram plot
`histc`	Histogram count
`horzcat`	Horizontally concatenate multiple `fi` objects
`imag`	Imaginary part of complex number
`innerprodintbits`	Number of integer bits needed for fixed-point inner product
`int`	Smallest built-in integer fitting stored integer value of `fi` object
`int16`	Stored integer value of `fi` object as built-in `int16`
`int32`	Stored integer value of `fi` object as built-in `int32`
`int64`	Stored integer value of `fi` object as built-in `int64`
`int8`	Stored integer value of `fi` object as built-in `int8`
`intmax`	Largest positive stored integer value representable by `numerictype` of `fi` object
`intmin`	Smallest stored integer value representable by `numerictype` of `fi` object
`ipermute`	Inverse permute dimensions of multidimensional array
`isboolean`	Determine whether input is Boolean
`iscolumn`	Determine whether `fi` object is column vector
`isdouble`	Determine whether input is double-precision data type
`isempty`	Determine whether array is empty

`isequal`	Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isfi`	Determine whether variable is `fi` object
`isfimathlocal`	Determine whether `fi` object has local fimath
`isfinite`	Determine whether array elements are finite
`isfixed`	Determine whether input is fixed-point data type
`isfloat`	Determine whether input is floating-point data type
`isinf`	Determine whether array elements are infinite
`isnan`	Determine whether array elements are NaN
`isnumeric`	Determine whether input is numeric array
`isobject`	Determine whether input is MATLAB object
`ispropequal`	Determine whether properties of two `fi` objects are equal
`isreal`	Determine whether array elements are real
`isrow`	Determine whether `fi` object is row vector
`isscalar`	Determine whether input is scalar
`isscaleddouble`	Determine whether input is scaled double data type

`isscaledtype`	Determine whether input is fixed-point or scaled double data type
`issigned`	Determine whether `fi` object is signed
`issingle`	Determine whether input is single-precision data type
`isvector`	Determine whether input is vector
`le`	Determine whether real-world value of `fi` object is less than or equal to another
`length`	Vector length
`line`	Create line object
`logical`	Convert numeric values to logical
`loglog`	Create log-log scale plot
`logreport`	Quantization report
`lowerbound`	Lower bound of range of `fi` object
`lsb`	Scaling of least significant bit of `fi` object, or value of least significant bit of `quantizer` object
`lt`	Determine whether real-world value of one `fi` object is less than another
`max`	Largest element in array of `fi` objects
`maxlog`	Log maximums
`mean`	Average or mean value of fixed-point array
`median`	Median value of fixed-point array
`mesh`	Create mesh plot
`meshc`	Create mesh plot with contour plot
`meshz`	Create mesh plot with curtain plot

`min`	Smallest element in array of `fi` objects
`minlog`	Log minimums
`minus`	Matrix difference between `fi` objects
`mpower`	Fixed-point matrix power (^)
`mrdivide`	Forward slash (/) or right-matrix division
`mtimes`	Matrix product of `fi` objects
`ndgrid`	Generate arrays for N-D functions and interpolation
`ndims`	Number of array dimensions
`ne`	Determine whether real-world values of two `fi` objects are not equal
`nearest`	Round toward nearest integer with ties rounding toward positive infinity
`not`	Find logical NOT of array or scalar input
`noverflows`	Number of overflows
`numberofelements`	Number of data elements in `fi` array
`numerictype`	Construct `numerictype` object
`nunderflows`	Number of underflows
`oct`	Octal representation of stored integer of `fi` object
`or`	Find logical OR of array or scalar inputs
`patch`	Create patch graphics object
`pcolor`	Create pseudocolor plot
`permute`	Rearrange dimensions of multidimensional array
`plot`	Create linear 2-D plot

`plot3`	Create 3-D line plot
`plotmatrix`	Draw scatter plots
`plotyy`	Create graph with y-axes on right and left sides
`plus`	Matrix sum of `fi` objects
`polar`	Plot polar coordinates
`pow2`	Efficient fixed-point multiplication by 2^K
`power`	Fixed-point array power (.^)
`quantizer`	Construct `quantizer` object
`quiver`	Create quiver or velocity plot
`quiver3`	Create 3-D quiver or velocity plot
`range`	Numerical range of `fi` or `quantizer` object
`rdivide`	Right-array division (./)
`real`	Real part of complex number
`realmax`	Largest positive fixed-point value or quantized number
`realmin`	Smallest positive normalized fixed-point value or quantized number
`reinterpretcast`	Convert fixed-point data types without changing underlying data
`repmat`	Replicate and tile array
`rescale`	Change scaling of `fi` object
`resetlog`	Clear log for `fi` or `quantizer` object
`reshape`	Reshape array
`rgbplot`	Plot colormap
`ribbon`	Create ribbon plot

`rose`	Create angle histogram
`round`	Round `fi` object toward nearest integer or round input data using `quantizer` object
`scatter`	Create scatter or bubble plot
`scatter3`	Create 3-D scatter or bubble plot
`sdec`	Signed decimal representation of stored integer of `fi` object
`semilogx`	Create semilogarithmic plot with logarithmic x-axis
`semilogy`	Create semilogarithmic plot with logarithmic y-axis
`sfi`	Construct signed fixed-point numeric object
`shiftdata`	Shift data to operate on specified dimension
`shiftdim`	Shift dimensions
`sign`	Perform signum function on array
`single`	Single-precision floating-point real-world value of `fi` object
`size`	Array dimensions
`slice`	Create volumetric slice plot
`sort`	Sort elements of real-valued fi object in ascending or descending order
`spy`	Visualize sparsity pattern
`sqrt`	Square root of `fi` object
`squeeze`	Remove singleton dimensions
`stairs`	Create stairstep graph
`stem`	Plot discrete sequence data
`stem3`	Plot 3-D discrete sequence data

`streamribbon`	Create 3-D stream ribbon plot
`streamslice`	Draw streamlines in slice planes
`streamtube`	Create 3-D stream tube plot
`stripscaling`	Stored integer of `fi` object
`subsasgn`	Subscripted assignment
`subsref`	Subscripted reference
`sum`	Sum of array elements
`surf`	Create 3-D shaded surface plot
`surfc`	Create 3-D shaded surface plot with contour plot
`surfl`	Create surface plot with colormap-based lighting
`surfnorm`	Compute and display 3-D surface normals
`text`	Create text object in current axes
`times`	Element-by-element multiplication of `fi` objects
`toeplitz`	Create Toeplitz matrix
`transpose`	Transpose operation
`treeplot`	Plot picture of tree
`tril`	Lower triangular part of matrix
`trimesh`	Create triangular mesh plot
`triplot`	Create 2-D triangular plot
`trisurf`	Create triangular surface plot
`triu`	Upper triangular part of matrix
`ufi`	Construct unsigned fixed-point numeric object
`uint16`	Stored integer value of `fi` object as built-in `uint16`

`uint32`	Stored integer value of `fi` object as built-in `uint32`
`uint64`	Stored integer value of `fi` object as built-in `uint64`
`uint8`	Stored integer value of `fi` object as built-in `uint8`
`uminus`	Negate elements of `fi` object array
`unshiftdata`	Inverse of `shiftdata`
`uplus`	Unary plus
`upperbound`	Upper bound of range of `fi` object
`vertcat`	Vertically concatenate multiple `fi` objects
`voronoi`	Create Voronoi diagram
`voronoin`	Create n-D Voronoi diagram
`waterfall`	Create waterfall plot
`xlim`	Set or query x-axis limits
`xor`	Logical exclusive-OR
`ylim`	Set or query y-axis limits
`zlim`	Set or query z-axis limits

fimath Object Operations

`add`	Add two objects using `fimath` object
`disp`	Display object
`fimath`	Construct `fimath` object
`globalfimath`	Configure global fimath and return handle object
`isequal`	Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isfimath`	Determine whether variable is `fimath` object
`mpy`	Multiply two objects using `fimath` object
`removedefaultfimathpref`	Remove global fimath preference
`removeglobalfimathpref`	Remove global fimath preference
`resetdefaultfimath`	Set global fimath to MATLAB factory default
`resetglobalfimath`	Set global fimath to MATLAB factory default
`savedefaultfimathpref`	Save global fimath for next MATLAB session
`saveglobalfimathpref`	Save global fimath for next MATLAB session
`setdefaultfimath`	Set MATLAB global fimath
`sqrt`	Square root of `fi` object
`sub`	Subtract two objects using `fimath` object

fipref Object Operations

`disp`	Display object
`fipref`	Construct `fipref` object
`isfipref`	Determine whether input is `fipref` object
`reset`	Reset objects to initial conditions
`savefipref`	Save `fi` preferences for next MATLAB session

numerictype Object Operations

`disp`	Display object
`divide`	Divide two objects
`isboolean`	Determine whether input is Boolean
`isdouble`	Determine whether input is double-precision data type
`isequal`	Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal
`isfixed`	Determine whether input is fixed-point data type
`isfloat`	Determine whether input is floating-point data type
`isnumerictype`	Determine whether input is `numerictype` object
`isscaleddouble`	Determine whether input is scaled double data type
`isscaledtype`	Determine whether input is fixed-point or scaled double data type
`issingle`	Determine whether input is single-precision data type
`isslopebiasscaled`	Determine whether `numerictype` object has nontrivial slope and bias
`sqrt`	Square root of `fi` object
`tostring`	Convert `numerictype` or `quantizer` object to string

quantizer Object Operations

`bin2num`	Convert two's complement binary string to number using `quantizer` object
`copyobj`	Make independent copy of `quantizer` object
`denormalmax`	Largest denormalized quantized number for `quantizer` object
`denormalmin`	Smallest denormalized quantized number for `quantizer` object
`disp`	Display object
`eps`	Quantized relative accuracy for `fi` or `quantizer` objects
`errmean`	Mean of quantization error
`errpdf`	Probability density function of quantization error
`errvar`	Variance of quantization error
`exponentbias`	Exponent bias for `quantizer` object
`exponentlength`	Exponent length of `quantizer` object
`exponentmax`	Maximum exponent for `quantizer` object
`exponentmin`	Minimum exponent for `quantizer` object
`fractionlength`	Fraction length of `quantizer` object
`get`	Property values of object
`hex2num`	Convert hexadecimal string to number using `quantizer` object

isequal	Determine whether real-world values of two fi objects are equal, or determine whether properties of two fimath, numerictype, or quantizer objects are equal
isfixed	Determine whether input is fixed-point data type
isfloat	Determine whether input is floating-point data type
isquantizer	Determine whether input is quantizer object
length	Vector length
lsb	Scaling of least significant bit of fi object, or value of least significant bit of quantizer object
max	Largest element in array of fi objects
maxlog	Log maximums
min	Smallest element in array of fi objects
minlog	Log minimums
noperations	Number of operations
noverflows	Number of overflows
num2bin	Convert number to binary string using quantizer object
num2hex	Convert number to hexadecimal equivalent using quantizer object
num2int	Convert number to signed integer
nunderflows	Number of underflows
quantize	Apply quantizer object to data
quantizer	Construct quantizer object

`randquant`	Generate uniformly distributed, quantized random number using `quantizer` object
`range`	Numerical range of `fi` or `quantizer` object
`realmax`	Largest positive fixed-point value or quantized number
`realmin`	Smallest positive normalized fixed-point value or quantized number
`reset`	Reset objects to initial conditions
`resetlog`	Clear log for `fi` or `quantizer` object
`round`	Round `fi` object toward nearest integer or round input data using `quantizer` object
`set`	Set or display property values for `quantizer` objects
`tostring`	Convert `numerictype` or `quantizer` object to string
`unitquantize`	Quantize except numbers within `eps` of `+1`
`unitquantizer`	Constructor for `unitquantizer` object
`wordlength`	Word length of `quantizer` object

3

Functions — Alphabetical List

Purpose Absolute value of `fi` object

Syntax

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

Description

`c = abs(a)` returns the absolute value of `fi` object `a` with the same `numerictype` object as `a`. Intermediate quantities are calculated using the `fimath` associated with `a`.

`c = abs(a,T)` returns a `fi` object with a value equal to the absolute value of `a` and `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` associated with `a`. See "Data Type Propagation Rules" on page 3-3.

`c = abs(a,F)` returns a `fi` object with a value equal to the absolute value of `a` and the same `numerictype` object as `a`. Intermediate quantities are calculated using the `fimath` object `F`, and the output `fi` object `c` is always associated with the global fimath.

`c = abs(a,T,F)` returns a `fi` object with a value equal to the absolute value of `a` and the `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` object `F`, and the output `fi` object `c` is always associated with the global fimath. See "Data Type Propagation Rules" on page 3-3.

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

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

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

Data Type Propagation Rules

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

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

Examples

Example 1

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

```
P = fipref('NumericTypeDisplay','full',...
           'FimathDisplay','full');
a = fi(-128)

a =
```

```
  -128

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

abs(a)

ans =

  127.9961

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

a.OverflowMode = 'wrap'

a =

  -128

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

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

```
abs(a)

ans =

  -128

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

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

Example 2

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

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

re =

    -1

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

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

```
im =

     0

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

a = complex(re,im)

a =

    -1

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

abs(a,re.numerictype,fimath('overflowmode','wrap'))

ans =

    1.0000

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

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

ans =

    -1
```

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

Example 3

The following example shows how to specify `numerictype` and `fimath` objects as optional arguments to control the result of the `abs` function for real inputs. When you specify a `fimath` object as an argument, that `fimath` object is used to compute intermediate quantities, and the resulting `fi` object is always associated with the global fimath.

```
a = fi(-1,1,6,5,'overflowmode','wrap')

a =

    -1

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

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

abs(a)

ans =

    -1
```

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

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

f = fimath('overflowmode','saturate')

f =


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

abs(a,f)

ans =

    0.9688

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

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

```
t =


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

abs(a,t,f)

ans =

     1

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

Example 4

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

```
a = fi(-1-i,1,16,15,'overflowmode','wrap')

a =

  -1.0000 - 1.0000i

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

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

t = numerictype(a.numerictype,'signed',false)

t =


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

abs(a,t)

ans =

    1.4142

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

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

f = fimath('overflowmode','saturate','summode',...
```

```
        'keepLSB','sumwordlength',a.wordlength,...
        'productmode','specifyprecision',...
        'productwordlength',a.wordlength,...
        'productfractionlength',a.fractionlength)

f =


             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 16
 ProductFractionLength: 15
               SumMode: KeepLSB
         SumWordLength: 16
         CastBeforeSum: true

abs(a,t,f)

ans =

    1.4142

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

Algorithm

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

y = a if a >= 0

y = -a if a < 0

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

```
y = sqrt(real(a)*real(a) + imag(a)*imag(a))
```

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

1. Calculate the real and imaginary parts of `a` using the following equations:

   ```
   re = real(a)

   im = imag(a)
   ```

2. Compute the squares of `re` and `im` using one of the following objects:
   - The `fimath` object `F` if `F` is specified as an argument.
   - The `fimath` associated with `a` if `F` is not specified as an argument.
3. Cast the squares of `re` and `im` to unsigned types if the input is signed.
4. Add the squares of `re` and `im` using one of the following objects:
   - The `fimath` object `F` if `F` is specified as an argument.
   - The `fimath` object associated with `a` if `F` is not specified as an argument.
5. Compute the square root of the sum computed in step four using the `sqrt` function with the following additional arguments:
   - The `numerictype` object `T` if `T` is specified, or the `numerictype` object of `a` otherwise.
   - The `fimath` object `F` if `F` is specified, or the `fimath` object associated with `a` otherwise.

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

Purpose

Add two objects using `fimath` object

Syntax

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

Description

`c = F.add(a,b)` adds objects `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` is always associated with the global fimath.

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

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

Examples

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

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

c =

    5.8599


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

Algorithm

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

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

```
c = a + b
c =
    5.8599

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

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

but not identical. When you use `add`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` is associated with the global fimath. When you use the syntax `c = a + b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See "fimath Rules for Fixed-Point Arithmetic" in the *Fixed-Point Toolbox User's Guide* for more information.

See Also

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

Purpose Determine whether all array elements are nonzero

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

Purpose Find logical AND of array or scalar inputs

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

Purpose Determine whether any array elements are nonzero

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

Purpose Create filled area 2-D plot

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

Purpose Assignment `quantizer` object of `fi` object

Syntax

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

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

See Also `quantize`, `quantizer`

Purpose Create vertical bar graph

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

Purpose Create horizontal bar graph

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

Purpose Binary representation of stored integer of `fi` object

Syntax

```
bin(a)
```

Description

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

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

Examples

The following code

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

returns

```
y =

  10000000   01111111

z =

  10000000   01111111
```

See Also `dec`, `hex`, `int`, `oct`

bin2num

Purpose

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

Syntax

`y = bin2num(q,b)`

Description

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

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

Examples

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

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

b =

0111
0110
0101
0100
0011
0010
0001
0000
1111
1110
1101
```

```
1100
1011
1010
1001
1000
```

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

```
y=bin2num(q,b)

y =

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

See Also

`hex2num`, `num2bin`, `num2hex`, `num2int`

bitand

Purpose

Bitwise AND of two fi objects

Syntax

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

Description

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

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

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

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

See Also

bitcmp, bitget, bitor, bitset, bitxor

Purpose

Bitwise AND of consecutive range of bits

Syntax

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

Description

`c = bitandreduce(a)` performs a bitwise AND operation on the entire set of bits in the `fi` object `a` and returns the result as a `u1,0` (unsigned integer of word length 1).

`c = bitandreduce(a, lidx)` performs a bitwise AND operation on a consecutive range of bits starting at position `lidx` and ending at the LSB (the bit at position 1). `lidx` is a constant that represents the position in the range closest to the MSB.

`c = bitandreduce(a, lidx, ridx)` performs a bitwise AND operation on a consecutive range of bits starting at position `lidx` and ending at position `ridx`. `ridx` is a constant that represents the position in the range closest to the LSB.

The `bitandreduce` arguments must satisfy the following condition:

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

`a` can be a scalar `fi` object or a vector `fi` object.

`bitandreduce` only supports `fi` objects with fixed-point data types; it does not support inputs with complex data types.

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

Example

This example shows how to perform a bitwise AND operation on a range of bits of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value `5`, word length `4`, and fraction length `0`:

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

```
disp(bin(a))

0101
```

Get the bitwise AND of the consecutive set of bits starting at position 2 and ending at position 1:

```
disp(bin(bitandreduce(a,2,1)))

0
```

See Also

bitconcat, bitorreduce, bitsliceget, bitxorreduce

Purpose

Bitwise complement of `fi` object

Syntax

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

Description

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

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

Example

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

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

1010
```

Complement the values of the bits in `a`:

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

0101
```

See Also

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

bitconcat

Purpose

Concatenate bits of `fi` objects

Syntax

```
y = bitconcat(a, b)
y = bitconcat([a, b, c])
y = bitconcat(a, b, c, d, ...)
```

Description

`y = bitconcat(a, b)` concatenates the bits in the `fi` objects `a` and `b`.

`a` and `b` can both be vectors if the vectors are the same size. If `a` and `b` are vectors, `bitconcat` performs element-wise concatenation. `bitconcat` only supports vector input when both `a` and `b` are vectors.

`y = bitconcat([a, b, c])` performs element-wise concatenation of the bits of `fi` objects `a, b,` and `c`, as given by the input vector.

`y = bitconcat(a, b, c, d, ...)` concatenates the bits of the `fi` objects `a, b, c, d, ...`.

`bitconcat` returns an unsigned fixed value with a word length equal to the sum of the word lengths of the input objects and a fraction length of zero. The bit representation of the stored integer is in two's complement representation.

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

`bitconcat` only supports `fi` objects with fixed-point data types. `bitconcat` does not support inputs with complex data types. Scaling does not affect the result type and value. `bitconcat` accepts `varargin` number of inputs for concatenation.

Example

This example shows how to get the binary representation of the concatenated bits of two `fi` objects. Consider the following unsigned fixed-point `fi` objects. The first has a value of 5, word length 4, and fraction length 0. The second has a value of 10, word length 4, and fraction length 0:

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

```
0101

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

1010
```

Concatenate the objects:

```
c = bitconcat(a,b);
disp(bin(c))

01011010
```

See Also

`bitand`, `bitcmp`, `bitor`, `bitreplicate`, `bitset`, `bitsliceget`, `bitxor`

bitget

Purpose

Bit at certain position

Syntax

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

Description

`c = bitget(a, bit)` returns the value of the bit at position `bit` in `a` as a `u1,0` (unsigned integer of word length 1). `bit` must be an integer between `1` and the word length of `a`, inclusive. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

`bitget` only supports `fi` objects with fixed-point data types. `bitget` does not support inputs with complex data types.

`bitget` supports variable indexing. This means that `bit` can be a variable instead of a constant.

`a` and `bit` can be vectors or scalars. `a` and `bit` must be the same size unless one is a scalar. If `a` is a vector and `bit` is a scalar, `c` is a vector of `u1,0` values of the bits at position `bit` in each `fi` object in `a`. If `a` is a scalar and `bit` is a vector, `c` is a vector of `u1,0` values of the bits in `a` at the positions specified in `bit`.

`bit` does not need to be a vector of sequential bit positions.

Examples

Example 1

This example shows how to get the binary representation of the bit at a specific position in a `fi` object. Consider the following unsigned fixed-point `fi` object with a value of 85, word length 8, and fraction length 0:

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

01010101
```

Get the binary representation of the bit at position 4:

```
bit4 = bitget(a,4);
disp(bin(bit4))
```

```
0
```

Example 2

This example shows how to get the binary representation of the bits at a vector of positions in a `fi` object. Consider the following signed fixed-point `fi` object with a value of 55, word length 16, and best-precision fraction length 9:

```
a = fi(55);
disp(bin(a))

0110111000000000
```

Get the binary representation of the bits at positions 16, 14, 12, 10, 8, 6, 4, and 2:

```
bitvec = bitget(a,[16:-2:1]);
disp(bin(bitvec))

0 1 1 1 0 0 0 0
```

See Also

`bitand`, `bitcmp`, `bitor`, `bitset`, `bitxor`

bitor

Purpose

Bitwise OR of two fi objects

Syntax

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

Description

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

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

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

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

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

Examples

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

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

c =

   -18

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

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

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

binary_a =

100010

binary_b =

001100

binary_c =

101110
```

See Also

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

bitorreduce

Purpose

Bitwise `OR` of consecutive range of bits

Syntax

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

Description

`c = bitorreduce(a)` performs a bitwise `OR` operation on the entire set of bits in the `fi` object `a` and returns the result as a `u1,0` (unsigned integer of word length 1).

`c = bitorreduce(a, lidx)` performs a bitwise `OR` operation on a consecutive range of bits starting at position `lidx` and ending at the LSB (the bit at position 1). `lidx` is a constant that represents the position in the range closest to the MSB.

`c = bitorreduce(a, lidx, ridx)` performs a bitwise `OR` operation on a consecutive range of bits starting at position `lidx` and ending at position `ridx`. `ridx` is a constant that represents the position in the range closest to the LSB.

The `bitorreduce` arguments must satisfy the following condition:

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

`a` can be a scalar `fi` object or a vector `fi` object.

`bitorreduce` only supports `fi` objects with fixed-point data types; it does not support inputs with complex data types.

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

Example

This example shows how to perform a bitwise `OR` operation on a range of bits of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0:

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

```
disp(bin(a))

0101
```

Get the bitwise `OR` of the consecutive set of bits starting at position 4 and ending at position 3:

```
disp(bin(bitorreduce(a,4,3)))

1
```

See Also

`bitandreduce`, `bitconcat`, `bitsliceget`, `bitxorreduce`

bitreplicate

Purpose Replicate and concatenate bits of `fi` object

Syntax

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

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

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

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

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

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

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

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

MATLAB returns the following:

```
a_binary =

001110


c_binary =

001110001110
```

See Also

`bitand`, `bitconcat`, `bitget`, `bitset`, `bitor`, `bitsliceget`, `bitxor`

bitrol

Purpose

Bitwise rotate left

Syntax

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

Description

`c = bitrol(a, k)` returns the value of the `fi` object `a` rotated left by `k` bits.

`a` can be a scalar `fi` object or a vector `fi` object. It can be any fixed-point numeric type. The `OverflowMode` and `RoundMode` properties are ignored. `bitrol` operates on both signed and unsigned fixed point inputs and does not check overflow or underflow. `bitrol` rotates bits from the MSB side into the LSB side.

`k` is an integer constant that must be greater than zero. `k` can be greater than the word length of `a`. It is always normalized to `mod(a.WordLength,k)`.

`a` and `c` have the same `fimath` and the `numerictype` objects.

Example

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

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

1010
```

Rotate `a` left one bit:

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

0101
```

Rotate `a` left two bits:

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

1010
```

See Also `bitconcat`, `bitror`, `bitshift`, `bitsliceget`, `bitsll`, `bitsra`, `bitsrl`

bitror

Purpose

Bitwise rotate right

Syntax

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

Description

`c = bitror(a, k)` returns the value of the `fi` object `a` rotated right by `k` bits.

`a` can be a scalar `fi` object or a vector `fi` object. It can be any fixed-point numeric type. The `OverflowMode` and `RoundMode` properties are ignored. `bitror` operates on both signed and unsigned fixed point inputs and does not check overflow or underflow. `bitror` rotates bits from the LSB side into the MSB side.

`k` is an integer constant that must be greater than zero. `k` can be greater than the word length of `a`. It is always normalized to `mod(a.WordLength,k)`.

`a` and `c` have the same `fimath` and the `numerictype` objects.

Example

This example shows how to rotate the bits of a `fi` object right. Consider the following unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0:

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

0101
```

Rotate `a` right one bit:

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

1010
```

Rotate `a` right two bits:

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

0101
```

See Also

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

bitset

Purpose

Set bit at certain position

Syntax

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

Description

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

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

`bit` must be a number between `1` and the word length of `a`, inclusive. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

`bitset` only supports `fi` objects with fixed-point data types. `a` can be a scalar `fi` object or a vector `fi` object. `bit` and `v` can be scalars or vectors.

Example

This example shows how to set a bit of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value of 5, word length 4, and fraction length 0:

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

0101
```

Set the bit at position 2 to 1:

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

0111
```

See Also

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

Purpose

Shift bits specified number of places

Syntax

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

Description

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

The `OverflowMode` property of `a` is obeyed, but the `RoundMode` is always `floor`. If obeying the `RoundMode` property of `a` is important, try using the `pow2` function.

When the overflow mode is `saturate` the sign bit is always preserved. The sign bit is also preserved when the overflow mode is `wrap`, and `k` is negative. When the overflow mode is `wrap` and `k` is positive, the sign bit is not preserved.

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

Example

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

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

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

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

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

Now change `OverflowMode` to `wrap`. In this case, most significant bits shift off the "top" of `a` until the value is zero:

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

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

```
11. 0001100000000000
12. 0011000000000000
13. 0110000000000000
14. 1100000000000000
15. 1000000000000000
16. 0000000000000000
```

See Also

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

bitsliceget

Purpose

Consecutive slice of bits

Syntax

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

Description

`c = bitsliceget(a)` returns the entire set of bits in the `fi` object `a`. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

`c = bitsliceget(a, lidx)` returns a consecutive slice of bits from `a` starting at position `lidx` and ending at the LSB (the bit at position 1). `lidx` is a constant that represents the position in the slice that is closest to the MSB.

`c = bitsliceget(a, lidx, ridx)` returns a consecutive slice of bits from `a` starting at position `lidx` and ending at position `ridx`. `ridx` is a constant that represents the position in the slice that is closest to the LSB.

The `bitsliceget` arguments must satisfy the following condition:

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

If `lidx` and `ridx` are equal, `bitsliceget` only slices one bit, and `bitsliceget(a, lidx, ridx)` is the same as `bitget(a, lidx)`.

`bitsliceget` only supports `fi` objects with fixed-point data types. `bitsliceget` always returns a fixed point number with no scaling and with word length equal to slice length, `lidx-ridx+1`.

Example

This example shows how to get the binary representation of a specified set of consecutive bits in a `fi` object. Consider the following unsigned fixed-point `fi` object with a value of 85, word length 8, and fraction length 0:

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

```
01010101
```

Get the binary representation of the consecutive set of bits starting at position 8 and ending at position 3:

```
bits8to3 = bitsliceget(a,8,3);
disp(bin(bits8to3))

010101
```

See Also

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

bitsll

Purpose

Bit shift left logical

Syntax

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

Description

`c = bitsll(a, k)` returns the value of the input operand `a` shifted left logical by `k` bits.

The input operand `a` can be a built-in integer or a `fi` object with a fixed-point data type. For fixed-point operations, the `OverflowMode` and `RoundMode` properties are ignored. `bitsll` operates on both signed and unsigned inputs and does not check overflow or underflow. `bitsll` shifts zeros into the positions of bits that it shifts left.

`k` is an integer constant in the following range:

```
a.WordLength > k >= 0
```

`a` and `c` have the same associated `fimath` and `numerictype` objects.

Example

This example shows how to shift bits using the `bitsll` function. Consider the following unsigned fixed-point `fi` object with a value of 10, word length 4, and fraction length 0:

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

1010
```

Shift `a` left by one bit:

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

0100
```

Shift `a` left by one more bit:

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

```
1000
```

Unlike the `bitshift` function, the output value does not saturate.

The `bitsll` function also supports built-in integer inputs. The following example shows the `uint8` input being shifted left by four bits:

```
x = uint8(50);
bitsll(x,4)

ans =
   32
```

See Also

`bitconcat`, `bitrol`, `bitror`, `bitshift`, `bitsliceget`, `bitsra`, `bitsrl`, `pow2`

bitsra

Purpose

Bit shift right arithmetic

Syntax

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

Description

`c = bitsra(a, k)` performs an arithmetic right shift by `k` bits on input operand `a`.

`a` can be any numeric type, including double, single, integer, or fixed-point. For fixed-point operations, the `OverflowMode` and `RoundMode` properties are ignored. `bitsra` operates on both signed and unsigned inputs and does not check overflow or underflow. `bitsra` shifts zeros into the positions of bits that it shifts right if the input is unsigned. `bitsra` shifts the MSB into the positions of bits that it shifts right if the input is signed.

`k` is an integer constant in the following range:

```
a.WordLength > k >= 0
```

`a` and `c` have the same associated `fimath` and `numerictype` objects.

Example

This example shows how to shift bits using the `bitsra` function. Consider the following signed fixed-point `fi` object with a value of -8, word length 4, and fraction length 0:

```
a = fi(-8,1,4,0);
disp(bin(a))

1000
```

Shift `a` right by one bit:

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

1100
```

`bitsra` shifts the MSB into the position of the bit that it shifts right.

The `bitsra` function also supports built-in integer inputs. For example, you can use `bitsra` to shift the `int8` input right by two bits:

```
x = int8(64);
bitsra(x,2)

ans =
   16
```

You can also use `bitsra` with floating-point inputs. The following example shifts the `double` input right by three bits:

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

ans =
    16
```

See Also

`bitconcat`, `bitshift`, `bitsliceget`, `bitsll`, `bitsrl`, `pow2`

Purpose

Bit shift right logical

Syntax

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

Description

`c = bitsrl(a, k)` returns the value of `a` shifted right logical by `k` bits.

The input operand `a` can be a built-in integer or a `fi` object with a fixed-point data type. For fixed-point operations, the `OverflowMode` and `RoundMode` properties are ignored. `bitsrl` operates on both signed and unsigned inputs and does not check overflow or underflow. `bitsrl` shifts zeros into the positions of bits that it shifts right.

`k` is an integer constant in the following range:

```
a.WordLength > k >= 0
```

`a` and `c` have the same associated `fimath` and `numerictype` objects.

Example

This example shows how to shift bits using the `bitsrl` function. Consider the following signed fixed-point `fi` object with a value of -8, word length 4, and fraction length 0:

```
a = fi(-8,1,4,0);
disp(bin(a))

1000
```

Shift `a` right by one bit:

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

0100
```

`bitsrl` shifts a zero into the position of the bit that it shifts right.

The `bitsrl` function also supports built-in integer inputs. The following example shows the `uint8` input being shifted right by two bits:

```
x = uint8(64);
```

```
bitsrl(x,2)

ans =
   16
```

See Also

`bitconcat`, `bitrol`, `bitror`, `bitshift`, `bitsliceget`, `bitsll`, `bitsra`, `pow2`

bitxor

Purpose

Bitwise exclusive `OR` of two `fi` objects

Syntax

`c = bitxor(a,b)`

Description

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

- Elements in the output array `c` are assigned a value of `1` when exactly one of the corresponding bits in the input arrays has a value of `1`.
- Elements in the output array `c` are assigned a value of `0` when the corresponding bits in the input arrays have the same value (e.g. both `1`'s or both `0`'s).

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

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

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

Examples

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

```
a = fi(-28,1,6,0);
b = fi(12, 1, 6, 0);
c = bitxor(a,b)

c =

   -24

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

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

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

binary_a =

100100

binary_b =

001100

binary_c =

101000
```

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

bitxorreduce

Purpose

Bitwise exclusive OR of consecutive range of bits

Syntax

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

Description

`c = bitxorreduce(a)` performs a bitwise exclusive OR operation on the entire set of bits in the `fi` object `a` and returns the result as a `u1,0` (unsigned integer of word length 1).

`c = bitxorreduce(a, lidx)` performs a bitwise exclusive OR operation on a consecutive range of bits starting at position `lidx` and ending at the LSB (the bit at position 1). `lidx` is a constant that represents the position in the range closest to the MSB.

`c = bitxorreduce(a, lidx, ridx)` performs a bitwise exclusive OR operation on a consecutive range of bits starting at position `lidx` and ending at position `ridx`. `ridx` is a constant that represents the position in the range closest to the LSB.

The `bitxorreduce` arguments must satisfy the following condition:

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

`a` can be a scalar `fi` object or a vector `fi` object.

`bitxorreduce` only supports `fi` objects with fixed-point data types; it does not support inputs with complex data types.

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

Example

This example shows how to perform a bitwise exclusive OR operation on a range of bits of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0:

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

```
disp(bin(a))

0101
```

Get the bitwise exclusive OR of the consecutive set of bits starting at position 4 and ending at position 2:

```
disp(bin(bitxorreduce(a,4,2)))

1
```

See Also

bitandreduce, bitconcat, bitorreduce, bitsliceget

Purpose

Buffer signal vector into matrix of data frames

Description

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

Purpose

Round toward positive infinity

Syntax

```
y = ceil(a)
```

Description

`y = ceil(a)` rounds `fi` object `a` to the nearest integer in the direction of positive infinity and returns the result in `fi` object `y`.

`y` and `a` have the same `fimath` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a` plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`ceil` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Examples

Example 1

The following example demonstrates how the `ceil` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =

    3.1250
```

```
            DataTypeMode: Fixed-point: binary point scaling
              Signedness: Signed
              WordLength: 8
          FractionLength: 3

y = ceil(a)

y =

     4

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

Example 2

The following example demonstrates how the `ceil` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

            DataTypeMode: Fixed-point: binary point scaling
              Signedness: Signed
              WordLength: 8
          FractionLength: 12

y = ceil(a)

y =
```

```
    1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

a	ceil(a)	fix(a)	floor(a)
– 2.5	–2	–2	–3
–1.75	–1	–1	–2
–1.25	–1	–1	–2
–0.5	0	0	–1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

See Also

`convergent`, `fix`, `floor`, `nearest`, `round`

clabel

Purpose Create contour plot elevation labels

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

Purpose Create 2-D comet plot

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

comet3

Purpose Create 3-D comet plot

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

Purpose Plot arrows emanating from origin

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

complex

Purpose

Construct complex `fi` object from real and imaginary parts

Syntax

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

Description

The `complex` function constructs a complex `fi` object from real and imaginary parts.

`c = complex(a,b)` returns the complex result `a + bi`, where `a` and `b` are identically sized real N-D arrays, matrices, or scalars of the same data type. When `b` is all zero, `c` is complex with an all-zero imaginary part. This is in contrast to the addition of `a` + 0i, which returns a strictly real result.

`c = complex(a)` for a real `fi` object `a` returns the complex result `a + bi` with real part `a` and an all-zero imaginary part. Even though its imaginary part is all zero, `c` is complex.

The output `fi` object `c` has the same `numerictype` and `fimath` properties as the input `fi` object `a`. If `a` is associated with the global fimath, the output `fi` object `c` is also associated with the global fimath.

See Also

`imag`, `real`

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

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

Purpose Complex conjugate of `fi` object

Syntax

```
conj(a)
```

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

When `a` is complex,

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

The `numerictype` and `fimath` properties associated with the input `a` are applied to the output.

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

Purpose Create contour graph of matrix

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

contour3

Purpose Create 3-D contour plot

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

Purpose Create two-level contour plot computation

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

contourf

Purpose Create filled 2-D contour plot

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

Purpose

Convolution and polynomial multiplication of fi objects

Syntax

```
c = conv(a,b)
c = conv(a,b,'shape')
```

Description

c = conv(a,b) outputs the convolution of input vectors a and b, at least one of which must be a fi object.

c = conv(a,b,'shape') returns a subsection of the convolution, as specified by the shape parameter:

- full — Returns the full convolution. This option is the default shape.
- same — Returns the central part of the convolution that is the same size as input vector a.
- valid — Returns only those parts of the convolution that the function computes without zero-padded edges. In this case, the length of output vector c is max(length(a)-max(0,length(b)-1), 0).

The fimath properties associated with the inputs determine the numerictype properties of output fi object c:

- If either a or b has a local fimath object, conv uses that fimath object to compute intermediate quantities and determine the numerictype properties of c.
- If both a and b are associated with the global fimath, conv uses the global fimath to compute intermediate quantities and determine the numerictype properties of c.

If either input is a built-in data type, conv casts it into a fi object using best-precision rules before the performing the convolution operation.

The output fi object c is always associated with the global fimath.

Refer to the MATLAB conv reference page for more information on the convolution algorithm.

Examples

The following example illustrates the convolution of a 22-sample sequence with a 16-tap FIR filter.

First, make sure the `SumMode` of the global fimath is set to `FullPrecision`:

```
globalfimath('SumMode', 'FullPrecision');
```

Next, define the variables:

- `x` is a 22-sample sequence of signed values with a word length of 16 bits and a fraction length of 15 bits.
- `h` is the 16 tap FIR filter.

```
u = (pi/4)*[1 1 1 -1 -1 -1 1 -1 -1 1 -1];
x = fi(kron(u,[1 1]));
h = firls(15, [0 .1 .2 .5]*2, [1 1 0 0]);
```

Because `x` is a `fi` object, you do not need to cast `h` into a `fi` object before performing the convolution operation. The `conv` function does so using best-precision scaling.

Finally, use the `conv` function to convolve the two vectors:

```
y = conv(x,h);
```

The operation results in a signed `fi` object `y` with a word length of 36 bits and a fraction length of 31 bits. The `fimath` properties associated with the inputs determine the `numerictype` of the output. In this case, both inputs are associated with the global fimath, so the global fimath determines the `numerictype` of the output.

See Also

`conv`

Purpose

Round toward nearest integer with ties rounding to nearest even integer

Syntax

```
y = convergent(a)
y = convergent(x)
```

Description

`y = convergent(a)` rounds `fi` object `a` to the nearest integer. In the case of a tie, `convergent(a)` rounds to the nearest even integer.

`y` and `a` have the same `fimath` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`convergent` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

`y = convergent(x)` rounds the elements of `x` to the nearest integer. In the case of a tie, `convergent(x)` rounds to the nearest even integer.

Examples

Example 1

The following example demonstrates how the `convergent` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =
```

```
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3

y = convergent(a)

y =

     3

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

Example 2

The following example demonstrates how the `convergent` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12

y = convergent(a)
```

```
y =

     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

The following table illustrates these differences for a given `fi` object `a`.

a	convergent(a)	nearest(a)	round(a)
–3.5	–4	–3	–4
–2.5	–2	–2	–3
–1.5	–2	–1	–2
–0.5	0	0	–1
0.5	0	1	1
1.5	2	2	2
2.5	2	3	3
3.5	4	4	4

See Also `ceil`, `fix`, `floor`, `nearest`, `round`

Purpose Make independent copy of quantizer object

Syntax

```
q1 = copyobj(q)
[q1,q2,...] = copyobj(obja,objb,...)
```

Description q1 = copyobj(q) makes a copy of quantizer object q and returns it in q1.

[q1,q2,...] = copyobj(obja,objb,...)copies obja into q1, objb into q2, and so on.

Using copyobj to copy a quantizer object is not the same as using the command syntax q1 = q to copy a quantizer object. quantizer objects have memory (their read-only properties). When you use copyobj, the resulting copy is independent of the original item; it does not share the original object's memory, such as the values of the properties min, max, noverflows, or noperations. Using q1 = q creates a new object that is an alias for the original and shares the original object's memory, and thus its property values.

Examples

```
q = quantizer([8 7]);
q1 = copyobj(q)
```

See Also quantizer, get, set

Purpose CORDIC-based approximation of complex exponential

Syntax

```
y = cordiccexp(theta, niters)
```

Description

`y = cordiccexp(theta, niters)` computes `cos(theta) + j*sin(theta)` using a "CORDIC" on page 3-93 algorithm approximation. `y` contains the approximated complex result.

Input Arguments

`theta`

`theta` can be a scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of *`theta`* must be real and in the range `[0, 2*pi)`.

`niters`

`niters` is the number of iterations the CORDIC algorithm performs. *`niters`* must be a positive, integer-valued scalar that is less than the word length of *`theta`*. Increasing the number of iterations may produce more accurate results, but increasing the number of iterations also increases the expense of computation and adds latency.

Output Arguments

`y`

`y` is the approximated complex result of the `cordiccexp` function. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` – `2`.

Definitions

CORDIC

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

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

Examples

The following example illustrates the effect of the number of iterations on the result of the `cordiccexp` approximation.

```
wrdLn = 8;
theta = fi(pi/2, 1, wrdLn);
fprintf( ...
 '\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n'
fprintf( ...
  '------\t\t-------\t ------\t ----\t\t-------\t ------\t ----\n')
for niters = 1:(wrdLn - 1)
  cis    = cordiccexp(theta, niters);
  fl     = cis.FractionLength;
  x      = real(cis);
  y      = imag(cis);
  x_dbl  = double(x);
  x_err  = abs(x_dbl - cos(double(theta)));
  y_dbl  = double(y);
  y_err  = abs(y_dbl - sin(double(theta)));
  fprintf( ...
   '%d\t\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
    niters, y_dbl, y_err, (y_err * pow2(fl)), ...
    x_dbl, x_err, (x_err * pow2(fl)));
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS    Y (SIN)  ERROR   LSBs    X (COS)  ERROR   LSBs
------     -------  ------  ----    -------  ------  ----
1          0.7031   0.2968  19.0     0.7031   0.7105  45.5
2          0.9375   0.0625  4.0      0.3125   0.3198  20.5
```

```
3          0.9844   0.0156  1.0     0.0938   0.1011  6.5
4          0.9844   0.0156  1.0    -0.0156   0.0083  0.5
5          1.0000   0.0000  0.0     0.0312   0.0386  2.5
6          1.0000   0.0000  0.0     0.0000   0.0073  0.5
7          1.0000   0.0000  0.0     0.0156   0.0230  1.5
```

References

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

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* Feb. 22–24, 1998, pp. 191–200.

See Also

`cordiccos` | `cordicsin` | `cordicsincos`

Tutorials

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

Purpose

CORDIC-based approximation of cosine

Syntax

`y = cordiccos(theta, niters)`

Description

`y = cordiccos(theta, niters)` computes the cosine of `theta` using a “CORDIC” on page 3-93 algorithm approximation.

Input Arguments

theta

`theta` can be a scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range `[0, 2*pi)`.

niters

`niters` is the number of iterations the CORDIC algorithm performs. `niters` must be a positive, integer-valued scalar that is less than the word length of `theta`. Increasing the number of iterations may produce more accurate results, but it also increases the expense of computation and adds latency.

Output Arguments

y

`y` is the CORDIC-based approximation of the cosine of `theta`. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` – `2`.

Definitions

CORDIC

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

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

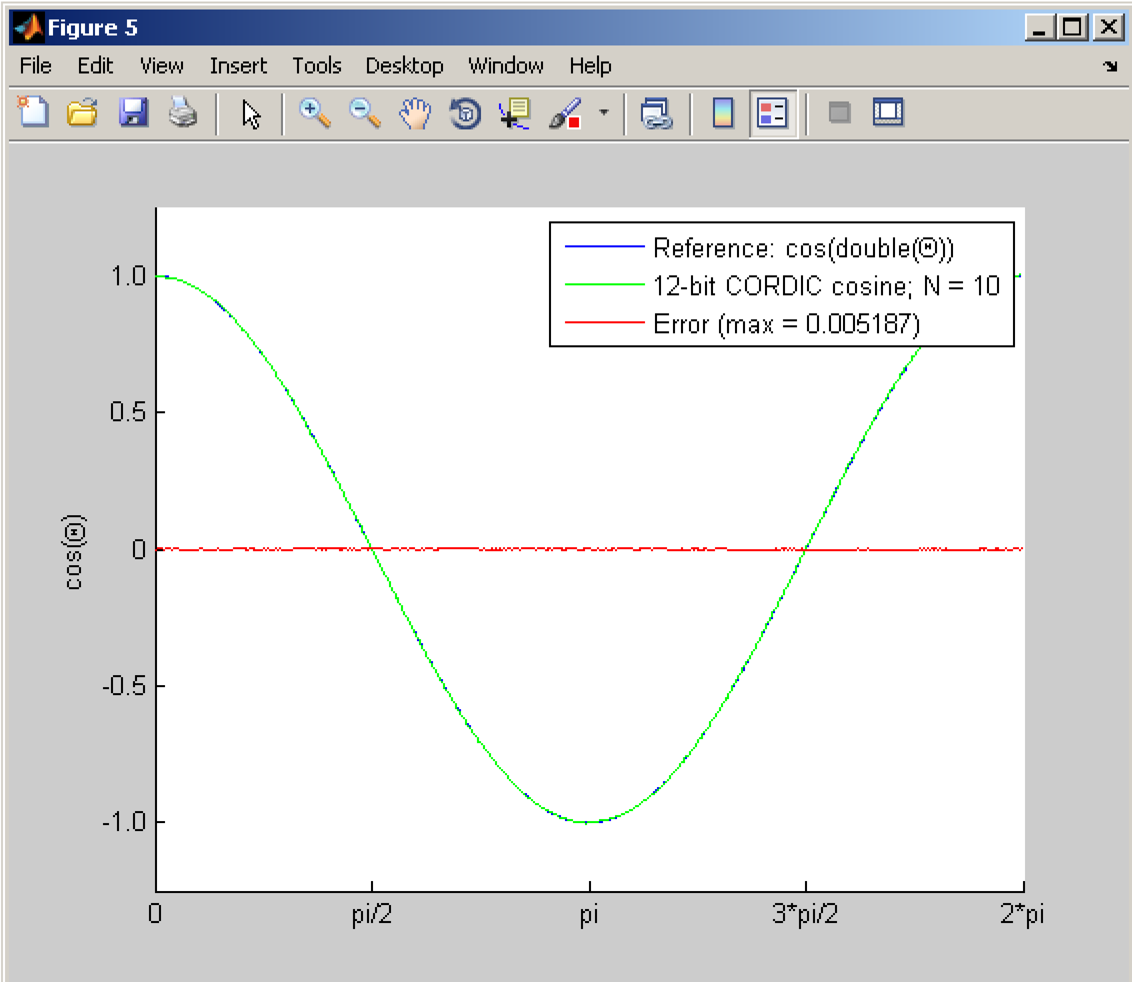
Examples

Compare the results produced by various iterations of the `cordiccos` algorithm to the results of the double-precision `cos` function:

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

% Use 12-bit quantized inputs and vary the number
% of iterations from 2 to 10.
% Compare the  fixed-point CORDIC results to the
% double-precision trig function results.
for niters = 2:2:10
    cdcCosTh  = cordiccos(thRadFxp,  niters);
    errCdcRef = cosThRef - double(cdcCosTh);
    figure; hold on; axis([0 2*pi -1.25 1.25]);
    plot(thRadFxp, cosThRef,  'b');
    plot(thRadFxp, cdcCosTh,  'g');
    plot(thRadFxp, errCdcRef, 'r');
    ylabel('cos(\Theta)');
    set(gca,'XTick',0:pi/2:2*pi);
    set(gca,'XTickLabel',{'0','pi/2','pi','3*pi/2','2*pi'});
    set(gca,'YTick',-1:0.5:1);
    set(gca,'YTickLabel',{'-1.0','-0.5','0','0.5','1.0'});
    ref_str = 'Reference: cos(double(\Theta))';
    cdc_str = sprintf('12-bit CORDIC cosine; N = %d', niters);
    err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
    legend(ref_str, cdc_str, err_str);
end
```

After 10 iterations, the CORDIC algorithm has approximated the cosine of *theta* to within 0.005187 of the double-precision cosine result.



References

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

[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* Feb. 22–24, 1998, pp. 191–200.

See Also

`cordiccexp` | `cordicsin` | `cordicsincos`

Tutorials

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

Purpose

CORDIC-based approximation of sine

Syntax

`y = cordicsin(theta, niters)`

Description

`y = cordicsin(theta, niters)` computes the sine of *theta* using a "CORDIC" on page 3-93 algorithm approximation.

Input Arguments

theta

> *theta* can be a scalar, vector, matrix, or N-dimensional array containing the angle values in radians. All values of *theta* must be real and in the range `[0, 2*pi)`.

niters

> *niters* is the number of iterations the CORDIC algorithm performs. *niters* must be a positive, integer-valued scalar that is less than the word length of *theta*. Increasing the number of iterations may produce more accurate results, but it also increases the expense of computation and adds latency.

Output Arguments

y

> *y* is the CORDIC-based approximation of the sine of `theta`. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` – `2`.

Definitions

CORDIC

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

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

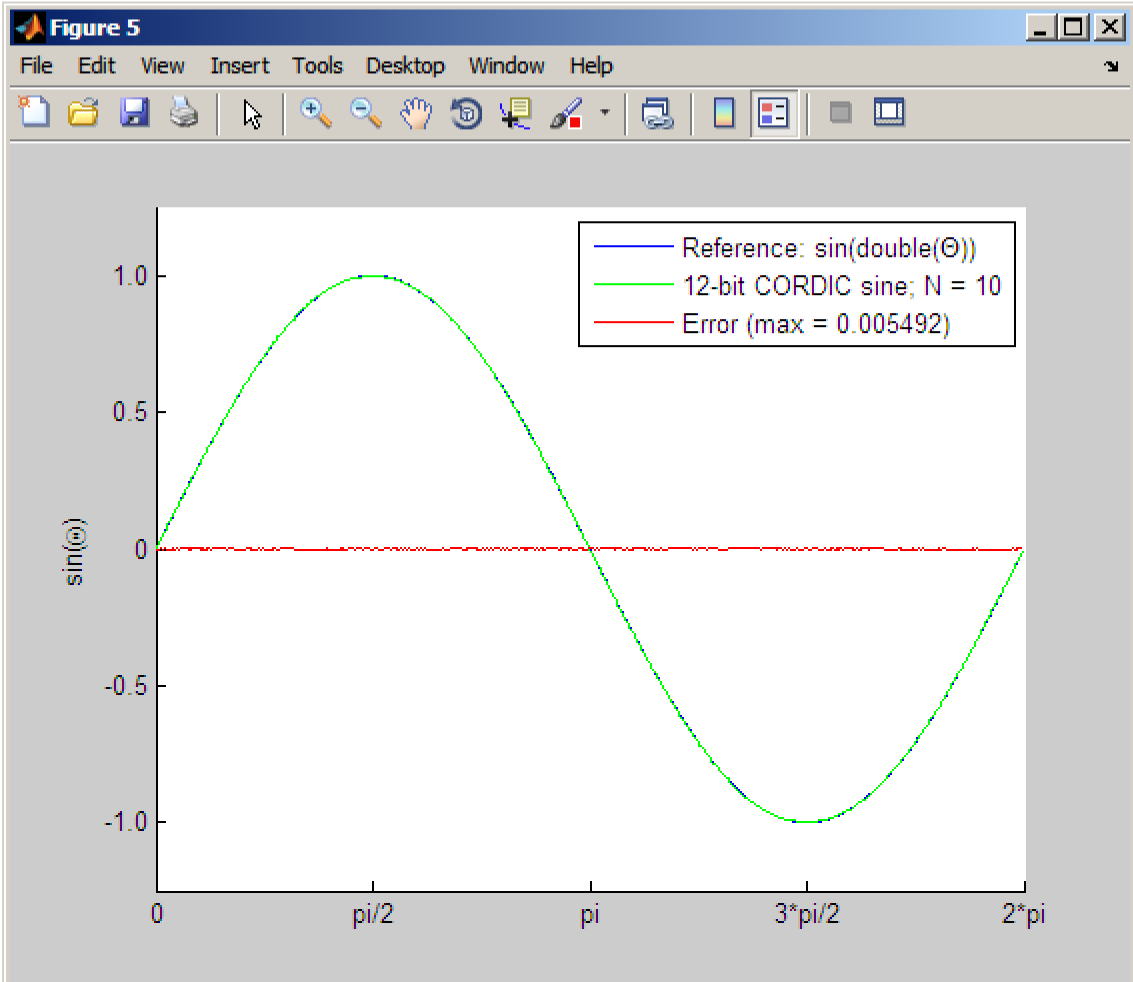
Examples

Compare the results produced by various iterations of the `cordicsin` algorithm to the results of the double-precision `sin` function:

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

% Use 12-bit quantized inputs and vary the number of iterations
% from 2 to 10.
% Compare the fixed-point cordicsin function results to the
% results of the double-precision sin function.
for niters = 2:2:10
    cdcSinTh  = cordicsin(thRadFxp,  niters);
    errCdcRef = sinThRef - double(cdcSinTh);
    figure; hold on; axis([0 2*pi -1.25 1.25]);
    plot(thRadFxp, sinThRef,  'b');
    plot(thRadFxp, cdcSinTh,  'g');
    plot(thRadFxp, errCdcRef, 'r');
    ylabel('sin(\Theta)');
    set(gca,'XTick',0:pi/2:2*pi);
    set(gca,'XTickLabel',{'0','pi/2','pi','3*pi/2','2*pi'});
    set(gca,'YTick',-1:0.5:1);
    set(gca,'YTickLabel',{'-1.0','-0.5','0','0.5','1.0'});
    ref_str = 'Reference: sin(double(\Theta))';
    cdc_str = sprintf('12-bit CORDIC sine; N = %d', niters);
    err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
    legend(ref_str, cdc_str, err_str);
end
```

After 10 iterations, the CORDIC algorithm has approximated the sine of *theta* to within 0.005492 of the double-precision sine result.



References

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

[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* Feb. 22–24, 1998, pp. 191–200.

See Also

`cordiccexp` | `cordiccos` | `cordicsincos`

Tutorials

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

Purpose

CORDIC-based approximation of sine and cosine

Syntax

`[y, x] = cordicsincos(theta, niters)`

Description

`[y, x] = cordicsincos(theta, niters)` computes the sine and cosine of *`theta`* using a "CORDIC" on page 3-93 algorithm approximation. *`y`* contains the approximated sine result, and *`x`* contains the approximated cosine result.

Input Arguments

`theta`

`theta` can be a scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of *`theta`* must be real and in the range `[0, 2*pi)`.

`niters`

`niters` is the number of iterations the CORDIC algorithm performs. *`niters`* must be a positive, integer-valued scalar that is less than the word length of *`theta`*. Increasing the number of iterations may produce more accurate results, but increasing the number of iterations also increases the expense of computation and adds latency.

Output Arguments

`y`

`[y, x]` contains the CORDIC-based approximation of the sine and cosine of `theta`, where *`y`* is the approximated sine and *`x`* is the approximated cosine. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` – `2`.

Definitions

CORDIC

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

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

Examples

The following example illustrates the effect of the number of iterations on the result of the `cordicsincos` approximation.

```
wrdLn = 8;
theta = fi(pi/2, 1, wrdLn);
fprintf( ...
 '\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n');
fprintf( ...
 '------\t\t-------\t ------\t ----\t\t-------\t ------\t ----\n');
for niters = 1:(wrdLn - 1)
  [y, x] = cordicsincos(theta, niters);
  y_FL   = y.FractionLength;
  y_dbl  = double(y);
  x_dbl  = double(x);
  y_err  = abs(y_dbl - sin(double(theta)));
  x_err  = abs(x_dbl - cos(double(theta)));
  fprintf( ...
   '  %d\t\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
    niters, y_dbl, y_err, (y_err * pow2(y_FL)), ...
    x_dbl, x_err, (x_err * pow2(y_FL)));
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS    Y (SIN)  ERROR   LSBs   X (COS)  ERROR   LSBs
------    -------  ------  ----   -------  ------  ----
1         0.7031   0.2968  19.0   0.7031   0.7105  45.5
```

2	0.9375	0.0625	4.0	0.3125	0.3198	20.5
3	0.9844	0.0156	1.0	0.0938	0.1011	6.5
4	0.9844	0.0156	1.0	-0.0156	0.0083	0.5
5	1.0000	0.0000	0.0	0.0312	0.0386	2.5
6	1.0000	0.0000	0.0	0.0000	0.0073	0.5
7	1.0000	0.0000	0.0	0.0156	0.0230	1.5

References

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

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays.* Feb. 22–24, 1998, pp. 191–200.

See Also

`cordiccexp` | `cordiccos` | `cordicsin`

Tutorials

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

ctranspose

Purpose Complex conjugate transpose of `fi` object

Syntax `ctranspose(a)`

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

See Also `transpose`

Purpose

Unsigned decimal representation of stored integer of `fi` object

Syntax

```
dec(a)
```

Description

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

.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

Examples

The code

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

returns

```
y =

  128   127


z =

  128   127
```

See Also

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

denormalmax

Purpose Largest denormalized quantized number for `quantizer` object

Syntax

```
x = denormalmax(q)
```

Description `x = denormalmax(q)` is the largest positive denormalized quantized number where `q` is a `quantizer` object. Anything larger than `x` is a normalized number. Denormalized numbers apply only to floating-point format. When `q` represents fixed-point numbers, this function returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
x = denormalmax(q)

x =

    0.1875
```

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

```
denormalmax(q) = realmin(q) - denormalmin(q)
```

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

```
denormalmax(q) = eps(q)
```

See Also `denormalmin`, `eps`, `quantizer`

Purpose

Smallest denormalized quantized number for `quantizer` object

Syntax

```
x = denormalmin(q)
```

Description

`x = denormalmin(q)` is the smallest positive denormalized quantized number where `q` is a `quantizer` object. Anything smaller than `x` underflows to zero with respect to the `quantizer` object `q`. Denormalized numbers apply only to floating-point format. When `q` represents a fixed-point number, `denormalmin` returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
x = denormalmin(q)

x =

    0.0625
```

Algorithm

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

$$x = 2^{E_{min} - f}$$

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

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

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

where f is equal to `fractionlength(q)`.

See Also

`denormalmax`, `eps`, `quantizer`

diag

Purpose Diagonal matrices or diagonals of matrix

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

Purpose Display object

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

divide

Purpose

Divide two objects

Syntax

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

Description

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

If `a` and `b` are both `fi` objects, `c` has the same `fimath` object as `a`. If `c` has a `fi Fixed` data type, and any one of the inputs have `fi` floating point data types, then the `fi` floating point is converted into a fixed-point value. Intermediate quantities are calculated using the `fimath` object of `a`. See "Data Type Propagation Rules" on page 3-102.

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

If either `a` or `b` is a `fi` object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the `fi` object, preserving best-precision fraction length. Intermediate quantities are calculated using the `fimath` object of the input `fi` object. See "Data Type Propagation Rules" on page 3-102.

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

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

Data Type Propagation Rules

For syntaxes for which Fixed-Point Toolbox software uses the `numerictype` object `T`, the `divide` function follows the data type propagation rules listed in the following table. In general, these rules can be summarized as "floating-point data types are propagated." This allows you to write code that can be used with both fixed-point and floating-point inputs.

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

Data Type of Input fi Objects a and b		Data Type of numerictype object T	Data Type of Output c
fi ScaledDouble	fi ScaledDouble	fi double	fi double
fi ScaledDouble	fi ScaledDouble	fi single	fi single
fi ScaledDouble	fi ScaledDouble	fi ScaledDouble	fi ScaledDouble with properties of numerictype object T

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 =

11001100110011001100110011001100110011001100110011010000
000000000000000000000000
      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 =

110011001100110011001100110011001100110011001100110011001100
11001100110011001100
```

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 =

11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100110011001100
110011001100110011001100110011001100110011001100
```

See Also

`add`, `fi`, `fimath`, `mpy`, `mrdivide`, `numerictype`, `rdivide`, `sub`, `sum`

Purpose

Double-precision floating-point real-world value of `fi` object

Syntax

`double(a)`

Description

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

Fixed-point numbers can be represented as

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

or, equivalently as

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

Examples

The code

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

returns

```
y =

            -1        0.9922
z =

            -1        0.9922
```

See Also

`single`

Purpose Last index of array

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

Purpose

Quantized relative accuracy for `fi` or `quantizer` objects

Syntax

```
eps(obj)
```

Description

`eps(obj)` returns the value of the least significant bit of the value of the `fi` object or `quantizer` object `obj`. The result of this function is equivalent to that given by the Fixed-Point Toolbox function `lsb`.

See Also

`intmax`, `intmin`, `lowerbound`, `lsb`, `range`, `realmax`, `realmin`, `upperbound`

eq

Purpose

Determine whether real-world values of two `fi` objects are equal

Syntax

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

Description

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

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

See Also

`ge`, `gt`, `isequal`, `le`, `lt`, `ne`

Purpose

Mean of quantization error

Syntax

```
m = errmean(q)
```

Description

`m = errmean(q)` returns the mean of a uniformly distributed random quantization error that arises from quantizing a signal by `quantizer` object `q`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples

Find `m`, the mean of the quantization error for `quantizer` `q`:

```
q = quantizer;
m = errmean(q)

m =

   -1.525878906250000e-005
```

Now compare `m` to `m_est`, the sample mean from a Monte Carlo experiment:

```
r = realmax(q);
u = 2*r*rand(1000,1)-r;  % Original signal
y = quantize(q,u);       % Quantized signal
e = y - u;               % Error
m_est = mean(e)          % Estimate of the error mean

m_est =

   -1.519507450175317e-005
```

See Also

`errpdf`, `errvar`, `quantize`

errorbar

Purpose Plot error bars along curve

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

Purpose

Probability density function of quantization error

Syntax

```
[f,x] = errpdf(q)
f = errpdf(q,x)
```

Description

`[f,x] = errpdf(q)` returns the probability density function `f` evaluated at the values in `x`. The vector `x` contains the uniformly distributed random quantization errors that arise from quantizing a signal by `quantizer` object `q`.

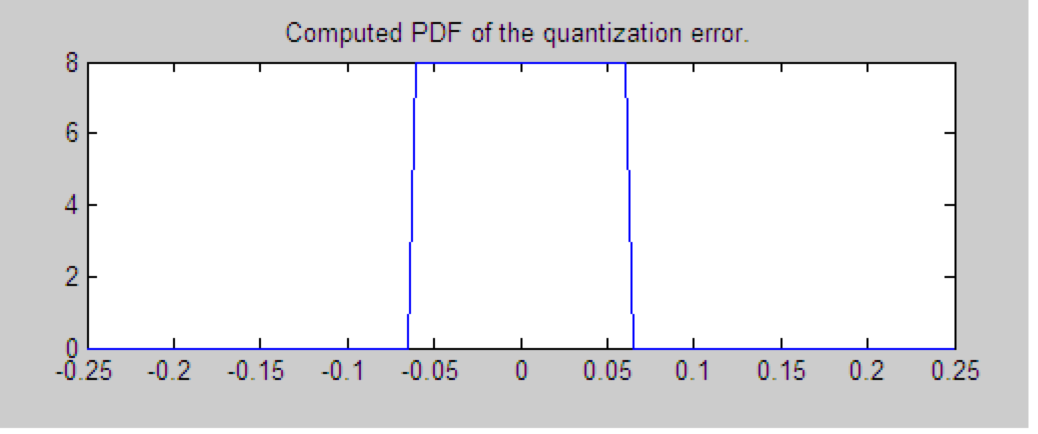
`f = errpdf(q,x)` returns the probability density function `f` evaluated at the values in vector `x`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples

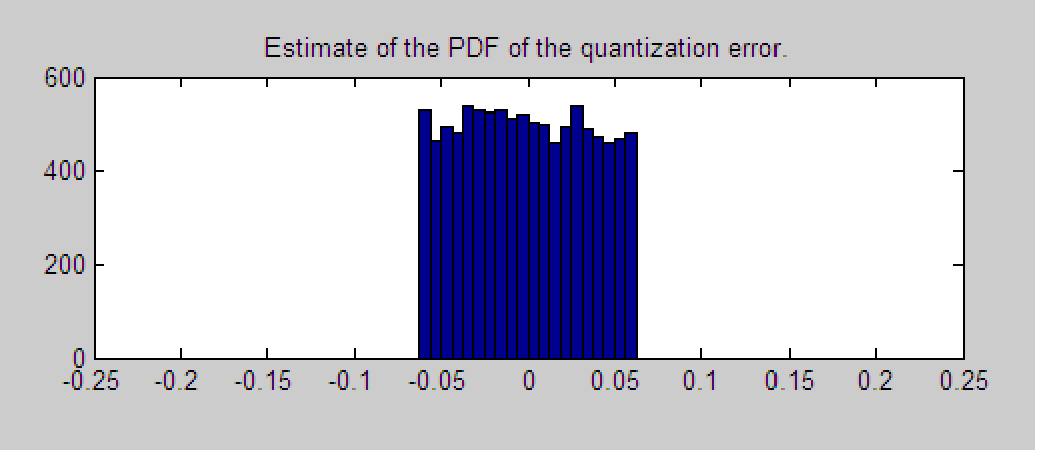
```
q = quantizer('nearest',[4 3]);
[f,x] = errpdf(q);
subplot(211)
plot(x,f)
title('Computed PDF of the quantization error.')
```

The output plot shows the probability density function of the quantization error.



Compare this result to a plot of the sample probability density function from a Monte Carlo experiment:

```
r = realmax(q);
u = 2*r*rand(10000,1)-r;  % Original signal
y = quantize(q,u);        % Quantized signal
e = y - u;                % Error
subplot(212)
hist(e,20);set(gca,'xlim',[min(x) max(x)])
title('Estimate of the PDF of the quantization error.')
```



See Also `errmean`, `errvar`, `quantize`

errvar

Purpose Variance of quantization error

Syntax

```
v = errvar(q)
```

Description `v = errvar(q)` returns the variance of a uniformly distributed random quantization error that arises from quantizing a signal by `quantizer` object `q`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples Find `v`, the variance of the quantization error for `quantizer` object `q`:

```
q = quantizer;
v = errvar(q)

v =

    7.761021455128987e-011
```

Now compare `v` to `v_est`, the sample variance from a Monte Carlo experiment:

```
r = realmax(q);
      u = 2*r*rand(1000,1)-r;  % Original signal
      y = quantize(q,u);     % Quantized signal
      e = y - u;           % Error
      v_est = var(e)     % Estimate of the error variance

v_est =

    7.520208858166330e-011
```

See Also `errmean`, `errpdf`, `quantize`

Purpose

Plot elimination tree

Description

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

exponentbias

Purpose

Exponent bias for `quantizer` object

Syntax

```
b = exponentbias(q)
```

Description

`b = exponentbias(q)` returns the exponent bias of the `quantizer` object `q`. For fixed-point `quantizer` objects, `exponentbias(q)` returns 0.

Examples

```
q = quantizer('double');
b = exponentbias(q)

b =

        1023
```

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

Exponent length of `quantizer` object

Syntax

`e = exponentlength(q)`

Description

`e = exponentlength(q)` returns the exponent length of `quantizer` object `q`. When `q` is a fixed-point `quantizer` object, `exponentlength(q)` returns `0`. This is useful because exponent length is valid whether the `quantizer` object mode is floating point or fixed point.

Examples

```
q = quantizer('double');
e = exponentlength(q)

e =

    11
```

Algorithm

The exponent length is part of the format of a floating-point `quantizer` object `[w e]`. For fixed-point `quantizer` objects, $e = 0$ by definition.

See Also

`eps`, `exponentbias`, `exponentmax`, `exponentmin`

exponentmax

Purpose Maximum exponent for `quantizer` object

Syntax `exponentmax(q)`

Description `exponentmax(q)` returns the maximum exponent for `quantizer` object `q`. When `q` is a fixed-point `quantizer` object, it returns `0`.

Examples

```
q = quantizer('double');
emax = exponentmax(q)

emax =

        1023
```

Algorithm For floating-point `quantizer` objects,

$$E_{max} = 2^{e-1} - 1$$

For fixed-point `quantizer` objects, $E_{max} = 0$ by definition.

See Also `eps`, `exponentbias`, `exponentlength`, `exponentmin`

Purpose Minimum exponent for `quantizer` object

Syntax

```
emin = exponentmin(q)
```

Description `emin = exponentmin(q)` returns the minimum exponent for `quantizer` object `q`. If `q` is a fixed-point `quantizer` object, `exponentmin` returns 0.

Examples

```
q = quantizer('double');
emin = exponentmin(q)

emin =

        -1022
```

Algorithm For floating-point `quantizer` objects,

$$E_{min} = -2^{e-1} + 2$$

For fixed-point `quantizer` objects, $E_{min} = 0$.

See Also `eps`, `exponentbias`, `exponentlength`, `exponentmax`

Purpose Easy-to-use contour plotter

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

Purpose Easy-to-use filled contour plotter

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

Purpose Easy-to-use 3-D mesh plotter

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

Purpose Easy-to-use function plotter

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

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

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

Purpose Easy-to-use polar coordinate plotter

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

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

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

Purpose Easy-to-use combination surface/contour plotter

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

feather

Purpose Plot velocity vectors

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

Purpose

Construct fixed-point numeric object

Syntax

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

Description

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

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

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

word length w, slopeadjustmentfactor, fixedexponent, bias, and embedded.fimath F.

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

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

- "Data Properties" on page 3-133
- "fimath Properties" on page 3-134
- "numerictype Properties" on page 3-135

Note These properties are described in detail in "fi Object Properties" on page 1-2 in the Properties Reference.

Data Properties

The data properties of a fi object are always writable.

- bin — Stored integer value of a fi object in binary
- data — Numerical real-world value of a fi object
- dec — Stored integer value of a fi object in decimal
- double — Real-world value of a fi object, stored as a MATLAB double
- hex — Stored integer value of a fi object in hexadecimal
- int — Stored integer value of a fi object, stored in a built-in MATLAB integer data type. You can also use int8, int16, int32, int64, uint8, uint16, uint32, and uint64 to get the stored integer value of a fi object in these formats
- oct — Stored integer value of a fi object in octal

These properties are described in detail in "fi Object Properties" on page 1-2.

fimath Properties

When you create a fi object and specify fimath object properties in the fi constructor, a fimath object is created as a property of the fi object. If you do not specify any fimath properties in the fi constructor, the resulting fi object associates itself with the global fimath. See "Working with the Global fimath" for more information.

- fimath — fimath properties associated with a fi object

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

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

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

- MaxProductWordLength — Maximum allowable word length for the product data type
- MaxSumWordLength — Maximum allowable word length for the sum data type
- OverflowMode — Overflow mode
- ProductBias — Bias of the product data type
- ProductFixedExponent — Fixed exponent of the product data type
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined

- ProductSlope — Slope of the product data type
- ProductSlopeAdjustmentFactor — Slope adjustment factor of the product data type
- ProductWordLength — Word length, in bits, of the product data type
- RoundMode — Rounding mode
- SumBias — Bias of the sum data type
- SumFixedExponent — Fixed exponent of the sum data type
- SumFractionLength — Fraction length, in bits, of the sum data type
- SumMode — Defines how the sum data type is determined
- SumSlope — Slope of the sum data type
- SumSlopeAdjustmentFactor — Slope adjustment factor of the sum data type
- SumWordLength — The word length, in bits, of the sum data type

These properties are described in detail in "fimath Object Properties" on page 1-4.

numerictype Properties

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

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

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

- Bias — Bias of a fi object
- DataType — Data type category associated with a fi object

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

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

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

For further details on these properties, see “numerictype Object Properties” on page 1-15.

Examples

Note For information about the display format of `fi` objects, refer to Display Settings.

For examples of casting, see “Casting fi Objects”.

Example 1

For example, the following creates a signed `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = fi(pi, 1, 8, 3)

a =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

Example 2

The value `v` can also be an array:

```
a = fi((magic(3)/10), 1, 16, 12)

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

Example 3

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

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

a =

    3.1563

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

Example 4

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

```
a = fi(pi, 1)

a =

    3.1416

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

Example 5

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

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

a =
    3.1415
```

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

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

Example 6

You can remove a local `fimath` object from a `fi` object at any time using the following syntax:

```
a = fi(pi, 'roundmode', 'floor', 'overflowmode', 'wrap')
a.fimath = []

a =
    3.1415

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

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

a =
    3.1415
```

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

fi object a is now associated with the global fimath. To reassign it a local fimath object, use dot notation:

```
a.ProductMode = 'KeepLSB'

a =
    3.1415

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

                 RoundMode: nearest
              OverflowMode: saturate
               ProductMode: KeepLSB
         ProductWordLength: 32
                   SumMode: FullPrecision
          MaxSumWordLength: 128
```

fi object a now has a local fimath object with a ProductMode of KeepLSB. The values of the remaining fimath object properties are inherited from the current global fimath.

See Also

fimath, fipref, isfimathlocal, numerictype, quantizer, sfi, ufi

Purpose

One-dimensional digital filter of fi objects

Syntax

```
y = filter(b,1,x)
[y,zf] = filter(b,1,x,zi)
y = filter(b,1,x,zi,dim)
```

Description

`y = filter(b,1,x)` filters the data in the fixed-point vector `x` using the filter described by the fixed-point vector *b*. The function returns the filtered data in the output `fi` object `y`. Inputs `b` and `x` must be `fi` objects. `filter` always operates along the first non-singleton dimension. Thus, the filter operates along the first dimension for column vectors and nontrivial matrices, and along the second dimension for row vectors.

`[y,zf] = filter(b,1,x,zi)` gives access to initial and final conditions of the delays, `zi` and `zf`. `zi` is a vector of length `length(b)-1`, or an array with the leading dimension of size `length(b)-1` and with remaining dimensions matching those of `x`. `zi` must be a `fi` object with the same data type as `y` and `zf`. If you do not specify a value for `zi`, it defaults to a fixed-point array with a value of `0` and the appropriate `numerictype` and size.

`y = filter(b,1,x,zi,dim)` performs the filtering operation along the specified dimension. If you do not want to specify the vector of initial conditions, use `[]` for the input argument `zi`.

Tips

- The `filter` function only supports FIR filters.
- The `numerictype` of `b` can be different than the `numerictype` of `x`.
- If you want to specify initial conditions, but do not know what `numerictype` to use, first try filtering your data without initial conditions. You can do so by specifying `[]` for the input `zi`. After performing the filtering operation, you have the `numerictype` of `y` and `zf` (if requested). Because the `numerictype` of `zi` must match that of `y` and `zf`, you now know the `numerictype` to use for the initial conditions.

Input Arguments

b

Fixed-point vector of the filter coefficients.

x

Fixed-point vector containing the data for the function to filter.

zi

Fixed-point vector containing the initial conditions of the delays. If the initial conditions of the delays are zero, you can specify zero, or, if you do not know the appropriate size and `numerictype` for *zi*, use `[]`.

If you do not specify a value for *zi*, the parameter defaults to a fixed-point vector with a value of zero and the same `numerictype` and size as the output *zf* (default).

dim

Dimension along which to perform the filtering operation.

Output Arguments

y

Output vector containing the filtered fixed-point data.

zf

Fixed-point output vector containing the final conditions of the delays.

Definitions

Filter length (L)

The filter length is `length(b)`, or the number of filter coefficients specified in the fixed-point vector *b*.

Filter order (N)

The filter order is the number of states (delays) of the filter, and is equal to *L*-1.

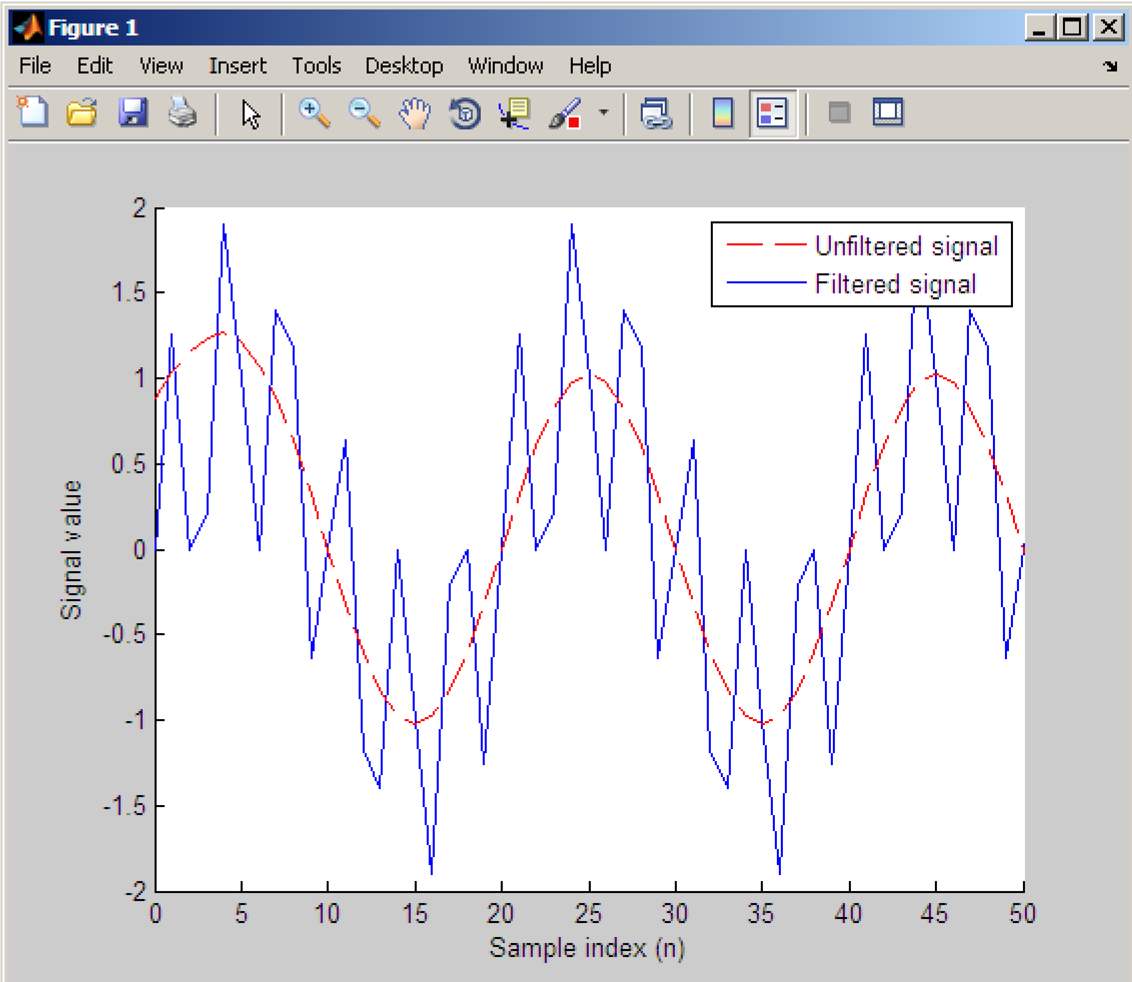
Examples

The following example filters a high-frequency fixed-point sinusoid from a signal that contains both a low- and high-frequency fixed-point sinusoid.

```
w1 = .1*pi;
w2 = .6*pi;
n  = 0:999;
xd = sin(w1*n) + sin(w2*n);
x  = sfi(xd,12);
b  = ufi([.1:.1:1,1-.1:-.1:.1]/4,10);
gd = (length(b)-1)/2;
y  = filter(b,1,x);

%% Plot results, accomodate for group-delay of filter
plot(n(1:end-gd),x(1:end-gd))
hold on
plot(n(1:end-gd),y(gd+1:end),'r--')
axis([0 50 -2 2])
legend('Unfiltered signal','Filtered signal')
xlabel('Sample index (n)')
ylabel('Signal value')
```

The resulting plot shows both the unfiltered and filtered signals.



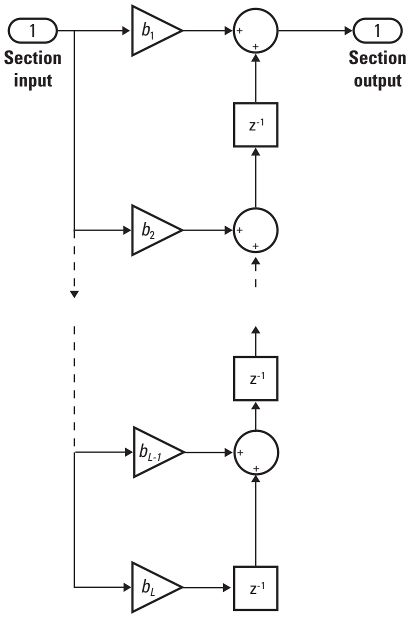
Algorithm

The `filter` function uses a Direct-Form Transposed FIR implementation of the following difference equation:

$$y(n) = b_1 * x_n + b_2 * x_{n-1} + \ldots + b_L * x_{n-N}$$

where L is the filter length and N is the filter order.

The following diagram shows the direct-form transposed FIR filter structure used by the `filter` function:



See Also

`conv` | `filter`

fimath

Purpose

Construct `fimath` object

Syntax

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

Description

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

- `F = fimath` creates a `fimath` object with the same properties as the current global fimath. The factory default configuration of the global fimath has the following properties:

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

  You can request a handle object to the global fimath and change any of its property values using `globalfimath`. For more information about configuring the global fimath, see "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide*.

- `F = fimath(...'PropertyName',PropertyValue...)` allows you to set the attributes of a `fimath` object using property name/property value pairs. All property names that you do not specify in the constructor get their values from the current global fimath.

The properties of the `fimath` object are listed below. These properties are described in detail in "fimath Object Properties" on page 1-4 in the Properties Reference.

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

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

- MaxProductWordLength — Maximum allowable word length for the product data type
- MaxSumWordLength — Maximum allowable word length for the sum data type
- OverflowMode — Overflow-handling mode
- ProductBias — Bias of the product data type
- ProductFixedExponent — Fixed exponent of the product data type
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined
- ProductSlope — Slope of the product data type
- ProductSlopeAdjustmentFactor — Slope adjustment factor of the product data type
- ProductWordLength — Word length, in bits, of the product data type
- RoundMode — Rounding mode
- SumBias — Bias of the sum data type
- SumFixedExponent — Fixed exponent of the sum data type
- SumFractionLength — Fraction length, in bits, of the sum data type
- SumMode — Defines how the sum data type is determined
- SumSlope — Slope of the sum data type
- SumSlopeAdjustmentFactor — Slope adjustment factor of the sum data type
- SumWordLength — Word length, in bits, of the sum data type

Examples

Example 1

Type

```
F = fimath
```

to create a default `fimath` object. If you are using the factory default setting of the global fimath, you get the following output:

```
F =


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

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
```

See Also

`fi`, `fipref`, `numerictype`, `quantizer`, `removeglobalfimathpref`, `resetglobalfimath`, `saveglobalfimathpref`, `globalfimath`

fipref

Purpose

Construct `fipref` object

Syntax

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

Description

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

- `P = fipref` creates a default `fipref` object.
- `P = fipref(...'PropertyName',PropertyValue...)` allows you to set the attributes of a object using property name/property value pairs.

The properties of the `fipref` object are listed below. These properties are described in detail in “fipref Object Properties” on page 1-12.

- `FimathDisplay` — Display options for the local `fimath` attributes of `fi` objects. When `fi` objects are associated with the global fimath, their `fimath` attributes are never displayed.
- `DataTypeOverride` — Data type override options.
- `LoggingMode` — Logging options for operations performed on `fi` objects.
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object.
- `NumberDisplay` — Display options for the value of a `fi` object.

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

See “Display Settings” in the *Fixed-Point Toolbox User’s Guide* for more information on the display preferences used for most code examples in the documentation.

Examples

Example 1

Type

```
P = fipref
```

to create a default `fipref` object.

```
P =

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

Example 2

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

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

P =

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

See Also

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

Purpose Round toward zero

Syntax

```
y = fix(a)
```

Description

`y = fix(a)` rounds `fi` object `a` to the nearest integer in the direction of zero and returns the result in `fi` object `y`.

`y` and `a` have the same `fimath` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`fix` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Examples

Example 1

The following example demonstrates how the `fix` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =

    3.1250
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3

y = fix(a)

y =

     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 5
        FractionLength: 0
```

Example 2

The following example demonstrates how the `fix` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12

y = fix(a)

y =
```

```
0

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 2
    FractionLength: 0
```

Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

a	ceil(a)	fix(a)	floor(a)
– 2.5	–2	–2	–3
–1.75	–1	–1	–2
–1.25	–1	–1	–2
–0.5	0	0	–1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

See Also

`ceil`, `convergent`, `floor`, `nearest`, `round`

Purpose Flip array along specified dimension

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

fliplr

Purpose Flip matrix left to right

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

Purpose Flip matrix up to down

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

floor

Purpose

Round toward negative infinity

Syntax

```
y = floor(a)
```

Description

`y = floor(a)` rounds `fi` object `a` to the nearest integer in the direction of negative infinity and returns the result in `fi` object `y`.

`y` and `a` have the same `fimath` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`floor` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Examples

Example 1

The following example demonstrates how the `floor` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =

    3.1250
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3

y = floor(a)

y =

     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 5
        FractionLength: 0
```

Example 2

The following example demonstrates how the `floor` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12

y = floor(a)

y =

     0
```

```
   DataTypeMode: Fixed-point: binary point scaling
     Signedness: Signed
     WordLength: 2
 FractionLength: 0
```

Example 3

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

- The `ceil` function rounds values to the nearest integer toward positive infinity
- The `fix` function rounds values toward zero
- The `floor` function rounds values to the nearest integer toward negative infinity

The following table illustrates these differences for a given `fi` object `a`.

a	ceil(a)	fix(a)	floor(a)
– 2.5	–2	–2	–3
–1.75	–1	–1	–2
–1.25	–1	–1	–2
–0.5	0	0	–1
0.5	1	0	0
1.25	2	1	1
1.75	2	1	1
2.5	3	2	2

See Also

`ceil`, `convergent`, `fix`, `nearest`, `round`

Purpose Plot function between specified limits

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

fractionlength

Purpose Fraction length of `quantizer` object

Syntax

```
fractionlength(q)
```

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

Algorithm For floating-point `quantizer` objects, $f = w - e - 1$, where w is the word length and e is the exponent length.

For fixed-point `quantizer` objects, f is part of the format $[w\ f]$.

See Also `fi`, `numerictype`, `quantizer`, `wordlength`

Purpose

Determine whether real-world value of one `fi` object is greater than or equal to another

Syntax

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

Description

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

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

See Also

`eq`, `gt`, `le`, `lt`, `ne`

get

Purpose

Property values of object

Syntax

```
value = get(o,'propertyname')
structure = get(o)
```

Description

`value = get(o,'propertyname')` returns the property value of the property `'propertyname'` for the object `o`. If you replace the string `'propertyname'` by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(o)` returns a structure containing the properties and states of object `o`.

`o` can be a `fi`, `fimath`, `fipref`, `numerictype`, or `quantizer` object.

See Also

`set`

Purpose

Least significant bit

Syntax

```
c = getlsb(a)
```

Description

`c = getlsb(a)` returns the value of the least significant bit in `a` as a `u1,0`.

`a` can be a scalar `fi` object or a vector `fi` object.

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

Examples

The following example uses `getlsb` to find the least significant bit in the `fi` object *a*.

```
a = fi(-26, 1, 6, 0);
c = getlsb(a)

c =

     0

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

You can verify that the least significant bit in the `fi` object *a* is `0` by looking at the binary representation of *a*.

```
disp(bin(a))

100110
```

See Also

`bitand`, `bitandreduce`, `bitconcat`, `bitget`, `bitor`, `bitorreduce`, `bitset`, `bitxor`, `bitxorreduce`, `getmsb`

getmsb

Purpose

Most significant bit

Syntax

```
c = getmsb(a)
```

Description

`c = getmsb(a)` returns the value of the most significant bit in `a` as a `u1,0`.

`a` can be a scalar `fi` object or a vector `fi` object.

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

Examples

The following example uses `getmsb` to find the most significant bit in the `fi` object *a*.

```
a = fi(-26, 1, 6, 0);
c = getmsb(a)

c =

     1

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

You can verify that the most significant bit in the `fi` object *a* is `1` by looking at the binary representation of *a*.

```
disp(bin(a))

100110
```

See Also

`bitand`, `bitandreduce`, `bitconcat`, `bitget`, `bitor`, `bitorreduce`, `bitset`, `bitxor`, `bitxorreduce`, `getlsb`

Purpose

Configure global fimath and return handle object

Syntax

```
G = globalfimath
G = globalfimath(f)
G = globalfimath('PropertyName1',PropertyValue1,...)
```

Description

`G = globalfimath` returns a handle object to the global fimath.

`G = globalfimath(f)` sets the properties of the global fimath to match those of the input `fimath` object *f*, and returns a handle object to it.

`G = globalfimath('PropertyName1',PropertyValue1,...)` sets the global fimath using the named properties and their corresponding values. Properties that you do not specify in this syntax are automatically set to that of the current global fimath.

Examples

This example shows you how to use the `globalfimath` function to set, change and reset the global fimath.

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap');
globalfimath(F);
F1 = fimath; % Will be the same as F
A = fi(pi); % A associates with the global fimath

% Now set the "SumMode" property of the global fimath to
% "KeepMSB" and retain all the other property values
% of the current global fimath.
G = globalfimath('SumMode','KeepMSB');

% It is also possible to change the global fimath by
% directly interacting with the handle object G.
G.ProductMode = 'SpecifyPrecision';

% The global fimath may also be reset to the factory
% default by calling the reset method on G. This is
% equivalent to using the resetglobalfimath function.
reset(G);
```

See Also

fimath | removeglobalfimathpref | resetglobalfimath | saveglobalfimathpref

How To

- "Working with the Global fimath"

Purpose Plot set of nodes using adjacency matrix

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

Purpose

Determine whether real-world value of one `fi` object is greater than another

Syntax

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

Description

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

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

See Also

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

Purpose Hankel matrix

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

hex

Purpose

Hexadecimal representation of stored integer of `fi` object

Syntax

`hex(a)`

Description

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

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

Examples

Viewing fi Objects in Hexadecimal Format

The following code

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

returns

```
y =

  80   7f

z =

  80   7f
```

Writing Hex Data to a File

The following example shows how to write hex data from the MATLAB workspace into a text file.

First, define your data and create a writable text file called `hexdata.txt`:

```
x = (0:15)'/16;
a = fi(x,0,16,16);

h = fopen('hexdata.txt','w');
```

Use the `fprintf` function to write your data to the `hexdata.txt` file:

```
for k=1:length(a)
    fprintf(h,'%s\n',hex(a(k)));
end
fclose(h);
```

To see the contents of the file you created, use the `type` function:

```
type hexdata.txt
```

MATLAB returns:

```
0000
1000
2000
3000
4000
5000
6000
7000
8000
9000
a000
b000
c000
```

```
d000
e000
f000
```

Reading Hex Data from a File

The following example shows how to read hex data from a text file back into the MATLAB workspace.

Open hexdata.txt for reading and read its contents into a workspace variable:

```
h = fopen(hexdata.txt','r');

nextline = '';
str='';
while ischar(nextline)
    nextline = fgetl(h);
    if ischar(nextline)
        str = [str;nextline];
    end
end
```

Create a fi object with the correct scaling and assign it the hex values stored in the str variable:

```
b = fi([],0,16,16);
b.hex = str

b =
        0
   0.0625
   0.1250
   0.1875
   0.2500
   0.3125
   0.3750
   0.4375
```

```
0.5000
0.5625
0.6250
0.6875
0.7500
0.8125
0.8750
0.9375

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

See Also

bin, dec, int, oct

hex2num

Purpose

Convert hexadecimal string to number using `quantizer` object

Syntax

```
x = hex2num(q,h)
[x1,x2,...] = hex2num(q,h1,h2,...)
```

Description

`x = hex2num(q,h)` converts hexadecimal string `h` to numeric matrix `x`. The attributes of the numbers in `x` are specified by `quantizer` object `q`. When `h` is a cell array containing hexadecimal strings, `hex2num` returns `x` as a cell array of the same dimension containing numbers. For fixed-point hexadecimal strings, `hex2num` uses two's complement representation. For floating-point strings, the representation is IEEE Standard 754 style.

When there are fewer hexadecimal digits than needed to represent the number, the fixed-point conversion zero-fills on the left. Floating-point conversion zero-fills on the right.

`[x1,x2,...]  = hex2num(q,h1,h2,...)` converts hexadecimal strings `h1`, `h2`,... to numeric matrices `x1`, `x2`,....

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

Examples

To create all the 4-bit fixed-point two's complement numbers in fractional form, use the following code.

```
q = quantizer([4 3]);
h = ['7 3 F B';'6 2 E A';'5 1 D 9';'4 0 C 8'];
x = hex2num(q,h)

x =

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

See Also

`bin2num`, `num2bin`, `num2hex`, `num2int`

Purpose Create histogram plot

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

Purpose Histogram count

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

Purpose

Horizontally concatenate multiple `fi` objects

Syntax

```
c = horzcat(a,b,...)
[a, b, ...]
```

Description

`c = horzcat(a,b,...)` is called for the syntax `[a, b, ...]` when any of `a`, `b`, ... , is a `fi` object.

`[a b, ...]` or `[a,b, ...]` is the horizontal concatenation of matrices `a` and `b`. `a` and `b` must have the same number of rows. Any number of matrices can be concatenated within one pair of brackets. N-D arrays are horizontally concatenated along the second dimension. The first and remaining dimensions must match.

Horizontal and vertical concatenation can be combined together as in `[1 2;3 4]`.

`[a b; c]` is allowed if the number of rows of `a` equals the number of rows of `b`, and if the number of columns of `a` plus the number of columns of `b` equals the number of columns of `c`.

The matrices in a concatenation expression can themselves be formed via a concatenation as in `[a b;[c d]]`.

Note The `fimath` and `numerictype` properties of a concatenated matrix of `fi` objects `c` are taken from the leftmost `fi` object in the list `(a,b,...)`.

See Also

`vertcat`

imag

Purpose Imaginary part of complex number

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

Purpose

Number of integer bits needed for fixed-point inner product

Syntax

`innerprodintbits(a,b)`

Description

`innerprodintbits(a,b)` computes the minimum number of integer bits necessary in the inner product of `a'*b` to guarantee that no overflows occur and to preserve best precision.

- `a` and `b` are `fi` vectors.
- The values of `a` are known.
- Only the numeric type of `b` is relevant. The values of `b` are ignored.

Examples

The primary use of this function is to determine the number of integer bits necessary in the output `Y` of an FIR filter that computes the inner product between constant coefficient row vector `B` and state column vector `Z`. For example,

```
for k=1:length(X);
  Z = [X(k);Z(1:end-1)];
  Y(k) = B * Z;
end
```

Algorithm

In general, an inner product grows `log2(n)` bits for vectors of length `n`. However, in the case of this function the vector `a` is known and its values do not change. This knowledge is used to compute the smallest number of integer bits that are necessary in the output to guarantee that no overflow will occur.

The largest gain occurs when the vector `b` has the same sign as the constant vector `a`. Therefore, the largest gain due to the vector `a` is `a*sign(a')`, which is equal to `sum(abs(a))`.

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

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

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

Purpose

Smallest built-in integer fitting stored integer value of `fi` object

Syntax

```
c = int(a)
```

Description

`c = int(a)` returns the smallest built-in integer of the data type in which the stored integer value of `fi` object `a` fits. `int(a)` is equivalent to `a.int`.

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

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 bits < word length <= 64 bits	`int64`	`uint64`
64 < word length	`double`	`double`

Note When the word length is greater than 52 bits, the return value can have quantization error. For bit-true integer representation of very large word lengths, use `bin`, `oct`, `dec`, `hex`, or `sdec`.

Examples

The following code

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

returns

```
y =

   -128   127

z =

   -128   127
```

See Also

`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`

Purpose

Stored integer value of `fi` object as built-in `int8`

Syntax

```
c = int8(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`

int16

Purpose

Stored integer value of `fi` object as built-in `int16`

Syntax

`c = int16(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`

Purpose Stored integer value of `fi` object as built-in `int32`

Syntax `c = int32(a)`

Description Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`

int64

Purpose

Stored integer value of `fi` object as built-in `int64`

Syntax

```
c = int64(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = int64(a)` returns the stored integer value of `fi` object `a` as a built-in `int64`. If the stored integer word length is too big for an `int64`, or if the stored integer is unsigned, the returned value saturates to an `int64`.

See Also

`int`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`, `uint64`

Purpose

Largest positive stored integer value representable by `numerictype` of `fi` object

Syntax

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

Description

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

See Also

`eps`, `intmin`, `lowerbound`, `lsb`, `range`, `realmax`, `realmin`, `stripscaling`, `upperbound`

intmin

Purpose

Smallest stored integer value representable by `numerictype` of `fi` object

Syntax

`x = intmin(a)`

Description

`x = intmin(a)` returns the smallest stored integer value representable by the `numerictype` of `a`.

Examples

```
a = fi(pi, true, 16, 12);
x = intmin(a)


x =

      -32768

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

See Also

`eps`, `intmax`, `lowerbound`, `lsb`, `range`, `realmax`, `realmin`, `stripscaling`, `upperbound`

Purpose Inverse permute dimensions of multidimensional array

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

isboolean

Purpose

Determine whether input is Boolean

Syntax

```
y = isboolean(a)
y = isboolean(T)
```

Description

`y = isboolean(a)` returns `1` when the `DataType` property of `fi` object `a` is `boolean`, and `0` otherwise.

`y = isboolean(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `boolean`, and `0` otherwise.

See Also

`isdouble`, `isfixed`, `isfloat`, `isscaleddouble`, `issingle`

Purpose

Determine whether `fi` object is column vector

Syntax

```
y = iscolumn(a)
```

Description

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

See Also

`isrow`

isdouble

Purpose

Determine whether input is double-precision data type

Syntax

```
y = isdouble(a)
y = isdouble(T)
```

Description

`y = isdouble(a)` returns `1` when the `DataType` property of `fi` object `a` is `double`, and `0` otherwise.

`y = isdouble(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `double`, and `0` otherwise.

See Also

`isboolean`, `isdouble`, `isfixed`, `isfloat`, `isscaleddouble`, `isscaledtype`, `issingle`

Purpose Determine whether array is empty

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

isequal

Purpose

Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal

Syntax

```
y = isequal(a,b,...)
y = isequal(F,G,...)
y = isequal(T,U,...)
y = isequal(q,r,...)
```

Description

`y = isequal(a,b,...)` returns `1` if all the `fi` object inputs have the same real-world value. Otherwise, the function returns `0`.

`y = isequal(F,G,...)` returns `1` if all the `fimath` object inputs have the same properties. Otherwise, the function returns `0`.

`y = isequal(T,U,...)` returns `1` if all the `numerictype` object inputs have the same properties. Otherwise, the function returns `0`.

`y = isequal(q,r,...)` returns `1` if all the `quantizer` object inputs have the same properties. Otherwise, the function returns `0`.

See Also

`eq`, `ispropequal`

Purpose Determine whether variable is `fi` object

Syntax `y = isfi(a)`

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

See Also `fi`, `isfimath`, `isfipref`, `isnumerictype`, `isquantizer`

isfimath

Purpose Determine whether variable is `fimath` object

Syntax

```
y = isfimath(F)
```

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

See Also `fimath`, `isfi`, `isfipref`, `isnumerictype`, `isquantizer`

Purpose

Determine whether `fi` object has local fimath

Syntax

```
y = isfimathlocal(a)
```

Description

`y = isfimathlocal(a)` returns `1` if the `fi` object `a` has a local `fimath` object, and 0 if `a` is associated with the global fimath.

See Also

`fimath`, `isfi`, `isfipref`, `isnumerictype`, `isquantizer`, `sfi`, `ufi`

isfinite

Purpose Determine whether array elements are finite

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

Purpose Determine whether input is `fipref` object

Syntax

```
y = isfipref(P)
```

Description `y = isfipref(P)` returns `1` if `P` is a `fipref` object, and `0` otherwise.

See Also `fipref`, `isfi`, `isfimath`, `isnumerictype`, `isquantizer`

isfixed

Purpose

Determine whether input is fixed-point data type

Syntax

```
y = isfixed(a)
y = isfixed(T)
y = isfixed(q)
```

Description

`y = isfixed(a)` returns `1` when the `DataType` property of `fi` object `a` is `Fixed`, and `0` otherwise.

`y = isfixed(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `Fixed`, and `0` otherwise.

`y = isfixed(q)` returns `1` when `q` is a fixed-point `quantizer`, and `0` otherwise.

See Also

`isboolean`, `isdouble`, `isfloat`, `isscaleddouble`, `isscaledtype`, `issingle`

Purpose Determine whether input is floating-point data type

Syntax

```
y = isfloat(a)
y = isfloat(T)
y = isfloat(q)
```

Description

`y = isfloat(a)` returns `1` when the `DataType` property of `fi` object `a` is `single` or `double`, and `0` otherwise.

`y = isfloat(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `single` or `double`, and `0` otherwise.

`y = isfloat(q)` returns `1` when `q` is a floating-point `quantizer`, and `0` otherwise.

See Also `isboolean`, `isdouble`, `isfixed`, `isscaleddouble`, `isscaledtype`, `issingle`

isinf

Purpose Determine whether array elements are infinite

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

Purpose Determine whether array elements are NaN

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

Purpose Determine whether input is numeric array

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

Purpose Determine whether input is `numerictype` object

Syntax

```
y = isnumerictype(T)
```

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

See Also `isfi`, `isfimath`, `isfipref`, `isquantizer`, `numerictype`

Purpose Determine whether input is MATLAB object

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

Purpose

Determine whether properties of two `fi` objects are equal

Syntax

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

Description

`y = ispropequal(a,b,...)` returns `1` if all the inputs are `fi` objects and all the inputs have the same properties. Otherwise, the function returns `0`.

To compare the real-world values of two `fi` objects `a` and `b`, use `a == b` or `isequal(a,b)`.

See Also

`fi`, `isequal`

isquantizer

Purpose Determine whether input is `quantizer` object

Syntax

```
y = isquantizer(q)
```

Description `y = isquantizer(q)` returns `1` when `q` is a `quantizer` object, and `0` otherwise.

See Also `quantizer`, `isfi`, `isfimath`, `isfipref`, `isnumerictype`

Purpose Determine whether array elements are real

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

isrow

Purpose Determine whether `fi` object is row vector

Syntax

```
y = isrow(a)
```

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

See Also `iscolumn`

Purpose Determine whether input is scalar

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

isscaleddouble

Purpose

Determine whether input is scaled double data type

Syntax

```
y = isscaleddouble(a)
y = isscaleddouble(T)
```

Description

`y = isscaleddouble(a)` returns `1` when the `DataType` property of `fi` object `a` is `ScaledDouble`, and `0` otherwise.

`y = isscaleddouble(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `ScaledDouble`, and `0` otherwise.

See Also

`isboolean`, `isdouble`, `isfixed`, `isfloat`, `isscaledtype`, `issingle`

Purpose

Determine whether input is fixed-point or scaled double data type

Syntax

```
y = isscaledtype(a)
y = isscaledtype(T)
```

Description

`y = isscaledtype(a)` returns `1` when the `DataType` property of `fi` object `a` is `Fixed` or `ScaledDouble`, and `0` otherwise.

`y = isscaledtype(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `Fixed` or `ScaledDouble`, and `0` otherwise.

See Also

`isboolean`, `isdouble`, `isfixed`, `isfloat`, `numerictype`, `isscaleddouble`, `issingle`

issigned

Purpose

Determine whether `fi` object is signed

Syntax

```
y = issigned(a)
```

Description

`y = issigned(a)` returns `1` if the `fi` object `a` is signed, and `0` if it is unsigned.

Purpose

Determine whether input is single-precision data type

Syntax

```
y = issingle(a)
y = issingle(T)
```

Description

`y = issingle(a)` returns `1` when the `DataType` property of `fi` object `a` is `single`, and `0` otherwise.

`y = issingle(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `single`, and `0` otherwise.

See Also

`isboolean`, `isdouble`, `isfixed`, `isfloat`, `isscaleddouble`, `isscaledtype`

isslopebiasscaled

Purpose

Determine whether `numerictype` object has nontrivial slope and bias

Syntax

```
y = isslopebiasscaled(T)
```

Description

`y = isslopebiasscaled(T)` returns `1` when `numerictype` object `T` has nontrivial slope and bias scaling, and `0` otherwise. Slope and bias scaling is trivial when the slope is an integer power of 2, and the bias is `0`.

See Also

`isboolean`, `isdouble`, `isfixed`, `isfloat`, `isscaleddouble`, `isscaledtype`, `issingle`, `numerictype`

Purpose Determine whether input is vector

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

Purpose Determine whether real-world value of `fi` object is less than or equal to another

Syntax

```
c = le(a,b)
a <= b
```

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

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

See Also `eq`, `ge`, `gt`, `lt`, `ne`

Purpose Vector length

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

Purpose

Create line object

Description

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

Purpose Convert numeric values to logical

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

loglog

Purpose Create log-log scale plot

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

Purpose

Quantization report

Syntax

```
logreport(a)
logreport(a, b, ...)
```

Description

`logreport(a)` displays the `minlog`, `maxlog`, `lowerbound`, `upperbound`, `noverflows`, and `nunderflows` for the `fi` object `a`.

`logreport(a, b, ...)` displays the report for each `fi` object `a, b, ...` .

Examples

The following example produces a `logreport` for `fi` objects `a` and `b`:

```
fipref('LoggingMode','On');
a = fi(pi);
b = fi(randn(10),1,8,7);

Warning: 27 overflows occurred in the fi assignment operation.
Warning: 1 underflow occurred in the fi assignment operation.

logreport(a,b)
        minlog      maxlog  lowerbound    upperbound   noverflows   nunderflows
 a    3.141602    3.141602          -4      3.999878            0             0
 b          -1   0.9921875          -1     0.9921875           27             1
```

See Also

`fipref`, `quantize`, `quantizer`

lowerbound

Purpose Lower bound of range of `fi` object

Syntax `lowerbound(a)`

Description `lowerbound(a)` returns the lower bound of the range of `fi` object `a`. If `L=lowerbound(a)` and `U=upperbound(a)`, then `[L,U]=range(a)`.

See Also `eps`, `intmax`, `intmin`, `lsb`, `range`, `realmax`, `realmin`, `upperbound`

Purpose

Scaling of least significant bit of `fi` object, or value of least significant bit of `quantizer` object

Syntax

```
b = lsb(a)
p = lsb(q)
```

Description

`b = lsb(a)` returns the scaling of the least significant bit of `fi` object `a`. The result is equivalent to the result given by the `eps` function.

`p = lsb(q)` returns the quantization level of `quantizer` object `q`, or the distance from `1.0` to the next largest floating-point number if `q` is a floating-point `quantizer` object.

Examples

This example uses the `lsb` function to find the value of the least significant bit of the `quantizer` object `q`.

```
q = quantizer('fixed',[8 7]);
p = lsb(q)

p =

    0.0078
```

See Also

`eps`, `intmax`, `intmin`, `lowerbound`, `quantize`, `range`, `realmax`, `realmin`, `upperbound`

Purpose

Determine whether real-world value of one `fi` object is less than another

Syntax

```
c = lt(a,b)
a < b
```

Description

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

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

See Also

`eq`, `ge`, `gt`, `le`, `ne`

Purpose

Largest element in array of `fi` objects

Syntax

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

Description

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

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

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

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

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

See Also

`mean`, `median`, `min`, `sort`

maxlog

Purpose

Log maximums

Syntax

```
y = maxlog(a)
y = maxlog(q)
```

Description

`y = maxlog(a)` returns the largest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = maxlog(q)` is the maximum value after quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the maximum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `maxlog(q)` is equivalent to `get(q,'maxlog')` and `q.maxlog`.

Examples

Example 1: Using maxlog with fi objects

```
P = fipref('LoggingMode','on');
format long g
a = fi([-1.5 eps 0.5], true, 16, 15);
a(1) = 3.0;
maxlog(a)

Warning: 1 overflow occurred in the fi assignment operation.
> In embedded.fi.fi at 510
  In fi at 220
Warning: 1 underflow occurred in the fi assignment operation.
> In embedded.fi.fi at 510
  In fi at 220
Warning: 1 overflow occurred in the fi assignment operation.

ans =

         0.999969482421875
```

The largest value maxlog can return is the maximum representable value of its input. In this example, a is a signed fi object with word length 16, fraction length 15 and range:

$$-1 \le x \le 1 - 2^{-15}$$

You can obtain the numerical range of any fi object a using the range function:

```
format long g
r = range(a)

r =

                                    -1          0.999969482421875
```

Example 2: Using maxlog with quantizer objects

```
q = quantizer;
warning on
format long g
x = [-20:10];
y = quantize(q,x);
maxlog(q)

Warning: 29 overflows.
> In embedded.quantizer.quantize at 74

ans =

   .999969482421875
```

The largest value maxlog can return is the maximum representable value of its input. You can obtain the range of x after quantization using the range function:

```
format long g
r = range(q)
```

```
r =

                -1    0.999969482421875
```

See Also

`fipref`, `minlog`, `noverflows`, `nunderflows`, `reset`, `resetlog`

Purpose

Average or mean value of fixed-point array

Syntax

```
c = mean(a)
c = mean(a,dim)
```

Description

`c = mean(a)` computes the mean value of the fixed-point array *a* along its first nonsingleton dimension.

`c = mean(a,dim)` computes the mean value of the fixed-point array *a* along dimension *dim*. *dim* must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The input to the `mean` function must be a real-valued fixed-point array.

The fixed-point output array *c* has the same `numerictype` properties as the fixed-point input array *a* and is always associated with the global fimath.

When *a* is an empty fixed-point array (value = `[]`), the value of the output array is zero.

Examples

Compute the mean value along the first dimension (rows) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 28-bits
mx1 = mean(x,1)
```

Compute the mean value along the second dimension (columns) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 28 bits
mx2 = mean(x,2)
```

Algorithm

The general equation for computing the `mean` of an array *a*, across dimension *dim* is:

```
sum(a,dim)/size(a,dim)
```

Because `size(a,dim)` is always a positive integer, the algorithm casts `size(a,dim)` to an unsigned 32-bit `fi` object with a fraction length of zero (`SizeA`). The algorithm then computes the mean of *a* according to the following equation, where `Tx` represents the `numerictype` properties of the fixed-point input array *a*:

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

See Also

`max` | `median` | `min`

Purpose

Median value of fixed-point array

Syntax

```
c = median(a)
c = median(a,dim)
```

Description

`c = median(a)` computes the median value of the fixed-point array *a* along its first nonsingleton dimension.

`c = median(a,dim)` computes the median value of the fixed-point array *a* along dimension *dim*. *dim* must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The input to the `median` function must be a real-valued fixed-point array.

The fixed-point output array *c* has the same `numerictype` properties as the fixed-point input array *a* and is always associated with the global fimath.

When *a* is an empty fixed-point array (value = `[]`), the value of the output array is zero.

Examples

Compute the median value along the first dimension of a fixed-point array.

```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx1 = median(x,1)
```

Compute the median value along the second dimension (columns) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx2 = median(x, 2)
```

See Also

`max` | `mean` | `min`

Purpose Create mesh plot

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

Purpose

Create mesh plot with contour plot

Description

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

Purpose Create mesh plot with curtain plot

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

min

Purpose Smallest element in array of `fi` objects

Syntax

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

Description

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

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

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

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

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

See Also `max`, `mean`, `median`, `sort`

Purpose Log minimums

Syntax

```
y = minlog(a)
y = minlog(q)
```

Description

`y = minlog(a)` returns the smallest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = minlog(q)` is the minimum value after quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the minimum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `minlog(q)` is equivalent to `get(q,'minlog')` and `q.minlog`.

Examples

Example 1: Using minlog with fi objects

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

ans =

     -1
```

The smallest value `minlog` can return is the minimum representable value of its input. In this example, `a` is a signed `fi` object with word length `16`, fraction length `15` and range:

$$-1 \leq x \leq 1 - 2^{-15}$$

You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)

r =

                        -1         0.999969482421875
```

Example 2: Using minlog with quantizer objects

```
q = quantizer;
warning on
x = [-20:10];
y = quantize(q,x);
minlog(q)

Warning: 29 overflows.
> In embedded.quantizer.quantize at 74

ans =

    -1
```

The smallest value `minlog` can return is the minimum representable value of its input. You can obtain the range of `x` after quantization using the `range` function:

```
format long g
r = range(q)

r =

                        -1         0.999969482421875
```

See Also

`fipref`, `maxlog`, `noverflows`, `nunderflows`, `reset`, `resetlog`

Purpose

Matrix difference between `fi` objects

Syntax

`minus(a,b)`

Description

`minus(a,b)` is called for the syntax `a - b` when `a` or `b` is an object.

`a - b` subtracts matrix `b` from matrix `a`. `a` and `b` must have the same dimensions unless one is a scalar value (a 1-by-1 matrix). A scalar value can be subtracted from any other value.

`minus` does not support `fi` objects of data type `Boolean`.

Note For information about the `fimath` properties involved in Fixed-Point Toolbox calculations, see "Using fimath Properties to Perform Fixed-Point Arithmetic" and "Using fimath ProductMode and SumMode" in the *Fixed-Point Toolbox User's Guide*.

For information about calculations using Simulink® Fixed Point™ software, see the "Arithmetic Operations" chapter of the *Simulink Fixed Point User's Guide*.

See Also

`mtimes`, `plus`, `times`, `uminus`

Purpose

Fixed-point matrix power (^)

Syntax

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

Description

`c = mpower(a,k)` and `c = a^k` compute matrix power. The exponent *k* requires a positive, real-valued integer value.

The fixed-point output array *c* is always associated with the global fimath.

Tips

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

Examples

Compute the power of a 2-dimensional square matrix for exponent values 0, 1, 2, and 3.

```
x = fi([0 1; 2 4], 1, 32);

px0 = x^0
px1 = x^1
px2 = x^2
px3 = x^3
```

See Also

`arithmeticoperators` | `power`

Purpose

Multiply two objects using `fimath` object

Syntax

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

Description

`c = F.mpy(a,b)` performs elementwise multiplication on `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` is always associated with the global fimath.

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

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

Examples

In this example, `c` is the 40-bit product of `a` and `b` with fraction length 30.

```
a = fi(pi);
b = fi(exp(1));
F = fimath('ProductMode','SpecifyPrecision',...
  'ProductWordLength',40,'ProductFractionLength',30);
c = F.mpy(a, b)

c =

    8.5397


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

Algorithm

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

```
a.fimath = F;
```

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

c =
    8.5397

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

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 40
 ProductFractionLength: 30
               SumMode: FullPrecision
      MaxSumWordLength: 128
```

but not identical. When you use `mpy`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` is associated with the global fimath. When you use the syntax `c = a .* b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See "fimath Rules for Fixed-Point Arithmetic" in the *Fixed-Point Toolbox User's Guide* for more information.

See Also

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

Purpose

Forward slash (/) or right-matrix division

Syntax

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

Description

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

When one or both of the inputs is a `fi` object, the denominator input, `b`, must be a scalar and the output `fi` object `c` is equivalent to `c = rdivide(a,b)` or `c = a./b` (right-array division).

The numerator input `a` can be complex, but the denominator input `b` must always be real-valued. When the numerator input `a` is complex, the real and imaginary parts of `a` are independently divided by `b`.

For information on the data type rules used by the `mrdivide` function, see the `rdivide` reference page.

Examples

In this example, you use the forward slash (/) to perform right matrix division on a 3-by-3 magic square of `fi` objects. Because the numerator input is a `fi` object, the denominator input `b` must be a scalar:

```
a = fi(magic(3))
b = fi(3, 1, 12, 8)
c = a/b
```

The `mrdivide` function outputs a signed 3-by-3 array of `fi` objects, each of which has a word length of 16 bits and a fraction length of 3 bits.

```
a =

      8     1     6
      3     5     7
      4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
b =

     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 8

c =

    2.6250    0.3750    2.0000
    1.0000    1.6250    2.3750
    1.3750    3.0000    0.6250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 3
```

See Also

add, divide, fi, fimath, numerictype, rdivide, sub, sum

Purpose Matrix product of `fi` objects

Syntax `mtimes(a,b)`

Description `mtimes(a,b)` is called for the syntax `a * b` when `a` or `b` is an object.

`a * b` is the matrix product of `a` and `b`. A scalar value (a 1-by-1 matrix) can multiply any other value. Otherwise, the number of columns of `a` must equal the number of rows of `b`.

`mtimes` does not support `fi` objects of data type `Boolean`.

Note For information about the `fimath` properties involved in Fixed-Point Toolbox calculations, see "Using fimath Properties to Perform Fixed-Point Arithmetic" and "Using fimath ProductMode and SumMode" in the *Fixed-Point Toolbox User's Guide*.

For information about calculations using Simulink Fixed Point software, see the "Arithmetic Operations" chapter of the *Simulink Fixed Point User's Guide*.

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

ndgrid

Purpose Generate arrays for N-D functions and interpolation

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

Purpose Number of array dimensions

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

Purpose

Determine whether real-world values of two `fi` objects are not equal

Syntax

```
c = ne(a,b)
a ~= b
```

Description

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

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

See Also

`eq`, `ge`, `gt`, `le`, `lt`

Purpose

Round toward nearest integer with ties rounding toward positive infinity

Syntax

`y = nearest(a)`

Description

`y = nearest(a)` rounds `fi` object `a` to the nearest integer or, in case of a tie, to the nearest integer in the direction of positive infinity, and returns the result in `fi` object `y`.

`y` and `a` have the same `fimath` object and `DataType` property.

When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`nearest` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Examples

Example 1

The following example demonstrates how the `nearest` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =
```

```
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
y = nearest(a)

y =

     3

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

Example 2

The following example demonstrates how the `nearest` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12

y = nearest(a)

y =
```

```
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

The following table illustrates these differences for a given `fi` object `a`.

a	convergent(a)	nearest(a)	round(a)
–3.5	–4	–3	–4
–2.5	–2	–2	–3
–1.5	–2	–1	–2
–0.5	0	0	–1
0.5	0	1	1
1.5	2	2	2
2.5	2	3	3
3.5	4	4	4

See Also

`ceil`, `convergent`, `fix`, `floor`, `round`

noperations

Purpose Number of operations

Syntax

```
noperations(q)
```

Description noperations(q) is the number of quantization operations during a call to quantize(q,...) for quantizer object q. This value accumulates over successive calls to quantize. You reset the value of noperations to zero by issuing the command resetlog(q).

Each time any data element is quantized, noperations is incremented by one. The real and complex parts are counted separately. For example, (complex * complex) counts four quantization operations for products and two for sum, because(a+bi)*(c+di) = (a*c - b*d) + (a*d + b*c). In contrast, (real*real) counts one quantization operation.

In addition, the real and complex parts of the inputs are quantized individually. As a result, for a complex input of length 204 elements, noperations counts 408 quantizations: 204 for the real part of the input and 204 for the complex part.

If any inputs, states, or coefficients are complex-valued, they are all expanded from real values to complex values, with a corresponding increase in the number of quantization operations recorded by noperations. In concrete terms, (real*real) requires fewer quantizations than (real*complex) and (complex*complex). Changing all the values to complex because one is complex, such as the coefficient, makes the (real*real) into (real*complex), raising noperations count.

See Also maxlog, minlog

Purpose Find logical NOT of array or scalar input

Description Refer to the MATLAB `not` reference page for more information.

noverflows

Purpose

Number of overflows

Syntax

```
y = noverflows(a)
y = noverflows(q)
```

Description

`y = noverflows(a)` returns the number of overflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = noverflows(q)` returns the accumulated number of overflows resulting from quantization operations performed by a `quantizer` object `q`.

See Also

`maxlog`, `minlog`, `nunderflows`, `resetlog`

Purpose

Convert number to binary string using `quantizer` object

Syntax

```
y = num2bin(q,x)
```

Description

`y = num2bin(q,x)` converts numeric array `x` into binary strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to binary. If `x` is a structure, each numeric field of `x` is converted to binary.

`num2bin` and `bin2num` are inverses of one another, differing in that `num2bin` returns the binary strings in a column.

Examples

```
x = magic(3)/9;
q = quantizer([4,3]);
y = num2bin(q,x)

Warning: 1 overflow.

y =

0111
0010
0011
0000
0100
0111
0101
0110
0001
```

See Also

`bin2num`, `hex2num`, `num2hex`, `num2int`

num2hex

Purpose

Convert number to hexadecimal equivalent using `quantizer` object

Syntax

```
y = num2hex(q,x)
```

Description

`y = num2hex(q,x)` converts numeric array `x` into hexadecimal strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to hexadecimal. If `x` is a structure, each numeric field of `x` is converted to hexadecimal.

For fixed-point `quantizer` objects, the representation is two's complement. For floating-point `quantizer` objects, the representation is IEEE Standard 754 style.

For example, for `q = quantizer('double')`

```
num2hex(q,nan)

ans =

fff8000000000000
```

The leading fraction bit is 1, all other fraction bits are 0. Sign bit is 1, exponent bits are all 1.

```
num2hex(q,inf)

ans =

7ff0000000000000
```

Sign bit is 0, exponent bits are all 1, all fraction bits are 0.

```
num2hex(q,-inf)

ans =

fff0000000000000
```

Sign bit is 1, exponent bits are all 1, all fraction bits are 0.

`num2hex` and `hex2num` are inverses of each other, except that `num2hex` returns the hexadecimal strings in a column.

Examples

This is a floating-point example using a `quantizer` object `q` that has 6-bit word length and 3-bit exponent length.

```
x = magic(3);
q = quantizer('float',[6 3]);
y = num2hex(q,x)

y =

18
12
14
0c
15
18
16
17
10
```

See Also

`bin2num`, `hex2num`, `num2bin`, `num2int`

num2int

Purpose

Convert number to signed integer

Syntax

```
y = num2int(q,x)
[y1,y,...] = num2int(q,x1,x,...)
```

Description

`y = num2int(q,x)` uses `q.format` to convert numeric `x` to an integer.

`[y1,y,...]  = num2int(q,x1,x,...)` uses `q.format` to convert numeric values `x1, x2,...` to integers `y1,y2,...`

Examples

All the two's complement 4-bit numbers in fractional form are given by

```
x = [0.875 0.375 -0.125 -0.625
     0.750 0.250 -0.250 -0.750
     0.625 0.125 -0.375 -0.875
     0.500 0.000 -0.500 -1.000];

q=quantizer([4 3]);

y = num2int(q,x)

y =

     7     3    -1    -5
     6     2    -2    -6
     5     1    -3    -7
     4     0    -4    -8
```

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`

Purpose

Number of data elements in `fi` array

Syntax

`numberofelements(a)`

Description

`numberofelements(a)` returns the number of data elements in a `fi` array. `numberofelements(a) == prod(size(a))`.

Note that `fi` is a MATLAB object, and therefore `numel(a)` returns `1` when `a` is a `fi` object. Refer to the information about classes in the MATLAB `numel` reference page.

See Also

`max`, `min`, `numel`

numerictype

Purpose

Construct `numerictype` object

Syntax

```
T = numerictype
T = numerictype(s)
T = numerictype(s,w)
T = numerictype(s,w,f)
T = numerictype(s,w,slope,bias)
T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)
T = numerictype(property1,value1, ...)
T = numerictype(T1, property1, value1, ...)
T = numerictype('double')
T = numerictype('single')
T = numerictype('boolean')
```

Description

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

- `T = numerictype` creates a default `numerictype` object.
- `T = numerictype(s)` creates a `numerictype` object with `Fixed-point:  unspecified scaling`, `Signed` property value `s`, and 16-bit word length.
- `T = numerictype(s,w)` creates a `numerictype` object with `Fixed-point:  unspecified scaling`, `Signed` property value `s`, and word length `w`.
- `T = numerictype(s,w,f)` creates a `numerictype` object with `Fixed-point:  binary point scaling`, `Signed` property value `s`, word length `w` and fraction length `f`.
- `T = numerictype(s,w,slope,bias)` creates a `numerictype` object with `Fixed-point:  slope and bias scaling`, `Signed` property value `s`, word length `w`, `slope`, and `bias`.
- `T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)` creates a `numerictype` object with `Fixed-point:  slope and bias scaling`, `Signed` property value `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

- `T = numerictype(property1,value1, ...)` allows you to set properties for a `numerictype` object using property name/property value pairs. All properties for which you do not specify a value get assigned their default value.
- `T = numerictype(T1, property1, value1, ...)` allows you to make a copy of an existing `numerictype` object, while modifying any or all of the property values.
- `T = numerictype('double')` creates a `double` `numerictype`.
- `T = numerictype('single')` creates a `single` `numerictype`.
- `T = numerictype('boolean')` creates a `Boolean` `numerictype`.

The properties of the `numerictype` object are listed below. These properties are described in detail in "numerictype Object Properties" on page 1-15.

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

Examples

Example 1

Type

```
T = numerictype
```

to create a default `numerictype` object.

```
T =


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

Example 2

The following code creates a signed `numerictype` object with a 32-bit word length and 30-bit fraction length.

```
T = numerictype(1, 32, 30)

T =

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

Example 3

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

```
T = numerictype(1, 32)

T =

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

Example 4

If you omit the arguments `w` and `f`, the word length is automatically set to 16 bits and the scaling is unspecified.

```
T = numerictype(1)

T =

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

Example 5

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

```
T = numerictype('Signed', true, ...
      'DataTypeMode', 'Fixed-point: slope and bias', ...
      'WordLength', 32, 'Slope', 2^-2, 'Bias', 4)

T =

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

Note When you create a `numerictype` object using property name/property value pairs, Fixed-Point Toolbox software first creates a default `numerictype` object, and then, for each property name you specify in the constructor, assigns the corresponding value. This behavior differs from the behavior that occurs when you use a syntax such as `T = numerictype(s,w)`. See "Example: Constructing a numerictype Object with Property Name and Property Value Pairs" in the *Fixed-Point Toolbox User's Guide* for more information.

Example 6

You can create a `numerictype` object with an unspecified sign by using property name/property values pairs to set the `Signedness` property to `Auto`.

```
T = numerictype('Signedness', 'Auto')

T =


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

Note Although you can create `numerictype` objects with an unspecified sign (`Signedness:  Auto`), all `fi` objects must have a `Signedness` of `Signed` or `Unsigned`. If you use a `numerictype` object with `Signedness: Auto` to construct a `fi` object, the `Signedness` property of the `fi` object automatically defaults to `Signed`.

See Also

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

Purpose

Determine numeric type for data

Syntax

```
H = NumericTypeScope
show(H)
step(H, data)
reset(H)
```

Description

The `NumericTypeScope` is an object that provides information about the dynamic range of your data. You can use information from the `NumericTypeScope` to help you select appropriate data types. The scope provides a visual representation of the dynamic range of your data in the form of a `log2` histogram with the bit weights represented along the X-axis, and the percentage of occurrences along the Y-axis. Each bin of the histogram corresponds to a bit in the binary word. For example, 2^0 corresponds to the first integer bit in the binary word, 2^{-1} corresponds to the first fractional bit in the binary word, and the binary point lies between them.

`H = NumericTypeScope` returns a `NumericTypeScope` object that you can use to view the dynamic range of data in MATLAB. To view the `NumericTypeScope` window after creating *H*, use the `show` method.

`show(H)` opens the `NumericTypeScope` object *H* and brings it into view. Closing the scope window does not delete the object from your workspace. If the scope object still exists in your workspace, you can open it and bring it back into view using the `show` method.

`step(H, data)` processes your data and allows you to visualize the dynamic range. The object *H* retains previously collected information about the variable between each call to `step`.

`reset(H)` clears all stored information from the `NumericTypeScope` object *H*. Resetting the object does not clear the information displayed in the scope window. The object does not clear the scope window until the next time you use the `step` method.

Identifying Overflows and Underflows

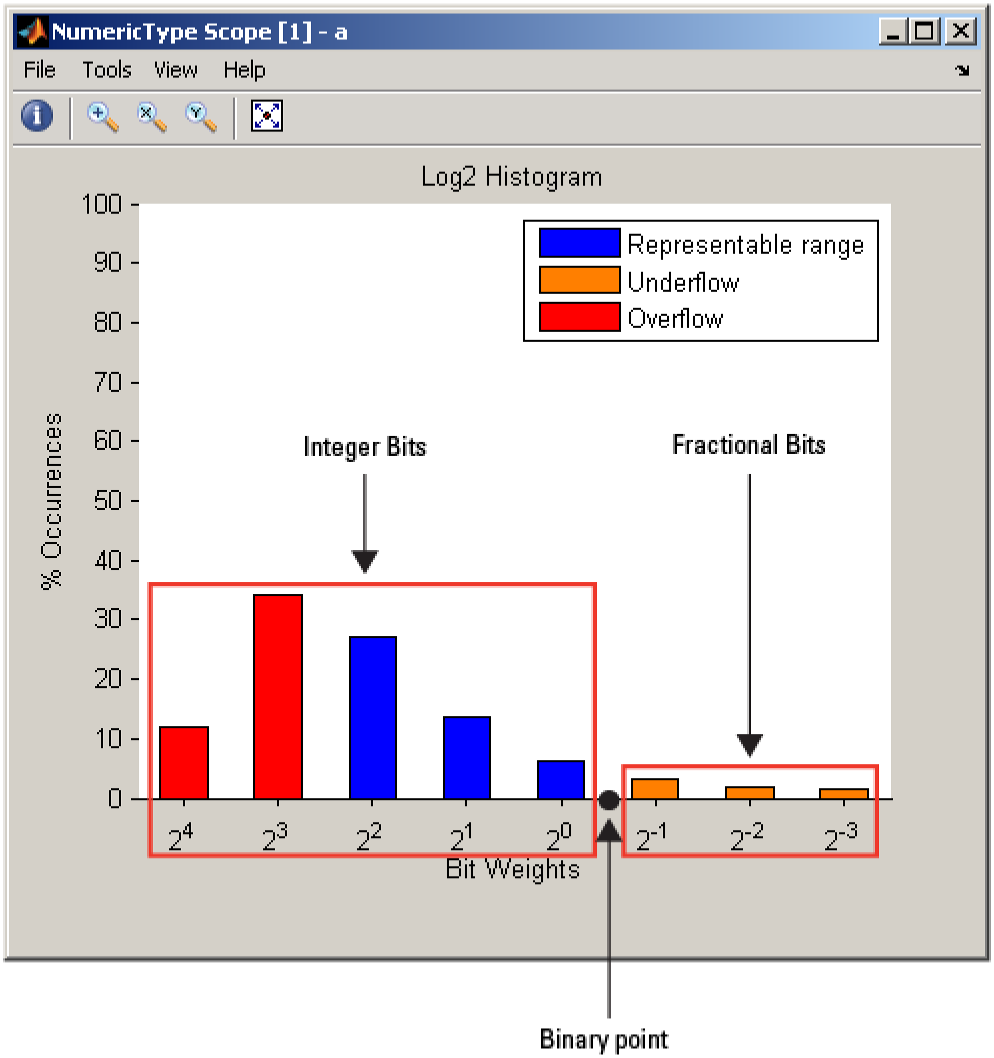
The `NumericTypeScope` object can also help you identify any overflows or underflows that occur, based on the current data type. To prepare the NumericTypeScope to identify overflows and underflows, you must provide an input variable that is a fi object and verify that one of the following conditions is true:

- The DataTypeMode of the fi object is set to Scaled doubles: binary point scaling.
- The DataTypeOverride property of the Fixed-Point Toolbox fipref object is set to ScaledDoubles.

When the information is available, the scope indicates overflow, underflow and the representable range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are within the representable range of the current data type.
- Red — Histogram bin contains values that overflow in the current data type.
- Orange — Histogram bin contains values that underflow in the current data type.

For an example of the scope color coding, see the following figure.



You can choose to show or hide the legend in the scope window by selecting **View > Show Legend** from the the `NumericTypeScope` menu.

Note The scope hides the legend when you call the `step` method on your `NumericTypeScope` object. You can turn it back on at any time by selecting **View > Show Legend**.

See the "Examples" on page 3-281 section to learn more about using the `NumericTypeScope` to select data types.

Methods

reset

Use this method to clear the information stored in the object *H*. Doing so allows you to reuse *H* to process data from a different variable.

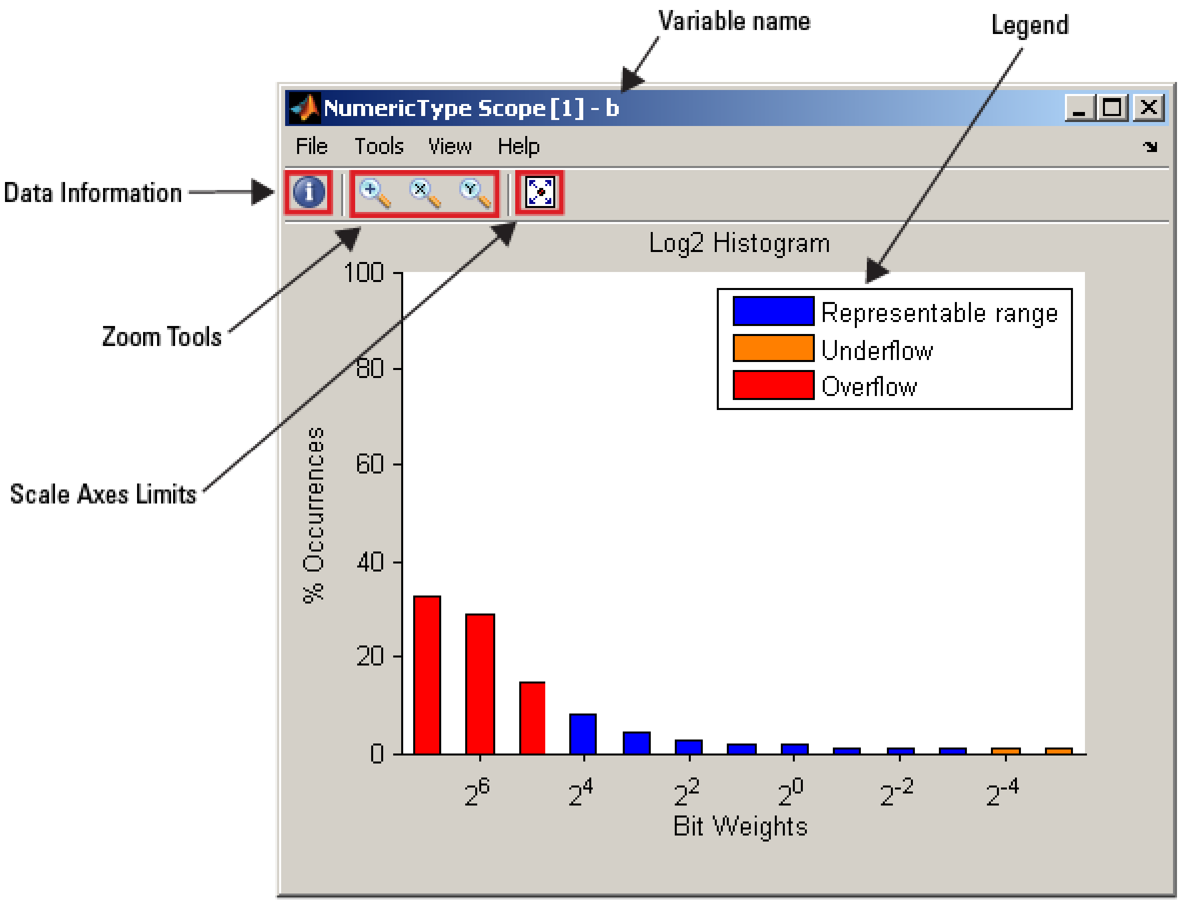
show

Use this method to open the scope window and bring it into view.

step

Use this method to process your data and visualize the dynamic range in the scope window.

Toolbar and Dialog Boxes



Toolbar

The scope toolbar includes the tools described in the following table.

Icon	Menu Location	Shortcut Keys	Description
	**View > Data Information**	**D**	Click this button to display the **Data Information** dialog box for the variable currently displayed in the scope window. For more information about this dialog box, see the Data Information Dialog Box section.
	**Tools > Zoom In**	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.
	**Tools > Zoom X**	N/A	When this tool is active, you can zoom in on the X-axis. To do so, click inside the scope window, or click and drag your cursor along the X-axis over your area of interest.
	**Tools > Zoom Y**	N/A	When this tool is active, you can zoom in on the Y-axis. To do so, click inside the scope window, or click and drag your cursor along the Y-axis over your area of interest.
	**Tools > Scale Axes Limits**	**Ctrl+A**	Click this button to scale the axes of the active scope window.

After zooming in on your data, you can zoom out incrementally by right-clicking inside the scope window and selecting **Zoom Out** from the context menu. Alternatively, you can return directly to the original view by right-clicking inside the scope window and selecting **Reset to Original View**.

You can control whether or not this toolbar appears in the scope window by selecting **View > Toolbar** from the scope menu.

Configuration Dialog Box

The `NumericTypeScope` configuration allows you to control the behavior and appearance of the scope window.

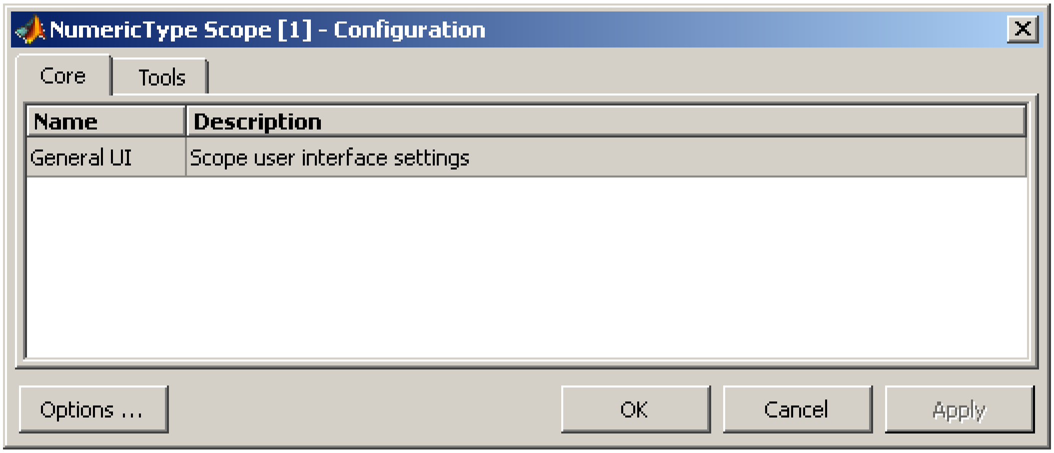
- To open the **Configuration** dialog box, select **File > Configuration > Edit**, or, with the scope as your active window, press the **N** key.
- To save the configuration settings for future use, select **File > Configuration > Save as**. The configuration settings you save become the default configuration settings for the `NumericTypeScope`.

  If you choose to save your configuration settings for future use, you must save them in the `matlab/toolbox/fixedpoint/fixedpoint` folder with the file name `NumericTypeScopeComponent.cfg`. You can resave your configuration settings at anytime, but you must do so in the specified folder using the specified file name.

  Note Before saving your own set of configuration settings in the `matlab/toolbox/fixedpoint/fixedpoint` folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

Core Pane

The Core pane in the **Configuration** dialog box controls the general settings of the scope.



General UI
Click **General UI**, and click the **Options** button to open the General UI Options dialog box.

- **Display the full source path in the title bar** — When you select this check box, the scope displays the file name and variable name in the title bar. If the scope is not from a file, or if you clear this check box, the scope displays only the variable name in the title bar.
- **Open message log** — Use this parameter to control when the Message log window opens. The Message log window helps you debug

any issues with the scope. You can choose to open the Message log window under any of the following conditions:

- `for any new messages`
- `for warn/fail messages`
- `only for fail messages`
- `manually`

You can open the Message Log at any time by selecting **Help > Message Log**. The Message Log dialog box provides a system level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by **Type** and **Category**.

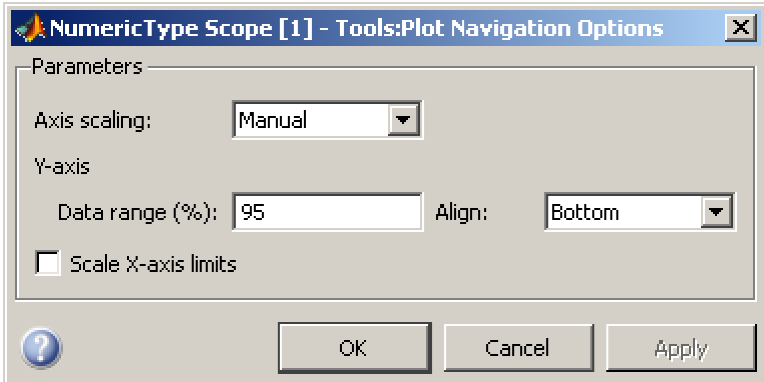
The **Type** parameter allows you to select which types of messages to display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`.

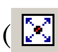
The **Category** parameter allows you to select the category of messages to display in the Message Log. You can select `All`, `Configuration` or `Extension`. The scope uses `Configuration` messages to indicate when new configuration files are loaded, and `Extension` messages to indicate when components are registered.

Tools Pane

The Tools pane in the **Configuration** dialog box contains the Plot Navigation tool, which allows you to control how the scope scales the axes and displays your data.

Plot Navigation
Click **Plot Navigation**, and then click the **Options** button to open the **Tools:Plot Navigation Options** dialog box.



- **Axis Scaling** — You must scale the axes of the `NumericTypeScope` manually, so by default, this parameter is set to `Manual`. You can scale the axes in any of the following ways:
  - Select **Tools > Scale axes limits**.
  - Press the **Scale Axes Limits** toolbar button ().
  - When the scope is the active window, press **Ctrl** and **A** simultaneously.

Note The `NumericTypeScope` does not support automatic axes scaling. You must always manually scale the axes, even if you set the **Axis Scaling** parameter to `Auto` or `Once at stop`. The scope respects the settings of the other parameters on this dialog box, but only applies them when you manually scale the axes limits.

- **Do not allow Y-axis limits to shrink** — When you select this parameter, the Y-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the Y-axis limits may shrink during axes scaling operations.

This parameter appears only when you select `Auto` for the **Axis Scaling** parameter. When you set the **Axis Scaling** parameter to `Manual` or `Once at stop`, the Y-axis limits are allowed to shrink.

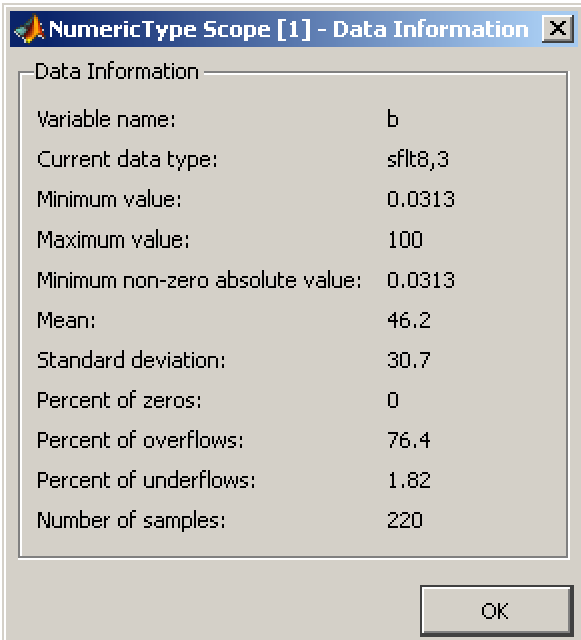
- **Y-axis Data range (%)** — Set the percentage of the Y-axis the scope uses to display the data when scaling the axes (valid values are between 1 and 100). For example, if you set this parameter to 100, the scope scales the Y-axis limits such that your data uses the entire Y-axis range. If you then set this parameter to 30, the scope increases the Y-axis range and scales the Y-axis limits such that your data only uses 30% of the Y-axis range. This parameter has a default value of `95` in the `NumericTypeScope`.
- **Y-axis Align** — Specify where the scope should align your data with respect to the Y-axis when it scales the axes. You can select `Top`, `Center` or `Bottom`. This parameter has a default value of `Bottom`.
- **Scale X-axis limits** — Check this box to allow the scope to scale the X-axis limits when it scales the axes.
- **X-axis Data range (%)** — Set the percentage of the X-axis the scope should use to display the data when scaling the axes (valid values are between 1 and 100). For example, if you set this parameter to 100, the scope scales the X-axis limits such that your data uses the entire X-axis range. If you then set this parameter to 30, the scope increases the X-axis range and scales the X-axis limits such that your data only uses 30% of the X-axis range. Use the X-axis **Align** parameter to specify where the scope should place your data with respect to the X-axis.

  This parameter appears only when you select the **Scale X-axis limits** check box. This parameter has a default value of `100` in the `NumericTypeScope`.
- **X-axis Align** — Specify how the scope should align your data with respect to the X-axis: `Left`, `Center` or `Right`. This parameter appears only when you select the **Scale X-axis limits** check box.

Data Information Dialog Box

The **Data Information** dialog box is a textual display of information about the variable the scope is currently displaying. You can access the **Data Information** dialog box in the following ways:

- Click the **Data Information** toolbar button ( ).
- Select **View > Data Information** from the scope window.
- With the `NumericTypeScope` as your active window, press the **D** key.



The name and current data type of the variable the scope is displaying appear on the first two lines of this dialog box. The dialog box also provides statistical information about the variable, including the minimum, maximum, mean, and standard deviation values.

The **Percent of zeros** shown on this dialog box reflects the percentage of your original data that had a value of zero. This value does not include any zeros resulting from underflow.

You can view overflow and underflow information about a variable when that variable is a `fi` object with a scaled double data type, or the `DataTypeOverride` property of the `fipref` object is set to `Scaled Doubles`. See “Identifying Overflows and Underflows” on page 3-269 for more information.

Examples

Set the `DataTypeOverride` to `Scaled Doubles`, and view the dynamic range of a `fi` object.

```
fp = fipref;
initialDTOSetting = fp.DataTypeOverride;
fp.DataTypeOverride = 'ScaledDoubles';
a = fi(magic(10),1,8,2);
b = fi([a; 2.^(-5:4)],1,8,3);
h = NumericTypeScope;
step(h,b);
fp.DataTypeOverride = initialDTOSetting;
```

From the `log2` histogram display, you can see that both overflows and underflows occur in the variable `b` with its current data type of `numerictype(1,8,3)`. The `numerictype(1,8,3)` data type provides 5 integer bits (including the signed bit), and 3 fractional bits. Thus, this data type can only represent values between -2^4 and $2^4 - 2^{-3}$ (from `16` to `15.8750`). Given the range and precision of this data type, values greater than 2^4 overflow and values less than 2^{-3} underflow.

Looking at the `NumericTypeScope` display, you can see that the overflows occurred for values requiring bits 5, 6 and 7, and underflows occurred for values requiring fractional bits 4 and 5. Given this information, you can eliminate overflows and underflows by changing the data type of the variable `b` to `numerictype(0,12,5)`.

View the dynamic range, and determine an appropriate numeric type for a fi object with a DataTypeMode of Scaled double: binary point scaling.

Create a numerictype object with a DataTypeMode of Scaled double: binary point scaling. You can then use that numerictype object to construct your fi objects. Because you set the DataTypeMode to Scaled double: binary point scaling, the NumericTypeScope can now identify overflows in your data.

```
T = numerictype;
T.DataTypeMode = 'Scaled double: binary point scaling';
T.WordLength = 8; T.FractionLength = 6;
a = fi(sin(0:100)*3.5, T);
b = fi(cos(0:100)*1.75,T);
acc = fi(0,T);
h = NumericTypeScope;
for i = 1:length(a)
    acc(:) = a(i)*0.7+b(i);
    step(h,acc);
end
```

You can see from the dynamic range analysis that the entire range of data in the accumulator can be represented with 5 bits; three to the left of the binary point (integer bits) and two to the right of it (fractional bits). You can verify that this data type is able to represent all the values by changing the WordLength and FractionLength properties of the numerictype object T. Then, use T to redefine the accumulator.

To view the dynamic range analysis based on this new data type, reset the NumericTypeScope object h, and rerun the loop:

```
T.WordLength = 5; T.FractionLength = 2;
acc = fi(0,T);
reset(h);
for i = 1:length(a)
    acc(:) = a(i)*0.7 + b(i);
    step(h,acc);
```

```
end
```

See Also

hist | log2

nunderflows

Purpose Number of underflows

Syntax

```
y = nunderflows(a)
y = nunderflows(q)
```

Description

`y = nunderflows(a)` returns the number of underflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = nunderflows(q)` returns the accumulated number of underflows resulting from quantization operations performed by a `quantizer` object `q`.

See Also `maxlog`, `minlog`, `noverflows`, `resetlog`

Purpose

Octal representation of stored integer of `fi` object

Syntax

```
oct(a)
```

Description

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

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

Examples

The following code

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

returns

```
y =

  200   177

z =

  200   177
```

See Also

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

Purpose Find logical OR of array or scalar inputs

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

Purpose Create patch graphics object

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

Purpose Create pseudocolor plot

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

Purpose Rearrange dimensions of multidimensional array

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

plot

Purpose Create linear 2-D plot

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

Purpose Create 3-D line plot

Description Refer to the MATLAB `plot3` reference page for more information.

Purpose Draw scatter plots

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

Purpose Create graph with y-axes on right and left sides

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

plus

Purpose Matrix sum of `fi` objects

Syntax `plus(a,b)`

Description `plus(a,b)` is called for the syntax `a + b` when `a` or `b` is an object.

`a + b` adds matrices `a` and `b`. `a` and `b` must have the same dimensions unless one is a scalar value (a 1-by-1 matrix). A scalar value can be added to any other value.

`plus` does not support `fi` objects of data type `Boolean`.

Note For information about the `fimath` properties involved in Fixed-Point Toolbox calculations, see “Using fimath Properties to Perform Fixed-Point Arithmetic” and “Using fimath ProductMode and SumMode” in the *Fixed-Point Toolbox User’s Guide*.

For information about calculations using Simulink Fixed Point software, see the “Arithmetic Operations” chapter of the *Simulink Fixed Point User’s Guide*.

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

Purpose Plot polar coordinates

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

Purpose

Efficient fixed-point multiplication by 2^K

Syntax

`b = pow2(a,K)`

Description

`b = pow2(a,K)` returns the value of `a` shifted by `K` bits where `K` is an integer and `a` and `b` are `fi` objects. The output `b` always has the same word length and fraction length as the input `a`.

Note In fixed-point arithmetic, shifting by `K` bits is equivalent to, and more efficient than, computing $b = a*2^k$.

If `K` is a non-integer, the `pow2` function will round it to `floor` before performing the calculation.

The scaling of `a` must be equivalent to binary point-only scaling; in other words, it must have a power of 2 slope and a bias of 0.

`a` can be real or complex. If `a` is complex, `pow2` operates on both the real and complex portions of `a`.

The `pow2` function obeys the `OverflowMode` and `RoundMode` properties associated with `a`. If obeying the `RoundMode` property associated with `a` is not important, try using the `bitshift` function.

The `pow2` function does not support `fi` objects of data type `Boolean`.

The function also does not support the syntax `b = pow2(a)` when `a` is a `fi` object.

Examples

Example 1

In the following example, `a` is a real-valued `fi` object, and `K` is a positive integer.

The `pow2` function shifts the bits of `a` 3 places to the left, effectively multiplying `a` by 2^3.

```
a = fi(pi,1,16,8)
```

```
b = pow2(a,3)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

```
a =

    3.1406

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

b =

   25.1250

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

binary_a =

0000001100100100


binary_b =

0001100100100000
```

Example 2

In the following example, `a` is a real-valued `fi` object, and `K` is a negative integer.

The pow2 function shifts the bits of a 4 places to the right, effectively multiplying a by 2^{-4}.

```
a = fi(pi,1,16,8)
b = pow2(a,-4)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

```
a =

    3.1406

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

b =

    0.1953

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

binary_a =

0000001100100100


binary_b =

0000000000110010
```

Example 3

The following example shows the use of `pow2` with a complex `fi` object:

```
format long g
P = fipref('NumericTypeDisplay', 'short');
a = fi(57 - 2i, 1, 16, 8)

a =
                          57 -                     2i
      s16,8

pow2(a, 2)

ans =
               127.99609375 -                     8i
      s16,8
```

See Also

`bitshift`, `bitsll`, `bitsra`, `bitsrl`

Purpose

Fixed-point array power (.^)

Syntax

```
c = power(a,k)
c = a.^k
```

Description

`c = power(a,k)` and `c = a.^k` compute element-by-element power. The exponent `k` requires a positive, real-valued integer value.

The fixed-point output array `c` is always associated with the global fimath.

Tips

For more information about the `power` function, see the MATLAB `arithmeticoperators` reference page.

Examples

Compute the power of a 2-dimensional array for exponent values 0, 1, 2, and 3.

```
x = fi([0 1 2; 3 4 5], 1, 32);

px0 = x.^0
px1 = x.^1
px2 = x.^2
px3 = x.^3
```

See Also

`arithmeticoperators` | `mpower`

Purpose

Apply `quantizer` object to data

Syntax

```
y = quantize(q, x)
[y1,y2,...] = quantize(q,x1,x2,...)
```

Description

`y = quantize(q, x)` uses the `quantizer` object `q` to quantize `x`. When `x` is a numeric array, each element of `x` is quantized. When `x` is a cell array, each numeric element of the cell array is quantized. When `x` is a structure, each numeric field of `x` is quantized. Quantize does not change nonnumeric elements or fields of `x`, nor does it issue warnings for nonnumeric values. The output `y` is a built-in double. When the input `x` is a structure or cell array, the fields of `y` are built-in doubles.

`[y1,y2,...]  = quantize(q,x1,x2,...)` is equivalent to

```
y1 = quantize(q,x1), y2 = quantize(q,x2),...
```

The `quantizer` object states

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

are updated during the call to `quantize`, and running totals are kept until a call to `resetlog` is made.

Examples

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.

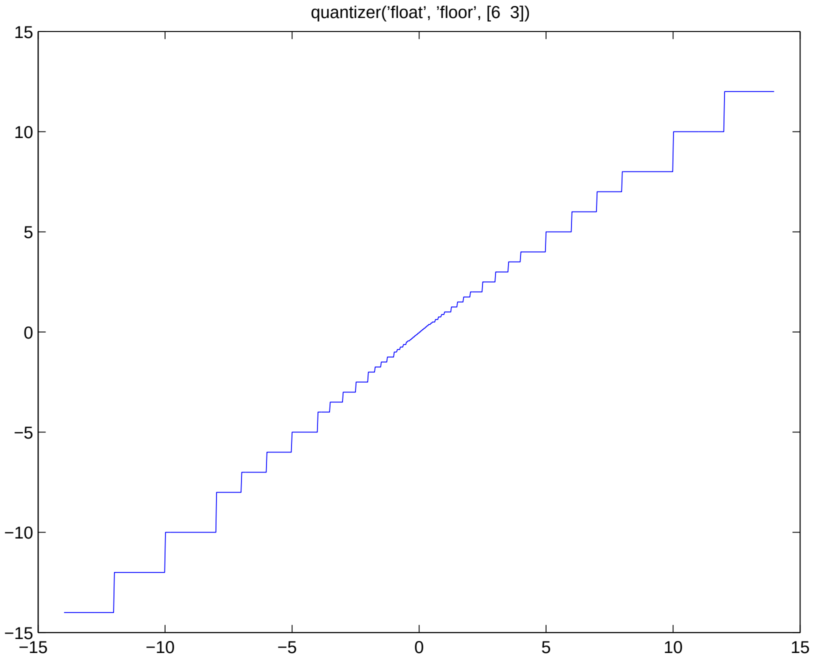
```
u=linspace(-15,15,1000);
q=quantizer([6 3],'float');
```

```
range(q)

ans =

   -14    14
y=quantize(q,u);
plot(u,y);title(tostring(q))

Warning: 68 overflows.
```



Example 2 - Fixed-Point

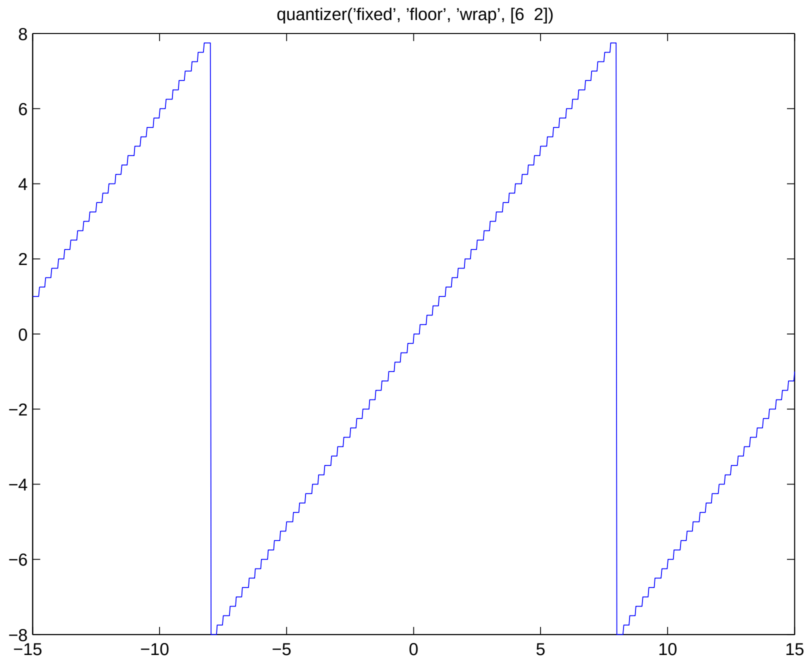
The code listed here produces the plot shown in the following figure.

```
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

assignmentquantizer, quantizer, set, unitquantize, unitquantizer

quantizer

Purpose

Construct `quantizer` object

Syntax

```
q = quantizer
q = quantizer('PropertyName1',PropertyValue1,...)
q = quantizer(PropertyValue1,PropertyValue2,...)
q = quantizer(struct)
q = quantizer(pn,pv)
```

Description

`q = quantizer` creates a `quantizer` object with properties set to their default values.

`q = quantizer('PropertyName1',PropertyValue1,...)` uses property name/ property value pairs.

`q = quantizer(PropertyValue1,PropertyValue2,...)` creates a `quantizer` object with the listed property values. When two values conflict, `quantizer` sets the last property value in the list. Property values are unique; you can set the property names by specifying just the property values in the command.

`q = quantizer(struct)`, where `struct` is a structure whose field names are property names, sets the properties named in each field name with the values contained in the structure.

`q = quantizer(pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

The `quantizer` object property values are listed below. These properties are described in detail in "quantizer Object Properties" on page 1-20.

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.
	'ufixed'	Unsigned fixed-point mode.
roundmode	'ceil'	Round toward positive infinity.
	'convergent'	Round to nearest integer with ties rounding to nearest even integer.
	'fix'	Round toward zero.
	'floor'	Round toward negative infinity.
	'nearest'	Round to nearest integer with ties rounding toward positive infinity.
	'round'	Round to nearest integer with ties rounding to nearest integer with greater absolute value.

Property Name	Property Value	Description
overflowmode (fixed-point only)	'saturate'	Saturate on overflow.
	'wrap'	Wrap on overflow.
format	[wordlength fractionlength]	Format for fixed or ufixed mode.
	[wordlength exponentlength]	Format for float mode.

The default property values for a quantizer object are

```
mode = 'fixed';
roundmode = 'floor';
overflowmode = 'saturate';
format = [16 15];
```

Along with the preceding properties, quantizer objects have read-only states: max, min, noverflows, nunderflows, and noperations. They can be accessed through quantizer/get or q.maxlog, q.minlog, q.noverflows, q.nunderflows, and q.noperations, but they cannot be set. They are updated during the quantizer/quantize method, and are reset by the resetlog function.

The following table lists the read-only quantizer object states:

Property Name	Description
max	Maximum value before quantizing
min	Minimum value before quantizing
noverflows	Number of overflows
nunderflows	Number of underflows
noperations	Number of data points quantized

Examples

The following example operations are equivalent.

Setting `quantizer` object properties by listing property values only in the command,

```
q = quantizer('fixed', '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

`assignmentquantizer`, `fi`, `fimath`, `fipref`, `numerictype`, `quantize`, `set`, `unitquantize`, `unitquantizer`

Purpose Create quiver or velocity plot

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

Purpose Create 3-D quiver or velocity plot

Description Refer to the MATLAB `quiver3` reference page for more information.

Purpose

Generate uniformly distributed, quantized random number using quantizer object

Syntax

```
randquant(q,n)
randquant(q,m,n)
randquant(q,m,n,p,...)
randquant(q,[m,n])
randquant(q,[m,n,p,...])
```

Description

randquant(q,n) uses quantizer object q to generate an n-by-n matrix with random entries whose values cover the range of q when q is a fixed-point quantizer object. When q is a floating-point quantizer object, randquant populates the n-by-n array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

randquant(q,m,n) uses quantizer object q to generate an m-by-n matrix with random entries whose values cover the range of q when q is a fixed-point quantizer object. When q is a floating-point quantizer object, randquant populates the m-by-n array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

randquant(q,m,n,p,...) uses quantizer object q to generate an m-by-n-by-p-by ... matrix with random entries whose values cover the range of q when q is fixed-point quantizer object. When q is a floating-point quantizer object, randquant populates the matrix with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

randquant(q,[m,n]) uses quantizer object q to generate an m-by-n matrix with random entries whose values cover the range of q when q is a fixed-point quantizer object. When q is a floating-point quantizer object, randquant populates the m-by-n array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

randquant(q,[m,n,p,...]) uses quantizer object q to generate p m-by-n matrices containing random entries whose values cover the range of q when q is a fixed-point quantizer object. When q is a floating-point quantizer object, randquant populates the m-by-n arrays with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

randquant produces pseudorandom numbers. The number sequence randquant generates during each call is determined by the state of the generator. Because MATLAB resets the random number generator state at startup, the sequence of random numbers generated by the function remains the same unless you change the state.

randquant works like rand in most respects, including the generator used, but it does not support the 'state' and 'seed' options available in rand.

Examples

```
q=quantizer([4 3]);
rand('state',0)
randquant(q,3)

ans =

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

See Also

quantizer, rand, range, realmax

Purpose

Numerical range of `fi` or `quantizer` object

Syntax

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

Description

`range(a)` returns a fi object with the minimum and maximum possible values of `fi` object `a`. All possible quantized real-world values of `a` are in the range returned. If `a` is a complex number, then all possible values of `real(a)` and `imag(a)` are in the range returned.

`[min, max]= range(a)` returns the minimum and maximum values of `fi` object `a` in separate output variables.

`r = range(q)` returns the two-element row vector $r = [a\ b]$ such that for all real x, `y = quantize(q,x)` returns y in the range $a \le y \le b$.

`[min, max] = range(q)` returns the minimum and maximum values of the range in separate output variables.

Examples

```
q = quantizer('float',[6 3]);
r = range(q)

r =

   -14    14
q = quantizer('fixed',[4 2],'floor');
[min,max] = range(q)

min =

    -2


max =

    1.7500
```

Algorithm

If q is a floating-point `quantizer` object, a = `-realmax`(q), b = `realmax`(q).

If q is a signed fixed-point `quantizer` object (`datamode = 'fixed'`),

$$a = -\text{realmax}(q) - \text{eps}(q) = \frac{-2^{w-1}}{2^f}$$

$$b = \text{realmax}(q) = \frac{2^{w-1} - 1}{2^f}$$

If q is an unsigned fixed-point `quantizer` object (`datamode = 'ufixed'`),

$$a = 0$$

$$b = \text{realmax}(q) = \frac{2^w - 1}{2^f}$$

See `realmax` for more information.

See Also

`eps`, `exponentmax`, `exponentmin`, `fractionlength`, `intmax`, `intmin`, `lowerbound`, `lsb`, `max`, `min`, `realmax`, `realmin`, `upperbound`

Purpose

Right-array division (./)

Syntax

```
c = rdivide(a,b)
c = a./b
```

Description

`c = rdivide(a,b)` and `c = a./b` perform right-array division by dividing each element of `a` by the corresponding element of `b`. If inputs `a` and `b` are not the same size, one of them must be a scalar value.

The numerator input `a` can be complex, but the denominator `b` requires a real-valued input. If `a` is complex, the real and imaginary parts of `a` are independently divided by `b`.

The following table shows the rules used to assign property values to the output of the `rdivide` function.

Output Property	Rule
`Signedness`	If either input is `Signed`, the output is `Signed`. If both inputs are `Unsigned`, the output is `Unsigned`.
`WordLength`	The output word length equals the maximum of the input word lengths.
`FractionLength`	For `c = a./b`, the fraction length of output `c` equals the fraction length of `a` minus the fraction length of `b`.

The following table shows the rules the `rdivide` function uses to handle inputs with different data types.

Case	Rule
Interoperation of `fi` objects and built-in integers	Built-in integers are treated as fixed-point objects. For example, `B = int8(2)` is treated as an `s8,0` `fi` object.
Interoperation of `fi` objects and constants	The Embedded MATLAB® subset treats constant integers as fixed-point objects with the same word length as the `fi` object and a fraction length of `0`.
Interoperation of mixed data types	Similar to all other `fi` object functions, when inputs `a` and `b` have different data types, the data type with the higher precedence determines the output data type. The order of precedence is as follows: **1** `ScaledDouble` **2** `Fixed-point` **3** Built-in `double` **4** Built-in `single` When both inputs are `fi` objects, the only data types that are allowed to mix are `ScaledDouble` and `Fixed-point`.

Examples

In this example, you perform right-array division on a 3-by-3 magic square of `fi` objects. Each element of the 3-by-3 magic square is divided by the corresponding element in the 3-by-3 input array `b`.

```
a = fi(magic(3))
b = int8([3  3 4; 1 2 4 ; 3 1 2 ])
c = a./b
```

The `mrdivide` function outputs a 3-by-3 array of signed `fi` objects, each of which has a word length of 16 bits and fraction length of 11 bits.

```
a =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11

b =

    3    3    4
    1    2    4
    3    1    2


c =

    2.6665    0.3335    1.5000
    3.0000    2.5000    1.7500
    1.3335    9.0000    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

See Also

`add`, `divide`, `fi`, `fimath`, `mrdivide`, `numerictype`, `sub`, `sum`

Purpose Real part of complex number

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

Purpose

Largest positive fixed-point value or quantized number

Syntax

```
realmax(a)
realmax(q)
```

Description

`realmax(a)` is the largest real-world value that can be represented in the data type of `fi` object `a`. Anything larger overflows.

`realmax(q)` is the largest quantized number that can be represented where `q` is a `quantizer` object. Anything larger overflows.

Examples

```
q = quantizer('float',[6 3]);
x = realmax(q)

x =

    14
```

Algorithm

If `q` is a floating-point `quantizer` object, the largest positive number, x, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point `quantizer` object, the largest positive number, x, is

$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point `quantizer` object (`datamode = 'ufixed'`), the largest positive number, x, is

$$x = \frac{2^w - 1}{2^f}$$

See Also

`eps`, `exponentmax`, `exponentmin`, `fractionlength`, `intmax`, `intmin`, `lowerbound`, `lsb`, `quantizer`, `range`, `realmin`, `upperbound`

Purpose

Smallest positive normalized fixed-point value or quantized number

Syntax

```
realmin(a)
realmin(q)
```

Description

`realmin(a)` is the smallest real-world value that can be represented in the data type of `fi` object `a`. Anything smaller underflows.

`realmin(q)` is the smallest positive normal quantized number where `q` is a `quantizer` object. Anything smaller than `x` underflows or is an IEEE “denormal” number.

Examples

```
q = quantizer('float',[6 3]);
x = realmin(q)

x =

    0.2500
```

Algorithm

If `q` is a floating-point `quantizer` object, $x = 2^{E_{min}}$ where $E_{min} = \mathrm{exponentmin}(q)$ 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

`eps`, `exponentmax`, `exponentmin`, `fractionlength`, `intmax`, `intmin`, `lowerbound`, `lsb`, `range`, `realmax`, `upperbound`

reinterpretcast

Purpose

Convert fixed-point data types without changing underlying data

Syntax

```
c = reinterpretcast(a, T)
```

Description

`c = reinterpretcast(a, T)` converts the input `a` to the data type specified by `numerictype` object `T` without changing the underlying data. The result is returned in `fi` object `c`.

The input `a` must be a built-in integer or a `fi` object with a fixed-point data type. `T` must be a `numerictype` object with a fully specified fixed-point data type. The word length of inputs `a` and `T` must be the same.

The `reinterpretcast` function differs from the MATLAB `typecast` and `cast` functions in that it only operates on `fi` objects and built-in integers, and it does not allow the word length of the input to change.

Examples

In the following example, `a` is a signed `fi` object with a word length of 8 bits and a fraction length of 7 bits. The `reinterpretcast` function converts `a` into an unsigned `fi` object `c` with a word length of 8 bits and a fraction length of 0 bits. The real-world values of `a` and `c` are different, but their binary representations are the same.

```
 a = fi([-1 pi/4], true, 8, 7)
 T = numerictype(false, 8, 0);
 c = reinterpretcast(a, T)
a =

   -1.0000    0.7891

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7

c =

   128   101
```

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

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

```
binary_a = bin(a)
binary_c = bin(c)
binary_a =

10000000   01100101


binary_c =

10000000   01100101
```

See Also

cast, fi, numerictype, typecast

removedefaultfimathpref

Purpose

Remove global fimath preference

Note `removedefaultfimathpref` will be removed in a future version. Use `removeglobalfimathpref` instead.

Syntax

`removedefaultfimathpref`

Description

`removedefaultfimathpref` removes your global fimath from the MATLAB preferences. Doing so forces MATLAB to use the MATLAB factory default setting of the global fimath in future MATLAB sessions.

The `removedefaultfimathpref` function does not change the global fimath for your current MATLAB session. To revert back to the factory default setting of the global fimath in your current MATLAB session, use the `resetdefaultfimath` command.

For more information on the global fimath, see "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide*.

Examples

Removing Your Global fimath from the MATLAB Preferences

Typing

```
removedefaultfimathpref;
```

at the MATLAB command line removes your global fimath from the MATLAB preferences. Using the `removedefaultfimathpref` function allows you to:

- Continue using your global fimath in the current MATLAB session
- Use the MATLAB factory default setting of the global fimath in all future MATLAB sessions

To revert back to the MATLAB factory default setting of the global fimath in both your current and future MATLAB sessions, use both the `resetdefaultfimath` and the `removedefaultfimathpref` commands:

```
resetdefaultfimath;
removedefaultfimath;
```

See Also

`fimath`, `globalfimath`, `resetglobalfimath`, `saveglobalfimathpref`

removeglobalfimathpref

Purpose

Remove global fimath preference

Syntax

```
removeglobalfimathpref
```

Description

`removeglobalfimathpref` removes your global fimath from the MATLAB preferences. Doing so forces MATLAB to use the MATLAB factory default setting of the global fimath in future MATLAB sessions.

The `removeglobalfimathpref` function does not change the global fimath for your current MATLAB session. To revert back to the factory default setting of the global fimath in your current MATLAB session, use the `resetglobalfimath` command.

For more information on the global fimath, see "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide*.

Examples

Removing Your Global fimath from the MATLAB Preferences

Typing

```
removeglobalfimathpref;
```

at the MATLAB command line removes your global fimath from the MATLAB preferences. Using the `removeglobalfimathpref` function allows you to:

- Continue using your global fimath in the current MATLAB session
- Use the MATLAB factory default setting of the global fimath in all future MATLAB sessions

To revert back to the MATLAB factory default setting of the global fimath in both your current and future MATLAB sessions, use both the `resetglobalfimath` and the `removeglobalfimathpref` commands:

```
resetglobalfimath;
removeglobalfimath;
```

See Also

`fimath` | `globalfimath` | `resetglobalfimath` | `saveglobalfimathpref`

How To

- "Working with the Global fimath"

Purpose Replicate and tile array

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

Purpose

Change scaling of `fi` object

Syntax

```
b = rescale(a, fractionlength)
b = rescale(a, slope, bias)
b = rescale(a, slopeadjustmentfactor, fixedexponent, bias)
b = rescale(a, ..., PropertyName, PropertyValue, ...)
```

Description

The `rescale` function acts similarly to the `fi` copy function with the following exceptions:

- The `fi` copy constructor preserves the real-world value, while `rescale` preserves the stored integer value.
- `rescale` does not allow the `Signed` and `WordLength` properties to be changed.

Examples

In the following example, `fi` object `a` is rescaled to create `fi` object `b`. The real-world values of `a` and `b` are different, while their stored integer values are the same:

```
p = fipref('FimathDisplay','none',...
  'NumericTypeDisplay','short');
a = fi(10, 1, 8, 3)

a =

    10
      s8,3

b = rescale(a, 1)

b =

    40
      s8,1
```

```
stored_integer_a = a.int;
stored_integer_b = b.int;
isequal(stored_integer_a, stored_integer_b)

ans =

1
```

See Also fi

Purpose Reset objects to initial conditions

Syntax

```
reset(P)
reset(q)
```

Description `reset(P)` resets the `fipref` object `P` to its initial conditions.

`reset(q)` resets the following `quantizer` object properties to their initial conditions:

- `minlog`
- `maxlog`
- `noverflows`
- `nunderflows`
- `noperations`

See Also `resetlog`

resetdefaultfimath

Purpose

Set global fimath to MATLAB factory default

Note `resetdefaultfimath` will be removed in a future version. Use `resetglobalfimath` instead.

Syntax

```
resetdefaultfimath
```

Description

`resetdefaultfimath` sets the global fimath to the MATLAB factory default in your current MATLAB session. The MATLAB factory default has the following properties:

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

For more information on the global fimath, see “Working with the Global fimath” in the *Fixed-Point Toolbox User’s Guide*.

Examples

In this example, you create your own `fimath` object `F` and set it as the global fimath. Then, use the `resetdefaultfimath` command to reset the global fimath to the MATLAB factory default setting.

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap');
setdefaultfimath(F);
F1 = fimath
a = fi(pi)

F1 =


             RoundMode: floor
          OverflowMode: wrap
```

```
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                 SumMode: FullPrecision
        MaxSumWordLength: 128


a =

    3.1416

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

Now, set the global fimath back to the factory default setting:

```
resetdefaultfimath;
F2 = fimath
a = fi(pi)

F2 =


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


a =

    3.1416
```

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

You've now set the global fimath in your current MATLAB session back to the factory default setting. To use the factory default setting of the global fimath in future MATLAB sessions, you must use the `removedefaultfimathpref` command.

See Also

`fimath`, `globalfimath`, `removeglobalfimathpref`, `saveglobalfimathpref`

Purpose

Set global fimath to MATLAB factory default

Syntax

```
resetglobalfimath
```

Description

`resetglobalfimath` sets the global fimath to the MATLAB factory default in your current MATLAB session. The MATLAB factory default has the following properties:

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

For more information on the global fimath, see “Working with the Global fimath” in the *Fixed-Point Toolbox User’s Guide*.

Examples

In this example, you create your own `fimath` object `F` and set it as the global fimath. Then, using the `resetglobalfimath` command, reset the global fimath to the MATLAB factory default setting.

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap');
globalfimath(F);
F1 = fimath
a = fi(pi)

F1 =


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

```
a =

    3.1416

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

Now, set the global fimath back to the factory default setting using `resetglobalfimath`:

```
resetglobalfimath;
F2 = fimath
a = fi(pi)

F2 =


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

a =

    3.1416

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

You've now set the global fimath in your current MATLAB session back to the factory default setting. To use the factory default setting of the global fimath in future MATLAB sessions, you must use the `removeglobalfimathpref` command.

Alternatives

`reset(G)` — If `G` is a handle to the global fimath, `reset(G)` is equivalent to using the `resetglobalfimath` command.

See Also

`fimath` | `globalfimath` | `removeglobalfimathpref` | `saveglobalfimathpref`

How To

- "Working with the Global fimath"

resetlog

Purpose Clear log for fi or quantizer object

Syntax

```
resetlog(a)
resetlog(q)
```

Description resetlog(a) clears the log for fi object a.

resetlog(q) clears the log for quantizer object q.

Turn logging on or off by setting the fipref property LoggingMode.

See Also fipref, maxlog, minlog, noperations, noverflows, nunderflows, reset

Purpose Reshape array

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

Purpose Plot colormap

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

Purpose Create ribbon plot

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

rose

Purpose Create angle histogram

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

Purpose

Round fi object toward nearest integer or round input data using quantizer object

Syntax

```
y = round(a)
y = round(q,x)
```

Description

y = round(a) rounds fi object a to the nearest integer. In the case of a tie, round rounds values to the nearest integer with greater absolute value. The rounded value is returned in fi object y.

y and a have the same fimath object and DataType property.

When the DataType of a is single, double, or boolean, the numerictype of y is the same as that of a.

When the fraction length of a is zero or negative, a is already an integer, and the numerictype of y is the same as that of a.

When the fraction length of a is positive, the fraction length of y is 0, its sign is the same as that of a, and its word length is the difference between the word length and the fraction length of a, plus one bit. If a is signed, then the minimum word length of y is 2. If a is unsigned, then the minimum word length of y is 1.

For complex fi objects, the imaginary and real parts are rounded independently.

round does not support fi objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

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

Examples

Example 1

The following example demonstrates how the round function affects the numerictype properties of a signed fi object with a word length of 8 and a fraction length of 3.

```
a = fi(pi, 1, 8, 3)

a =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3

y = round(a)

y =

     3

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

Example 2

The following example demonstrates how the `round` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)

a =

    0.0249

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

```
             WordLength: 8
         FractionLength: 12

y = round(a)

y =

     0

           DataTypeMode: Fixed-point: binary point scaling
             Signedness: Signed
             WordLength: 2
         FractionLength: 0
```

Example 3

The functions `convergent`, `nearest` and `round` differ in the way they treat values whose least significant digit is 5:

- The `convergent` function rounds ties to the nearest even integer
- The `nearest` function rounds ties to the nearest integer toward positive infinity
- The `round` function rounds ties to the nearest integer with greater absolute value

The following table illustrates these differences for a given `fi` object `a`.

a	convergent(a)	nearest(a)	round(a)
–3.5	–4	–3	–4
–2.5	–2	–2	–3
–1.5	–2	–1	–2
–0.5	0	0	–1
0.5	0	1	1
1.5	2	2	2

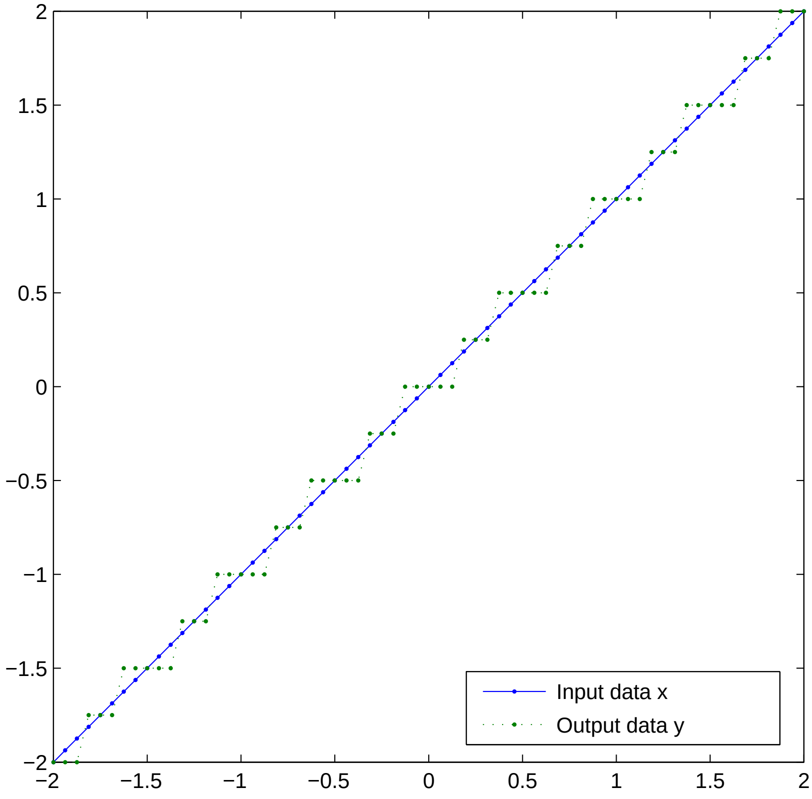
a	convergent(a)	nearest(a)	round(a)
2.5	2	3	3
3.5	4	4	4

Example 4

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 in the following figure. Linear data input results in output where `y` shows distinct quantization levels.



See Also

`ceil`, `convergent`, `fix`, `floor`, `nearest`, `quantize`, `quantizer`

savedefaultfimathpref

Purpose

Save global fimath for next MATLAB session

Note `savedefaultfimathpref` will be removed in a future version. Use `saveglobalfimathpref` instead.

Syntax

```
savedefaultfimathpref
```

Description

`savedefaultfimathpref` saves the current global fimath as the global fimath to be used in all future MATLAB sessions.

For more information on the global fimath, see “Working with the Global fimath” in the *Fixed-Point Toolbox User’s Guide*.

See Also

`fimath`, `globalfimath`, `removeglobalfimathpref`, `resetglobalfimath`

Purpose

Save global fimath for next MATLAB session

Syntax

```
saveglobalfimathpref
```

Description

`saveglobalfimathpref` saves the current global fimath as the global fimath to be used in all future MATLAB sessions.

For more information on the global fimath, see “Working with the Global fimath” in the *Fixed-Point Toolbox User’s Guide*.

See Also

`fimath` | `globalfimath` | `removeglobalfimathpref` | `resetglobalfimath`

How To

- “Working with the Global fimath”

savefipref

Purpose

Save `fi` preferences for next MATLAB session

Syntax

```
savefipref
```

Description

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

See Also

`fipref`

Purpose Create scatter or bubble plot

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

Purpose Create 3-D scatter or bubble plot

Description Refer to the MATLAB `scatter3` reference page for more information.

Purpose

Signed decimal representation of stored integer of `fi` object

Syntax

`sdec(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

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

Examples

The code

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

returns

```
-128   127
```

See Also

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

semilogx

Purpose Create semilogarithmic plot with logarithmic x-axis

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

Purpose Create semilogarithmic plot with logarithmic y-axis

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

Purpose

Set or display property values for `quantizer` objects

Syntax

```
set(q, PropertyValue1, PropertyValue2,...)
set(q,s)
set(q,pn,pv)
set(q,'PropertyName1',PropertyValue1,'PropertyName2',
PropertyValue2,...)
q.PropertyName = Value
s = set(q)
```

Description

`set(q, PropertyValue1, PropertyValue2,...)` sets the properties of `quantizer` object `q`. If two property values conflict, the last value in the list is the one that is set.

`set(q,s)`, where `s` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(q,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

`set(q,'PropertyName1',PropertyValue1,'PropertyName2', PropertyValue2,...)` sets multiple property values with a single statement.

Note You can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`q.PropertyName = Value` uses dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of `quantizer` object `q`.

`s = set(q)` returns a structure containing the possible values for the properties of `quantizer` object `q`.

Note The `set` function operates on `quantizer` objects. To learn about setting the properties of other objects, see properties of `fi`, `fimath`, `fipref`, and `numerictype` objects.

See Also

`get`

setdefaultfimath

Purpose

Set MATLAB global fimath

Note `setdefaultfimath` will be removed in a future version. Use `globalfimath` instead.

Syntax

```
setdefaultfimath(F)
setdefaultfimath('PropertyName1',PropertyValue1,...)
```

Description

`setdefaultfimath(F)` sets a copy of the `fimath` object `F` as the global fimath for your current MATLAB session.

`setdefaultfimath('PropertyName1',PropertyValue1,...)` changes the specified properties of the current global fimath to the values you specify. All properties that are not specified as inputs to the function retain the same values as the current global fimath.

For more information on working with the global fimath, see "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide*.

Examples

Setting the Global fimath Using a Workspace Variable

If you create a `fi` object in the MATLAB workspace and do not specify any `fimath` properties in the constructor, Fixed-Point Toolbox software associates it with the global fimath. To change the global fimath, you must use the `setdefaultfimath` command.

In this example, you create your own `fimath` object `F` and set it as the global fimath for your current MATLAB session:

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap')

F =


            RoundMode: floor
         OverflowMode: wrap
          ProductMode: FullPrecision
```

```
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128

setdefaultfimath(F);
```

Because all `fi` and `fimath` objects you create without specifying `fimath` properties in the constructor get associated with the global fimath, the `fimath` properties of both `F1` and `a` match that of `F`.

```
 F1 = fimath
 a = fi(pi)

F1 =


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

a =

    3.1416

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

Because `a` is associated with the global fimath, MATLAB does not display its `fimath` properties. To verify that `a` is associated with the global fimath, use the `isfimathlocal` command. To see the `fimath` properties associated with `a`, use dot notation:

```
isfimathlocal(a)
a.fimath

ans =
     0
ans =

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

To use the current global fimath in future MATLAB sessions, you must use the `savedefaultfimathpref` command.

Setting the Global fimath Using Property Name/Property Value Pairs

You can use the property name/property value pairs syntax to set select properties of the global fimath. For example, to change the `SumMode` of the global fimath to `KeepMSB`, do the following:

```
setdefaultfimath('SumMode', 'KeepMSB');
```

See Also

`fimath`, `removeglobalfimathpref`, `resetglobalfimath`, `saveglobalfimathpref`

Purpose

Construct signed fixed-point numeric object

Syntax

```
a = sfi
a = sfi(v)
a = sfi(v,w)
a = sfi(v,w,f)
a = sfi(v,w,slope,bias)
a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

Description

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

- `a = sfi` is the default constructor and returns a signed `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = sfi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = sfi(v,w)` returns a signed fixed-point object with value `v`, word length `w`, and best-precision fraction length.
- `a = sfi(v,w,f)` returns a signed fixed-point object with value `v`, word length `w`, and fraction length `f`.
- `a = sfi(v,w,slope,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slope`, and `bias`.
- `a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`fi` objects created by the `sfi` constructor function have the following general types of properties:

- "Data Properties" on page 3-133
- "fimath Properties" on page 3-362
- "numerictype Properties" on page 3-135

These properties are described in detail in "fi Object Properties" on page 1-2 in the Properties Reference.

Note fi objects created by the sfi constructor function are always associated with the global fimath. See "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide* for more information.

Data Properties

The data properties of a fi object are always writable.

- bin — Stored integer value of a fi object in binary
- data — Numerical real-world value of a fi object
- dec — Stored integer value of a fi object in decimal
- double — Real-world value of a fi object, stored as a MATLAB double
- hex — Stored integer value of a fi object in hexadecimal
- int — Stored integer value of a fi object, stored in a built-in MATLAB integer data type. You can also use int8, int16, int32, int64, uint8, uint16, uint32, and uint64 to get the stored integer value of a fi object in these formats
- oct — Stored integer value of a fi object in octal

These properties are described in detail in "fi Object Properties" on page 1-2.

fimath Properties

When you create a fi object with the sfi constructor function, that fi object does not have a local fimath object. Instead, the fi object is associated with the global fimath. When a fi object is associated with the global fimath, you can change its fimath properties by reconfiguring the global fimath, or by assigning the fi object a local fimath object.

For more information, see "Working with the Global fimath" in the Fixed-Point Toolbox User's Guide.

- `fimath` — fixed-point math object

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

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

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

- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow mode
- `ProductBias` — Bias of the product data type
- `ProductFixedExponent` — Fixed exponent of the product data type
- `ProductFractionLength` — Fraction length, in bits, of the product data type
- `ProductMode` — Defines how the product data type is determined
- `ProductSlope` — Slope of the product data type
- `ProductSlopeAdjustmentFactor` — Slope adjustment factor of the product data type
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode

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

These properties are described in detail in “fimath Object Properties” on page 1-4.

numerictype Properties

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

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

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

- Bias — Bias of a fi object
- DataType — Data type category associated with a fi object
- DataTypeMode — Data type and scaling mode of a fi object
- FixedExponent — Fixed-point exponent associated with a fi object
- SlopeAdjustmentFactor — Slope adjustment associated with a fi object

- `FractionLength` — Fraction length of the stored integer value of a `fi` object in bits
- `Scaling` — Fixed-point scaling mode of a `fi` object
- `Signed` — Whether a `fi` object is signed or unsigned
- `Signedness` — Whether a `fi` object is signed or unsigned

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

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

For further details on these properties, see "numerictype Object Properties" on page 1-15.

Examples

> **Note** For information about the display format of `fi` objects, refer to Display Settings.
>
> For examples of casting, see "Casting fi Objects".

Example 1

For example, the following creates a signed `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = sfi(pi,8,3)

a =
```

```
     3.1250

              DataTypeMode: Fixed-point: binary point scaling
                Signedness: Signed
                WordLength: 8
            FractionLength: 3
```

The `fimath` properties associated with `a` come from the global fimath. When a `fi` object does not have a local `fimath` object, it associates itself with the global fimath, and no `fimath` object properties are displayed in its output. To determine whether a `fi` object is associated with the global fimath, or has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)

ans =
     0
```

A returned value of `0` means the `fi` object is associated with the global fimath and does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

Example 2

The value `v` can also be an array:

```
a = sfi((magic(3)/10),16,12)

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

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

```
        FractionLength: 12
```

Example 3

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

```
a = sfi(pi,8)

a =

    3.1563

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

Example 4

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

```
a = sfi(pi)

a =

    3.1416

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

See Also

`fi`, `fimath`, `fipref`, `isfimathlocal`, `numerictype`, `quantizer`, `ufi`

shiftdata

Purpose

Shift data to operate on specified dimension

Syntax

```
[x,perm,nshifts] = shiftdata(x,dim)
```

Description

`[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first non-singleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples

Example 1

This example shifts `x`, a `3-by-3` magic square, permuting dimension `2` to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a `3-by-3` magic square:

```
x = fi(magic(3))

x =

     8     1     6
     3     5     7
     4     9     2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =

     8     3     4
     1     5     9
     6     7     2


perm =

     2     1


nshifts =

     []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2
```

Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5
```

```
x =

     1     2     3     4     5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

`x` is returned as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =

     1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =

     1     2     3     4     5
```

See Also

`permute`, `shiftdim`, `unshiftdata`

Purpose Shift dimensions

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

Purpose

Perform signum function on array

Syntax

```
c = sign(a)
```

Description

`c = sign(a)` returns an array `c` the same size as `a`, where each element of `c` is

- `1` if the corresponding element of `a` is greater than zero
- `0` if the corresponding element of `a` is zero
- `-1` if the corresponding element of `a` is less than zero

The elements of `c` are of data type `int8`.

`sign` does not support complex `fi` inputs.

Purpose Single-precision floating-point real-world value of `fi` object

Syntax `single(a)`

Description Fixed-point numbers can be represented as

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

or, equivalently as

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

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

See Also `double`

Purpose Array dimensions

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

slice

Purpose Create volumetric slice plot

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

Purpose Sort elements of real-valued fi object in ascending or descending order

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

Purpose

Visualize sparsity pattern

Description

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

Purpose

Square root of `fi` object

Syntax

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

Description

This function computes the square root of a `fi` object using a bisection algorithm.

`c = sqrt(a)` returns the square root of `fi` object `a`. Intermediate quantities are calculated using the `fimath` associated with `a`. The `numerictype` object of `c` is determined automatically for you using an internal rule.

`c = sqrt(a,T)` returns the square root of `fi` object `a` with `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` associated with `a`. See "Data Type Propagation Rules" on page 3-380.

`c = sqrt(a,F)` returns the square root of `fi` object `a`. Intermediate quantities are calculated using the `fimath` object `F`. The `numerictype` object of `c` is determined automatically for you using an internal rule. When `a` is a built-in `double` or `single` data type, this syntax is equivalent to `c = sqrt(a)` and the `fimath` object `F` is ignored.

`c = sqrt(a,T,F)` returns the square root `fi` object `a` with `numerictype` object `T`. Intermediate quantities are also calculated using the `fimath` object `F`. See "Data Type Propagation Rules" on page 3-380.

`sqrt` does not support complex, negative-valued, or [Slope Bias] inputs.

Internal Rule

For syntaxes where the `numerictype` object of the output is not specified as an input to the `sqrt` function, it is automatically calculated according to the following internal rule:

$$sign_c = sign_a$$

$$WL_c = \text{ceil}(\frac{WL_a}{2})$$

$$FL_c = WL_c - \text{ceil}(\frac{WL_a - FL_a}{2})$$

Data Type Propagation Rules

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

Data Type of Input fi Object a	Data Type of numerictype object T	Data Type of Output c
Built-in `double`	Any	Built-in `double`
Built-in `single`	Any	Built-in `single`
`fi Fixed`	`fi Fixed`	Data type of `numerictype` object `T`
`fi ScaledDouble`	`fi Fixed`	`ScaledDouble` with properties of `numerictype` object `T`
`fi double`	`fi Fixed`	`fi double`
`fi single`	`fi Fixed`	`fi single`
Any `fi` data type	`fi double`	`fi double`
Any `fi` data type	`fi single`	`fi single`

Purpose Remove singleton dimensions

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

stairs

Purpose Create stairstep graph

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

Purpose Plot discrete sequence data

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

stem3

Purpose Plot 3-D discrete sequence data

Description Refer to the MATLAB `stem3` reference page for more information.

Purpose Create 3-D stream ribbon plot

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

streamslice

Purpose Draw streamlines in slice planes

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

Purpose Create 3-D stream tube plot

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

stripscaling

Purpose Stored integer of fi object

Syntax

```
I = stripscaling(a)
```

Description

I = stripscaling(a) returns the stored integer of a as a fi object with binary-point scaling, zero fraction length and the same word length and sign as a.

Examples

Stripscaling is useful for converting the value of a fi object to its stored integer value.

```
fipref('NumericTypeDisplay','short', ...
       'FimathDisplay','none');
format long g
a = fi(0.1,true,48,47)

a =

          0.100000000000001
      s48,47
b = stripscaling(a)

b =

               14073748835533
      s48,0
bin(a)

ans =

000011001100110011001100110011001100110011001101

bin(b)

ans =

000011001100110011001100110011001100110011001101
```

Notice that the stored integer values of `a` and `b` are identical, while their real-world values are different.

sub

Purpose

Subtract two objects using `fimath` object

Syntax

```
c = 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` properties associated with `a` and `b` are different. The output `fi` object `c` is always associated with the global fimath.

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

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

Examples

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

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

c =

    0.4233


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

Algorithm

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

```
a.fimath = F;
```

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

c =
    0.4233

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

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

but not identical. When you use `sub`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` is associated with the global fimath. When you use the syntax `c = a - b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See "fimath Rules for Fixed-Point Arithmetic" in the *Fixed-Point Toolbox User's Guide* for more information.

See Also

`add`, `divide`, `fi`, `fimath`, `mpy`, `mrdivide`, `numerictype`, `rdivide`

subsasgn

Purpose Subscripted assignment

Syntax

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

Description

`a(I) = b` assigns the values of `b` into the elements of `a` specified by the subscript vector `I`. `b` must have the same number of elements as `I` or be a scalar value.

`a(I,J) = b` assigns the values of `b` into the elements of the rectangular submatrix of `a` specified by the subscript vectors `I` and `J`. `b` must have `LENGTH(I)` rows and `LENGTH(J)` columns.

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

For multidimensional arrays, `a(I,J,K,...) = b` assigns `b` to the specified elements of `a`. `b` must be `length(I)`-by-`length(J)`-by-`length(K)`-... or be shiftable to that size by adding or removing singleton dimensions.

`a = subsasgn(a,S,b)` is called for the syntax `a(i)=b`, `a{i}=b`, or `a.i=b` when `a` is an object. `S` is a structure array with the following fields:

- type — String containing `'()'`, `'{}'`, or `'.'` specifying the subscript type
- subs — Cell array or string containing the actual subscripts

For instance, the syntax `a(1:2,:) = b` calls `a=subsasgn(a,S,b)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

Examples

Example 1

For fi objects a and b, there is a difference between

```
a = b
```

and

```
a(:) = b
```

In the first case, a = b replaces a with b while a assumes the value, numerictype object and fimath object associated with b.

In the second case, a(:) = b assigns the value of b into a while keeping the numerictype object of a. You can use this to cast a value with one numerictype object into another numerictype object.

For example, cast a 16-bit number into an 8-bit number:

```
a = fi(0, 1, 8, 7)

a =

     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7

b = fi(pi/4, 1, 16, 15)

b =

    0.7854

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

```
a(:) = b

a =

    0.7891

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Example 2

This example defines a variable `acc` to emulate a 40–bit accumulator of a DSP. The products and sums in this example are assigned into the accumulator using the syntax `acc(1) = ...`. Assigning values into the accumulator is like storing a value in a register.

To begin, turn the logging mode on and define the variables. In this example, `n` is the number of points in the input data `x` and output data `y`, and `t` represents time. The remaining variables are all defined as `fi` objects. The input data `x` is a high-frequency sinusoid added to a low-frequency sinusoid.

```
fipref('LoggingMode','on');
n = 100;
t = (0:n-1)/n;
x = fi(sin(2*pi*t) + 0.2*cos(2*pi*50*t));
b = fi([.5 .5]);
y = fi(zeros(size(x)), numerictype(x));
acc = fi(0.0, true, 40, 30);
```

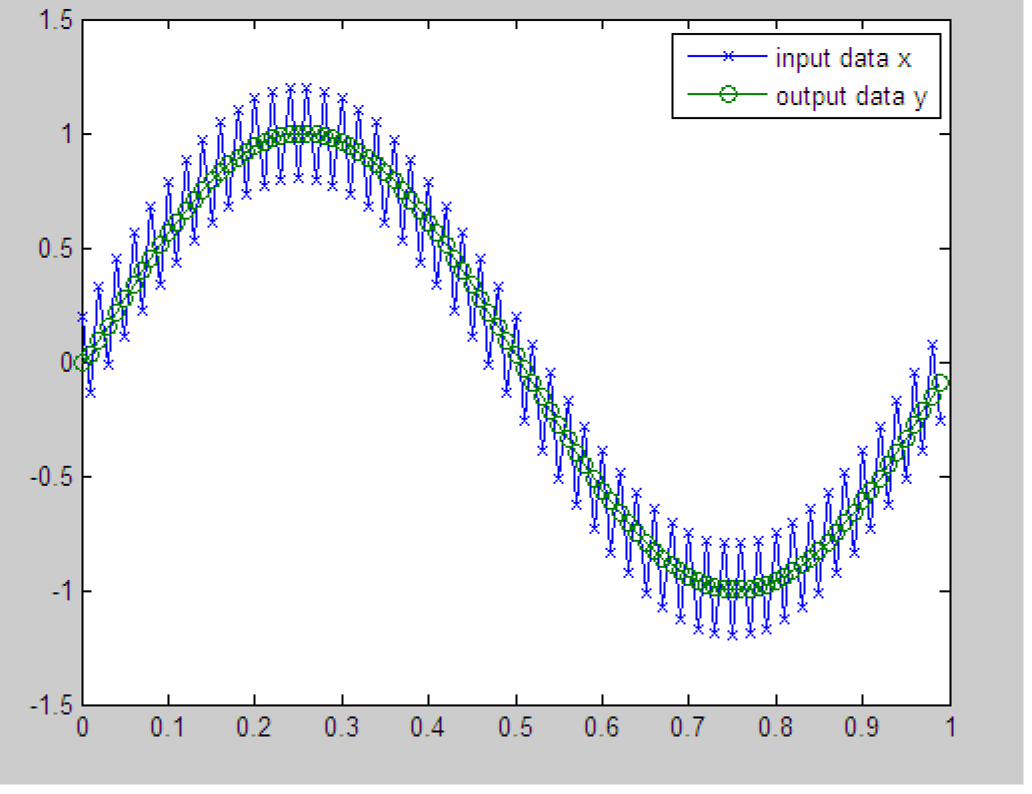
The following loop takes a running average of the input `x` using the coefficients in `b`. Notice that `acc` is assigned into `acc(1) = ...` versus using `acc = ...`, which would overwrite and change the data type of `acc`.

```
for k = 2:n
```

```
        acc(1) = b(1)*x(k);
        acc(1) = acc + b(2)*x(k-1);
        y(k) = acc;
    end
```

By averaging every other sample, the loop shown above passes the low-frequency sinusoid through and attenuates the high-frequency sinusoid.

```
plot(t,x,'x-',t,y,'o-')
legend('input data x','output data y')
```



The log report shows the minimum and maximum logged values and ranges of the variables used. Because acc is assigned into, rather than over written, these logs reflect the accumulated minimum and maximum values.

```
logreport(x,y,b,acc)
```

The table below shows selected output from the log report:

Value	minlog	maxlog	lowerbound	upperbound
x	–1.200012	1.197998	–2	1.999939
y	–0.9990234	0.9990234	–2	1.999939
b	0.5	0.5	–1	0.9999695
acc	–0.9990234	0.9989929	–512	512

Display acc to verify that its data type did not change:

```
acc

acc =

   -0.0941

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

See Also

subsref

Purpose Subscripted reference

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

Purpose

Sum of array elements

Syntax

```
b = sum(a)
b = sum(a, dim)
```

Description

`b = sum(a)` returns the sum along different dimensions of the `fi` array `a`.

If `a` is a vector, `sum(a)` returns the sum of the elements.

If `a` is a matrix, `sum(a)` treats the columns of `a` as vectors, returning a row vector of the sums of each column.

If `a` is a multidimensional array, `sum(a)` treats the values along the first nonsingleton dimension as vectors, returning an array of row vectors.

`b = sum(a, dim)` sums along the dimension `dim` of `a`.

The `fimath` object is used in the calculation of the sum. If `SumMode` is `FullPrecision`, `KeepLSB`, or `KeepMSB`, then the number of integer bits of growth for `sum(a)` is `ceil(log2(length(a)))`.

`sum` does not support `fi` objects of data type `Boolean`.

See Also

`add`, `divide`, `fi`, `fimath`, `mpy`, `mrdivide`, `numerictype`, `rdivide`, `sub`

Purpose Create 3-D shaded surface plot

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

Purpose Create 3-D shaded surface plot with contour plot

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

Purpose Create surface plot with colormap-based lighting

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

surfnorm

Purpose Compute and display 3-D surface normals

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

Purpose Create text object in current axes

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

times

Purpose

Element-by-element multiplication of `fi` objects

Syntax

`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 value. A scalar value can be multiplied by any other value.

`times` does not support `fi` objects of data type `Boolean`.

Note For information about the `fimath` properties involved in Fixed-Point Toolbox calculations, see "Using fimath Properties to Perform Fixed-Point Arithmetic" and "Using fimath ProductMode and SumMode" in the *Fixed-Point Toolbox User's Guide*.

For information about calculations using Simulink Fixed Point software, see the "Arithmetic Operations" chapter of the *Simulink Fixed Point User's Guide*.

See Also

`plus`, `minus`, `mtimes`, `uminus`

Purpose

Create Toeplitz matrix

Syntax

```
t = toeplitz(a,b)
t = toeplitz(b)
```

Description

`t = toeplitz(a,b)` returns a nonsymmetric Toeplitz matrix having `a` as its first column and `b` as its first row. `b` is cast to the `numerictype` of `a`.

`t = toeplitz(b)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `b`, where `b` is the first row of the matrix.

The output `fi` object `t` has the same `numerictype` properties as the leftmost `fi` object input. If the leftmost `fi` object input has a local fimath, the output `fi` object `t` is assigned the same local fimath. Otherwise, the output `fi` object `t` is associated with the global fimath.

Examples

`toeplitz(a,b)` casts `b` into the data type of `a`. In this example, overflow occurs:

```
fipref('NumericTypeDisplay','short');
format short g
a = fi([1 2 3],true,8,5)

a =

     1     2     3
      s8,5
b = fi([1 4 8],true,16,10)

b =

     1     4     8
      s16,10
```

```
toeplitz(a,b)

ans =

          1     3.9688     3.9688
          2          1     3.9688
          3          2          1
     s8,5
```

`toeplitz(b,a)` casts `a` into the data type of `b`. In this example, overflow does not occur:

```
toeplitz(b,a)

ans =

    1    2    3
    4    1    2
    8    4    1
     s16,10
```

If one of the arguments of `toeplitz` is a built-in data type, it is cast to the data type of the `fi` object.

```
x = [1 exp(1) pi]

x =

          1     2.7183     3.1416

toeplitz(a,x)

ans =

          1     2.7188     3.1563
          2          1     2.7188
          3          2          1
     s8,5
```

```
toeplitz(x,a)

ans =

           1              2              3
      2.7188              1              2
      3.1563         2.7188              1
    s8,5
```

tostring

Purpose

Convert `numerictype` or `quantizer` object to string

Syntax

```
s = tostring(T)
s = tostring(q)
```

Description

`s = tostring(T)` converts `numerictype` object `T` to a string `s` such that `eval(s)` would create a `numerictype` object with the same properties as `T`.

`s = tostring(q)` converts `quantizer` object `q` to a string `s`. After converting `q` to a string, the function `eval(s)` can use `s` to create a `quantizer` object with the same properties as `q`.

Examples

This example uses the `tostring` function to convert a `numerictype` object `T` to a string `s`

```
T = numerictype(true,16,15);
s = tostring(T);
T1 = eval(s);
isequal(T,T1)

ans =

     1
```

See Also

`eval`, `numerictypequantizer`

Purpose Transpose operation

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

treeplot

Purpose Plot picture of tree

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

Purpose Lower triangular part of matrix

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

trimesh

Purpose Create triangular mesh plot

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

Purpose

Create 2-D triangular plot

Description

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

Purpose Create triangular surface plot

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

Purpose Upper triangular part of matrix

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

Purpose

Construct unsigned fixed-point numeric object

Syntax

```
a = ufi
a = ufi(v)
a = ufi(v,w)
a = ufi(v,w,f)
a = ufi(v,w,slope,bias)
a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

Description

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

- `a = ufi` is the default constructor and returns an unsigned `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = ufi(v)` returns an unsigned fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = ufi(v,w)` returns an unsigned fixed-point object with value `v`, word length `w`, and best-precision fraction length.
- `a = ufi(v,w,f)` returns an unsigned fixed-point object with value `v`, word length `w`, and fraction length `f`.
- `a = ufi(v,w,slope,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slope`, and `bias`.
- `a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`fi` objects created by the `ufi` constructor function have the following general types of properties:

- “Data Properties” on page 3-133
- “fimath Properties” on page 3-417
- “numerictype Properties” on page 3-135

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

Note fi objects created by the ufi constructor function are always associated with the global fimath. See “Working with the Global fimath” in the *Fixed-Point Toolbox User’s Guide* for more information.

Data Properties

The data properties of a fi object are always writable.

- bin — Stored integer value of a fi object in binary
- data — Numerical real-world value of a fi object
- dec — Stored integer value of a fi object in decimal
- double — Real-world value of a fi object, stored as a MATLAB double
- hex — Stored integer value of a fi object in hexadecimal
- int — Stored integer value of a fi object, stored in a built-in MATLAB integer data type. You can also use int8, int16, int32, int64, uint8, uint16, uint32, and uint64 to get the stored integer value of a fi object in these formats
- oct — Stored integer value of a fi object in octal

These properties are described in detail in “fi Object Properties” on page 1-2.

fimath Properties

When you create a fi object with the ufi constructor function, that fi object does not have a local fimath object. Instead, the fi object is associated with the global fimath. When a fi object is associated with the global fimath, you can change its fimath properties by reconfiguring the global fimath, or by assigning the fi object a local fimath object.

For more information, see "Working with the Global fimath" in the *Fixed-Point Toolbox User's Guide*.

- `fimath` — fixed-point math object

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

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

---

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

- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow mode
- `ProductBias` — Bias of the product data type
- `ProductFixedExponent` — Fixed exponent of the product data type
- `ProductFractionLength` — Fraction length, in bits, of the product data type
- `ProductMode` — Defines how the product data type is determined
- `ProductSlope` — Slope of the product data type
- `ProductSlopeAdjustmentFactor` — Slope adjustment factor of the product data type
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode

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

These properties are described in detail in “fimath Object Properties” on page 1-4.

numerictype Properties

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

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

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

- `Bias` — Bias of a `fi` object
- `DataType` — Data type category associated with a `fi` object
- `DataTypeMode` — Data type and scaling mode of a `fi` object
- `FixedExponent` — Fixed-point exponent associated with a `fi` object
- `SlopeAdjustmentFactor` — Slope adjustment associated with a `fi` object

- `FractionLength` — Fraction length of the stored integer value of a `fi` object in bits
- `Scaling` — Fixed-point scaling mode of a `fi` object
- `Signed` — Whether a `fi` object is signed or unsigned
- `Signedness` — Whether a `fi` object is signed or unsigned

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

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

For further details on these properties, see "numerictype Object Properties" on page 1-15.

Examples

> **Note** For information about the display format of `fi` objects, refer to Display Settings.
>
> For examples of casting, see "Casting fi Objects".

Example 1

For example, the following creates an unsigned `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = ufi(pi,8,3)

a =
```

```
      3.1250

            DataTypeMode: Fixed-point: binary point scaling
              Signedness: Unsigned
              WordLength: 8
          FractionLength: 3
```

The `fimath` properties associated with `a` come from the global fimath. When a `fi` object does not have a local `fimath` object, it associates itself with the global fimath, and no `fimath` object properties are displayed in its output. To determine whether a `fi` object is associated with the global fimath, or has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)

ans =
     0
```

A returned value of `0` means the `fi` object is associated with the global fimath and does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

Example 2

The value `v` can also be an array:

```
a = ufi((magic(3)/10),16,12)

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

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

```
        FractionLength: 12
>>
```

Example 3

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

```
a = ufi(pi,8)

a =

    3.1406

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 6
```

Example 4

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

```
a = ufi(pi)

a =

    3.1416

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

See Also

`fi`, `fimath`, `fipref`, `isfimathlocal`, `numerictype`, `quantizer`, `sfi`

Purpose

Stored integer value of `fi` object as built-in `uint8`

Syntax

```
c = uint8(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint16`, `uint32`, `uint64`

uint16

Purpose

Stored integer value of `fi` object as built-in `uint16`

Syntax

```
c = uint16(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint8`, `uint32`, `uint64`

Purpose

Stored integer value of `fi` object as built-in `uint32`

Syntax

```
c = uint32(a)
```

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = 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`, `int64`, `uint8`, `uint16`, `uint64`

uint64

Purpose

Stored integer value of `fi` object as built-in `uint64`

Syntax

`c = uint64(a)`

Description

Fixed-point numbers can be represented as

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

or, equivalently as

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

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

`c = uint64(a)` returns the stored integer value of `fi` object `a` as a built-in `uint64`. If the stored integer word length is too big for a `uint64`, or if the stored integer is signed, the returned value saturates to a `uint64`.

See Also

`int`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`

Purpose

Negate elements of `fi` object array

Syntax

`uminus(a)`

Description

`uminus(a)` is called for the syntax `-a` when `a` is an object. `-a` negates the elements of `a`.

`uminus` does not support `fi` objects of data type `Boolean`.

Examples

When wrap occurs, -(-1) = -1 :

```
fipref('NumericTypeDisplay','short', ...
       'fimathDisplay','none');
format short g
a = fi(-1,true,8,7,'overflowmode','wrap')

a =

    -1
      s8,7
-a

ans =

    -1
      s8,7
b = fi([-1-i -1-i],true,8,7,'overflowmode','wrap')

b =

           -1 -          1i           -1 -          1i
      s8,7
-b

ans =

           -1 -          1i           -1 -          1i
```

```
      s8,7
b'

ans =

            -1 -            1i
            -1 -            1i
      s8,7
```

When saturation occurs, -(-1) = 0.99... :

```
c = fi(-1,true,8,7,'overflowmode','saturate')

c =

    -1
      s8,7
-c

ans =

      0.99219
      s8,7
d = fi([-1-i -1-i],true,8,7,'overflowmode','saturate')

d =

            -1 -            1i            -1 -            1i
      s8,7
-d

ans =

      0.99219 +    0.99219i      0.99219 +    0.99219i
      s8,7
d'
```

```
ans =

              -1 +    0.99219i
              -1 +    0.99219i
       s8,7
```

See Also

plus, minus, mtimes, times

unitquantize

Purpose

Quantize except numbers within eps of +1

Syntax

```
y = unitquantize(q, x)
[y1,y2,...] = unitquantize(q,x1,x2,...)
```

Description

`y = unitquantize(q, x)` works the same as `quantize` except that numbers within `eps(q)` of +1 are made exactly equal to +1 .

`[y1,y2,...]  = unitquantize(q,x1,x2,...)` is equivalent to

```
y1 = unitquantize(q,x1), y2 = unitquantize(q,x2),...
```

Examples

This example demonstrates the use of `unitquantize` with a `quantizer` object `q` and a vector `x`.

```
q = quantizer('fixed','floor','saturate',[4 3]);
x = (0.8:.1:1.2)';
y = unitquantize(q,x);
z = [x y]
e = eps(q)
```

This quantization outputs an array containing the original values of `x` and the quantized values of `x`, followed by the value of `eps(q)`:

```
z =

    0.8000    0.7500
    0.9000    1.0000
    1.0000    1.0000
    1.1000    1.0000
    1.2000    1.0000


e =

    0.1250
```

See Also

`eps`, `quantize`, `quantizer`, `unitquantizer`

unitquantizer

Purpose Constructor for `unitquantizer` object

Syntax

```
q = unitquantizer(...)
```

Description `q = unitquantizer(...)` constructs a `unitquantizer` object, which is the same as a `quantizer` object in all respects except that its `quantize` method quantizes numbers within `eps(q)` of `+1` to exactly `+1`.

See `quantizer` for parameters.

Examples In this example, a vector `x` is quantized by a `unitquantizer` object `u` .

```
u = unitquantizer([4 3]);
x = (0.8:.1:1.2)';
y = quantize(u,x);
z = [x y]
e = eps(u)
```

This quantization outputs an array containing the original values of `x` and the values of `x` that were quantized by the `unitquantizer` object `u`. The output also includes `e`, the value of `eps(u)`.

```
z =

    0.8000    0.7500
    0.9000    1.0000
    1.0000    1.0000
    1.1000    1.0000
    1.2000    1.0000


e =

    0.1250
```

See Also `quantize`, `quantizer`, `unitquantize`

Purpose

Inverse of `shiftdata`

Syntax

```
y = unshiftdata(x,perm,nshifts)
```

Description

`y = unshiftdata(x,perm,nshifts)` restores the orientation of the data that was shifted with `shiftdata`. The permutation vector is given by `perm`, and `nshifts` is the number of shifts that was returned from `shiftdata`.

`unshiftdata` is meant to be used in tandem with `shiftdata`. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples

Example 1

This example shifts `x`, a `3-by-3` magic square, permuting dimension `2` to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a `3-by-3` magic square:

```
x = fi(magic(3))

x =

     8     1     6
     3     5     7
     4     9     2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

This command returns the permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =
```

```
     8     3     4
     1     5     9
     6     7     2


perm =

     2     1


nshifts =

     []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2
```

Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5

x =

     1     2     3     4     5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

This command returns `x` as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =

     1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     1     2     3     4     5
```

See Also

`ipermute`, `shiftdata`, `shiftdim`

uplus

Purpose Unary plus

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

Purpose Upper bound of range of `fi` object

Syntax `upperbound(a)`

Description `upperbound(a)` returns the upper bound of the range of `fi` object `a`. If `L = lowerbound(a)` and `U = upperbound(a)`, then `[L,U] = range(a)`.

See Also `eps`, `intmax`, `intmin`, `lowerbound`, `lsb`, `range`, `realmax`, `realmin`

Purpose

Vertically concatenate multiple `fi` objects

Syntax

```
c = vertcat(a,b,...)
[a; b; ...]
[a;b]
```

Description

`c = vertcat(a,b,...)` is called for the syntax `[a; b; ...]` when any of `a`, `b`, `...` , is a `fi` object.

`[a;b]` is the vertical concatenation of matrices `a` and `b`. `a` and `b` must have the same number of columns. Any number of matrices can be concatenated within one pair of brackets. N-D arrays are vertically concatenated along the first dimension. The remaining dimensions must match.

Horizontal and vertical concatenation can be combined, as in `[1 2;3 4]`.

`[a b; c]` is allowed if the number of rows of `a` equals the number of rows of `b`, and if the number of columns of `a` plus the number of columns of `b` equals the number of columns of `c`.

The matrices in a concatenation expression can themselves be formed via a concatenation, as in `[a b;[c d]]`.

Note The `fimath` and `numerictype` objects of a concatenated matrix of `fi` objects `c` are taken from the leftmost `fi` object in the list `(a,b,...)`.

See Also

`horzcat`

Purpose Create Voronoi diagram

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

voronoin

Purpose Create n-D Voronoi diagram

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

Purpose Create waterfall plot

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

wordlength

Purpose

Word length of `quantizer` object

Syntax

```
wordlength(q)
```

Description

`wordlength(q)` returns the word length of the `quantizer` object `q`.

Examples

```
q = quantizer([16 15]);
wordlength(q)

ans =

    16
```

See Also

`fi`, `fractionlength`, `exponentlength`, `numerictype`, `quantizer`

Purpose Set or query x-axis limits

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

Purpose Logical exclusive-OR

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

Purpose Set or query y-axis limits

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

zlim

Purpose Set or query z-axis limits

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

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe products from The MathWorks™ that have fixed-point support.

arithmetic shift

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

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

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 software.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000
0.000
00.00
000.0
0000.

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `0`.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, signedness, word length

data type override

Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

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

2. Fixed-point numbers can be represented as

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

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

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

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

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

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

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

or

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

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

logical shift

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

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

See also exponent, floating-point representation

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Toolbox software.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

0000__.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$__.

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

$Qm.n$

where

- Q indicates that the number is in Q format.
- m is the number of bits used to designate the two's complement integer part of the number.

- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, precision

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

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

See also integer

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaled double

A double data type that retains fixed-point scaling information. For example, in Simulink and Fixed-Point Toolbox software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

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

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, signedness, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, signedness, unsigned fixed-point

signedness

The signedness of a number or data type can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

See also data type, sign bit, sign/magnitude representation, signed fixed-point, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]

Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer

See **integer**

trivial scaling

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$real\text{-}world\ value = stored\ integer$$

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

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

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:

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

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

See also bias, binary point, binary point-only scaling, fixed-point representation, fraction length, integer, least significant bit, scaling, slope, [Slope Bias]

truncation

Rounding mode that drops one or more least significant bits from a number.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, zero (round toward)

two's complement representation

Common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

See also binary word, one's complement representation, sign/magnitude representation, signed fixed-point

unsigned fixed-point

Fixed-point number or data type that can only represent numbers greater than or equal to zero.

See also data type, fixed-point representation, signed fixed-point, signedness

word

Fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also binary word, data type

word length

Number of bits in a binary word or data type.

See also binary word, bit, data type

wrapping

Method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range the data type being used back into the representable range.

See also data type, overflow, range, saturation

zero (round toward)

Rounding mode that rounds to the closest representable number in the direction of zero. This is equivalent to the `fix` mode in Fixed-Point Toolbox software.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation

Index

F

G

H

I

Q

R

S