Fixed-Point Designer™

Reference

R2013a

MATLAB®

MathWorks®

How to Contact MathWorks

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Fixed-Point Designer™ Reference

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)

Contents

1 Property Reference

fi Object Properties **1-2**
bin **1-2**
data **1-2**
dec **1-2**
double **1-2**
fimath **1-2**
hex **1-3**
NumericType **1-3**
oct **1-3**

fimath Object Properties **1-4**
CastBeforeSum **1-4**
MaxProductWordLength **1-4**
MaxSumWordLength **1-4**
OverflowAction **1-4**
ProductBias **1-5**
ProductFixedExponent **1-5**
ProductFractionLength **1-5**
ProductMode **1-5**
ProductSlope **1-7**
ProductSlopeAdjustmentFactor **1-7**
ProductWordLength **1-7**
RoundingMethod **1-8**
SumBias **1-8**
SumFixedExponent **1-8**
SumFractionLength **1-9**
SumMode **1-9**
SumSlope **1-11**
SumSlopeAdjustmentFactor **1-11**
SumWordLength **1-11**

fipref Object Properties **1-12**
DataTypeOverride **1-12**
DataTypeOverrideAppliesTo **1-12**
FimathDisplay **1-12**

LoggingMode **1-13**
NumericTypeDisplay **1-13**
NumberDisplay **1-14**

numerictype Object Properties **1-15**
Bias **1-15**
DataType **1-15**
DataTypeMode **1-15**
DataTypeOverride **1-16**
FixedExponent **1-16**
FractionLength **1-17**
Scaling **1-17**
Signed **1-17**
Signedness **1-18**
Slope **1-18**
SlopeAdjustmentFactor **1-19**
WordLength **1-19**

quantizer Object Properties **1-20**
DataMode **1-20**
Format **1-20**
OverflowAction **1-21**
RoundingMethod **1-22**

2 Functions — Alphabetical List

Glossary

A Selected Bibliography

Index

Property Reference

- "fi Object Properties" on page 1-2
- "fimath Object Properties" on page 1-4
- "fipref Object Properties" on page 1-12
- "numerictype Object Properties" on page 1-15
- "quantizer Object Properties" on page 1-20

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 get their `fimath` properties from a local `fimath` object or from default values. The factory-default fimath values have the following settings:

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

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

hex

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

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 Designer™ User's Guide.

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

---

oct

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

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 `65535`.

MaxSumWordLength

Maximum allowable word length for the sum data type.

The MATLAB factory default value of this property is `65535`.

OverflowAction

Overflow-handling action. The value of the `OverflowAction` 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`.

RoundingMethod

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

- `Ceiling` — 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 Designer software.
- `Zero` — 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 `Nnearest`.

See "Rounding Methods" in the Fixed-Point Designer 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.

  $$W_s = \text{specified in the } \texttt{SumWordLength} \text{ property}$$
  $$F_s = \max(F_a, F_b)$$

- 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.

  $$W_s = \text{specified in the } \texttt{SumWordLength} \text{ property}$$
  $$F_s = W_s - \text{integer length}$$

  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.

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

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

  $$S_s = \text{specified in the } \texttt{SumSlope} \text{ property}$$
  $$B_s = \text{specified in the } \texttt{SumBias} \text{ property}$$

  [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`.

DataTypeOverrideAppliesTo

Data type override application to `fi` objects

- `AllNumericTypes` — Apply data type override to all `fi` data types
- `Fixed-Point` — Apply data type override only to fixed-point data types
- `Floating-Point` — Apply data type override only to floating-point `fi` data types

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

The default value of this property is `AllNumericTypes`.

FimathDisplay

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

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

- none — None of the fimath attributes are displayed

The default value of this property is full.

LoggingMode

Logging options for operations performed on fi objects

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

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

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

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

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

The default value of this property 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

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

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

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

where the slope can be expressed as

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

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`.

DataTypeOverride

Data type override for applying `fipref` data type override settings to `fi` objects. This property provides a convenient way to ignore a global `fipref` data type override setting. This property is not visible when its value is the default, `Inherit`. The possible values of this property are:

- `Inherit` — `fi` object uses the `fipref` `DataTypeOverride` setting.
- `Off` — `fi` object uses the `numerictype` data type settings and ignores `fipref` settings

The default value of this property is `Inherit`.

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

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

where the slope can be expressed as

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

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

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

`FixedExponent` must an integer.

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
- `[]` — auto

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

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

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

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

where the slope can be expressed as

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

`SlopeAdjustmentFactor` must be greater than or equal to 1 and less than 2.

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.

OverflowAction

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

- Saturate — Overflows saturate.

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

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

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

The default value of this property is `Saturate`.

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

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

RoundingMethod

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

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

The default value of this property is `Floor`.

Functions — Alphabetical List

Purpose

Absolute value of `fi` object

Syntax

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

Description

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

`c = abs(a,T)` returns a `fi` object with a value equal to the absolute value of `a` and `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` associated with `a` and the output `fi` object `c` has the same local fimath as `a`. See "Data Type Propagation Rules" on page 2-3.

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

`c = abs(a,T,F)` returns a `fi` object with a value equal to the absolute value of `a` and the `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` object `F`. The output `fi` object `c` has no local fimath. See "Data Type Propagation Rules" on page 2-3.

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

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

When the object `a` is real and has a signed data type, the absolute value of the most negative value is problematic since it is not representable.

In this case, the absolute value saturates to the most positive value representable by the data type if the `OverflowMode` property is set to `saturate`. If `OverflowMode` is `wrap`, the absolute value of the most negative value has no effect.

Data Type Propagation Rules

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

Data Type of Input 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

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

  abs(a)
```

```
ans =

  -128

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

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

Example 2

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

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

re =

    -1

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

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

im =

     0
```

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

a = complex(re,im)

a =

    -1

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

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

ans =

    1.0000

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

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

ans =

    -1

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

```
        FractionLength: 15
```

Example 3

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

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

a =

    -1

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

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

abs(a)

ans =

    -1

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

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

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

f =

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

abs(a,f)

ans =

    0.9688

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

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

t =


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

abs(a,t,f)
```

```
ans =

     1

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

Example 4

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

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

a =

  -1.0000 - 1.0000i

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

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

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

t =
```

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

abs(a,t)

ans =

    1.4142

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

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

f = fimath('OverflowMode','saturate','SumMode',...
        'keepLSB','SumWordLength',a.WordLength,...
        'ProductMode','specifyprecision',...
        'ProductWordLength',a.WordLength,...
        'ProductFractionLength',a.FractionLength)

f =


           RoundingMethod: Nearest
           OverflowAction: Saturate
              ProductMode: SpecifyPrecision
        ProductWordLength: 16
    ProductFractionLength: 15
                  SumMode: KeepLSB
```

```
        SumWordLength: 16
        CastBeforeSum: true

abs(a,t,f)

ans =

    1.4142

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

Algorithms

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

y = a 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

Subtract two fi objects or values

Syntax

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

Description

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

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

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

Input Arguments

a

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

b

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

RoundingMethod

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

Default: Floor

OverflowAction

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

Default: `Wrap`

Output Arguments

c

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

Examples

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

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

See Also

`accumpos`

Purpose

Add two `fi` objects or values

Syntax

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

Description

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

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

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

Input Arguments

a

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

b

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

RoundingMethod

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

Default: `Floor`

OverflowAction

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

Default: `Wrap`

Output Arguments

c

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

Examples

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

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

See Also

`accumneg`

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` has no local fimath.

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

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

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

c =

    5.8599


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

Algorithms `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` has no local fimath. When you use the syntax `c = a + b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See "fimath Rules for Fixed-Point Arithmetic" in the Fixed-Point Designer User's Guide for more information.

See Also

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

Purpose Determine whether all array elements are nonzero

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

Purpose Find logical AND of array or scalar inputs

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

Purpose Determine whether any array elements are nonzero

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

area

Purpose Create filled area 2-D plot

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

Purpose Assignment `quantizer` object of `fi` object

Syntax

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

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

See Also `quantize` | `quantizer`

atan2

Purpose

Four-quadrant inverse tangent of fixed-point values

Syntax

`z = atan2(y,x)`

Description

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

Input Arguments

y,x

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

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

Output Arguments

z

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

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

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

Definitions

Four-Quadrant Arctangent

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

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

Examples

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

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

z =

    0.2450

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

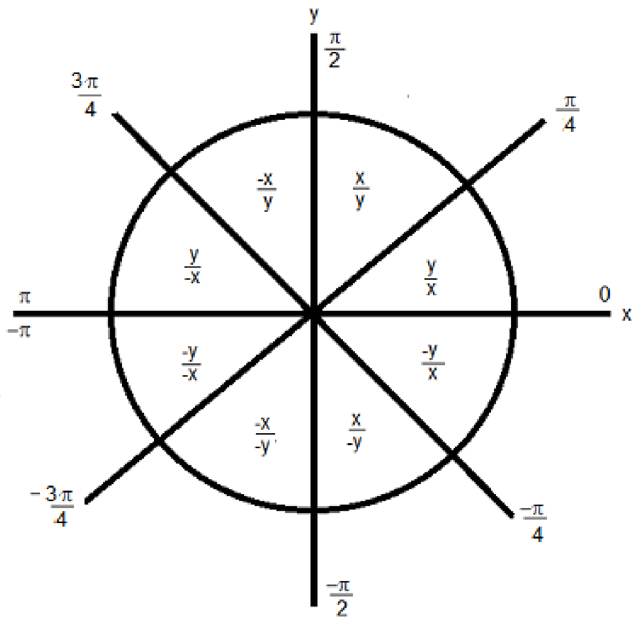
```
y = fi(-0.1,1,16);
x = fi(-0.9,1,16);
z = atan2(y,x)

z =

   -3.0309

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

Algorithms

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

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

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



2 Compute the table index, based on the 16-bit, unsigned, stored integer value:

a Use the 8 most-significant bits to obtain the first value from the table.

b Use the next-greater table value as the second value.

3 Use the 8 least-significant bits to interpolate between the first and second values using nearest neighbor linear interpolation. This interpolation produces a value in the range [0, pi/4).

4 Perform octant correction on the resulting angle, based on the values of the original `y` and `x` inputs.

See Also `sin` | `angle` | `cos`

Purpose

Automatically change scaling of fixed-point data types

Syntax

```
autofixexp
```

Description

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

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

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

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

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

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

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

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

See Also

`fxptdlg`

Purpose Create vertical bar graph

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

barh

Purpose Create horizontal bar graph

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

Purpose

Binary representation of stored integer of `fi` object

Syntax

`bin(a)`

Description

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

Fixed-point numbers can be represented as

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

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` | `storedInteger` | `oct`

bin2num

Purpose

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

Syntax

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

Description

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

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

Examples

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

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

b =

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

```
1100
1011
1010
1001
1000
```

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

```
y=bin2num(q,b)

y =

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

See Also

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

bitand

Purpose Bitwise AND of two fi objects

Syntax

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

Description

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

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

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

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

See Also bitcmp | bitget | bitor | bitset | bitxor

Purpose

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.

Examples

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.

Examples

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

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

1010
```

Complement the values of the bits in `a`:

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

0101
```

See Also

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

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.

Examples

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

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.

Examples

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`

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 `OverflowAction` and `RoundingMethod` 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.

Examples

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`

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 `OverflowAction` and `RoundingMethod` 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.

Examples

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.

Examples

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

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

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

Examples

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

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

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

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

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

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

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

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

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

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`.

Examples

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 any numeric type, including double, single, integer, or fixed-point. For fixed-point operations, the `OverflowAction` and `RoundingMethod` 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 a nonnegative, integer-valued constant.

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

Examples

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
```

You can also use `bitsll` with floating-point inputs. The following example scales the `double` input by 2^3:

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

ans =

        1024
```

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 `OverflowAction` and `RoundingMethod` 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 a nonnegative, integer-valued constant.

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

Examples

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 scales the `double` input by 2^{-3}:

```
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 `OverflowAction` and `RoundingMethod` 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 a nonnegative, integer-valued constant.

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

Examples

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.

Examples

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`

buffer

Purpose Buffer signal vector into matrix of data frames

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

Purpose Generate compiled C code function including logging instrumentation

Syntax

```
buildInstrumentedMex fcn -options
```

Description

`buildInstrumentedMex` *`fcn -options`* translates the MATLAB file *`fcn`*`.m` to a MEX function and enables instrumentation for logging minimum and maximum values of all named and intermediate variables. Optionally, you can enable instrumentation for log2 histograms of all named, intermediate and expression values.

Tips

- Arrays of structures are not logged. Only scalar (1x1) structures are logged.
- You cannot instrument MATLAB functions provided with the software. If your top-level function is such a MATLAB function, nothing is logged. You also cannot instrument scripts.
- Instrumentation results are accumulated every time the instrumented MEX function is called. Use `clearInstrumentationResults` to clear previous results in the log.
- Some coding patterns pass a significant amount of data, but only use a small portion of that data. In such cases, you may see degraded performance when using `buildInstrumentedMex`. In the following pattern, `subfun` only uses one element of input array, `A`. For normal execution, the amount of time to execute `subfun` once remians constant regardless of the size of `A`. The function `topfun` calls `subfun` `N` times, and thus the total time to execute `topfun` is proportional to `N`. When instrumented, however, the time to execute `subfun` once becomes proportional to `N^2`. This change occurs because the minimum and maximum data are calculated over the entire array. When `A` is large, the calculations can lead to significant performance degradation. Therefore, whenever possible, you should pass only the data that the function actually needs.

  ```
  function A = topfun(A)
      N = numel(A);
      for i=1:N
  ```

```
        A(i) = subfun(A,i);
    end
end
function b = subfun(A,i)
    b = 0.5 * A(i);
end

function A = topfun(A)
    N = numel(A);
    for i=1:N
        A(i) = subfun(A(i));
    end
end
function b = subfun(a)
    b = 0.5 * a;
end
```

Input Arguments

fcn

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

options

Choice of compiler options. `buildInstrumentedMex` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

`-args` *`example_inputs`*	Define the size, class, and complexity of all MATLAB function inputs. Use the values in *`example_inputs`* to define these properties. *`example_inputs`* must be a cell array that specifies the same number and order of inputs as the MATLAB function.
`-coder`	Use MATLAB Coder™ software to compile the MEX file, instead of the default Fixed-Point Designer `fiaccel` function. This option removes `fiaccel` restrictions and allows for full code generation support. You must have a MATLAB Coder license to use this option.
`-config` *`config_object`*	Specify MEX generation parameters, based on *`config_object`*, defined as a MATLAB variable using `coder.mexconfig`. For example: `cfg = coder.mexconfig;`
`-d` *`out_folder`*	Store generated files in the absolute or relative path specified by *`out_folder`*. If the folder specified by *`out_folder`* does not exist, `buildInstrumentedMex` creates it for you. If you do not specify the folder location, `buildInstrumentedMex` generates files in the default folder:

	fiaccel/mex/fcn. *fcn* is the name of the MATLAB function specified at the command line. The function does not support the following characters in folder names: asterisk (*), question-mark (?), dollar ($), and pound (#).
`-g`	Compiles the MEX function in debug mode, with optimization turned off. If not specified, `buildinstrumentedMex` generates the MEX function in optimized mode.
`-global` *global_values*	Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `buildInstrumentedMex`. If you do not provide initial values for global variables using the `-global` option, `buildInstrumentedMex` checks for the variable in the MATLAB global workspace. If you do not supply an initial

	value, `buildInstrumentedMex` generates an error. The generated MEX code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ.
`-histogram`	Compute the log2 histogram for all named, intermediate and expression values. A histogram column appears in the code generation report table.
`-I` *`include_path`*	Add *`include_path`* to the beginning of the code generation path. `buildInstrumentedMex` searches the code generation path *first* when converting MATLAB code to MEX code.
`-launchreport`	Generate and open a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-report` option.

`-o` *`output_file_name`*	Generate the MEX function with the base name *`output_file_name`* plus a platform-specific extension. *`output_file_name`* can be a file name or include an existing path. If you do not specify an output file name, the base name is *`fcn_mex`*, which allows you to run the original MATLAB function and the MEX function and compare the results.
`-O` *`optimization_option`*	Optimize generated MEX code, based on the value of *`optimization_option`*: • `enable:inline` — Enable function inlining • `disable:inline` — Disable function inlining If not specified, `buildInstrumentedMex` uses inlining for optimization.
`-report`	Generate a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-launchreport` option.

Examples

Create an instrumented MEX function. Run a test bench, then view logged results.

1 Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('buildInstrumentedMex')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
   'fidemos','fi_m_radix2fft_withscaling.m'),...
   'testfft.m','f')
```

2 Define prototype input arguments.

```
n = 128;
x = complex(zeros(n,1));
W = coder.Constant(fidemo.fi_radix2twiddles(n));
```

3 Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name. Use the `-histogram` option to compute histograms. (If you have a MATLAB Coder license, you may want to also add the `-coder` option. In this case, use `buildInstrumentedMex testfft -coder -o testfft_instrumented -args {x,W}` instead of the following line of code.)

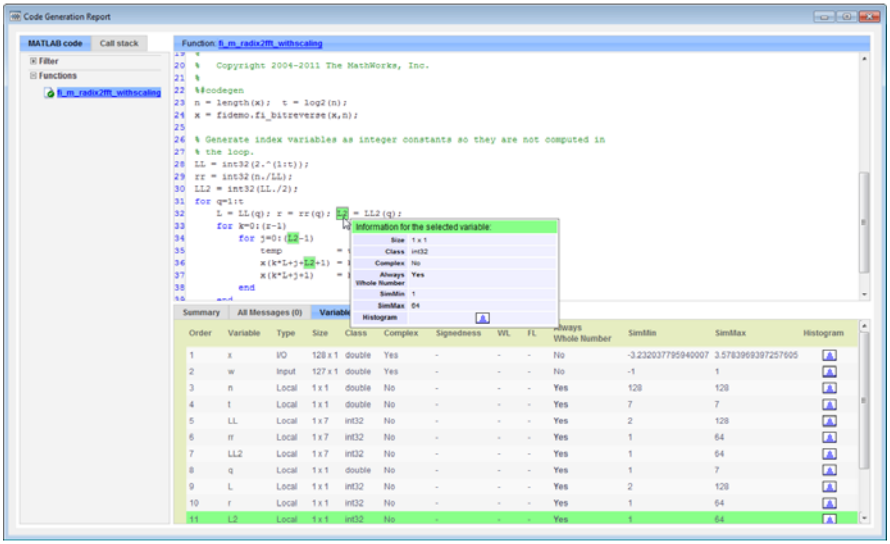
Note Like `fiaccel`, `buildInstrumentedMex` generates a MEX function. To generate C code, see the MATLAB Coder`codegen` function.

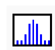
```
buildInstrumentedMex testfft -o testfft_instrumented...
-args {x,W} -histogram
```

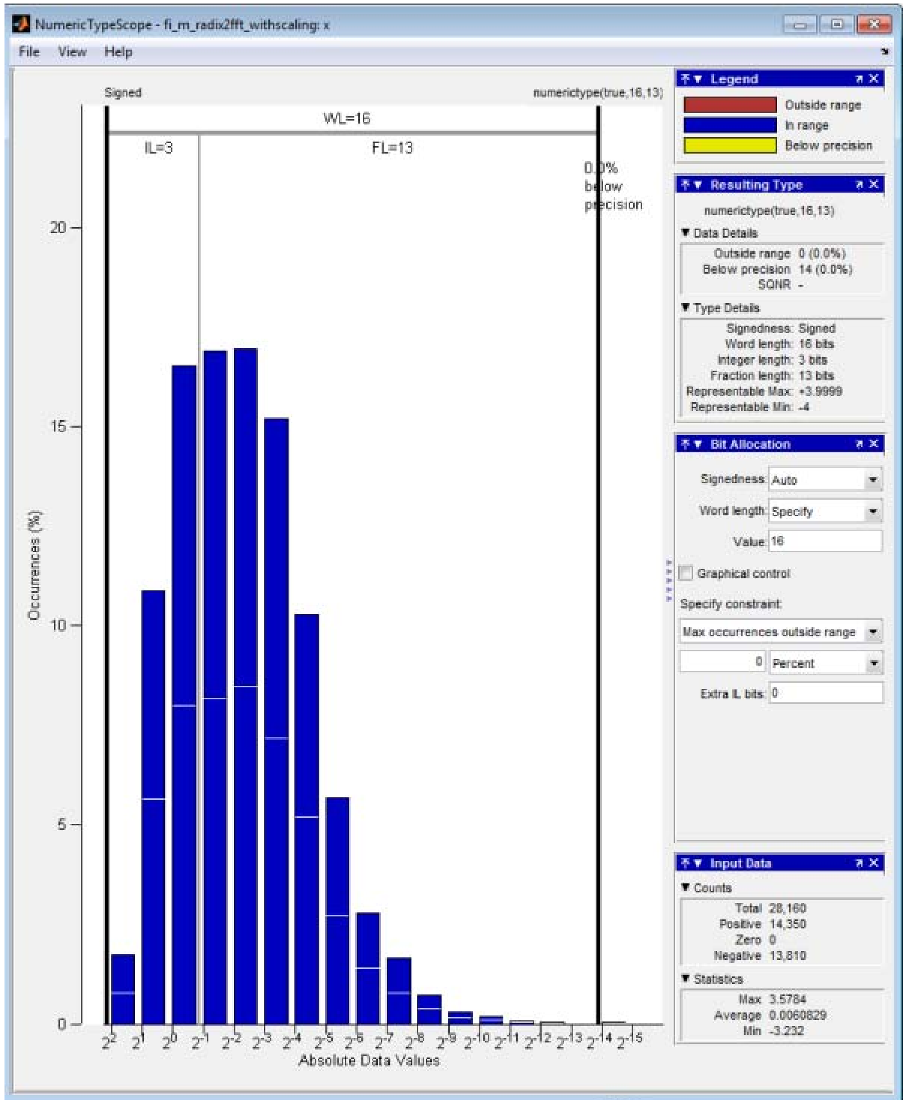
4 Run a test file to record instrumentation results. Call `showInstrumentationResults` to open the Code Generation Report. View the simulation minimum and maximum values and whole number status by hovering over a variable in the report. You can also see proposed data types for double precision numbers in the table.

```
for i=1:20
   y = testfft_instrumented(randn(size(x)));
```

```
end

showInstrumentationResults testfft_instrumented
```



5 View the histogram for a variable by clicking  in the **Variables** tab.



For information on the figure, refer to the `NumericTypeScope` reference page.

6 Close the histogram display and then, clear the results log.

```
clearInstrumentationResults testfft_instrumented;
```

7 Clear the MEX function, then delete temporary files.

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

See Also

`fiaccel` | `clearInstrumentationResults` | `showInstrumentationResults` | `NumericTypeScope` | `codegen` | `mex`

Purpose

Cast variable to different data type

Syntax

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

Description

`b = cast(a,'like',p)` converts `a` to the same `numerictype`, complexity (real or complex), and `fimath` as `p`. If `a` and `p` are both real, then `b` is also real. Otherwise, `b` is complex.

Input Arguments

a - Variable that you want to cast to a different data type

`fi` object | numeric variable

Variable, specified as a `fi` object or numeric variable.
Complex Number Support: Yes

p - Prototype

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.
Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Examples

Convert an int8 Value to Fixed Point

Define a scalar 8–bit integer.

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

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Convert `a` to fixed point with `numerictype`, complexity (real or complex), and `fimath` of the specified `fi` object, `p`.

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

```
b =

     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Match Data Type and Complex Nature of `p`

Define a complex `fi` object.

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

Define a scalar 8–bit integer.

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

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

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

```
b =
```

```
   5.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Convert an Array to Fixed Point

Define a 2-by-3 matrix of ones.

```
A = ones(2,3);
```

Create a signed `fi` object with word length of `16` and fraction length of `8`.

```
p = fi([],1,16,8);
```

Convert `A` to the same data type and complexity (real or complex) as `p`.

```
B = cast(A,'like',p)
```

```
B =

     1     1     1
     1     1     1

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

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
    a = [1.000000000000000  -0.942809041582063   0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);

    % Create a step input using ones with the
```

```
% floating-point data type
t = 0:20;
x_float = ones(size(t),'like',T_float.data);

% Initialize the states using zeros with the
% floating-point data type
z_float = zeros(1,2,'like',T_float.data);

% Run the floating-point algorithm
y_float = my_filter(b,a,x_float,z_float,T_float);

% Define fixed-point types
T_fixed.coeffs = fi([],true,8,6);
T_fixed.data   = fi([],true,8,6);

% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t),'like',T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2,'like',T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf','subplot','plot','legend');
clf
subplot(211)
plot(t,y_float,'co-',t,y_fixed,'kx-')
legend('Floating-point output','Fixed-point output');
title('Step response');
subplot(212)
plot(t,y_float - double(y_fixed),'rs-')
legend('Error')
figure(gcf)
```

```
end
```

See Also

`ones` | `zeros` | `cast`

Related Examples

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

Concepts

- “Workflow for Converting MATLAB Code to Fixed Point at the Command Line”
- “Best Practices for Converting MATLAB Code to Fixed Point at the Command Line”

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`

Purpose Create contour plot elevation labels

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

Purpose

Clear results logged by instrumented, compiled C code function

Syntax

```
clearInstrumentationResults('mex_fcn')
clearInstrumentationResults mex_fcn
clearInstrumentationResults all
```

Description

`clearInstrumentationResults('mex_fcn')` clears the results logged from calling the instrumented MEX function *`mex_fcn`*.

`clearInstrumentationResults` *`mex_fcn`* is alternative syntax for clearing the log.

`clearInstrumentationResults all` clears the results from all instrumented MEX functions.

Input Arguments

mex_fcn

Instrumented MEX function created using `buildInstrumentedMex`.

Examples

Run a test bench to log instrumentation, then use `clearInstrumentationResults` to clear the log.

1. Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('showInstrumentationResults')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
   'fidemos','fi_m_radix2fft_withscaling.m'),...
   'testfft.m','f')
```

2. Define prototype input arguments.

```
n = 128;
x = fi(zeros(n,1));
W = coder.Constant(fi(fidemo.fi_radix2twiddles(n));
```

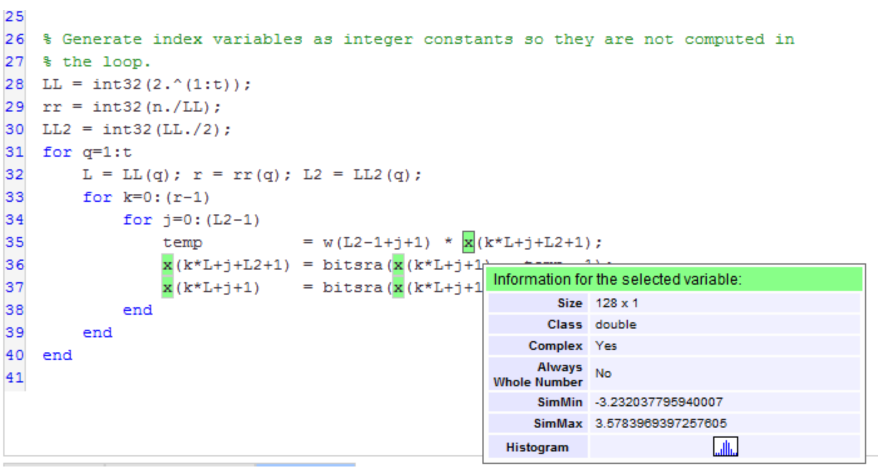
3. Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name.

clearInstrumentationResults

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

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

```
for i=1:20
    y = testfft_instrumented(randn(size(x)));
end

showInstrumentationResults testfft_instrumented
```



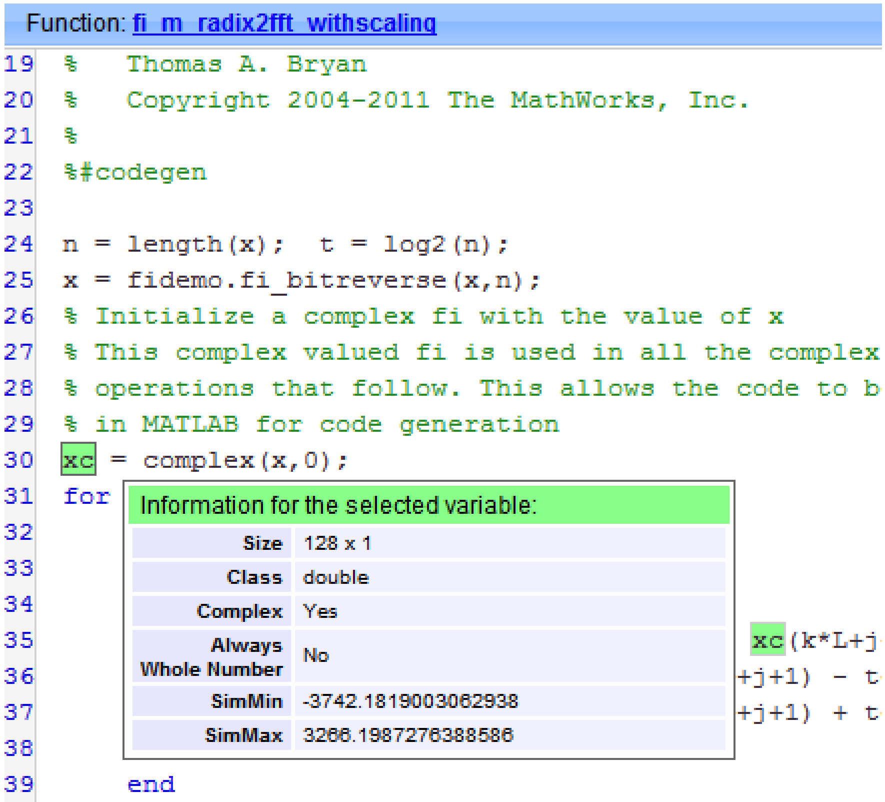
5 Clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

6 Run a different test bench, then view the new instrumentation results.

```
for i=1:20
   y = testfft_instrumented(randn(size(x)));
end
```

```
showInstrumentationResults testfft_instrumented
```



7 Clear the MEX function and delete temporary files.

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

See Also

fiaccel | showInstrumentationResults | buildInstrumentedMex | codegen | mex

coder.allowpcode

Purpose Control code generation from protected MATLAB files

Syntax `coder.allowpcode('plain')`

Description `coder.allowpcode('plain')` allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-file takes precedence.

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

Examples Generate optimized embeddable code from protected MATLAB code:

1. Write an function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

2. Generate protected P-code. At the MATLAB prompt, enter:

```
pcode p_abs
```

   The P-file, `p_abs.p`, appears in the current folder.

3 Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -args { int32(0) }
```

`codegen` generates a MEX function in the current folder.

4 Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` generates C library code in the `codegen\lib\p_abs` folder.

See Also

`pcode` | `codegen`

How To

- "Compilation Directive %#codegen"

coder.ArrayType

Superclasses

Type

Purpose

Represent set of MATLAB arrays

Description

Specifies the set of arrays that the generated code accepts. Use only with the `codegen` `fiaccel` `-args` option. Do not pass as an input to a generated MEX function.

Construction

`coder.ArrayType` is an abstract class. You cannot create instances of it directly. You can create `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, and `coder.StructType` objects that derive from this class.

Properties

ClassName

Class of values in this set

SizeVector

The upper-bound size of arrays in this set.

VariableDims

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`coder.Type` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `coder.Type` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `codegen` | `fiaccel`

Superclasses

Type

Purpose

Represent set containing one MATLAB value

Description

Use a `coder.Constant` object to define values that should be treated as constant during code generation. Use only with the `codegenfiaccel -args` options. Do not pass as an input to a generated MEX function.

Construction

`const_type=coder.Constant(v)` creates a `coder.Constant` type from the value `v`.

`const_type=coder.newtype('constant', v)` creates a `coder.Constant` type from the value `v`.

Input Arguments

v

Constant value used to construct the type.

Properties

Value

The actual value of the constant.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Generate code for a MATLAB function, `fcn`, specialized to the case where the input is exactly `42`.

```
k = coder.Constant(42);
```

Create a new constant type for use in code generation.

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

See Also

coder.Type | coder.newtype | coder.Type | coder.newtype | codegen | fiaccel

Purpose

Specify structure name in generated code

Syntax

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

Description

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

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

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

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

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

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

Tips

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

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

- To use `coder.cstructname` on arrays, use single indexing. For example, you cannot do `coder.cstructname(x(1,2))`. Instead, use single indexing, for example `coder.cstructname(x(n))`.
- Use of `coder.cstructname` with global variables is not supported.
- If you use `coder.cstructname` on an array, it sets the name of the base type of the array not the name of the array. Therefore, you cannot use `coder.cstructname` on the base element and then on the array. For example, the following code does not work because the second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previous `coder.cstructname`, `myStructName`.

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

- If you are using custom structure types, specify the name of the header file that includes the external definition of the structure using the `HeaderFile` input argument.
- If you have an Embedded Coder® license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. To take advantage of target-specific function implementations that require data to be aligned, use the `Alignment` input argument.

Input Arguments

structName

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

structType

`coder.StructType` object.

structVar

Structure variable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Alignment'

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary so it will not be matched by CRL functions that require alignment.

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

Default: -1

'HeaderFile'

Name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`. Specify the path to the file using the `codegen -I` option or the **Additional include directories** parameter on the MATLAB Coder **Project Settings** dialog box **Custom Code** tab.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty string.

Examples

Apply `coder.cstructname` to Top-Level Inputs

Generate code for a MATLAB function that takes structure inputs.

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

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

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

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

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

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

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

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

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

Assign a Name to a Structure and Pass it to a Function

Assign the name `MyStruct` to the structure `structVar` and pass the structure to a C function `use_struct`.

1 Create a C header file `use_struct.h` for a function `use_struct` that takes a parameter of type `MyStruct`. Define a structure of type `MyStruct` in the header file.

```
#include <tmwtypes.h>

typedef struct MyStruct
{
    real_T s1;
    real_T s2;
} MyStruct;

void use_struct(struct MyStruct *my_struct);
```

2 Write the C function `use_struct.c`.

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include "use_struct.h"

void use_struct(struct MyStruct *my_struct)
{
  real_T x = my_struct->s1;
  real_T y = my_struct->s2;
}
```

3 Write a MATLAB compliant function `m_use_struct` that declares a structure, assigns the name `MyStruct` to it, and then calls the C function `use_struct` using `coder.ceval`.

```
function m_use_struct   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Declare a MATLAB structure
structVar.s1 = 1;
structVar.s2 = 2;

% Assign the name MyStruct to the structure variable.
% extern indicates this is an externally defined
% structure.
coder.cstructname(structVar, 'MyStruct', 'extern');

% Call the C function use_struct. The type of structVar
% matches the signature of use_struct.
% Use coder.rref to pass the the variable structVar by
% reference as a read-only input to the external C
% function use_struct
coder.ceval('use_struct', coder.rref(structVar));
```

4 Generate C library code for function `m_use_struct`, passing `use_struct.h` to include the structure definition.

```
codegen -config:lib m_use_struct use_struct.c use_struct.h
```

`codegen` generates C code in the default folder `codegen\lib\m_use_struct`. The generated header file `m_use_struct_types.h` in this folder does not contain a definition of the structure `MyStruct` because `MyStruct` is an external type.

Create a `coder.StructType` Object

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

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

T =

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

See Also

`coder.ceval` | `coder.rref` | `codegen` | `fiaccel` | `coder` | `coder.StructType` | `coder.StructType`

How To

- “Structures”
- “Structures”

coder.EnumType

Superclasses ArrayType

Purpose Represent set of MATLAB enumerations

Description Specifies the set of MATLAB enumerations that the generated code should accept. Use only with the `codegenfiaccel -args` options. Do not pass as an input to a generated MEX function.

Construction `enum_type = coder.typeof(enum_value)` creates a `coder.EnumType` object representing a set of enumeration values of class (`enum_value`).

`enum_type = coder.typeof(enum_value, sz, variable_dims)` returns a modified copy of `coder.typeof(enum_value)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` do not change. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed; the unbounded dimensions are variable size. When `variable_dims` is a scalar, it applies to bounded dimensions that are not `1` or `0` (which are fixed).

`enum_type = coder.newtype(enum_name,sz,variable_dims)` creates a `coder.EnumType` object that has variable size with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. If you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to bounded dimensions that are not `1` or `0` (which are fixed).

Input Arguments

enum_value

Enumeration value defined in a file on the MATLAB path.

sz

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

variable_dims

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

Default: false(size(sz)) | sz==Inf for `coder.newtype`

enum_name

Name of a numeration defined in a file on the MATLAB path.

Properties

ClassName

Class of values in the set.

SizeVector

The upper-bound size of arrays in the set.

VariableDims

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `coder.EnumType` object using a value from an existing MATLAB enumeration.

1 Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

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

```
    end
end
```

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

```
t = coder.typeof(MyColors.red);
```

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

1 Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

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

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

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

See Also

`coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.newtype` | `coder.resize` | `coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.newtype` | `coder.resize` | `codegen` | `fiaccel`

How To

- "Enumerated Data"
- "Enumerated Data"

Purpose

Declare extrinsic function or functions

Syntax

```
coder.extrinsic('function_name');
```

```
coder.extrinsic('function_name_1', ...  ,
'function_name_n');
```

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

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

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

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

Arguments

function_name
function_name_1, ... , *function_name_n*

Declares *function_name* or *function_name_1* through *function_name_n* as extrinsic functions.

–sync:on

function_name or *function_name_1* through *function_name_n*.

By default, this option enables synchronization of global data between MATLAB and MEX functions at MEX function entry and exit, and before and after extrinsic calls. Use this option for maximum consistency between MATLAB and the MEX functions. If most extrinsic calls do not modify global data, but a few do, you can turn off synchronization before and after extrinsic calls. To do so, change the global synchronization mode to `At MEX-function entry and exit`. (To learn how, see “How to Synchronize Global Data” in the MATLAB Coder documentation.)

Use the *–sync:on* option to turn on synchronization for extrinsic calls that *do* modify global data. For more information, see “Synchronizing Global Data with MATLAB” in the MATLAB Coder documentation.

–sync:off

Disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions, *function_name* or *function_name_1* through *function_name_n*.

If most extrinsic calls modify global data, but a few do not, you can use the *–sync:off* option to turn off synchronization for the extrinsic calls that *do not* modify global data. For more information, see "Synchronizing Global Data with MATLAB".

Description

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

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

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

Tips

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

  During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable.

Provided that there is no change to the output, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, a compiler error is issued from MATLAB.

Examples

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

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
%  and displays the triangle as a patch object.

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

create_plot(a, b, color);

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

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

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

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

See Also

`coder.ceval` | `coder.ceval` | `coder.screener` | `coder.screener` | `coder.screener`

How To

- “Controlling Synchronization for Extrinsic Function Calls”
- “Resolution of Function Calls in MATLAB Generated Code”
- “Resolution of Function Calls in MATLAB Generated Code”

- "Resolution of Function Calls in MATLAB Generated Code"
- "Restrictions on Extrinsic Functions for Code Generation"
- "Restrictions on Extrinsic Functions for Code Generation"
- "Restrictions on Extrinsic Functions for Code Generation"

Superclasses

ArrayType

Purpose

Represent set of MATLAB fixed-point arrays

Description

Specifies the set of fixed-point array values that the generated code should accept. Use only with the `codegen` `fiaccel -args` options. Do not pass as an input to the generated MEX function.

Construction

`t=coder.typeof(v)` creates a `coder.FiType` object representing a set of fixed-point values whose properties are based on the fixed-point input `v`.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` do not change. If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not `1` or `0` (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When you do not specify `variable_dims`, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it applies to the bounded dimensions that are not `1` or `0` (which are fixed).

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.Type` object representing a set of fixed-point values with `numerictype` and additional options specified by one or more Name, Value pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

Input Arguments

v

Fixed-point value used to create new `coder.FiType` object.

sz

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

variable_dims

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

Default: false(size(sz)) | sz ==Inf for `coder.newtype`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

complex

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

Default: false

fimath

Specify local `fimath`. If not, uses default `fimath`.

Properties

ClassName

Class of values in the set.

Complex

Indicates whether fixed-point arrays in the set are real (`false`) or complex (`true`).

Fimath

Local `fimath` that the fixed-point arrays in the set use.

NumericType

numerictype that the fixed-point arrays in the set use.

SizeVector

The upper-bound size of arrays in the set.

VariableDims

A vector specifying whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a new fixed-point type `t`.

```
t = coder.typeof(fi(1));
% Returns
% coder.FiType
%   1x1 embedded.fi
%       DataTypeMode:Fixed-point: binary point scaling
%         Signedness:Signed
%         WordLength:16
%     FractionLength:14
```

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

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

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

This new type uses the default `fimath`.

See Also

`coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.resize` | `coder.newtype` | `coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.resize` | `coder.newtype` | `codegen` | `fiaccel`

Purpose

Control inlining in generated code

Syntax

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

Description

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

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

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

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

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

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

Examples

- “Preventing Function Inlining” on page 2-117
- “Using coder.inline In Control Flow Statements” on page 2-118

Preventing Function Inlining

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

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

Using coder.inline In Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

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

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

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

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

y = dividend / divisor;
```

Purpose

Load compile-time constants from MAT-file or ASCII file into caller workspace

Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename,'-mat',var1,...,varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

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

Input Arguments

filename - Name of file

string

Name of file, specified as a string constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

Data Types
`char`

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

string

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

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

Data Types
`char`

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

string

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

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

Data Types
`char`

Output Arguments

S - Loaded variables or data

structure array | m-by-n array

If `filename` is a MAT-file, `S` is a structure array.

If `filename` is an ASCII file, `S` is an m-by-n array of type `double`. m is the number of lines in the file and n is the number of values on a line.

Examples

Load compile-time constants from MAT-file

Generate code for a function `edgeDetect1` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect1` uses `coder.load` to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];

save('sobel.mat','k');
```

Write the function `edgeDetect1`.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k','same');
```

```
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect1`.

```
codegen -report -config cfg edgeDetect1
```

`codegen` generates C code in the `codegen\lib\edgeDetect1` folder.

Load compile-time constants from ASCII file

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
save('sobel.dat', 'k', '-ascii');
```

Write the function `edgeDetect2`.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = coder.load('sobel.dat');
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

`codegen` generates C code in the `codegen\lib\edgeDetect2` folder.

Limitations

- `coder.load` does not support loading objects.
- Arguments to `coder.load` must be compile-time constant strings.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S[i] = coder.load('myFile.mat')` is not allowed.
- You cannot use `save` to save workspace data to a file inside a function intended for code generation. The code generation software does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save workspace data to a file.

Tips

- `coder.load` loads data at compile time, not at run time. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function `load` to load run-time values.
- If the MAT-file contains unsupported constructs, use `coder.load(filename,var1,...,varN)` to load only the supported constructs.
- If you generate code in a MATLAB Coder project, the code generation software practices incremental code generation for the `coder.load` function. When the MAT-file or ASCII file used by `coder.load` changes, the software rebuilds the code.

See Also

`matfile | regexp | save`

Concepts

- "Regular Expressions"

coder.mexconfig

Purpose

Code acceleration configuration object

Syntax

`config_obj = coder.mexconfig`

Description

`config_obj = coder.mexconfig` creates a `coder.MexConfig` code generation configuration object for use with `fiaccel`, which generates a MEX function.

Output Arguments

config_obj

Code generation configuration object for use when generating MEX functions using `fiaccel`.

Examples

Create a configuration object to disable run-time checks

```
cfg = coder.mexconfig
% Turn off Integrity Checks, Extrinsic Calls,
% and Responsiveness Checks
cfg.IntegrityChecks = false;
cfg.ExtrinsicCalls = false;
cfg.ResponsivenessChecks = false;
% Use fiaccel to generate a MEX function for file foo.m
fiaccel -config cfg foo
```

See Also

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

Purpose

Code acceleration configuration object for use with `fiaccel`

Description

A `coder.MexConfig` object contains all the configuration parameters that the `fiaccel` function uses when accelerating fixed-point code via a generated MEX function. To use this object, first create it using the lowercase `coder.mexconfig` function and then, pass it to the `fiaccel` function using the `-config` option.

Construction

`cfg = coder.mexconfig` creates a `coder.MexConfig` object, `cfg`, for `fiaccel` MEX function generation.

Properties

ConstantFoldingTimeout

Maximum number of constant folder instructions

Specify, as a positive integer, the maximum number of instructions to be executed by the constant folder.

Default: 10000

DynamicMemoryAllocation

Dynamic memory allocation for variable-size data

By default, when this property is set to `'Threshold'` , dynamic memory allocation is enabled for all variable-size arrays whose size is greater than `DynamicMemoryAllocationThreshold` and `fiaccel` allocates memory for this variable-size data dynamically on the heap. Set this property to `'Off'` to allocate memory statically on the stack. Set it to`'AllVariableSizeArrays'` to allocate memory for all variable-size arrays dynamically on the heap . You must use dynamic memory allocation for all unbounded variable-size data.

This property, `DynamicMemoryAllocation`, is enabled only when `EnableVariableSizing` is `true`. When you set `DynamicMemoryAllocation` to `` `Threshold' ``, it enables the `DynamicMemoryAllocationThreshold` property.

Default: `Threshold`

DynamicMemoryAllocationThreshold

Memory allocation threshold

Specify the integer size of the threshold for variable-size arrays above which `fiaccel` allocates memory on the heap.

Default: `65536`

EchoExpressions

Show results of code not terminated with semicolons

Set this property to `true` to have the results of code instructions that do not terminate with a semicolon appear in the MATLAB Command Window. If you set this property to `false`, code results do not appear in the MATLAB Command Window.

Default: `true`

EnableDebugging

Compile generated code in debug mode

Set this property to `true` to compile the generated code in debug mode. Set this property to `false` to compile the code in normal mode.

Default: `false`

EnableVariableSizing

Variable-sized arrays support

Set this property to `true` to enable support for variable-sized arrays and to enable the `DynamicMemoryAllocation` property. If you set this property to `false`, variable-sized arrays are not supported.

Default: `true`

ExtrinsicCalls

Extrinsic function calls

An extrinsic function is a function on the MATLAB path that the generated code dispatches to MATLAB software for execution. `fiaccel` does not compile or generate code for extrinsic functions. Set this property to `true` to have `fiaccel` generate code for the call to a MATLAB function, but not generate the function's internal code. Set this property to `false` to have `fiaccel` ignore the extrinsic function and not generate code for the call to the MATLAB function. If the extrinsic function affects the output of `fiaccel`, a compiler error occurs.

`ExtrinsicCalls` affects how MEX functions built by `fiaccel` generate random numbers when using the MATLAB `rand`, `randi`, and `randn` functions. If extrinsic calls are enabled, the generated mex function uses the MATLAB global random number stream to generate random numbers. If extrinsic calls are not enabled, the MEX function built with `fiaccel` uses a self-contained random number generator.

Default: `true`

GenerateReport

Code generation report

Set this property to `true` to create an HTML code generation report. Set this property to `false` to not create the report.

Default: `false`

GlobalDataSyncMethod

MEX function global data synchronization with MATLAB global workspace

Set this property to `SyncAlways` so synchronize global data at MEX function entry and exit and for all extrinsic calls to ensure maximum consistency between MATLAB and the generated MEX function. If the extrinsic calls do not affect global data, use this option in conjunction with the `coder.extrinsic -sync:off` option to turn off synchronization for these calls to maximize performance.

If you set this property to `SyncAtEntryAndExits`, global data is synchronized only at MEX function entry and exit. If your code contains extrinsic calls, but only a few affect global data, use this option in conjunction with the `coder.extrinsic -sync:on` option to turn on synchronization for these calls to maximize performance.

If you set this property to `NoSync`, no synchronization occurs. Ensure that your MEX function does not interact with MATLAB globals before disabling synchronization otherwise inconsistencies between MATLAB and the MEX function might occur.

Default: `SyncAlways`

InlineStackLimit

Stack size for inlined functions

Specify, as a positive integer, the stack size limit on inlined functions.

Default: `4000`

InlineThreshold

Maximum size of functions to be inlined

Specify, as a positive integer, the maximum size of functions to be inlined.

Default: `10`

InlineThresholdMax

Maximum size of functions after inlining

Specify, as a positive integer, the maximum size of functions after inlining.

Default: `200`

IntegrityChecks

Memory integrity

Sset this property to `true` to detect any violations of memory integrity in code generated for MATLAB. When a violation is detected, execution stops and a diagnostic message displays. Set this property to `false` to disable both memory integrity checks and the runtime stack.

Default: `true`

LaunchReport

Code generation report display

Set this property to `true` to open the HTML code generation report automatically when code generation completes. Set this property to `false` to disable displaying the report automatically. This property applies only if you set the `GenerateReport` property to `true`.

Default: true

ResponsivenessChecks

Responsiveness checks

Set this property to `true` to turn on responsiveness checks. Set this property to `false` to disable responsiveness checks.

Default: `true`

SaturateOnIntegerOverflow

Integer overflow action

Overflows saturate to either the minimum or maximum value that the data type can represent. Set this property to `true` to have overflows saturate. Set this property to `false` to have overflows wrap to the appropriate value representable by the data type.

Default: `true`

StackUsageMax

Maximum stack usage per application

Specify, as a positive integer, the maximum stack usage per application in bytes. Set a limit that is lower than the available stack size. Otherwise, a runtime stack overflow might occur. Overflows are detected and reported by the C compiler, not by `fiaccel`.

Default: `200000`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Use the lowercase coder.mexconfig function to create a coder.MexConfig configuration object. Set this object to disable run-time checks.

```
cfg = coder.mexconfig
% Turn off Integrity Checks, Extrinsic Calls,
% and Responsiveness Checks
cfg.IntegrityChecks = false;
cfg.ExtrinsicCalls = false;
cfg.ResponsivenessChecks = false;
% Use fiaccel to generate a MEX function for file foo.m
fiaccel -config cfg foo
```

See Also

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

coder.newtype

Purpose Create a new `coder.Type` object

Syntax

```
t=coder.newtype(numeric_class, sz, variable_dims)
t=coder.newtype(numeric_class, sz, variable_dims, Name,
   Value)
t=coder.newtype('constant', value)
t=coder.newtype('struct', struct_fields, sz, variable_dims)
t=coder.newtype('embedded.fi', numerictype, sz,
   variable_dims, Name, Value)
t=coder.newtype(enum_value, sz, variable_dims)
```

Description

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

`t=coder.newtype(numeric_class, sz, variable_dims)` creates a `coder.Type` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `variable_dims` is not specified, the dimensions of the type are fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to dimensions of the type that are not `1` or `0`, which are fixed.

`t=coder.newtype(numeric_class, sz, variable_dims, Name, Value)` creates a `coder.Type` object with additional options specified by one or more Name, Value pair arguments.

`t=coder.newtype('constant', value)` creates a `coder.Constant` object representing a single value. Use this type to specify a value that should be treated as a constant in the generated code.

`t=coder.newtype('struct', struct_fields, sz, variable_dims)` creates a `coder.StructType` object for an array of structures of the given `sz` and `variable_dims` information with the same fields as the scalar structure `struct_fields`.

`t=coder.newtype('embedded.fi', numerictype, sz, variable_dims, Name, Value)` creates a `coder.FiType` object representing a set of fixed-point values with `numerictype` and additional options specified by one or more Name, Value pair arguments.

`t=coder.newtype(enum_value, sz, variable_dims)` creates a `coder.Type` object representing a set of enumeration values of class `enum_value`.

Input Arguments

numeric_class

Class of the set of values represented by the type object

struct_fields

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

sz

Size vector specifying each dimension of type object

> **Default:** [1 1]

variable_dims

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

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

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'complex'

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

Default: false

'fimath'

Specify local `fimath`. If `fimath` is not specified, uses default fimath values.

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

'sparse'

Set `sparse` to `true` to create a `coder.Type` object representing sparse data. The type must support sparse data.

Not for use with `t`=coder.newtype('embedded.fi', numerictype,`sz`, `variable_dims`, Name, Value)

Default: false

Output Arguments

t

New `coder.Type` object.

Examples

Create a new type for use in code generation.

```
t=coder.newtype('double',[2 3 4],[1 1 0])
% Returns double :2x:3x4
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size

```
coder.newtype('double',[inf,3])
%   returns double:inf x 3
```

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

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

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

Create a new structure type for use in code generation.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
coder.newtype('struct',struct('a',ta,'b',tb))
% returns struct 1x1
%            a: int8 1x1
%            b: double :1x:2
% ':' indicates variable-size dimensions
```

Create a new constant type for use in code generation.

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

Create a `coder.EnumType` object using the name of an existing MATLAB enumeration.

1 Define an enumeration `MyColors`. On the MATLAB path, create a file named 'MyColors' containing:

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

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

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

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

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

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

Alternatives

`coder.typeofcoder.typeof`

See Also

coder.resize | coder.Type | coder.ArrayType | coder.EnumType | coder.FiType | coder.PrimitiveType | coder.StructType | coder.resize | coder.Type | coder.ArrayType | coder.EnumType | coder.FiType | coder.PrimitiveType | coder.StructType | codegen | fiaccel

coder.nullcopy

Purpose

Declare uninitialized variables

Syntax

`X = coder.nullcopy(A)`

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. Preallocates memory for `X` without incurring the overhead of initializing memory.

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

Use With Caution

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

Examples

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

```
function X = foo

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
   if mod(i,2) == 0
      X(i) = i;
   else
      X(i) = 0;
   end
end
```

Using `coder.nullcopy` with `zeros` lets you specify the size of vector `X` without initializing each element to zero.

How To

- "Eliminate Redundant Copies of Variables in Generated Code"
- "Eliminate Redundant Copies of Variables in Generated Code"

Superclasses

ArrayType

Purpose

Represent set of logical, numeric, or char arrays

Description

Specifies the set of logical, numeric, or char values that the generated code should accept. Supported classes are `double`,`single`,`int8`,`uint8`,`int16`,`uint16`,`int32`,`uint32`,`char`, and `logical`. Use only with the `codegenfiaccel -args` option. Do not pass as an input to a generated MEX function.

Construction

`t=coder.typeof(v)` creates a `coder.PrimitiveType` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical or char.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of `coder.typeof(v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is [], the (upper bound) sizes of `v` remain unchanged. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to bounded dimensions that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims)` creates a `coder.PrimitiveType` object representing values of class `numeric_class` with (upper bound) sizes `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype(numeric_class, sz, variable_dims, Name, Value)` creates a `coder.PrimitiveType` object with additional options specified by one or more Name, Value pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as Name1,Value1, ,NameN,ValueN.

Input Arguments

v

Input that is not a coder.Type object

sz

Size for corresponding dimension of type object. Size must be a valid size vector.

Default: [1 1] for coder.newtype

variable_dims

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

Default: false(size(sz)) | sz==Inf for coder.newtype

numeric_class

Class of type object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

complex

Set complex to true to create a coder.PrimitiveType object that can represent complex values. The type must support complex data.

Default: false

sparse

Set `sparse` to `true` to create a `coder.PrimitiveType` object representing sparse data. The type must support sparse data.

Default: false

Properties

ClassName

Class of values in this set

Complex

Indicates whether the values in this set are real (`false`) or complex (`true`)

SizeVector

The upper-bound size of arrays in this set.

Sparse

Indicates whether the values in this set are sparse arrays (`true`)

VariableDims

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `coder.PrimitiveType` object.

```
z = coder.typeof(0,[2 3 4],[1 1 0]) % returns double :2x:3x4
% ':' indicates variable-size dimensions
```

Create a `coder.PrimitiveType` object then call `codegen` to generate a C library for a function `fcn.m` that has one input parameter of this type.

1 Create a `coder.PrimitiveType` object.

```
z = coder.typeof(0,[2 3 4],[1 1 0]) % returns double :2x:3x4
% ':' indicates variable-size dimensions
```

2 Call `codegen` to generate a C library for a MATLAB function `fcn.m` that has one input parameter type `z`.

```
% Use the config:lib option to generate a C library
codegen -config:lib fcn -args {z}
```

See Also

`coder.Type` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `coder.Type` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `codegen` | `fiaccel`

Purpose

Resize a `coder.Type` object

Syntax

```
t_out = coder.resize(t, sz, variable_dims)
t_out = coder.resize(t, sz)
t_out = coder.resize(t,[],variable_dims)
t_out = coder.resize(t, sz, variable_dims, Name, Value)
t_out = coder.resize(t, 'sizelimits', limits)
```

Description

`t_out = coder.resize(t, sz, variable_dims)` returns a modified copy of `coder.Type` `t` with upper-bound size `sz`, and variable dimensions `variable_dims`. If `variable_dims` or `sz` are scalars, they are applied to all dimensions of `t`. By default, `variable_dims` does not apply to dimensions where `sz` is `0` or `1`, which are fixed. Use the 'uniform' option to override this special case. `coder.resize` ignores `variable_dims` for dimensions with size `inf`. These dimensions are always variable size. `t` can be a cell array, in which case, `coder.resize` resizes all elements of the cell array.

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

`t_out = coder.resize(t,[],variable_dims)` changes `t` to have variable dimensions `variable_dims` while leaving the size unchanged.

`t_out = coder.resize(t, sz, variable_dims, Name, Value)` resizes `t` using additional options specified by one or more Name, Value pair arguments.

`t_out = coder.resize(t, 'sizelimits', limits)` resizes `t` with dimensions automatically becoming variable based on the `limits` vector. When the size `S` of a dimension is greater than or equal to the first threshold defined in `limits`, the dimension becomes variable size with upper bound `S`. When the size `S` of a dimension is greater than or equal to the second threshold defined in `limits`, the dimension becomes unbounded variable size.

Input Arguments

limits

Two-element vector (or a scalar-expanded one-element vector) of variable-sizing thresholds. If the size `sz` of a dimension of `t` is greater than or equal to the first threshold, the dimension becomes variable size with upper bound sz. If the size sz of a dimension of t is greater than or equal to the second threshold, the dimension becomes unbounded variable size.

sz

New size for coder.Type object, t_out

t

coder.Type object that you want to resize

variable_dims

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

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'recursive'

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

Default: false

'uniform'

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

Default: false

Output Arguments

t_out

Resized coder.Type object

Examples

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

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

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

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

Resize a structure field

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

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

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

See Also

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

Purpose

Determine if function is suitable for code generation

Syntax

```
coder.screener(fcn)
coder.screener(fcn_1,...,fcn_n )
```

Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function, `fcn`. It identifies unsupported functions and language features, such as recursion, cell arrays, nested functions, and function handles as code generation compliance issues and displays them in a report. If `fcn` calls other functions directly or indirectly that are not MathWorks® functions, `coder.screener` analyzes these functions too. It does not analyze MathWorks functions. `coder.screener` might not detect all code generation issues. Under certain circumstances, it might report false errors.

`coder.screener(fcn_1,...,fcn_n )` analyzes entry-point functions (`fcn_1,...,fcn_n`).

Tips

- Before using `coder.screener`, fix issues identified by the code analyzer.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.
- Because `coder.screener` might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.

Input Arguments

fcn

Name of entry-point MATLAB function that you want to analyze.

fcn_1,...,fcn_n

Comma-separated list of names of entry-point MATLAB functions that you want to analyze.

Examples

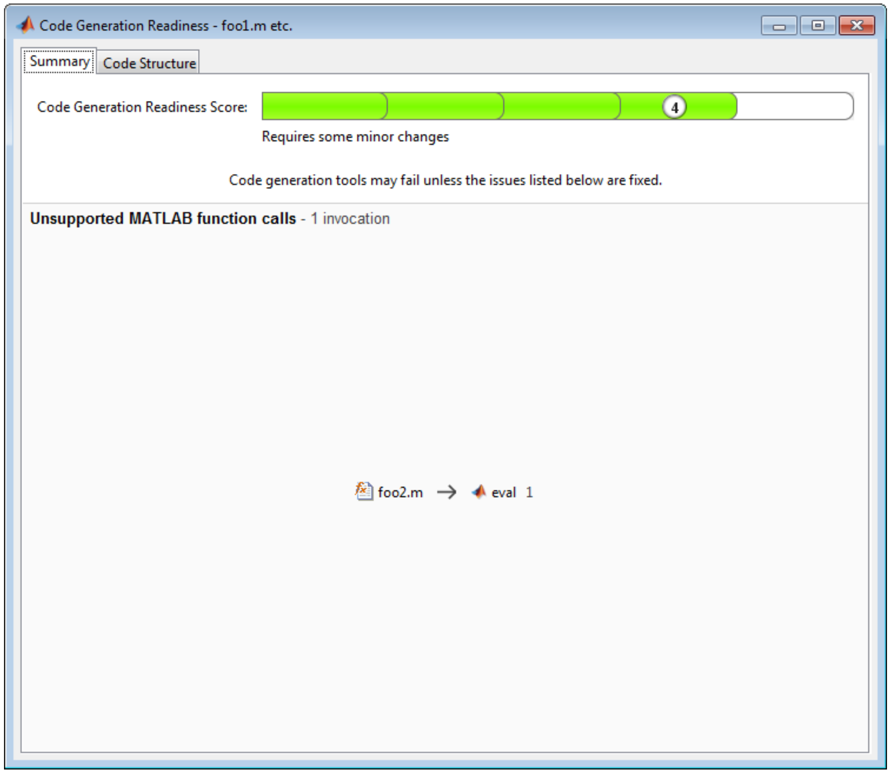
Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, `foo1`, and the function `foo2` that `foo1` calls.

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

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

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

coder.screener('foo1')
```

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



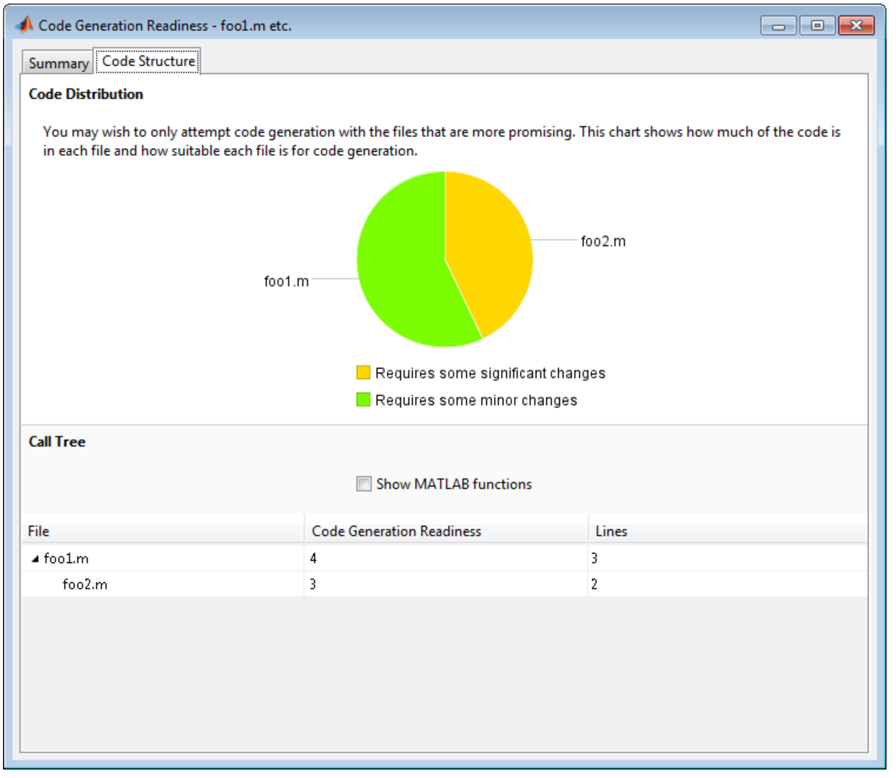
In the report, click the **Code Structure** tab and select **Show MATLAB functions**.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that some significant changes are required.
- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1 to 5. 1 indicates that

significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.

- Displays a call tree.



The report **Summary** tab indicates that `foo2.m` contains one call to the `eval` function which is not supported for code generation. To generate a MEX function for `foo2.m`, modify the code to make the call to `eval` extrinsic.

```
function out = foo2(in)
  coder.extrinsic('eval');
  out = eval(in);
```

```
end
```

Rerun the code generation readiness tool.

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

The report no longer flags that the `eval` function is not supported for code generation. When you generate a MEX function for `foo1`, the code generation software automatically calls out to MATLAB for `eval`. For standalone code generation, it does not generate code for it.

Identify Unsupported Data Types

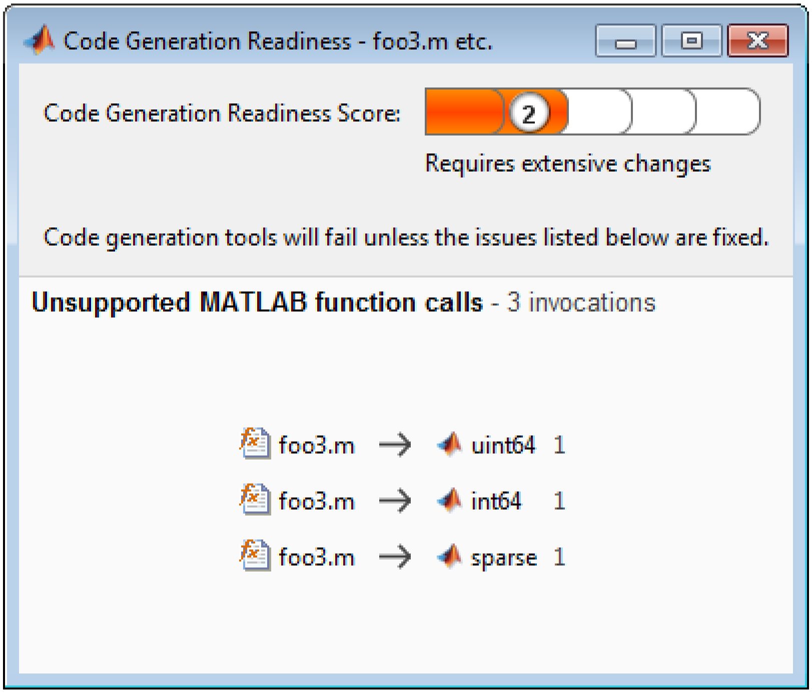
The `coder.screener` function identifies data types that are not supported for code generation.

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

```
function [outInt64,outUint64,outSparse,outCasts] = foo3(inVal)
  outInt64 = int64(inVal);
  outUint64 = uint64(inVal);
  outSparse = sparse(inVal);
  outCasts = sparse(uint64(int64(inVal)));
end
```

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

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



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

Before generating code, you must fix the reported issues.

Determine code generation readiness for multiple entry-point functions

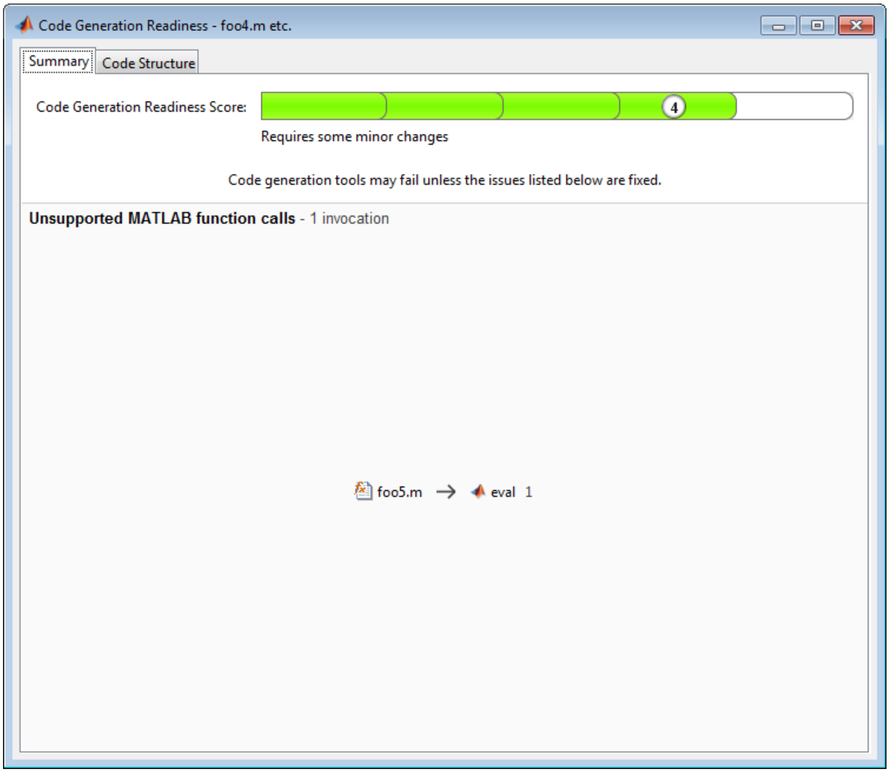
The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks the entry-point functions `foo4` and `foo5`.

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

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

```
end

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

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

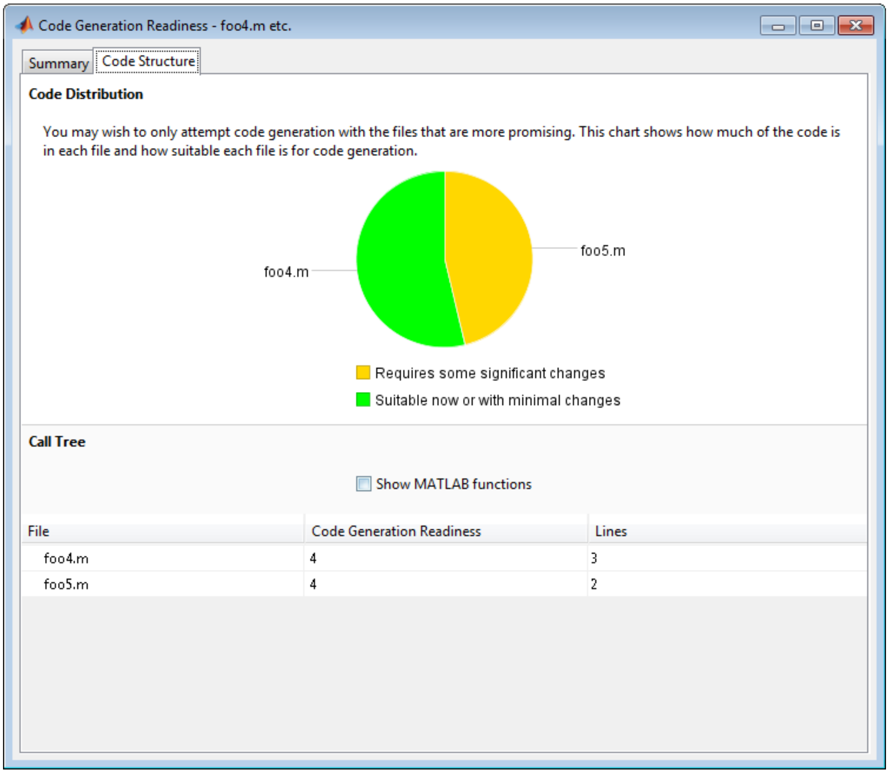
The code generation readiness report opens. It provides a summary of the unsupported MATLAB function calls. The function `foo5` calls one unsupported MATLAB function.



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

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:

- Colors `foo1.m` green to indicate that it is suitable for code generation.
- Colors `foo2.m` yellow to indicate that some significant changes are required.
- Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1 to 5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool cannot detect issues.
- Displays a call tree.



Alternatives

- "Run Code Generation Readiness Tool from the Current Folder Browser""Run the Code Generation Readiness Tool From the Current Folder Browser""Run the Code Generation Readiness Tool From the Current Folder Browser"
- "Run the Code Generation Readiness Tool in a Project".

See Also

`codegen` | `fiaccel`

Concepts

- "MATLAB Language Features Supported for C/C++ Code Generation"
- "Functions Supported for Code Acceleration or Generation"
- "Functions Supported for Code Generation — Alphabetical List"

- “Functions Supported for Code Generation — Categorical List”
- “Functions Supported for Code Generation — Alphabetical List”
- “System Objects Supported for Code Generation”
- “Code Generation Readiness Tool”
- “Code Generation Readiness Tool”
- “Code Generation Readiness Tool”

Superclasses

ArrayType

Purpose

Represent set of MATLAB structure arrays

Description

Specifies the set of structure arrays that the generated code should accept. Use only with the `codegenfiaccel -args` option. Do not pass as an input to a generated MEX function.

Construction

`t=coder.typeof(struct_v)` creates a `coder.StructType` object for a structure with the same fields as the scalar structure `struct_v`.

`t=coder.typeof(struct_v, sz, variable_dims)` returns a modified copy of `coder.typeof(struct_v)` with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `sz` is [], the (upper bound) sizes of `struct_v` remain unchanged. If the `variable_dims` input parameter is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the bounded dimensions that are not `1` or `0` (which are assumed to be fixed).

`t=coder.newtype('struct', struct_v, sz, variable_dims)` creates a `coder.StructType` object for an array of structures with the same fields as the scalar structure `struct_v` and (upper bound) size `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. When `variable_dims` is not specified, the dimensions of the type are assumed to be fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to the dimensions of the type, except if the dimension is `1` or `0`, which is assumed to be fixed.

Input Arguments

struct_v

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

sz

Size vector specifying each dimension of type object.

Default: [1 1] for `coder.newtype`

variable_dims

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

Default: false(size(sz)) `|` sz==Inf for `coder.newtype`

Properties

Alignment

The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on a specific boundary so it will not be matched by CRL functions that require alignment.

`Alignment` must be either `-1` or a power of `2` that is no more than `128`.

ClassName

Class of values in this set.

Extern

Whether the structure type is externally defined.

Fields

A structure giving the `coder.Type` of each field in the structure.

HeaderFile

If the structure type is externally defined, name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`. Specify the path to the file using the `codegen -I` option or the **Additional include directories** parameter on the MATLAB Coder **Project Settings** dialog box **Custom Code** tab.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` option, MATLAB Coder includes that header file exactly at the point where it is required.

Must be a non-empty string.

SizeVector

The upper-bound size of arrays in this set.

VariableDims

A vector used to specify whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
% coder.StructType
%    1x1 struct
%      a:  :3x:5 double
%      b:  3x3  double
% ':' indicates variable-size dimensions
```

Create a coder.StructType object then call codegen to generate a C library for a function fcn.m that has one input parameter of this type

1. Create a new structure type.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
z = coder.newtype('struct',struct('a',ta,'b',tb))
% Returns
% coder.StructType
%   1x1 struct
%      a: 1x1 int8
%      b: :1x:2 double
```

2. Call codegen to generate a C library for a MATLAB function fcn.m that has one input parameter of this type.

```
% Use the -config:lib option to generate a C library
codegen -config:lib fcn -args {z}
```

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

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

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

T =

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

```
    b: 1x1 single
```

2 View the types of the structure fields.

```
T.Fields

ans =

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

See Also

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

coder.target

Purpose

Determine code generation target

Syntax

`[y =] coder.target`

Description

`[y =] coder.target` returns a string representing the code generation target.

String	Description
`''`	Function is executing in MATLAB
`'rtw'`	MATLAB Coder target
`'sfun'`	S-function target (Simulation target)
`'mex'`	MEX-function target
`'hdl'`	Stateflow® HDL Coder target

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target` returns `''`.

Examples

Use `coder.target` to parameterize MATLAB functions that use custom C/C++ code so that they work in MATLAB or generated code.

```
if isempty(coder.target)
    % running in MATLAB
else
    % running in the generated code
end
```

See Also

`coder.ceval`

How To

- "Defining Class Properties for Code Generation"
- "Defining Class Properties for Code Generation"

Purpose

Represent set of MATLAB values

Description

Specifies the set of values that the generated code should accept. Use only with the `codegenfiaccel -args` option. Do not pass as an input to a generated MEX function.

Construction

`coder.Type` is an abstract class, and you cannot create instances of it directly. You can create `coder.Constant`, `coder.EnumType`, `coder.FiType`, `coder.PrimitiveType`, and `coder.StructType` objects that are derived from this class.

Properties

ClassName

Class of values in this set

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`coder.typeof` | `codegen` | `fiaccel` | `coder.newtype` | `coder.ArrayType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.typeof` | `coder.newtype` | `coder.ArrayType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType`

coder.typeof

Purpose

Convert a MATLAB value into its canonical type

Syntax

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

Description

`t=coder.typeof(v)` creates a `coder.Type` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical, char, enumeration, fixed-point array or a struct constructed from the preceding types. Use `coder.typeof` only to specify input parameter types. For example, use it with the `codegen` function `-args` option or in a MATLAB Coder project when you are defining an input type by example. Do not use it in MATLAB code from which you intend to generate code. For example, use it with the `fiaccel` function `-args` option. Do not use it in MATLAB code from which you intend to generate a MEX function.

`t=coder.typeof(v, sz, variable_dims)` returns a modified copy of t=coder.typeof(v) with (upper bound) size specified by `sz` and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is [], the (upper bound) sizes of `v` remain unchanged. If the `variable_dims` input parameter is not specified, the bounded dimensions of the type are fixed. When `variable_dims` is a scalar, it is applied to bounded dimensions or dimensions that are `1` or `0`, which are fixed.

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

Input Arguments

sz

Size vector specifying each dimension of type object

t

`coder.Type` object

v

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

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

variable_dims

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

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

Output Arguments

t

`coder.Type` object

Examples

Create a type for a simple fixed-sized `5x6` matrix of doubles

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

Create a type for a variable-sized matrix of doubles

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

Create a type for a structure with a variable-size field

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
% Returns
```

```
% coder.StructType
%    1x1 struct
%      a:  :3x:5 double
%      b:  3x3  double
% ':' indicates variable-size dimensions
```

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

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

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

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size

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

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

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

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

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

See Also

coder.newtype | coder.resize | coder.newtype | coder.resize | codegen | fiaccel

coder.unroll

Purpose

Copy body of `for`-loop in generated code for each iteration

Syntax

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

Description

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

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

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

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

This function is ignored outside of code generation.

Input Arguments

flag

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

`true`	Unroll the `for`-loop
`false`	Do not unroll the `for`-loop

range

Specifies the bounds of the `for`-loop iteration:

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

Examples

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

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

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

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

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

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

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

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

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

```
static void m_getrand(real_T y[8])
{
  int32_T i0;
  for(i0 = 0; i0 < 8; i0++) {
    y[i0] = 0.0;
  }
  /*  Loop body begins */
  y[2] = m_rand();
  y[4] = m_rand();
  /*  Loop body ends */
}
```

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

```
static void m_b_getrand(real_T y[50])
{
  int32_T i;
```

```
    for(i = 0; i < 50; i++) {
      y[i] = 0.0;
    }
    /*  Loop body begins */
    for(i = 0; i < 50; i += 2) {
      if((i + 1 > 2) && (i + 1 < 48)) {
        y[i] = m_rand();
      }
    }
    /*  Loop body ends */
  }
```

See Also

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

How To

- "Using Logicals in Array Indexing"
- "Unroll for-loops"

coder.varsize

Purpose

Declare variable-size data

Syntax

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

Description

`coder.varsize('`*var*$_1$`', '`*var*$_2$`', ...)` declares one or more variables as variable-size data, allowing subsequent assignments to extend their size. Each `'`*var*$_n$`'` must be a quoted string that represents a variable or structure field. If the structure field is a structure array, use colon (`:`) as the index expression, indicating that elements of the array are variable sized. For example, the expression `coder.varsize('data(:).A')` declares that the field `A` inside each element of `data` is variable sized.

`coder.varsize('`*var*$_1$`', '`*var*$_2$`', ..., `*`ubound`*`)` declares one or more variables as variable-size data with an explicit upper bound specified in *`ubound`*. The argument *`ubound`* must be a constant, integer-valued vector of upper bound sizes for every dimension of each `'`*var*$_n$`'`. If you specify more than one `'`*var*$_n$`'`, each variable must have the same number of dimensions.

`coder.varsize('`*var*$_1$`', '`*var*$_2$`', ..., `*`ubound`*`, `*`dims`*`)` declares one or more variables as variable-sized with an explicit upper bound and a mix of fixed and varying dimensions specified in *`dims`*. The argument *`dims`* is a logical vector, or double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` in *`dims`* have fixed size; dimensions that correspond to ones or `true` vary in size. If you specify more than one variable, each fixed dimension must have the same value across all `'`*var*$_n$`'`.

`coder.varsize('`*var*$_1$`', '`*var*$_2$`', ..., [], `*`dims`*`)` declares one or more variables as variable-sized with a mix of fixed and varying dimensions. The empty vector `[]` means that you do not specify an explicit upper bound.

When you do *not* specify *`ubound`*, the upper bound is computed for each `'`*var*$_n$`'` in generated code.

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

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

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

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

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

Examples

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

1 Write primary function `test_stack` to issue commands for pushing data on and popping data from a stack. Write local function `stack` to execute the push and pop commands.

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

function y = stack(command, varargin)
    persistent data;
    if isempty(data)
        data = ones(1,0);
```

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

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

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

2 Generate a MEX function for `test_stack`:

```
codegen test_stack
```

```
fiaccel test_stack
```

`codegenfiaccel` generates a MEX function in the current folder.

3 Run `test_stack` to get these results:

```
value =
    20

value =
```

```
     19

value =
     18

value =
     17

value =
     16

value =
     15

value =
     14

value =
     13

value =
     12

value =
     11
```

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

Declare a variable-size structure field.

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

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

     d = struct('values', zeros(1,0), 'color', 0);
   ```

```
data = repmat(d, [3 3]);
coder.varsize('data(:).values');

for i = 1:numel(data)
    data(i).color = rand-0.5;
    data(i).values = 1:i;
end

y = 0;
for i = 1:numel(data)
    if data(i).color > 0
        y = y + sum(data(i).values);
    end;
end
```

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

2 Generate a MEX function for `struct_example`:

```
codegen struct_example

fiaccel struct_example
```

3 Run `struct_example`.

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

Alternatives

You can use the `assert` function to constrain an upper bound within a range of values, such as when growing a variable in a loop.

See Also

`assert` | `codegen` | `fiaccel` | `size` | `varargin`

How To

- “Generate Code for Variable-Size Data”
- “Variable-Size Data Definition for Code Generation”

- "Defining Variable-Size Structure Fields"
- "Defining Variable-Size Structure Fields"
- "Compilation Directive %#codegen"
- "Compilation Directive %#codegen"
- "Defining Variable-Size Global Data"
- "Incompatibilities with MATLAB in Variable-Size Support for Code Generation"

comet

Purpose Create 2-D comet plot

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

Purpose Create 3-D comet plot

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

compass

Purpose Plot arrows emanating from origin

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

Purpose

Construct complex `fi` object from real and imaginary parts

Syntax

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

Description

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

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

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

The output `fi` object `c` has the same `numerictype` and `fimath` properties as the input `fi` object `a`.

See Also

`imag` | `real`

coneplot

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

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

Purpose

Complex conjugate of `fi` object

Syntax

```
conj(a)
```

Description

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

When `a` is complex,

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

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

See Also

`complex` | `imag` | `real`

contour

Purpose Create contour graph of matrix

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

Purpose Create 3-D contour plot

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

contourc

Purpose Create two-level contour plot computation

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

Purpose Create filled 2-D contour plot

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

Purpose

Convolution and polynomial multiplication of `fi` objects

Syntax

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

Description

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

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

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

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

- If either `a` or `b` has a local `fimath` object, `conv` uses that `fimath` object to compute intermediate quantities and determine the `numerictype` properties of `c`.
- If neither `a` nor `b` have an attached fimath, `conv` uses the default fimath to compute intermediate quantities and determine the `numerictype` properties of `c`.

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

The output `fi` object `c` always uses the default fimath.

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

Examples

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

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

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

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

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

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

The operation results in a signed fi object y with a word length of 36 bits and a fraction length of 31 bits. The default fimath properties associated with the inputs determine the numerictype of the output. The output does not have a local fimath.

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

cordicabs

Purpose

CORDIC-based absolute value

Syntax

```
r = cordicabs(c)
r = cordicabs(c,niters)
r = cordicabs(c,niters,'ScaleOutput',b)
r = cordicabs(c,'ScaleOutput',b)
```

Description

`r = cordicabs(c)` returns the magnitude of the complex elements of C.

`r = cordicabs(c,niters)` performs `niters` iterations of the algorithm.

`r = cordicabs(c,niters,'ScaleOutput',b)` specifies both the number of iterations and, depending on the Boolean value of `b`, whether to scale the output by the inverse CORDIC gain value.

`r = cordicabs(c,'ScaleOutput',b)` scales the output depending on the Boolean value of `b`.

Input Arguments

c

`c` is a vector of complex values.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`).

'ScaleOutput'

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: `true`

Output Arguments

r

`r` contains the magnitude values of the complex input values. If the inputs are fixed-point values, `r` is also fixed point (and is always signed, with binary point scaling). All input values must have the same data type. If the inputs are signed, then the word length of `r` is the input word length + 2. If the inputs are unsigned, then the word length of `r` is the input word length + 3. The fraction length of `r` is always the same as the fraction length of the inputs.

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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Compare `cordicabs` and `abs` of double values.

```
dblValues = complex(rand(5,4),rand(5,4));
r_dbl_ref = abs(dblValues)
r_dbl_cdc = cordicabs(dblValues)
```

Compute absolute values of fixed-point inputs.

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

References

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

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

See Also

`cordiccart2pol` | `cordicangle` | `abs`

Purpose

CORDIC-based phase angle

Syntax

```
theta = cordicangle(c)
theta = cordicangle(c),niters)
```

Description

`theta = cordicangle(c)` returns the phase angles, in radians, of matrix `c`, which contains complex elements.

`theta = cordicangle(c),niters)` performs `niters` iterations of the algorithm.

Input Arguments

c

Matrix of complex numbers

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Output Arguments

theta

`theta` contains the polar coordinates angle values, which are in the range [–pi, pi] radians. If `x` and `y` are floating-point, then `theta` has the same data type as `x` and `y`. Otherwise, `theta` is a fixed-point data type with the same word length as `x` and `y` and with a best-precision fraction length for the [-pi, pi] range.

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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Phase angle for double-valued input and for fixed-point-valued input.

```
dblRandomVals = complex(rand(5,4), rand(5,4));
theta_dbl_ref = angle(dblRandomVals);
theta_dbl_cdc = cordicangle(dblRandomVals)
fxpRandomVals = fi(dblRandomVals);
theta_fxp_cdc = cordicangle(fxpRandomVals)

theta_dbl_cdc =

    1.0422    1.0987    1.2536    0.6122
    0.5893    0.8874    0.3580    0.2020
    0.5840    0.2113    0.8933    0.6355
    0.7212    0.2074    0.9820    0.8110
    1.3640    0.3288    1.4434    1.1291

theta_fxp_cdc =

    1.0422    1.0989    1.2534    0.6123
    0.5894    0.8872    0.3579    0.2019
    0.5840    0.2112    0.8931    0.6357
    0.7212    0.2075    0.9819    0.8110
    1.3640    0.3289    1.4434    1.1289
```

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

References

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

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

See Also

`cordicatan2` | `cordiccart2pol` | `cordicabs` | `angle`

cordicatan2

Purpose

CORDIC-based four quadrant inverse tangent

Syntax

```
theta = cordicatan2(y,x)
theta = cordicatan2(y,x,niters)
```

Description

`theta = cordicatan2(y,x)` computes the four quadrant arctangent of `y` and `x` using a "CORDIC" on page 2-229 algorithm approximation.

`theta = cordicatan2(y,x,niters)` performs `niters` iterations of the algorithm.

Input Arguments

y,x

`y,x` are Cartesian coordinates. `y` and `x` must be the same size. If they are not the same size, at least one value must be a scalar value. Both `y` and `x` must have the same data type.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `y` or `x`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Output Arguments

theta

`theta` is the arctangent value, which is in the range [-pi, pi] radians. If `y` and `x` are floating-point numbers, then `theta` has the same data type as `y` and `x`. Otherwise, `theta` is a fixed-point data type with the same word length as `y` and `x` and with a best-precision fraction length for the [-pi, pi] range.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is 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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Floating-point CORDIC arctangent calculation.

```
theta_cdat2_float = cordicatan2(0.5,-0.5)

theta_cdat2_float =
    2.3562
```

Fixed- point CORDIC arctangent calculation.

```
theta_cdat2_fixpt = cordicatan2(fi(0.5,1,16,15),fi(-0.5,1,16,15));

theta_cdat2_fixpt =
    2.3562

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

References

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

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

See Also

`atan2`

Purpose

CORDIC-based approximation of Cartesian-to-polar conversion

Syntax

```
[theta,r] = cordiccart2pol(x,y)
[theta,r] = cordiccart2pol(x,y, niters)
[theta,r] = cordiccart2pol(x,y, niters,'ScaleOutput',b)
[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)
```

Description

`[theta,r] = cordiccart2pol(x,y)` using a CORDIC algorithm approximation, returns the polar coordinates, angle `theta` and radius `r`, of the Cartesian coordinates, `x` and `y`.

`[theta,r] = cordiccart2pol(x,y, niters)` performs `niters` iterations of the algorithm.

`[theta,r] = cordiccart2pol(x,y, niters,'ScaleOutput',b)` specifies both the number of iterations and, depending on the Boolean value of `b`, whether to scale the `r` output by the inverse CORDIC gain value.

`[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)` scales the `r` output by the inverse CORDIC gain value, depending on the Boolean value of `b`.

Input Arguments

x,y

`x,y` are Cartesian coordinates. `x` and `y` must be the same size. If they are not the same size, at least one value must be a scalar value. Both `x` and `y` must have the same data type.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`).

'ScaleOutput'

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: `true`

Output Arguments

theta

`theta` contains the polar coordinates angle values, which are in the range [–pi, pi] radians. If `x` and `y` are floating-point, then `theta` has the same data type as `x` and `y`. Otherwise, `theta` is a fixed-point data type with the same word length as `x` and `y` and with a best-precision fraction length for the [-pi, pi] range.

r

`r` contains the polar coordinates radius magnitude values. `r` is real-valued and can be a scalar value or have the same dimensions as `theta` If the inputs `x,y` are fixed-point values, `r` is also fixed point (and is always signed, with binary point scaling). Both `x,y` input values must have the same data type. If the inputs are signed, then the word length of `r` is the input word length + 2. If the inputs are unsigned, then the word length of `r` is the input word length + 3. The fraction length of `r` is always the same as the fraction length of the `x,y` inputs.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is 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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Convert fixed-point Cartesian coordinates to polar coordinates.

```
[thPos,r]=cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(0.5,16,15))

thPos =

    0.5881  0.7854  1.1072  1.5708  2.0344  2.3562  2.5535  2.6780

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

r =

    0.9014  0.7071  0.5591  0.5000  0.5591  0.7071  0.9014  1.1180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15

[thNeg,r]=...
```

```
  cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(-0.5,16,15))

thNeg =

 -0.5881 -0.7854 -1.1072 -1.5708 -2.0344 -2.3562 -2.5535 -2.6780

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

r =

 0.9014  0.7071  0.5591  0.5000  0.5591  0.7071  0.9014  1.1180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15
```

References

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

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

See Also

`cordicatan2` | `cordicpol2cart` | `cart2pol`

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 2-229 algorithm approximation. `y` contains the approximated complex result.

Input Arguments

theta

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

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y

`y` is the approximated complex result of the `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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

The following example illustrates the effect of the number of iterations on the result of the `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, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

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

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 2-229 algorithm approximation.

Input Arguments

theta

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

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y

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

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is 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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

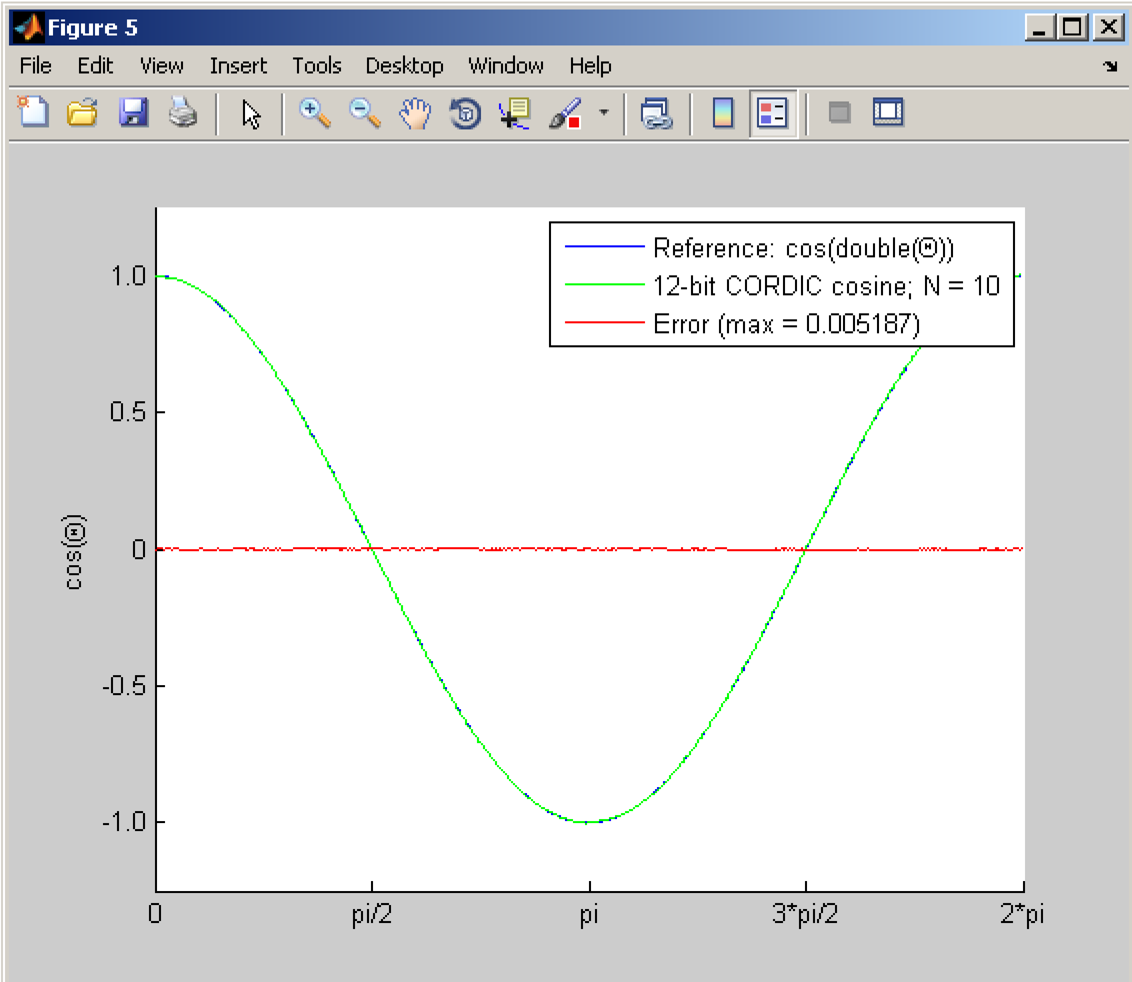
Examples

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

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

% Use 12-bit quantized inputs and vary the number
% of iterations from 2 to 10.
% Compare the  fixed-point CORDIC results to the
% double-precision trig function results.
for niters = 2:2:10
    cdcCosTh  = cordiccos(thRadFxp,  niters);
    errCdcRef = cosThRef - double(cdcCosTh);
    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, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

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

See Also

`cordiccexp` | `cordicsin` | `cordicsincos`

Tutorials

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

Purpose

CORDIC-based approximation of polar-to-Cartesian conversion

Syntax

```
[x,y] = cordicpol2cart(theta,r)
[x,y] = cordicpol2cart(theta,r,niters)
[x,y] = cordicpol2cart(theta,r,Name,Value)
[x,y] = cordicpol2cart(theta,r,niters,Name,Value)
```

Description

`[x,y] = cordicpol2cart(theta,r)` returns the Cartesian xy coordinates of `r*` e^(j`*theta`) using a CORDIC algorithm approximation.

`[x,y] = cordicpol2cart(theta,r,niters)` performs `niters` iterations of the algorithm.

`[x,y] = cordicpol2cart(theta,r,Name,Value)` scales the output depending on the Boolean value of `b`.

`[x,y] = cordicpol2cart(theta,r,niters,Name,Value)` specifies both the number of iterations and `Name,Value` pair for whether to scale the output.

Input Arguments

theta

`theta` can be a signed or unsigned scalar, vector, matrix, or *N*-dimensional array containing the angle values in radians. All values of `theta` must be in the range [–2π 2π).

r

`r` contains the input magnitude values and can be a scalar or have the same dimensions as `theta`. `r` must be real valued.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of r or one less than the word length of theta, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Name-Value Pair Arguments

Optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ('').

'ScaleOutput'

ScaleOutput is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set ScaleOutput to true or 1, the output values are multiplied by a constant, which incurs extra computations. If you set ScaleOutput to false or 0, the output is not scaled.

Default: true

Output Arguments

[x,y]

[x,y] contains the approximated Cartesian coordinates. When the input r is floating point, the output [x,y] has the same data type as the input.

When the input r is a *signed* integer or fixed point data type, the outputs [x,y] are signed fi objects. These fi objects have word lengths that are two bits larger than that of r. Their fraction lengths are the same as the fraction length of r.

When the input r is an *unsigned* integer or fixed point, the outputs [x,y] are signed fi objects. These fi objects have word lengths are three bits larger than that of r. Their fraction lengths are the same as the fraction length of r.

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is 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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Run the following code, and evaluate the accuracy of the CORDIC-based Polar-to-Cartesian conversion.

```
wrdLn = 16;
theta = fi(pi/3, 1, wrdLn);
u     = fi( 2.0, 1, wrdLn);

fprintf('\n\nNITERS\tX\t\t ERROR\t LSBs\t\tY\t\t ERROR\t LSBs\n');
fprintf('------\t-------\t ------\t ----\t\t-------\t ------\t ----\n');
for niters = 1:(wrdLn - 1)
 [x_ref, y_ref] = pol2cart(double(theta),double(u));
 [x_fi,  y_fi] = cordicpol2cart(theta, u, niters);
 x_dbl  = double(x_fi);
 y_dbl  = double(y_fi);
 x_err  = abs(x_dbl - x_ref);
 y_err  = abs(y_dbl - y_ref);
 fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
   niters,x_dbl,x_err,(x_err * pow2(x_fi.FractionLength)),...
   y_dbl,y_err,(y_err * pow2(y_fi.FractionLength)));
end
fprintf('\n');
```

```
NITERS  X        ERROR    LSBs      Y       ERROR   LSBs
------ -------  ------   ----  -------  ------  ----
   1    1.4142   0.4142   3392.8  1.4142   0.3178   2603.8
   2    0.6324   0.3676   3011.2  1.8973   0.1653   1354.2
   3    1.0737   0.0737   603.8   1.6873   0.0448   366.8
   4    0.8561   0.1440   1179.2  1.8074   0.0753   617.2
   5    0.9672   0.0329   269.2   1.7505   0.0185   151.2
   6    1.0214   0.0213   174.8   1.7195   0.0126   102.8
   7    0.9944   0.0056   46.2    1.7351   0.0031   25.2
   8    1.0079   0.0079   64.8    1.7274   0.0046   37.8
   9    1.0011   0.0011   8.8     1.7313   0.0007   5.8
   10   0.9978   0.0022   18.2    1.7333   0.0012   10.2
   11   0.9994   0.0006   5.2     1.7323   0.0003   2.2
   12   1.0002   0.0002   1.8     1.7318   0.0002   1.8
   13   0.9999   0.0002   1.2     1.7321   0.0000   0.2
   14   0.9996   0.0004   3.2     1.7321   0.0000   0.2
   15   0.9998   0.0003   2.2     1.7321   0.0000   0.2
```

References

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

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

See Also

`cordicrotate` | `cordicsincos` | `pol2cart`

Purpose

Rotate input using CORDIC-based approximation

Syntax

```
v = cordicrotate(theta,u)
v = cordicrotate(theta,u,niters)
v = cordicrotate(theta,u,Name,Value)
v = cordicrotate(theta,u,niters,Name,Value)
```

Description

`v = cordicrotate(theta,u)` rotates the input `u` by `theta` using a CORDIC algorithm approximation. The function returns the result of `u .* e^(j*theta)`.

`v = cordicrotate(theta,u,niters)` performs `niters` iterations of the algorithm.

`v = cordicrotate(theta,u,Name,Value)` scales the output depending on the Boolean value, `b`.

`v = cordicrotate(theta,u,niters,Name,Value)` specifies both the number of iterations and the `Name,Value` pair for whether to scale the output.

Input Arguments

theta

`theta` can be a signed or unsigned scalar, vector, matrix, or *N*-dimensional array containing the angle values in radians. All values of `theta` must be in the range [–2π 2π).

u

`u` can be a signed or unsigned scalar value or have the same dimensions as `theta`. `u` can be real or complex valued.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `u` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`).

'ScaleOutput'

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: true

Output Arguments

v

`v` contains the approximated result of the CORDIC rotation algorithm. When the input `u` is floating point, the output `v` has the same data type as the input.

When the input `u` is a *signed* integer or fixed point data type, the output `v` is a signed `fi` object. This `fi` object has a word length that is two bits larger than that of `u`. Its fraction length is the same as the fraction length of `u`.

When the input u is an *unsigned* integer or fixed point, the output `v` is a signed `fi` object. This `fi` object has a word length that is three bits larger than that of `u`. Its fraction length is the same as the fraction length of `u`.

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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

Run the following code, and evaluate the accuracy of the CORDIC-based complex rotation.

```
wrdLn = 16;
theta = fi(-pi/3, 1, wrdLn);
u     = fi(0.25 - 7.1i, 1, wrdLn);
uTeTh = double(u) .* exp(1i * double(theta));

fprintf('\n\nNITERS\tReal\t ERROR\t LSBs\t\tImag\tERROR\tLSBs\n');
fprintf('------\t-------\t ------\t ----\t\t-------\t------\t----\n');
for niters = 1:(wrdLn - 1)
 v_fi   = cordicrotate(theta, u, niters);
 v_dbl  = double(v_fi);
 x_err  = abs(real(v_dbl) - real(uTeTh));
  y_err  = abs(imag(v_dbl) - imag(uTeTh));
 fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
   niters, real(v_dbl),x_err,(x_err * pow2(v_fi.FractionLength)), ...
   imag(v_dbl),y_err, (y_err * pow2(v_fi.FractionLength)));
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS  Real     ERROR    LSBs     Imag     ERROR    LSBs
------ -------  ------   ----  -------  ------  ------
1      -4.8438   1.1800   4833.5   -5.1973  1.4306  5859.8
2      -6.6567   0.6329   2592.5   -2.4824  1.2842  5260.2
3      -5.8560   0.1678   687.5    -4.0227  0.2560  1048.8
4      -6.3098   0.2860   1171.5   -3.2649  0.5018  2055.2
5      -6.0935   0.0697   285.5    -3.6528  0.1138  466.2
6      -5.9766   0.0472   193.5    -3.8413  0.0746  305.8
7      -6.0359   0.0121   49.5     -3.7476  0.0191  78.2
8      -6.0061   0.0177   72.5     -3.7947  0.0280  114.8
9      -6.0210   0.0028   11.5     -3.7710  0.0043  17.8
10     -6.0286   0.0048   19.5     -3.7590  0.0076  31.2
11     -6.0247   0.0009   3.5      -3.7651  0.0015  6.2
12     -6.0227   0.0011   4.5      -3.7683  0.0017  6.8
13     -6.0237   0.0001   0.5      -3.7666  0.0001  0.2
14     -6.0242   0.0004   1.5      -3.7656  0.0010  4.2
15     -6.0239   0.0001   0.5      -3.7661  0.0005  2.2
```

References

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

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

See Also

`cordicpol2cart` | `cordiccexp`

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 2-229 algorithm approximation.

Input Arguments

theta

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

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y

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

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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

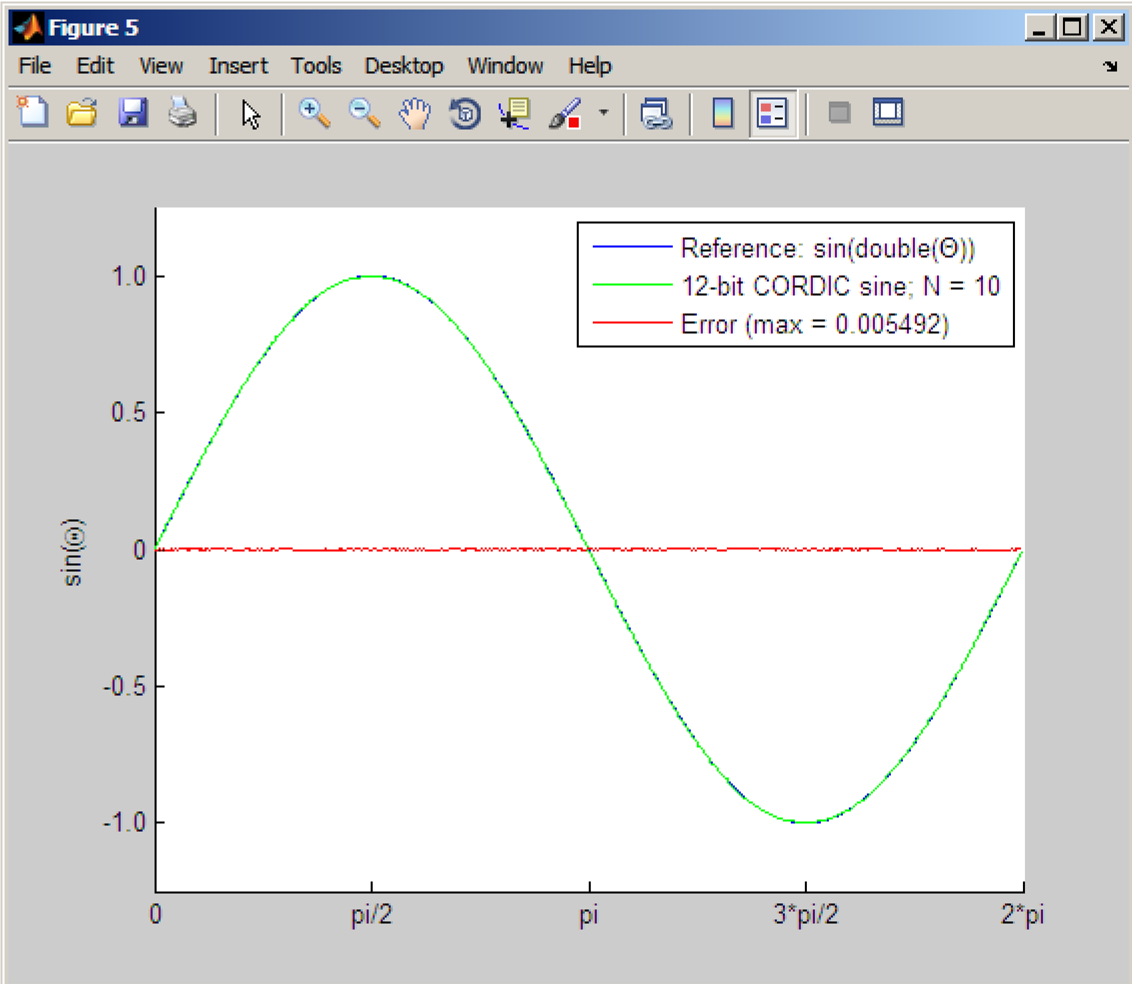
Examples

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

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

See Also

`cordiccexp` | `cordiccos` | `cordicsincos`

Tutorials

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

cordicsincos

Purpose

CORDIC-based approximation of sine and cosine

Syntax

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

Description

`[y, x] = cordicsincos(theta,niters)` computes the sine and cosine of `theta` using a "CORDIC" on page 2-229 algorithm approximation. *y* contains the approximated sine result, and *x* contains the approximated cosine result.

Input Arguments

theta

`theta` can be a signed or unsigned scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range [–2π 2π). When `theta` has a fixed-point data type, it must be signed.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y

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

x

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

Definitions

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is 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. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

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

Examples

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

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

```
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS    Y (SIN)  ERROR   LSBs   X (COS)  ERROR   LSBs
------    -------  ------  ----   -------  ------  ----
1         0.7031   0.2968  19.0   0.7031   0.7105  45.5
2         0.9375   0.0625  4.0    0.3125   0.3198  20.5
3         0.9844   0.0156  1.0    0.0938   0.1011  6.5
4         0.9844   0.0156  1.0   -0.0156   0.0083  0.5
5         1.0000   0.0000  0.0    0.0312   0.0386  2.5
6         1.0000   0.0000  0.0    0.0000   0.0073  0.5
7         1.0000   0.0000  0.0    0.0156   0.0230  1.5
```

References

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

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

See Also

`cordiccexp` | `cordiccos` | `cordicsin`

Tutorials

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

Purpose

Cosine of fi object

Syntax

```
y = cos(theta)
```

Description

y = cos(theta) returns the cosine of fi input theta using a table-lookup algorithm.

Input Arguments

theta

theta can be a real-valued, signed or unsigned scalar, vector, matrix, or N-dimensional array containing the fixed-point angle values in radians. Valid data types of theta are:

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

Output Arguments

y

y is the cosine of theta. y is a signed, fixed-point number in the range [-1,1]. It has a 16-bit word length and 15-bit fraction length (numerictype(1,16,15)). This cosine calculation is accurate only to within the top 16 most-significant bits of the input.

Definitions

Cosine

The cosine of angle Θ is defined as

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

Examples

Calculate the cosine of fixed-point input values.

```
theta = fi([0,pi/4,pi/3,pi/2,(2*pi)/3,(3*pi)/4,pi])
```

```
theta =

       0  0.7854  1.0472  1.5708  2.0944  2.3562  3.1416

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

y = cos(theta)

y =

    1.0000  0.7072  0.4999  0.0001 -0.4999  -0.7070  -1.0000

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

Algorithms

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

1. Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
2. Perform a modulo 2π, so the input is in the range [0,2π) radians.
3. Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
4. Use the 8 most-significant bits to obtain the first value from the table.
5. Use the next-greater table value as the second value.
6. Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

See Also

cos | angle | sin | atan2

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` | `storedInteger` | `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
```

Algorithms

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

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

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

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

See Also

`denormalmin` | `eps` | `quantizer`

Purpose

Smallest denormalized quantized number for `quantizer` object

Syntax

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

Description

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

Examples

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

x =

    0.0625
```

Algorithms

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

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

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

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

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

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

See Also

`denormalmax` | `eps` | `quantizer`

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.

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 2-240.

`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 2-240.

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

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 =

11001100110011001100110011001100110011001100110011001100
110011001100110011001100
```

Notice that when you use the `divide` function, the quotient is calculated to the full 80 bits, regardless of the precision of `a` and `b`. Thus, the `fi` object `c` represents 1/10 more precisely than IEEE standard double-precision floating-point number can.

With 1000 bits of precision,

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

```
c.bin

ans =

1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
1100110011001100110011001100110011001100110011001100
11001100110011001100110011001100110011001100
```

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`

end

Purpose Last index of array

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

Purpose

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

Syntax

```
eps(obj)
```

Description

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

See Also

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

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`

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`.

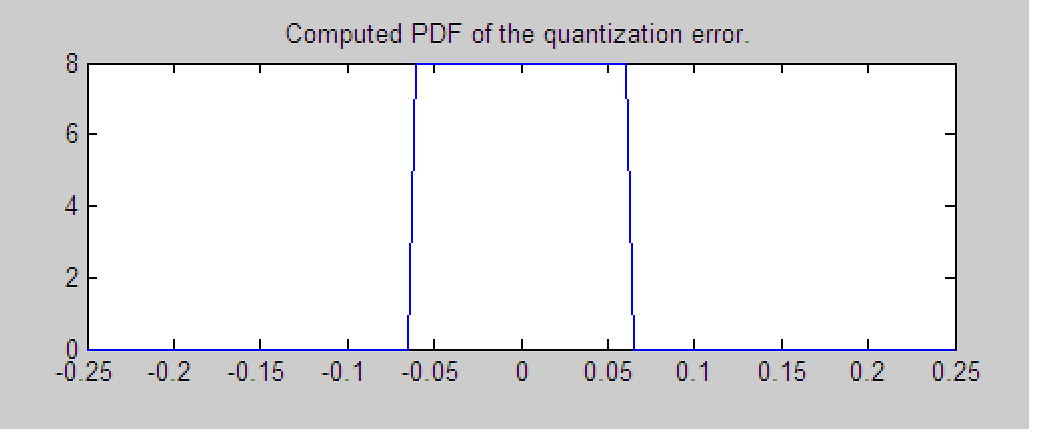
`f = errpdf(q,x)` returns the probability density function `f` evaluated at the values in vector `x`.

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

Examples

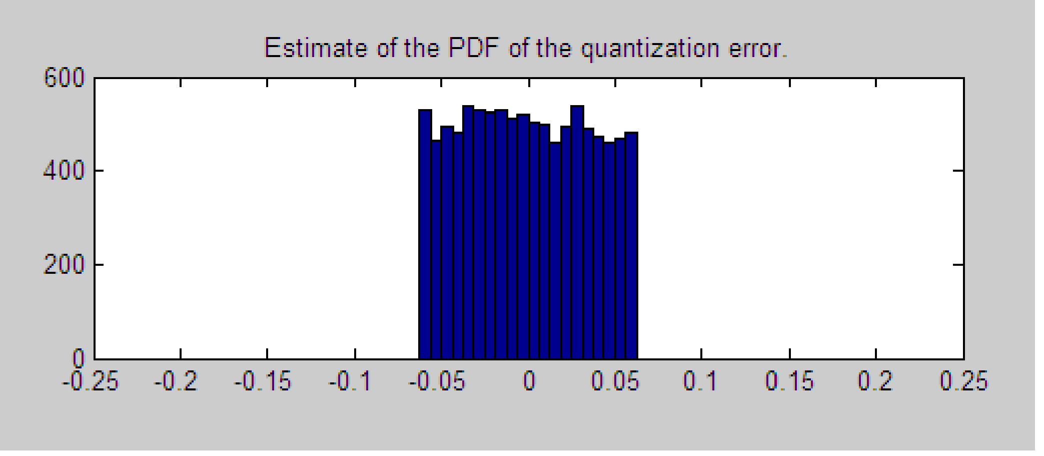
```
q = quantizer('nearest',[4 3]);
[f,x] = errpdf(q);
subplot(211)
plot(x,f)
title('Computed PDF of the quantization error.')
```

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
```

Algorithms

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
```

Algorithms

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

See Also

`eps` | `exponentbias` | `exponentmax` | `exponentmin`

exponentmax

Purpose Maximum exponent for `quantizer` object

Syntax

```
exponentmax(q)
```

Description

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

Examples

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

emax =

        1023
```

Algorithms

For floating-point `quantizer` objects,

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

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

See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmin`

exponentmin

Purpose

Minimum exponent for `quantizer` object

Syntax

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

Description

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

Examples

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

emin =

      -1022
```

Algorithms

For floating-point `quantizer` objects,

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

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

See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmax`

Purpose Easy-to-use contour plotter

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

Purpose Easy-to-use filled contour plotter

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

ezmesh

Purpose Easy-to-use 3-D mesh plotter

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

Purpose Easy-to-use function plotter

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

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

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

Purpose Easy-to-use polar coordinate plotter

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

ezsurf

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

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

Purpose Easy-to-use combination surface/contour plotter

Description Refer to the MATLAB `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 "numerictype Object Construction" 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 "fimath Object Construction" 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 2-271
- "fimath Properties" on page 2-272
- "numerictype Properties" on page 2-273

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

Data Properties

The data properties of a `fi` object are always writable.

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

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

fimath Properties

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

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

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

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

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

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

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

These properties are described in detail in "fimath Object Properties" 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 "View Fixed-Point Data".

For examples of casting, see "Cast fi Objects".

To see the number of elements in a `fi` array, use the `numberofelements` function, instead of the `numel` function.

Example 1

This example 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, 'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap')

a =
    3.1415

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

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

Example 6

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

```
a = fi(pi, 'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap')
a.fimath = []

a =
    3.1415

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

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

a =
    3.1415

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

`fi` object `a` now has no local fimath. To reassign it a local `fimath` object, use dot notation:

```
a.ProductMode = 'KeepLSB'
```

```
a =
    3.1415

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

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

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

Example 7

This example shows how to use a fi as an indexing argument.

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

aFi = fi(3,0,4,1);
aIdx = subsindex(aFi)
aIdx =
     2

y = x(aFi)
y =
     8
```

Example 8

This example shows how to use a fi in a switch statement. You can use a fi as the switch condition and as one or more of the cases in the switch expression.

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

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

>> y = test_switch(1,2.0)

y =
     3
```

See Also

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

Concepts

- “fi Object Functions”

Purpose

Accelerate fixed-point code

Syntax

```
fiaccel -options fcn
```

Description

`fiaccel -options fcn` translates the MATLAB file `fcn.m` to a MEX function, which accelerates fixed-point code. To use `fiaccel`, your code must meet one of these requirements:

- The top-level function has no inputs or outputs, and the code uses `fi`
- The top-level function has an output or a non-constant input, and at least one output or input is a `fi`.
- The top-level function has at least one input or output containing a built-in integer class (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, or `uint64`), and the code uses `fi`.

Note If your top-level file is on a path that contains Unicode characters, code generation might not be able to find the file.

Input Arguments

fcn

MATLAB function from which to generate a MEX function. *fcn* must be suitable for code generation. For information on code generation, see "Code Acceleration and Code Generation from MATLAB"

options

Choice of compiler options. `fiaccel` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

`-args` *`example_inputs`*	Define the size, class, and complexity of all MATLAB function inputs. Use the values in *`example_inputs`* to define these properties. *`example_inputs`* must be a cell array that specifies the same number and order of inputs as the MATLAB function.
`-config` *`config_object`*	Specify MEX generation parameters, based on *`config_object`*, defined as a MATLAB variable using `coder.mexconfig`. For example: `cfg = coder.mexconfig;`
`-d` *`out_folder`*	Store generated files in the absolute or relative path specified by *`out_folder`*. If the folder specified by *`out_folder`* does not exist, `fiaccel` creates it for you. If you do not specify the folder location, `fiaccel` generates files in the default folder: `fiaccel/`*`mex`*`/`*`fcn`*`.` *`fcn`* is the name of the MATLAB function specified at the command line. The function does not support the following characters in folder names: asterisk (`*`), question-mark (`?`), dollar (`$`), and pound (`#`).

`-g`	Compiles the MEX function in debug mode, with optimization turned off. If not specified, `fiaccel` generates the MEX function in optimized mode.
`-global` *`global_values`*	Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `fiaccel`. If you do not provide initial values for global variables using the `-global` option, `fiaccel` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `fiaccel` generates an error. The generated MEX code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ.

`-histogram`	Compute the log2 histogram for all named, intermediate and expression values. A histogram column appears in the code generation report table.
`-I` *`include_path`*	Add *`include_path`* to the beginning of the code generation path. `fiaccel` searches the code generation path *first* when converting MATLAB code to MEX code.
`-launchreport`	Generate and open a code generation report. If you do not specify this option, `fiaccel` generates a report only if error or warning messages occur or you specify the `-report` option.
`-o` *`output_file_name`*	Generate the MEX function with the base name *`output_file_name`* plus a platform-specific extension. *`output_file_name`* can be a file name or include an existing path. If you do not specify an output file name, the base name is *`fcn_mex`*, which allows you to run the original MATLAB function and the MEX function and compare the results.

`-O` *`optimization_option`*	Optimize generated MEX code, based on the value of *`optimization_option`*: • `enable:inline` — Enable function inlining • `disable:inline` — Disable function inlining If not specified, `fiaccel` uses inlining for optimization.
`-report`	Generate a code generation report. If you do not specify this option, `fiaccel` generates a report only if error or warning messages occur or you specify the `-launchreport` option.
`-?`	Display help for `fiaccel` command.

Examples

Create a test file and compute the moving average. Then, use `fiaccel` to accelerate the code and compare.

```
function avg = test_moving_average(x)
%#codegen
if nargin < 1,
    x = fi(rand(100,1),1,16,15);
end
z = fi(zeros(10,1),1,16,15);
avg = x;
for k = 1:length(x)
    [avg(k),z] = moving_average(x(k),z);
end

function [avg,z] = moving_average(x,z)
%#codegen
```

```
if nargin < 2,
    z = fi(zeros(10,1),1,16,15);
end
z(2:end) = z(1:end-1);   % Update buffer
z(1) = x;                % Add new value
avg = mean(z);          % Compute moving average

% Use fiaccel to create a MEX function and
% accelerate the code
x = fi(rand(100,1),1,16,15);
fiaccel test_moving_average -args {x} -report

% Compare the non-accelerated and accelerated code.
x = fi(rand(100,1),1,16,15);

% Non-compiled version
tic,avg = test_moving_average(x);toc
% Compiled version
tic,avg = test_moving_average_mex(x);toc
```

See Also

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

filter

Purpose

One-dimensional digital filter of `fi` objects

Syntax

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

Description

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

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

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

Tips

- The `filter` function only supports FIR filters. In the general filter representation, *b*/*a*, the denominator, *a*, of an FIR filter is the scalar 1, which is the second input of this function.
- The `numerictype` of `b` can be different than the `numerictype` of `x`.
- If you want to specify initial conditions, but do not know what `numerictype` to use, first try filtering your data without initial conditions. You can do so by specifying `[]` for the input `zi`. After performing the filtering operation, you have the `numerictype` of `y` and `zf` (if requested). Because the `numerictype` of `zi` must match that of `y` and `zf`, you now know the `numerictype` to use for the initial conditions.

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)

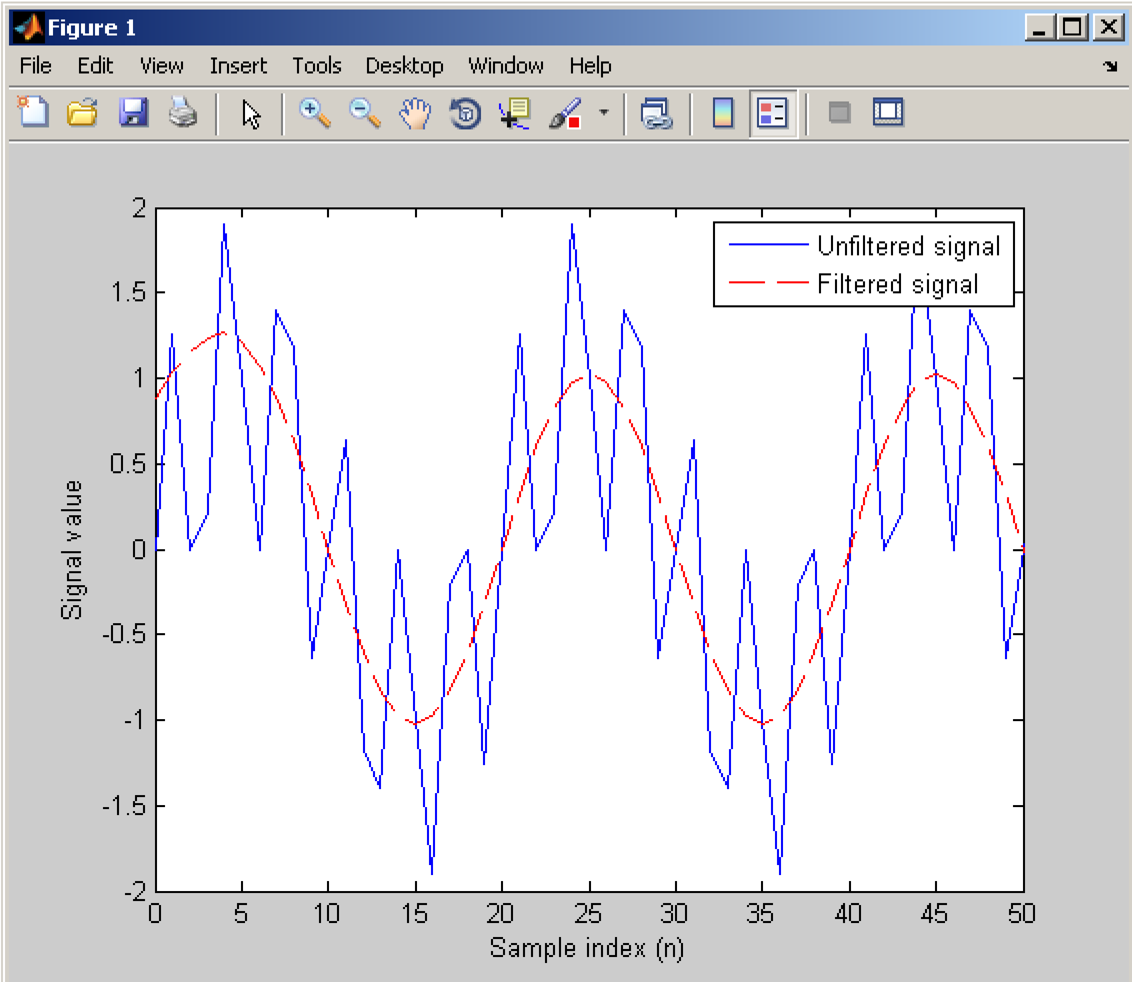
The filter order is the number of states (delays) of the filter, and is equal to *`L`*`-1`.

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.



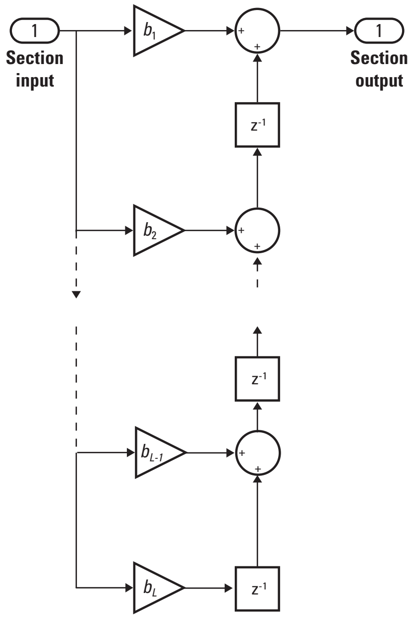
Algorithms

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

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

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`

Purpose

Set fixed-point math settings

Syntax

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

Description

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

- `F = fimath` creates a `fimath` object with default fimath property settings:

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

- `F = fimath(...'PropertyName',PropertyValue...)` allows you to set the attributes of a `fimath` object using property name/property value pairs. All property names that you do not specify in the constructor use default values.

The properties of the `fimath` object are listed below. These properties are described in detail in "fimath Object Properties" 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`.

- `OverflowAction` — Action to take on overflow
- `ProductBias` — Bias of the product data type
- `ProductFixedExponent` — Fixed exponent of the product data type
- `ProductFractionLength` — Fraction length, in bits, of the product data type

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

Examples

Example 1

Type

```
F = fimath
```

to create a default fimath object. It has these settings:

```
F =

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

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 action to `Saturate` and the rounding method to `Convergent`,

```
F = fimath('OverflowAction','Saturate',...
           'RoundingMethod','Convergent')

F =

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

See Also

`fi` | `fipref` | `numerictype` | `quantizer`

Related Examples

- “fimath Object Construction”

Concepts

- “fimath Object Properties”
- “fimath Properties Usage for Fixed-Point Arithmetic”

Purpose Set fixed-point preferences

Syntax

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

Description You can use the `fipref` constructor function in the following ways:

- `P = fipref` creates a default `fipref` object.
- `P = fipref(...'PropertyName',PropertyValue...)` allows you to set the attributes of a object using property name/property value pairs.

The properties of the `fipref` object are listed below. These properties are described in detail in "fipref Object Properties" on page 1-12.

- `FimathDisplay` — Display options for the local `fimath` attributes of `fi` objects. When `fi` objects to not have a local fimath, their `fimath` attributes are never displayed.
- `DataTypeOverride` — Data type override options.
- `DataTypeOverrideAppliesTo`— Data type override setting applicability.
- `LoggingMode` — Logging options for operations performed on `fi` objects.
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object.
- `NumberDisplay` — Display options for the value of a `fi` object.

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

See "View Fixed-Point Data" for more information on the display preferences used for most code examples in the documentation.

Examples

Example 1

Type

```
P = fipref
```

to create a default `fipref` object.

```
P =

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

Example 2

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

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

P =

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

See Also

`fi` | `fimath` | `numerictype` | `quantizer` | `savefipref`

Related Examples

- “fipref Object Construction”

Concepts

- "fipref Object Properties"

Purpose

Round toward zero

Syntax

```
y = fix(a)
```

Description

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

`y` and `a` have the same `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`

fixed.aggregateType

Purpose

Compute aggregate numerictype

Syntax

`aggNT = fixed.aggregateType(A,B)`

Description

`aggNT = fixed.aggregateType(A,B)` computes the smallest binary point scaled numerictype that is able to represent both the full range and precision of inputs `A` and `B`.

Input Arguments

A

An integer, binary point scaled fixed-point `fi` object, or numerictype object.

B

An integer, binary point scaled fixed-point `fi` object, or numerictype object.

Output Arguments

aggNT

A numerictype object.

Examples

Compute the aggregate numerictype of two numerictype objects.

```
% can represent range [-4,4) and precision 2^-13
a_nt = numerictype(true,16,13);
% can represent range [-2,2) and precision 2^-16
b_nt = numerictype(true,18,16);

% can represent range [-4,4) and precision 2^-16
aggNT = fixed.aggregateType(a_nt,b_nt)
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 19
        FractionLength: 16
```

Compute the aggregate numerictype of two `fi` objects.

```
% Unsigned, WordLength: 16, FractionLength: 14
a_fi = ufi(pi,16);
% Signed, WordLength: 24, FractionLength: 21
b_fi = sfi(-pi,24);

% Signed, WordLength: 24, FractionLength: 21
aggNT = fixed.aggregateType(a_fi,b_fi)
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 21
```

Compute the aggregate numerictype of a `fi` object and an integer.

```
% Unsigned, WordLength: 16, FractionLength: 14
% can represent range [0,3] and precision 2^-14
a_fi = ufi(pi,16);
% Unsigned, WordLength: 8, FractionLength: 0
% can represent range [0,255] and precision 2^0
cInt = uint8(0);

% Unsigned with WordLength: 14+8, FractionLength: 14
% can represent range [0,255] and precision 2^-14
aggNT = fixed.aggregateType(a_fi,cInt)
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 22
```

```
FractionLength: 14
```

See Also

numerictype | fi

Purpose Quantize fixed-point numbers

Syntax

```
q = fixed.Quantizer
q = fixed.Quantizer(nt,rm,oa)
q = fixed.Quantizer(s,wl,fl,rm,oa)
q = fixed.Quantizer(Name,Value)
```

Description

`q = fixed.Quantizer` creates a quantizer `q` that quantizes fixed-point (`fi`) numbers using default fixed-point settings.

`q = fixed.Quantizer(nt,rm,oa)` uses the `numerictype` (`nt`) object information and the `RoundingMethod` (`rm`) and `OverflowAction` (`oa`) properties.

The `numerictype`, rounding method, and overflow action apply only during the quantization. The resulting, quantized `q` does not have any fimath attached to it.

`q = fixed.Quantizer(s,wl,fl,rm,oa)` uses the `Signed` (`s`), `WordLength` (`wl`), `FractionLength` (`fl`), `RoundingMethod` (`rm`), and `OverflowAction` (`oa`) properties.

`q = fixed.Quantizer(Name,Value)` creates a quantizer with the property options specified by one or more `Name,Value` pair arguments. You separate pairs of `Name,Value` arguments with commas. `Name` is the argument name, and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

Tips

- Use `y = quantize(q,x)` to quantize input array `x` using the fixed-point settings of quantizer `q`. `x` can be any fixed-point number fi, except a Boolean value. If `x` is a scaled double, the `x` and `y` data will be the same, but `y` will have fixed-point settings. If `x` is a double or single then `y` = `x`. This functionality lets you share the same code for both floating-point data types and fi objects when quantizers are present.
- Use `n = numerictype(q)` to get a numerictype for the current settings of quantizer `q`.

- Use `clone(q)` to create a quantizer object with the same property values as `q`.
- If you use a `fixed.quantizer` in code generation, note that it is a handle object and must be declared as persistent.

Input Arguments

nt

Binary-point, scaled numerictype object or slope-bias scaled, fixed-point numerictype object. If your `fixed.Quantizer` uses a numerictype object that has either a `Signedness` of `Auto` or unspecified `Scaling`, an error occurs.

rm

Rounding method to apply to the output data. Valid rounding methods are: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, and `Zero`. The associated property name is `RoundingMethod`.

Default: `Floor`

oa

Overflow action to take in case of data overflow. Valid overflow actions are `Saturate` and `Wrap`. The associated property name is `OverflowAction`.

Default: `Wrap`

s

Logical value, `true` or `false`, indicating whether the output is signed or unsigned, respectively. The associated property name is `Signed`.

Default: `true`

wl

Word length (number of bits) of the output data. The associated property name is `WordLength`.

Default: 16

fl

Fraction length of the output data. The associated property name is `FractionLength`.

Default: 15

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Bias

The bias is part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number.

Default: 0

FixedExponent

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

The exponent of a fixed-point number is equal to the negative of the fraction length. `FixedExponent` must be an integer.

Default: -15

FractionLength

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

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

Default: 15

OverflowAction

Action to take in case of data overflow. Valid overflow actions are `Saturate` and `Wrap`. .

Default: `Wrap`

RoundingMethod

Rounding method to apply to the output data. Valid rounding methods are: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, and `Zero`.

Default: `Floor`

Scaling

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

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

Default: `BinaryPoint`

Signed

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

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

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

Default: `true`

Signedness

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

- `Signed` — signed
- `Unsigned` — unsigned

Default: `Signed`

Slope

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

Default: 2^-15

SlopeAdjustmentFactor

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

`SlopeAdjustmentFactor` must be greater than or equal to 1 and less than 2.

Default: 1

WordLength

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

Default: 16

Output Arguments

q

Quantizer that quantizes fi input numbers

Definitions

Fixed-point numbers

Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

Examples

Use `fixed.Quantizer` to reduce the word length that results from adding two fixed-point numbers.

```
q = fixed.Quantizer;
x1 = fi(0.1,1,16,15);
x2 = fi(0.8,1,16,15);
y  = quantize(q,x1+x2);
```

Use `fixed.Quantizer` object to change a binary point scaled fixed-point fi to a slope-bias scaled fixed-point fi

```
qsb = fixed.Quantizer(numerictype(1,7,1.6,0.2),...
  'Round','Saturate');
ysb = quantize(qsb,fi(pi,1,16,13));
```

See Also

`fi` | `numerictype` | `quantizer`

How To

- "Set numerictype Object Properties"

fixpt_instrument_purge

Purpose

Remove corrupt fixed-point instrumentation from model

Note `fixpt_instrument_purge` will be removed in a future release.

Syntax

```
fixpt_instrument_purge
fixpt_instrument_purge(modelName, interactive)
```

Description

The `fixpt_instrument_purge` script finds and removes fixed-point instrumentation from a model left by the Fixed-Point Tool and the fixed-point autoscaling script. The Fixed-Point Tool and the fixed-point autoscaling script each add callbacks to a model. For example, the Fixed-Point Tool appends commands to model-level callbacks. These callbacks make the Fixed-Point Tool respond to simulation events. Similarly, the autoscaling script adds instrumentation to some parameter values that gathers information required by the script.

Normally, these types of instrumentation are automatically removed from a model. The Fixed-Point Tool removes its instrumentation when the model is closed. The autoscaling script removes its instrumentation shortly after it is added. However, there are cases where abnormal termination of a model leaves fixed-point instrumentation behind. The purpose of `fixpt_instrument_purge` is to find and remove fixed-point instrumentation left over from abnormal termination.

`fixpt_instrument_purge(modelName, interactive)` removes instrumentation from model `modelName`. `interactive` is `true` by default, which prompts you to make each change. When `interactive` is set to `false`, all found instrumentation is automatically removed from the model.

See Also

`autofixexp` | `fxptdlg`

Purpose Flip array along specified dimension

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

Purpose Flip matrix left to right

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

Purpose Flip matrix up to down

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

Purpose

Round toward negative infinity

Syntax

```
y = floor(a)
```

Description

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

`y` and `a` have the same `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`.

Algorithms For floating-point `quantizer` objects, $f = w - e - 1$, where w is the word length and e is the exponent length.

For fixed-point `quantizer` objects, f is part of the format $[w\ f]$.

See Also `fi` | `numerictype` | `quantizer` | `wordlength`

Purpose

Determine whether real-world value of one `fi` object is greater than or equal to another

Syntax

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

Description

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

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

See Also

`eq` | `gt` | `le` | `lt` | `ne`

Purpose

Property values of object

Syntax

```
value = get(o,'propertyname')
structure = get(o)
```

Description

`value = get(o,'propertyname')` returns the property value of the property `'propertyname'` for the object `o`. If you replace the string `'propertyname'` by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(o)` returns a structure containing the properties and states of object `o`.

`o` can be a `fi`, `fimath`, `fipref`, `numerictype`, or `quantizer` object.

See Also

`set`

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

Purpose Plot set of nodes using adjacency matrix

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

Purpose

Determine whether real-world value of one `fi` object is greater than another

Syntax

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

Description

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

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

See Also

`eq` | `ge` | `le` | `lt` | `ne`

Purpose Hankel matrix

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

hdlram

Purpose

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

Description

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

Construction

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

`H = hdlram(Name,Value)` creates a single, simple dual, or dual port hdlram System object, `H`, with each specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. See "Properties" on page 2-328 for the list of available property names.

Properties

RAMType

Type of RAM to be created

Default: `Single port`

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

`Single Port`	Create a single port RAM, with 3 inputs and 1 output. Inputs: • Write Data • Write address • Write enable Outputs: Read/Write Data
`Simple dual port`	Create a simple dual-port RAM, with 4 inputs and 1 output. Inputs:

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

Outputs: Read Data

`Dual port`	Create a dual-port RAM, with 4 inputs and 2 outputs. Inputs: • Write Data • Read/Write address • Write enable • Read address Outputs: • Write Data • Read Data

WriteOutputValue

Behavior for Write output

Default: `New data`

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

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

Methods

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

Examples

Create Single-Port RAM System Object

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

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

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

```
H =

  System: hdlram

  Properties:
             RAMType: 'Single port'
    WriteOutputValue: 'Old data'
```

Create Simple Dual-Port RAM System Object

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

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

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

```
H =

  System: hdlram

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

Create Dual-Port RAM System Object

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

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

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

```
H =

  System: hdlram

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

Read/Write Single-Port RAM

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

Construct single-port RAM System object.

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

Preallocate memory.

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

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

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

Related Examples

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

Purpose Construct hdlram System object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates another instance of the System object, `H`, with the same property values. The clone method creates a new, unlocked object with uninitialized states.

Input Arguments

H

Instance of hdlram

Output Arguments

C

New hdlram System object with the same property values as the original System object. The clone method creates a new, unlocked object with uninitialized states.

See Also

`hdlram` | `hdlram.isLocked` | `hdlram.release` | `hdlram.step` |

hdlram.IsLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `L = isLocked(OBJ)`

Description `L = isLocked(OBJ)` returns a logical value, `L`, which indicates whether input attributes and nontunable properties are locked for the System object, `OBJ`. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After the System object is locked, the isLocked method returns a true value.

Input Arguments

OBJ

Instance of hdlram

Output Arguments

L

Logical value. Either `1` (true) or `0` (false).

See Also `hdlram` | `hdlram.clone` | `hdlram.release` | `hdlram.step` |

Purpose

Allow changes to property values and input characteristics

Syntax

`release(OBJ)`

Description

`release(OBJ)` releases system resources (such as memory, file handles, and hardware connections) of System object, `OBJ`. Releasing these resources allows all of the System object properties and input characteristics to be changed. After you call the release method on a System object, any subsequent calls to step or release that you make are not supported for code generation.

Input Arguments

OBJ

Instance of hdlram

See Also

`hdlram` | `hdlram.clone` | `hdlram.isLocked` | `hdlram.step` |

Purpose

Read or write input value to memory location

Syntax

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

Description

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

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

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

hdlram Input Requirements

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

Examples

Read/Write Single-Port RAM

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

Construct single-port RAM System object.

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

Preallocate memory.

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

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

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

See Also

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

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 | storedInteger | oct

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`

hist

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.

horzcat

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`

Purpose Imaginary part of complex number

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

innerprodintbits

Purpose

Number of integer bits needed for fixed-point inner product

Syntax

```
innerprodintbits(a,b)
```

Description

`innerprodintbits(a,b)` computes the minimum number of integer bits necessary in the inner product of `a'*b` to guarantee that no overflows occur and to preserve best precision.

- `a` and `b` are `fi` vectors.
- The values of `a` are known.
- Only the numeric type of `b` is relevant. The values of `b` are ignored.

Examples

The primary use of this function is to determine the number of integer bits necessary in the output `Y` of an FIR filter that computes the inner product between constant coefficient row vector `B` and state column vector `Z`. For example,

```
for k=1:length(X);
  Z = [X(k);Z(1:end-1)];
  Y(k) = B * Z;
end
```

Algorithms

In general, an inner product grows `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

Convert `fi` object to signed 8-bit integer

Syntax

```
c = int8(a)
```

Description

`c = int8(a)` returns the built-in `int8` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int8`.

Examples

This example shows the `int8` values of a fi object.

```
a = fi([-pi 0.1 pi],1,8);
c = int8(a)

c =

   -3    0    3
```

See Also

`storedInteger` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Purpose

Convert `fi` object to signed 16-bit integer

Syntax

```
c = int16(a)
```

Description

`c = int16(a)` returns the built-in `int16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int16`.

Examples

This example shows the `int16` values of a fi object.

```
a = fi([-pi 0.1 pi],1,16);
c = int16(a)

c =

   -3    0    3
```

See Also

`storedInteger` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Purpose

Convert `fi` object to signed 32-bit integer

Syntax

```
c = int32(a)
```

Description

`c = int32(a)` returns the built-in `int32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int32`.

Examples

This example shows the `int32` values of a fi object.

```
a = fi([-pi 0.1 pi],1,32);
c = int32(a)

c =

   -3    0    3
```

See Also

`storedInteger` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Purpose

Convert fi object to signed 64-bit integer

Syntax

```
c = int64(a)
```

Description

`c = int64(a)` returns the built-in `int64` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int64`.

Examples

This example shows the `int64` values of a fi object.

```
a = fi([-pi 0.1 pi],1,64);
c = int64(a)

c =

   -3    0    3
```

See Also

`storedInteger` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64`

intmax

Purpose

Largest positive stored integer value representable by `numerictype` of `fi` object

Syntax

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

Description

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

See Also

`eps` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

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`

ipermute

Purpose Inverse permute dimensions of multidimensional array

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

Purpose

Determine whether input is Boolean

Syntax

```
y = isboolean(a)
y = isboolean(T)
```

Description

`y = isboolean(a)` returns `1` when the `DataType` property of `fi` object `a` is `boolean`, and `0` otherwise.

`y = isboolean(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `boolean`, and `0` otherwise.

See Also

`isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

iscolumn

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`

Purpose

Determine whether input is double-precision data type

Syntax

```
y = isdouble(a)
y = isdouble(T)
```

Description

`y = isdouble(a)` returns `1` when the `DataType` property of `fi` object `a` is `double`, and `0` otherwise.

`y = isdouble(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `double`, and `0` otherwise.

See Also

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

isempty

Purpose Determine whether array is empty

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

Purpose

Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal

Syntax

```
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`

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`

isfimathlocal

Purpose Determine whether `fi` object has local fimath

Syntax `y = isfimathlocal(a)`

Description `y = isfimathlocal(a)` returns `1` if the `fi` object `a` has a local `fimath` object, and 0 if `a` does not have a local fimath.

See Also `fimath` | `isfi` | `isfipref` | `isnumerictype` | `isquantizer` | `sfi` | `ufi`

Purpose Determine whether array elements are finite

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

isfipref

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`

Purpose

Determine whether input is fixed-point data type

Syntax

```
y = isfixed(a)
y = isfixed(T)
y = isfixed(q)
```

Description

`y = isfixed(a)` returns `1` when the `DataType` property of `fi` object `a` is `Fixed`, and `0` otherwise.

`y = isfixed(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `Fixed`, and `0` otherwise.

`y = isfixed(q)` returns `1` when `q` is a fixed-point `quantizer`, and `0` otherwise.

See Also

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

isfloat

Purpose

Determine whether input is floating-point data type

Syntax

```
y = isfloat(a)
y = isfloat(T)
y = isfloat(q)
```

Description

`y = isfloat(a)` returns `1` when the `DataType` property of `fi` object `a` is `single` or `double`, and `0` otherwise.

`y = isfloat(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `single` or `double`, and `0` otherwise.

`y = isfloat(q)` returns `1` when `q` is a floating-point `quantizer`, and `0` otherwise.

See Also

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

Purpose Determine whether array elements are infinite

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

isnan

Purpose Determine whether array elements are NaN

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

Purpose Determine whether input is numeric array

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

isnumerictype

Purpose Determine whether 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.

ispropequal

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`

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.

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`

isscalar

Purpose Determine whether input is scalar

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

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` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isscaledtype

Purpose

Determine whether input is fixed-point or scaled double data type

Syntax

```
y = isscaledtype(a)
y = isscaledtype(T)
```

Description

`y = isscaledtype(a)` returns `1` when the `DataType` property of `fi` object `a` is `Fixed` or `ScaledDouble`, and `0` otherwise.

`y = isscaledtype(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `Fixed` or `ScaledDouble`, and `0` otherwise.

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `numerictype` | `isscaleddouble` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

Purpose

Determine whether input has binary point scaling

Syntax

```
y = isscalingbinarypoint(a)
y = isscalingbinarypoint(T)
```

Description

`y = isscalingbinarypoint(a)` returns `1` when the `fi` object `a` has binary point scaling or trivial slope and bias scaling. Otherwise, the function returns `0`. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`y = isscalingbinarypoint(T)` returns `1` when the `numerictype` object `T` has binary point scaling or trivial slope and bias scaling. Otherwise, the function returns `0`. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

See Also

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

isscalingslopebias

Purpose

Determine whether input has nontrivial slope and bias scaling

Syntax

```
y = isscalingslopebias(a)
y = isscalingslopebias(T)
```

Description

`y = isscalingslopebias(a)` returns `1` when the `fi` object `a` has nontrivial slope and bias scaling, and `0` otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`y = isscalingslopebias(T)` returns `1` when the `numerictype` object `T` has nontrivial slope and bias scaling, and `0` otherwise. Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

See Also

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

Purpose

Determine whether input has unspecified scaling

Syntax

```
y = isscalingunspecified(a)
y = isscalingunspecified(T)
```

Description

`y = isscalingunspecified(a)` returns `1` if `fi` object `a` has a fixed-point or scaled double data type and its scaling has not been specified.

`y = isscalingunspecified(T)` returns `1` if `numerictype` object `T` has a fixed-point or scaled double data type and its scaling has not been specified.

See Also

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

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` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified`

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.

line

Purpose Create line object

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

Purpose Convert numeric values to logical

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

loglog

Purpose Create log-log scale plot

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

Purpose

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:

$$-1 \leq x \leq 1 - 2^{-15}$$

You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)

r =

                         -1          0.999969482421875
```

Example 2: Using maxlog with quantizer objects

```
q = quantizer;
warning on
format long g
x = [-20:10];
y = quantize(q,x);
maxlog(q)

Warning: 29 overflows.
> In embedded.quantizer.quantize at 74

ans =

  .999969482421875
```

The largest value `maxlog` can return is the maximum representable value of its input. You can obtain the range of `x` after quantization using the `range` function:

```
format long g
r = range(q)

r =

                        -1          0.999969482421875
```

See Also

`fipref` | `minlog` | `noverflows` | `nunderflows` | `reset` | `resetlog`

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 has no local fimath.

When `a` is an empty fixed-point array (value = `[]`), the value of the output array is zero.

Examples

Compute the mean value along the first dimension (rows) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 28-bits
mx1 = mean(x,1)
```

Compute the mean value along the second dimension (columns) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5], 1, 32);
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 28 bits
mx2 = mean(x,2)
```

Algorithms

The general equation for computing the `mean` of an array *a*, across dimension *dim* is:

```
sum(a,dim)/size(a,dim)
```

Because `size(a,dim)` is always a positive integer, the algorithm casts `size(a,dim)` to an unsigned 32-bit `fi` object with a fraction length of zero (`SizeA`). The algorithm then computes the mean of *a* according to the following equation, where `Tx` represents the `numerictype` properties of the fixed-point input array *a*:

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

See Also

`max` | `median` | `min`

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 has no local 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.

meshc

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.

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 Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode" in the Fixed-Point Designer User's Guide.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

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

mod

Purpose

Modulus after division for `fi` objects

Syntax

```
M = mod(X,Y)
```

Description

`M = mod(X,Y)` if `Y ~= 0`, returns `X-n.*Y`, where `n = floor(X./Y)`. The inputs `X` and `Y` must be real arrays of the same size, or either can be a real scalar. For fixed-point or integer input arguments, the output data type is the aggregate type of both input signedness, word lengths, and fraction lengths. For fixed-point inputs, the word length of the internally computed aggregate fixed-point output data type cannot exceed 32 bits. For floating-point input arguments, the output data type is the same as the inputs.

Note The combination of fixed-point and floating-point inputs is not currently supported.

Input Arguments

X

Integer, fixed-point, or floating-point array, or real scalar.

Y

Array of the same size as `X`, or real scalar.

Output Arguments

M

Result of modulus operation. If both inputs `X` and `Y` are floating-point, then the data type of `M` is the same as the inputs. If either input `X` or `Y` is fixed-point, then the data type of `M` is the aggregate numerictype. This value equals that of `fixed.aggregateType(X,Y)`.

Examples

Calculate the `mod` of two `fi` objects.

```
% 7-bit signed fixed-point object
x = fi(-3,1,7,0);
% 15-bit signed fixed-point object
```

```
y = fi(2,1,15,0);
M1 = mod(x,y)
M1 =

     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 15
        FractionLength: 0
M2 = mod(y,x)
M2 =

    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 15
        FractionLength: 0
```

Convert the `fi` inputs in the previous example to double type, and calculate the `mod`.

```
Mf1 = mod(double(x),double(y))
Mf1 =

     1
Mf2 = mod(double(y),double(x))
Mf2 =

    -1
```

See Also

`fixed.aggregateType` | `mod`

Purpose

Fixed-point matrix power (^)

Syntax

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

Description

`c = mpower(a,k)` and `c = a^k` compute matrix power. The exponent *k* requires a positive, real-valued integer value.

The fixed-point output array *c* has the same local fimath as the input *a*. If *a* has no local fimath, the output *c* also has no local fimath. The matrix power operation is performed using default fimath settings.

Tips

For more information about the `mpower` function, see the MATLAB `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` has no local fimath.

`a` and `b` can both be `fi` objects with the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object. `a` and `b` can also be doubles, singles, or integers.

Examples

In this example, `c` is the 40-bit product of `a` and `b` with fraction length 30.

```
a = fi(pi);
b = fi(exp(1));
F = fimath('ProductMode','SpecifyPrecision',...
  'ProductWordLength',40,'ProductFractionLength',30);
c = F.mpy(a, b)

c =

    8.5397


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

Algorithms

`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

        RoundingMethod: nearest
        OverflowAction: saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 40
 ProductFractionLength: 30
               SumMode: FullPrecision
```

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

See Also

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

Purpose

Forward slash (/) or right-matrix division

Syntax

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

Description

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

When one or both of the inputs is a `fi` object, the denominator input, `b`, must be a scalar and the output `fi` object `c` is equivalent to `c = rdivide(a,b)` or `c = a./b` (right-array division).

The numerator input `a` can be complex, but the denominator input `b` must always be real-valued. When the numerator input `a` is complex, the real and imaginary parts of `a` are independently divided by `b`.

For information on the data type rules used by the `mrdivide` function, see the `rdivide` reference page.

Examples

In this example, you use the forward slash (/) to perform right matrix division on a 3-by-3 magic square of `fi` objects. Because the numerator input is a `fi` object, the denominator input `b` must be a scalar:

```
a = fi(magic(3))
b = fi(3, 1, 12, 8)
c = a/b
```

The `mrdivide` function outputs a signed 3-by-3 array of `fi` objects, each of which has a word length of 16 bits and a fraction length of 3 bits.

```
a =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
b =

     3

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

c =

    2.6250    0.3750    2.0000
    1.0000    1.6250    2.3750
    1.3750    3.0000    0.6250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 3
```

See Also

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

Purpose

Matrix product of `fi` objects

Syntax

`mtimes(a,b)`

Description

`mtimes(a,b)` is called for the syntax `a * b` when `a` or `b` is an object.

`a * b` is the matrix product of `a` and `b`. A scalar value (a 1-by-1 matrix) can multiply any other value. Otherwise, the number of columns of `a` must equal the number of rows of `b`.

`mtimes` does not support `fi` objects of data type `Boolean`.

Note For information about the `fimath` properties involved in Fixed-Point Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode" in the Fixed-Point Designer documentation.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

See Also

`plus` | `minus` | `times` | `uminus`

ndgrid

Purpose Generate arrays for N-D functions and interpolation

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

Purpose Number of array dimensions

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

Purpose

Determine whether real-world values of two `fi` objects are not equal

Syntax

```
c = ne(a,b)
a ~= b
```

Description

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

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

See Also

`eq` | `ge` | `gt` | `le` | `lt`

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

Determine fixed-point data type

Syntax

```
nts
nts({'block',PORT})
nts({line-handle})
nts({gsl})
```

Description

`nts` opens the `NumericTypeScope` window. To connect to a signal in a Simulink model, select the signal and then, in the `NumericTypeScope` window, select **File > Connect to Simulink Signal**.

The `NumericTypeScope` suggests a fixed-point data type in the form of a `numerictype` object based on the dynamic range of the input data and the criteria that you specify in the Bit Allocation Panel on page 437. The scope allows you to visualize the dynamic range of data in the form of a `log2` histogram. It displays the data values on the X-axis and the number or percentage of occurrences on the Y-axis. Each bin in the histogram corresponds to a bit in a word. For example, 2^0 corresponds to the first integer bit in the binary word, 2^{-1} corresponds to the first fractional bit in the binary word.

`nts({'block',PORT})` opens the `NumericTypeScope` window and connects the scope to the signal output from `block` on output port with index `PORT`. If the block has more than one output port, you must specify the port index. The scope cannot connect to more than one output port.

`nts({line-handle})` opens the `NumericTypeScope` window and connects the scope to the Simulink signal which has the line handle specified in `line-handle`.

`nts({gsl})` opens the `NumericTypeScope` window and connects the scope to the currently selected Simulink signal. You must select a signal in a Simulink model first, otherwise the scope opens with no signal selected.

Tips

- Use the `NumericTypeScope` to help you identify any values that are outside range or below precision based on the current data type.

When the information is available, the scope indicates values that are outside range, below precision, and in range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are in range of the current data type.
- Red — Histogram bin contains values that are outside range in the current data type.
- Yellow — Histogram bin contains values that are below precision in the current data type.

- Select **View > Vertical Units** to select whether to display values as a percentage or as an actual count.
- Use the **View > Bring All NumericTypeScope Windows Forward** menu option to manage your `NumericTypeScope` windows. Selecting this option or pressing **Ctrl+F** brings all `NumericTypeScope` windows into view.

Input Arguments

block

Full path to the specified block.

line-handle

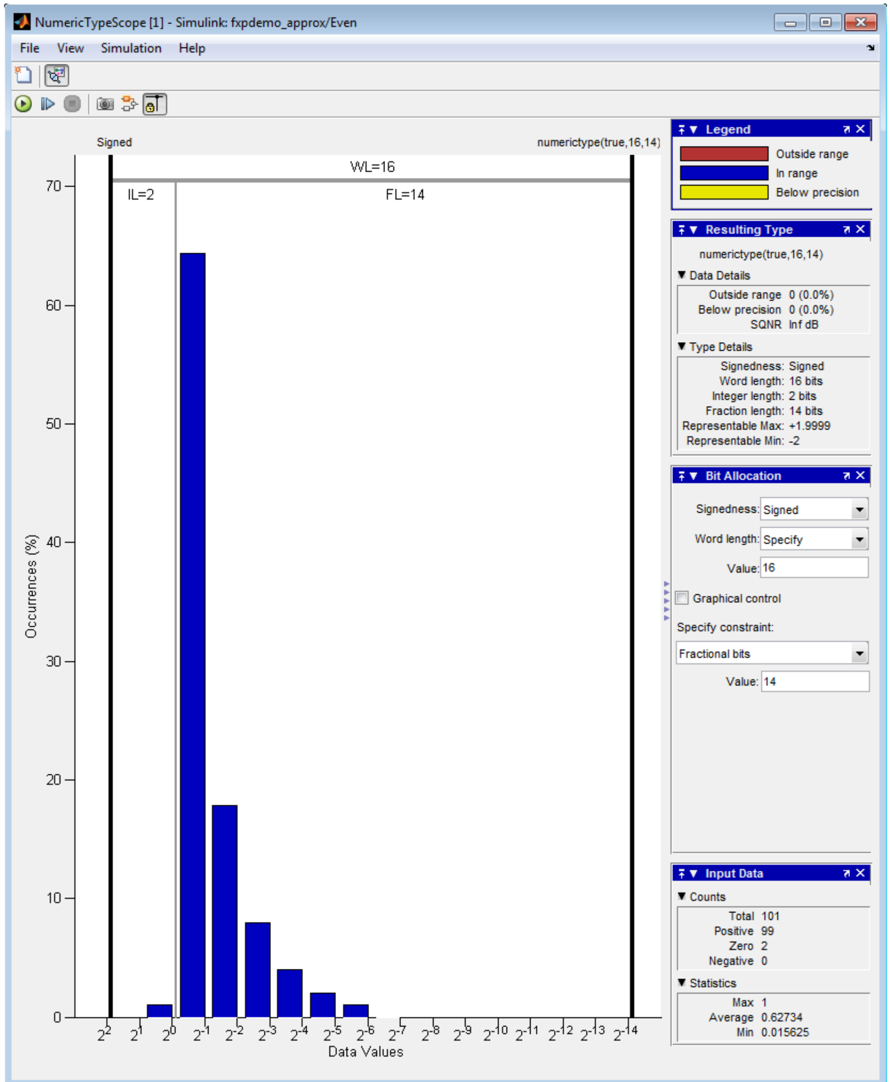
Handle of the Simulink signal that you want to view with the scope. To get the handle of the currently selected signal, at the MATLAB command line, enter `nts({gsl})`.

PORT

Index of the output port that you want to view with the scope. If the block has more than one output port, you must specify the index. The scope cannot connect to more than one output port.

The NumericTypeScope Window

The `NumericTypeScope` opens with the default toolbars displayed at the top of the window and the dialog panels to the right.



Toolbars

By default the scope displays a toolbar that provides these options:

Button	Action
	New NumericTypeScope.
	Connect to Simulink signal. The scope connects to the currently selected signal. If a block with only one output port is selected and the **Connect scope on selection of** is set to `Signal lines or blocks`, connects to the output port of the selected block. For more information, see Sources Pane on page 436.

After connecting the scope to a signal in a Simulink model, the scope displays an additional toolbar with the following options:

Button	Action
	Stop simulation
	Start simulation
	Simulate one step
	Snapshot. Freezes the display so that you can examine the results. To reenable display refreshing, click the button again.
	Highlight Simulink signal.
	Persistent. By default, the scope makes a persistent connection to the selected signal. If you want to view different signals during the simulation, click this button to make a floating connection. You can then select any signal in the model and the scope displays it.

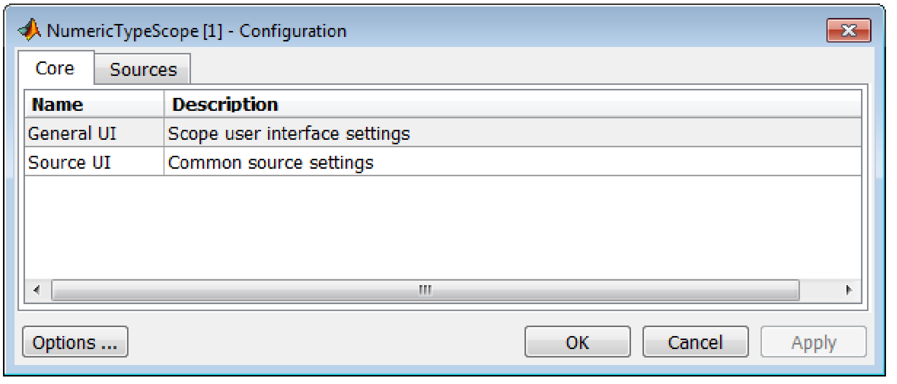
Dialog Boxes and Panels

- “Configuration Dialog Box” on page 2-433
- “Dialog Panels” on page 2-437

Configuration Dialog Box

Use the `NumericTypeScope` configuration dialog box to control the behavior and appearance of the scope window.

To open the **Configuration** dialog box, from the scope main menu, select **File > Configuration > Edit**, or, with the scope as your active window, press the **N** key.



For information about each pane, see Core Pane on page 434 and Sources Pane on page 436.

To save configuration settings for future use, select **File > Configuration > Save as**. The configuration settings you save become the default configuration settings for the `NumericTypeScope`.

Caution

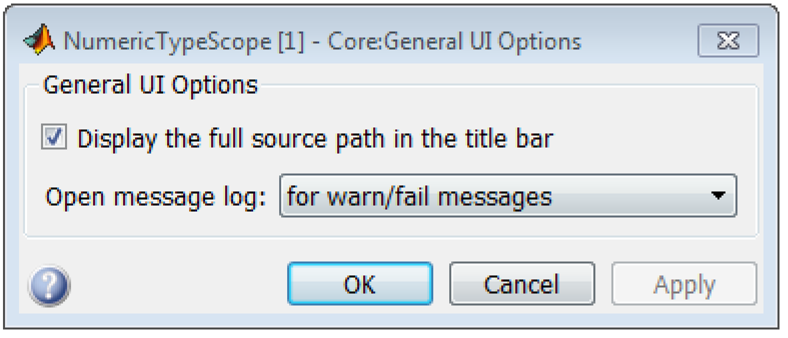
Before saving your own set of configuration settings in the `matlab/toolbox/fixpoint` folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

To save your configuration settings for future use, save them in the `matlab/toolbox/fixpoint` folder with the file name `NumericTypeScopeSL.cfg`. You can re-save your configuration settings at anytime, but you must save them in this folder with this filename.

Core Pane

The **Core** pane controls the general settings of the scope.

To open the **Core:General UI Options** dialog box, select **General UI** and then click **Options**.



- **Display the full source path in the title bar**—Select this check box to display the full path to the selected block in the model. Otherwise, the scope displays only the block name.
- **Open message log**—Control when the Message Log window opens. The Message log window helps you debug issues with the scope. Choose to open the Message Log window for any of these conditions:

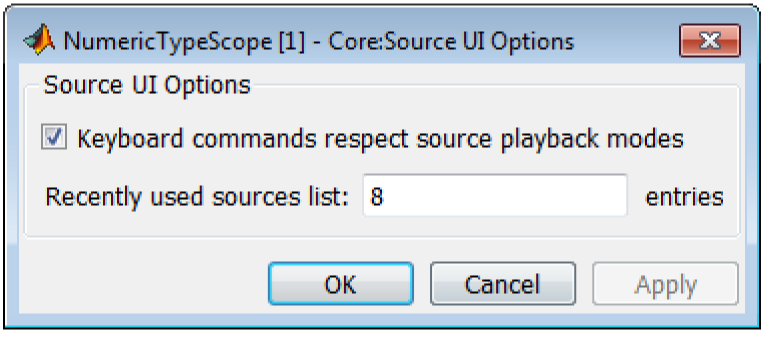
- for any new messages
- for warn/fail messages
- only for fail messages
- manually

The option defaults to `for warn/fail messages`.

You can open the Message Log at any time by selecting **Help > Message Log** or by pressing **Ctrl+M**. The **Message Log** dialog box provides a system level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by Type and Category.

- **Type**—Select the type of messages to display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`. Type defaults to `All`.
- **Category**—Select the category of messages to display in the Message Log. You can select `All`, `Configuration`, or `Extension`. The scope uses Configuration messages to indicate when new configuration files are loaded, and Extension messages to indicate when components are registered. Category defaults to `All`.

To open the **Core:Source UI Options** dialog box, select **General UI** and then click **Options**.



- **Keyboard commands respect source playback modes**—Has no effect. The following table shows the keyboard shortcut mapping. You cannot disable this mapping.

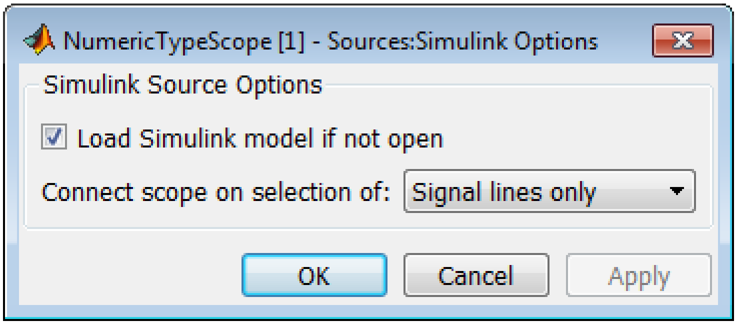
Action	Keyboard Shortcut
Open new NumericTypeScope	Insert
Change configuration	N
Display keyboard help	K
Play simulation	P
Pause simulation	Space
Stop simulation	S
Step forward	Right arrow, Page down

- **Recently used sources list**—Sets the maximum number of recently used sources displayed under the **Files** menu option.

Sources Pane

The **Sources** pane controls how the scope connects to Simulink. You cannot disable the Simulink source.

To open the **Sources:Simulink Options** dialog box, select the **Sources** tab and then click **Options**.

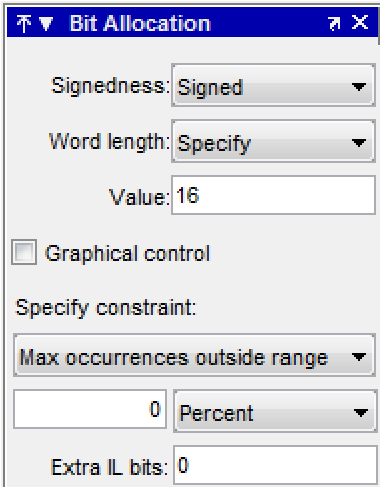


- **Load Simulink model if not open**—When selected, if you specify a signal in a Simulink model that is not currently open, the scope opens the model.
- **Connect scope on selection of**—Connects the scope only when you select signal lines or when you select signal lines or blocks. If you select `Signal lines or blocks`, the scope cannot connect to blocks that have more than one output port.

Dialog Panels

Bit Allocation Panel

The scope **Bit Allocation** panel provides options for specifying data type criteria. Adjust these criteria to observe the effect on suggested numerictype. For streaming data, the suggested numerictype adjusts over time in order to continue to satisfy the specified criteria.



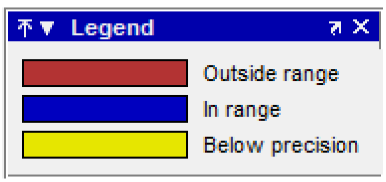
You can:

- Specify a known word length and signedness and, using **Specify constraint**, add additional constraints such as the maximum number of occurrences outside range or the smallest value that the suggested data type must represent.

- Specify **Integer length** and **Fraction length** constraints so that the scope suggests an appropriate word length.
- Set the **Signedness** and **Word length** to `Auto` so that the scope suggests values for these parameters.
- Enable **Graphical control** and use the cursors on either side of the binary point to adjust the fraction length and observe the effect on the suggested numerictype on the input data. For example, you can see the number of values that are outside range, below precision, or both. You can also view representable minimum and maximum values of the changed suggested data type.
- Specify extra bits for either the fraction length or the integer length. The extra bits act as a safety margin to minimize the risk of overflow and precision loss.

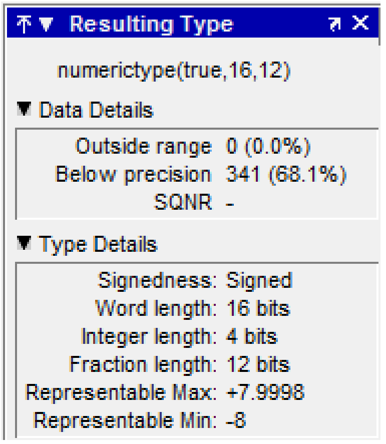
Legend

The scope **Legend** panel informs you which colors the scope uses to indicate values. These colors represent values that are outside range, in range, or below precision when displayed in the scope.



Resulting Type

The **Resulting Type** panel describes the fixed-point data type as defined by scope settings. By manipulating the visual display (via the **Bit Allocation** panel or with the cursors), you can change the data type specification.

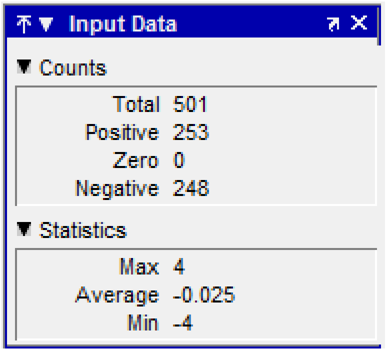


The **Data Details** section displays the percentage of values that fall outside range or below precision with the `numerictype` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data to calculate the SQNR. When scope information or the `numerictype` changes, the SQNR resets.

The **Type Details** section provides details about the fixed-point data type. You can copy the numerictype specification by right-clicking the **Resulting Type** pane and then selecting `Copy numerictype`.

Input Data

The **Input Data** panel provides statistical information about the values currently displayed in the `NumericScopeType`.



Examples

Connect a NumericTypeScope to a signal in a Simulink model

Open a `NumericTypeScope` window and connect to a signal.

Open the model.

```
fxpdemo_approx
```

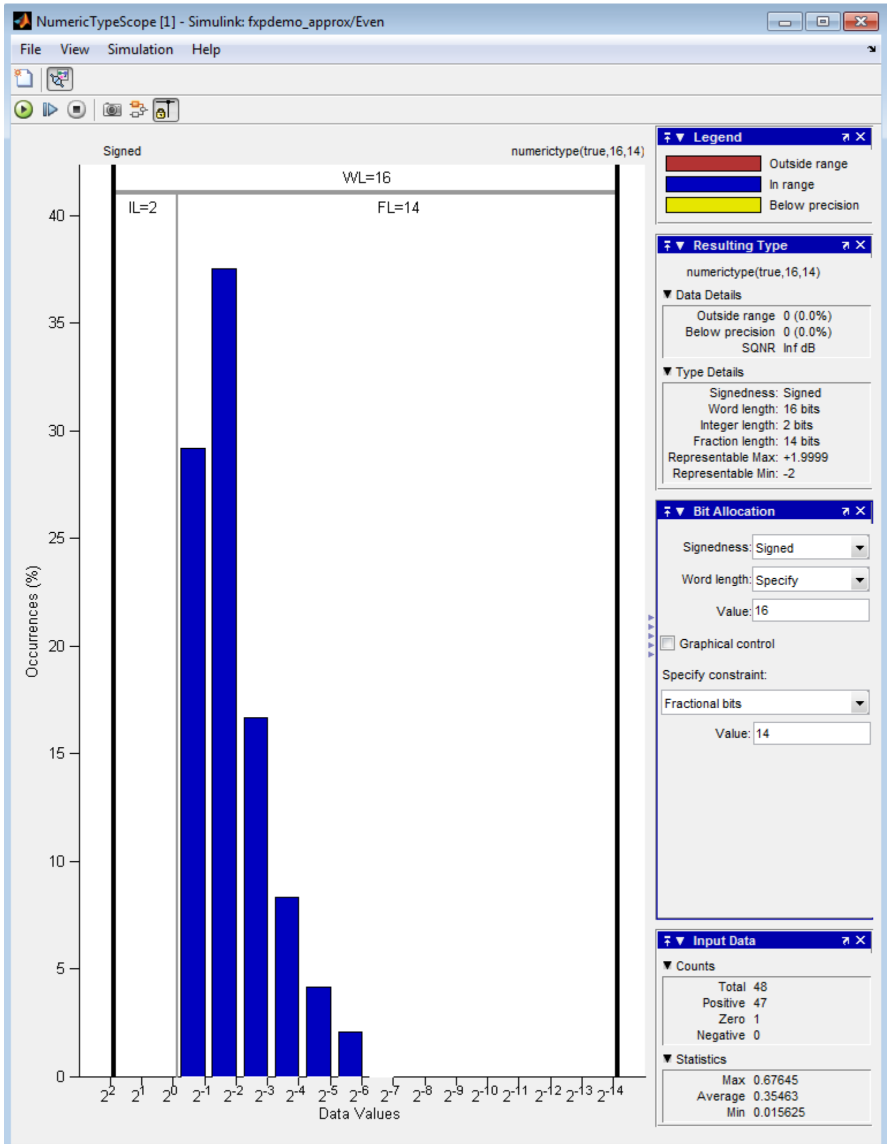
Open a NumericTypeScope.

```
nts
```

In the `fxpdemo_approx` model, select the `yEven` signal.

In the `NumericTypeScope` window, select **File > Connect to Simulink Signal**.

Run the simulation to view the dynamic range of the output. The `NumericTypeScope` suggests a data type for the output.


NumericTypeScope [1] - Simulink: fxpdemo_approx/Even
File View Simulation Help
Signed
numerictype(true,16,14)
WL=16
IL=2
FL=14
Occurrences (%)
Data Values
Legend
Outside range
In range
Below precision
Resulting Type
numerictype(true,16,14)
Data Details
Outside range 0 (0.0%)
Below precision 0 (0.0%)
SQNR Inf dB
Type Details
Signedness: Signed
Word length: 16 bits
Integer length: 2 bits
Fraction length: 14 bits
Representable Max: +1.9999
Representable Min: -2
Bit Allocation
Signedness: Signed
Word length: Specify
Value: 16
Graphical control
Specify constraint:
Fractional bits
Value: 14
Input Data
Counts
Total 48
Positive 47
Zero 1
Negative 0
Statistics
Max 0.67645
Average 0.35463
Min 0.015625

Connect a NumericTypeScope to a block output port

Connect a NumericTypeScope to a block output port and view the dynamic range of block output.

Specify the block path and name and the output port number.

```
blk='fxpdemo_approx/Even';
nts({blk,1})
```

Run the simulation to view the dynamic range of the output. The `NumericTypeScope` suggests a data type for the output.

Specify a Simulink signal to display

Connect a NumericTypeScope to a signal selected in a model.

Open the model.

```
fxpdemo_approx
```

In the `fxpdemo_approx` model, select the `yEven` signal.

Open a `NumericTypeScope`, specifying the line handle of the selected signal.

```
nts({gsl})
```

See Also

`hist` | `log2` | `numerictypescope`

Purpose

Convert number to binary string using `quantizer` object

Syntax

```
y = num2bin(q,x)
```

Description

`y = num2bin(q,x)` converts numeric array `x` into binary strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to binary. If `x` is a structure, each numeric field of `x` is converted to binary.

`num2bin` and `bin2num` are inverses of one another, differing in that `num2bin` returns the binary strings in a column.

Examples

```
x = magic(3)/9;
q = quantizer([4,3]);
y = num2bin(q,x)

Warning: 1 overflow.

y =

0111
0010
0011
0000
0100
0111
0101
0110
0001
```

See Also

`bin2num` | `hex2num` | `num2hex` | `num2int`

num2hex

Purpose Convert number to hexadecimal equivalent using `quantizer` object

Syntax

```
y = num2hex(q,x)
```

Description

`y = num2hex(q,x)` converts numeric array `x` into hexadecimal strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to hexadecimal. If `x` is a structure, each numeric field of `x` is converted to hexadecimal.

For fixed-point `quantizer` objects, the representation is two's complement. For floating-point `quantizer` objects, the representation is IEEE Standard 754 style.

For example, for `q = quantizer('double')`

```
num2hex(q,nan)

ans =

fff8000000000000
```

The leading fraction bit is 1, all other fraction bits are 0. Sign bit is 1, exponent bits are all 1.

```
num2hex(q,inf)

ans =

7ff0000000000000
```

Sign bit is 0, exponent bits are all 1, all fraction bits are 0.

```
num2hex(q,-inf)

ans =

fff0000000000000
```

Sign bit is 1, exponent bits are all 1, all fraction bits are 0.

`num2hex` and `hex2num` are inverses of each other, except that `num2hex` returns the hexadecimal strings in a column.

Examples

This is a floating-point example using a `quantizer` object `q` that has 6-bit word length and 3-bit exponent length.

```
x = magic(3);
q = quantizer('float',[6 3]);
y = num2hex(q,x)

y =

18
12
14
0c
15
18
16
17
10
```

See Also

`bin2num` | `hex2num` | `num2bin` | `num2int`

num2int

Purpose

Convert number to signed integer

Syntax

```
y = num2int(q,x)
[y1,y,...] = num2int(q,x1,x,...)
```

Description

`y = num2int(q,x)` uses `q.format` to convert numeric `x` to an integer.

`[y1,y,...]  = num2int(q,x1,x,...)` uses `q.format` to convert numeric values `x1, x2,...` to integers `y1,y2,...`

Examples

All the two's complement 4-bit numbers in fractional form are given by

```
x = [0.875 0.375 -0.125 -0.625
     0.750 0.250 -0.250 -0.750
     0.625 0.125 -0.375 -0.875
     0.500 0.000 -0.500 -1.000];

q=quantizer([4 3]);

y = num2int(q,x)

y =

     7     3    -1    -5
     6     2    -2    -6
     5     1    -3    -7
     4     0    -4    -8
```

Algorithms

When `q` is a fixed-point `quantizer` object, f is equal to `fractionlength(q)`, and x is numeric

$$y = x \times 2^f$$

When `q` is a floating-point `quantizer` object, $y = x$. `num2int` is meaningful only for fixed-point `quantizer` objects.

See Also

`bin2num` | `hex2num` | `num2bin` | `num2hex`

Purpose

Number of data elements in an array

Syntax

```
numberofelements(a)
```

Description

`numberofelements(a)` returns the number of data elements in an array. Using `numberofelements` in your MATLAB code returns the same result for built-in types and `fi` objects. Use `numberofelements` to write data-type independent MATLAB code for array handling.

Note that `fi` is a MATLAB object, and therefore `numel(a)` returns `1` when `a` is a `fi` object.

See Also

`numel` | `nargin` | `nargout` | `prod` | `size` | `subsref` | `subsasgn`

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
- `DataTypeOverride` — Data type override settings. Note that this property is not visible when its value is the default, `Inherit`.
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Signedness` — Signed, unsigned, or auto
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

Examples

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 Designer software first creates a default `numerictype` object, and then, for each property name you specify in the constructor, assigns the corresponding value. This behavior differs from the behavior that occurs when you use a syntax such as `T = numerictype(s,w)`. See “Example: Construct a numerictype Object with Property Name and Property Value Pairs” in the Fixed-Point Designer User’s Guide for more information.

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`

Related Examples

- "numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects"

Concepts

- "numerictype Object Properties"
- "numerictype Structure of Fixed-Point Objects"

NumericTypeScope

Purpose Determine fixed-point data type

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. The scope provides a visual representation of the dynamic range of your data in the form of a `log2` histogram. In this histogram, the bit weights appear along the X-axis, and the percentage of occurrences along the Y-axis. Each bin of the histogram corresponds to a bit in the binary word. For example, 2^0 corresponds to the first integer bit in the binary word, 2^{-1} corresponds to the first fractional bit in the binary word.

The scope suggests a data type in the form of a `numerictype` object that satisfies the specified criteria. See the section on Bit Allocation in "Dialog Panels" on page 2-461.

`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 clears the information displayed in the scope window.

Identifying Values Outside Range and Below Precision

The `NumericTypeScope` can also help you identify any values that are outside range or below precision based on the current data type. To prepare the `NumericTypeScope` to identify them, provide an input variable that is a `fi` object and verify that one of the following conditions is true:

- The `DataTypeMode` of the `fi` object is set to `Scaled doubles: binary point scaling`.
- The `DataTypeOverride` property of the Fixed-Point Designer `fipref` object is set to `ScaledDoubles`.

When the information is available, the scope indicates values that are outside range, below precision, and in range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are in range of the current data type.
- Red — Histogram bin contains values that are outside range in the current data type.
- Yellow — Histogram bin contains values that are below precision in the current data type.

For an example of the scope color coding, see the figures in "Vertical Units" on page 2-464.

See also Legend in "Dialog Panels" on page 2-461.

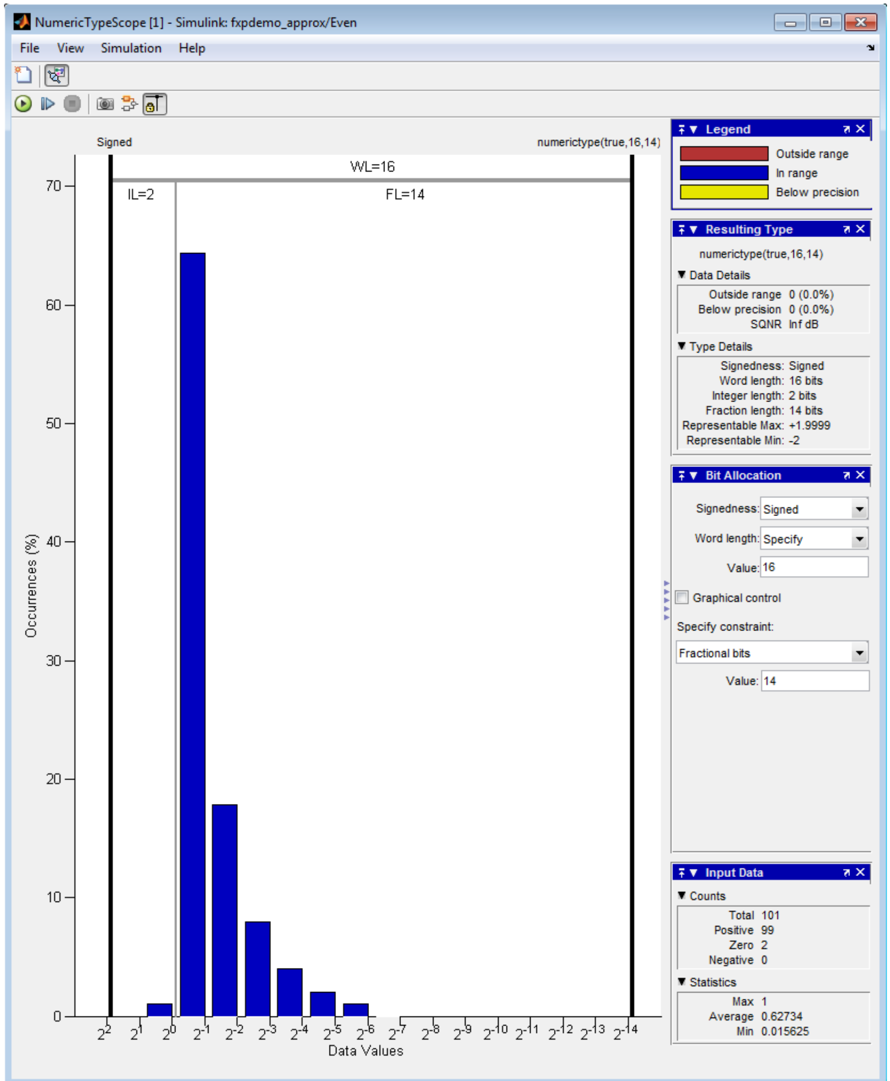
See the "Examples" on page 2-468 section to learn more about using the `NumericTypeScope` to select data types.

Dialog Boxes and Toolbar

- "The NumericTypeScope Window" on page 2-456
- "Configuration Dialog Box" on page 2-458
- "Dialog Panels" on page 2-461
- "Vertical Units" on page 2-464
- "Bring All NumericType Scope Windows Forward" on page 2-466
- "Toolbar (Mac Only)" on page 2-467

The NumericTypeScope Window

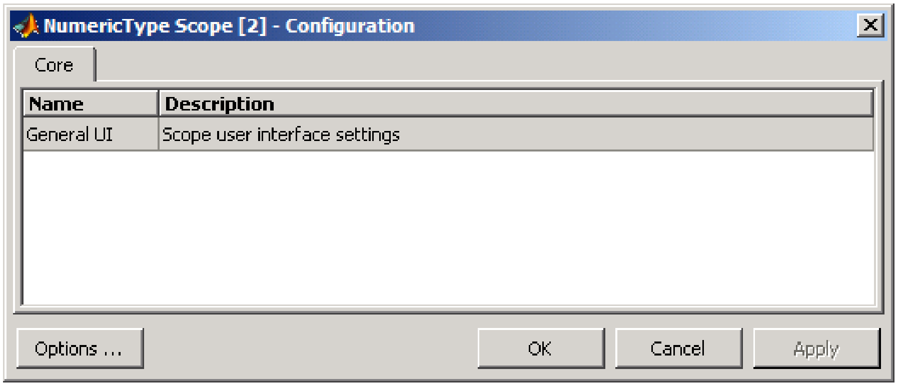
The `NumericTypeScope` opens with the default toolbars displayed at the top of the window and the dialog panels to the right.


NumericTypeScope [1] - Simulink: fxpdemo_approx/Even
File View Simulation Help
Signed
numerictype(true,16,14)
WL=16
IL=2
FL=14
Occurrences (%)
Data Values
Legend
Outside range
In range
Below precision
Resulting Type
numerictype(true,16,14)
Data Details
Outside range 0 (0.0%)
Below precision 0 (0.0%)
SQNR Inf dB
Type Details
Signedness: Signed
Word length: 16 bits
Integer length: 2 bits
Fraction length: 14 bits
Representable Max: +1.9999
Representable Min: -2
Bit Allocation
Signedness: Signed
Word length: Specify
Value: 16
Graphical control
Specify constraint:
Fractional bits
Value: 14
Input Data
Counts
Total 101
Positive 99
Zero 2
Negative 0
Statistics
Max 1
Average 0.62734
Min 0.015625

Configuration Dialog Box

The `NumericTypeScope` configuration allows you to control the behavior and appearance of the scope window.

To open the Configuration dialog box, select **File > Configuration > Edit**, or, with the scope as your active window, press the **N** key.



The Configuration Dialog box contains a series of panes each containing a table of configuration options. See the reference section for each pane for instructions on setting the options on each one. This dialog box has one pane, the Core pane, with only one option, for General UI settings for the scope user interface.

To save configuration settings for future use, select **File > Configuration > Save as**. The configuration settings you save become the default configuration settings for the `NumericTypeScope` object.

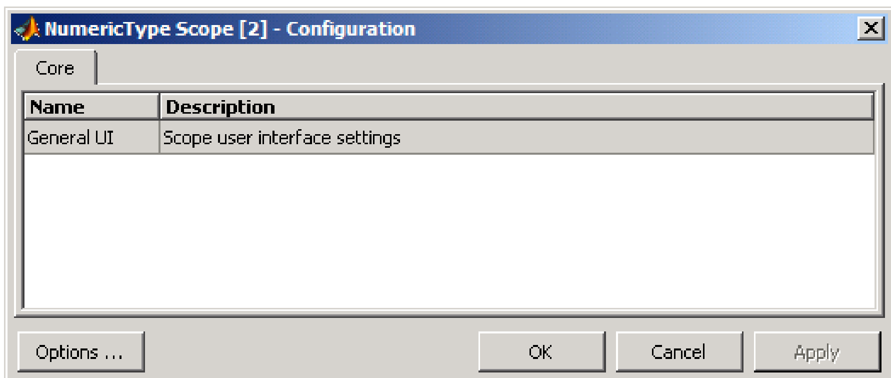
Caution

Before saving your own set of configuration settings in the matlab/toolbox/fixedpoint/fixedpoint folder, save a backup copy of the default configuration settings in another location. If you do not save a backup copy of the default configuration settings, you cannot restore these settings at a later time.

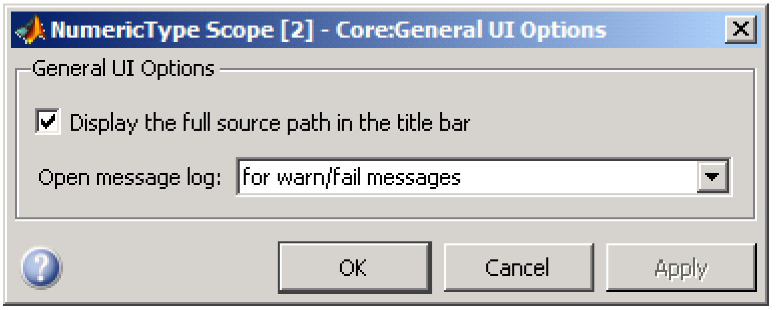
To save your configuration settings for future use, save them in the matlab/toolbox/fixedpoint/fixedpoint folder with the file name `NumericTypeScopeComponent.cfg`. You can re-save your configuration settings at anytime, but remember to do so in the specified folder using the specified file name.

Core Pane

The Core pane in the Configuration dialog box controls the general settings of the scope.



Click General UI and then click **Options** to open the Core:General UI Options dialog box.



- **Display the full source path in the title bar**—Select this check box to display the file name and variable name in the scope title bar. If the scope is not from a file, or if you clear this check box, the scope displays only the variable name in the title bar.

- **Open message log**—Control when the Message Log window opens. The Message log window helps you debug issues with the scope. Choose to open the Message Log window for any of these conditions:

  - `for any new messages`
  - `for warn/fail messages`
  - `only for fail messages`
  - `manually`

  The option defaults to `for warn/fail messages`.

  You can open the Message Log at any time by selecting **Help > Message Log** or by pressing **Ctrl+M**. The Message Log dialog box provides a system level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by Type and Category.

  - **Type**—Select the type of messages to display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`. Type defaults to `All`.
  - **Category**—Select the category of messages to display in the Message Log. You can select `All`, `Configuration`, or `Extension`.

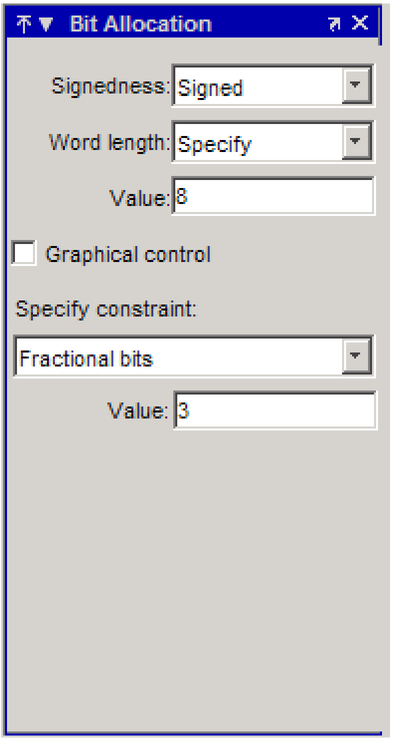
The scope uses Configuration messages to indicate when new configuration files are loaded, and Extension messages to indicate when components are registered. Category defaults to `All`.

Dialog Panels

- Bit Allocation on page 461
- Legend on page 463
- Resulting Type on page 463
- Input Data on page 464

Bit Allocation

The scope Bit Allocation dialog panel, as shown in the following figure, offers you several options for specifying data type criteria.

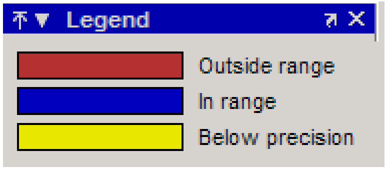


You can use this panel to specify a known word length and the desired maximum occurrences outside range. You can also use the panel to specify the desired number of occurrences outside range and the smallest value to be represented by the suggested data type. For streaming data, the suggested numerictype object adjusts over time in order to continue to satisfy the specified criteria.

The scope also allows you to interact with the histogram plot. When you select **Graphical control** on the Bit Allocation dialog panel, you enable cursors on either side of the binary point. You can interact with these cursors and observe the effect of the suggested numerictype on the input data. For example, you can see the number of values that are outside range, below precision, or both. You can also view representable minimum and maximum values of the data type.

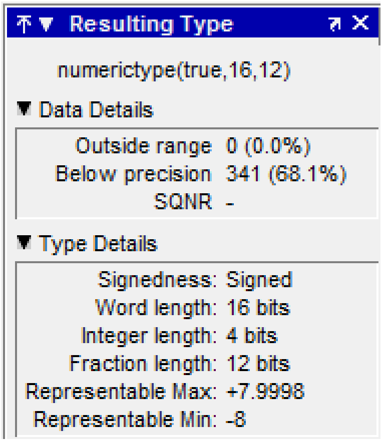
Legend

The scope Legend panel informs you which colors the scope uses to indicate values. These colors represent values that are outside range, in range, or below precision when displayed in the scope.



Resulting Type

The Resulting Type panel describes the fixed-point data type as defined by scope settings. By manipulating the visual display (via the Bit Allocation panel or with the cursors) you can change the value of the data type.

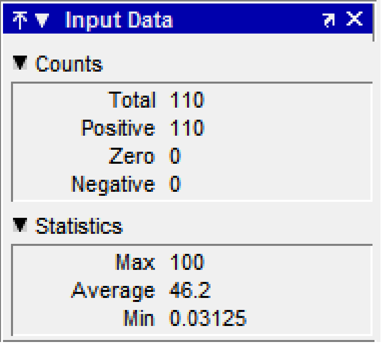


The Data Details section displays the percentage of values that fall outside range or below precision with the `numerictype` object located at the top of this panel. SQNR (Signal Quantization Noise Ratio) varies depending on the signal. If the parameter has no value, then there is not enough data to calculate the SQNR. When scope information or the `numerictype` changes, the SQNR resets.

Type Details section provides details about the fixed-point data type.

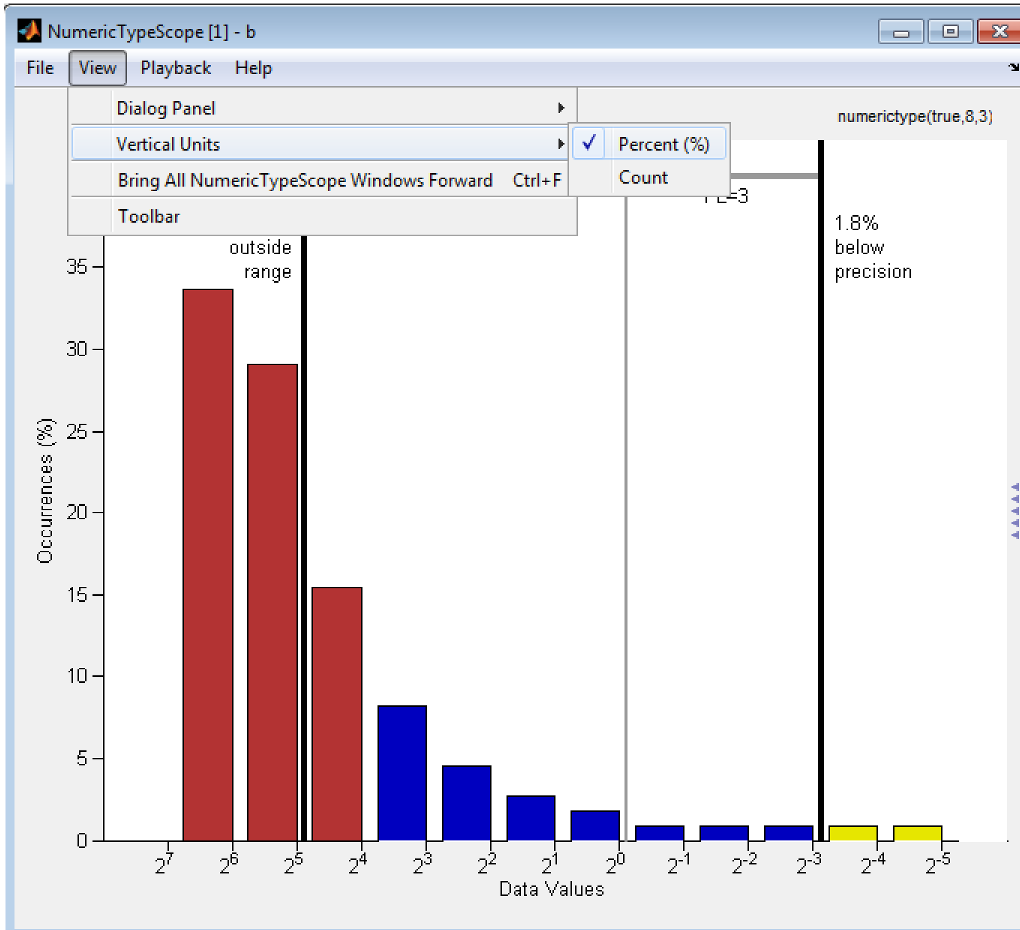
Input Data

The Input Data panel provides statistical information about the values currently displayed in the `NumericScopeType` object.

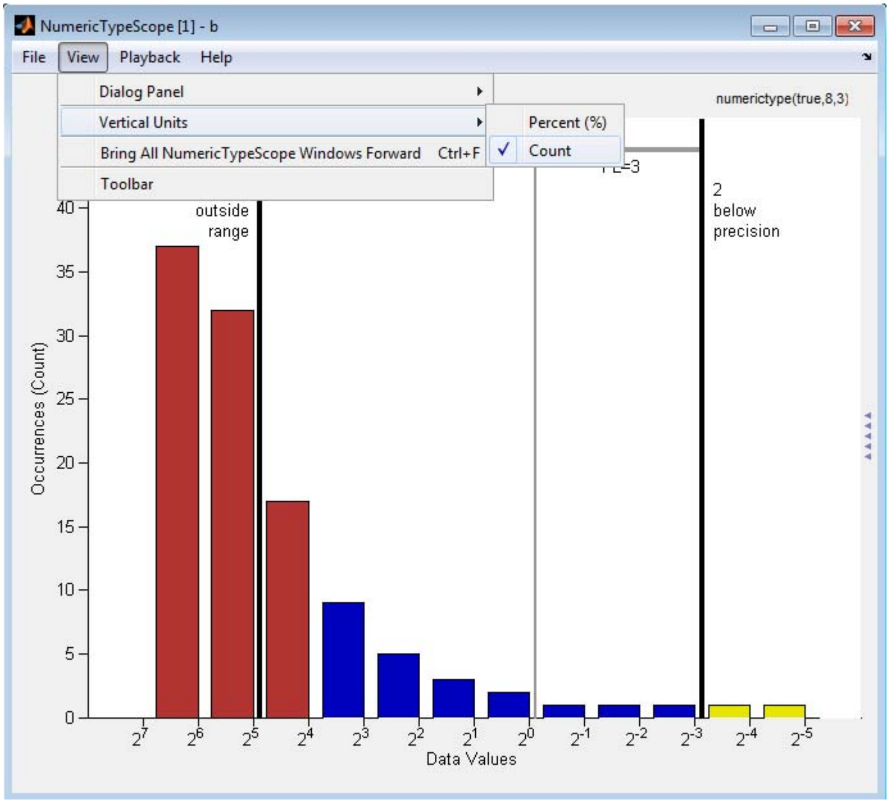


Vertical Units

Use the Vertical Units selection to display values that are outside range or below precision as a percentage or as an actual count. For example, the following image shows the values that are outside range or below precision as a percentage of the total values.

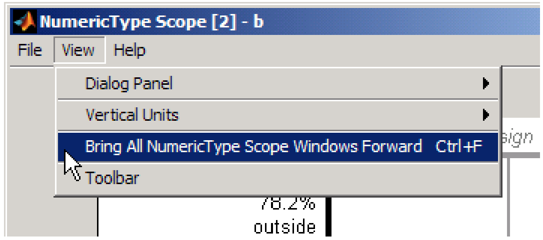


This next example shows the values that are outside range or below precision as an actual count.



Bring All NumericType Scope Windows Forward

The `NumericScopeType` GUI offers a **View > Bring All NumericType Scopes Forward** menu option to help you manage your `NumericTypeScope` windows. Selecting this option or pressing **Ctrl+F** brings all `NumericTypeScope` windows into view. If a `NumericTypeScope` window is not currently open, this menu option opens the window and brings it into view.



Toolbar (Mac Only)

Activate the Toolbar by selecting **View > Toolbar**. When this tool is active, you can dock or undock the scope from the GUI.

The toolbar feature is for the Mac only. Selecting **Toolbar** on Windows® and UNIX® versions displays only an empty toolbar. The docking icon always appears in the GUI in the upper-right corner for these versions.

Methods

reset

Use this method to clear the information stored in the object *H*. Doing so allows you to reuse *H* to process data from a different variable.

Example:

```
>>reset(H)
```

show

Use this method to open the scope window and bring it into view.

Example:

```
>>show(H)
```

step

Use this method to process your data and visualize the dynamic range in the scope window.

Example:

```
>>step(H, data)
```

Examples

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;
```

The `log2` histogram display shows that the values appear both outside range and below precision in the variable. In this case, `b` has a data type of `numerictype(1,8,3)`. The `numerictype(1,8,3)` data type provides 5 integer bits (including the signed bit), and 3 fractional bits. Thus, this data type can represent only values between -2^4 and $2^4 - 2^{-3}$ (from `16` to `15.8750`). Given the range and precision of this data type, values greater than 2^4 fall outside the range and values less than 2^{-3} fall below the precision of the data type.

When you examine the `NumericTypeScope` display, you can see that values requiring bits 5, 6, and 7 are outside range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numerictype(0,13,5)`.

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
```

This dynamic range analysis shows that you can represent the entire range of data in the accumulator 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` | `storedInteger`

Purpose

Create array of all ones with fixed-point properties

Syntax

```
X = ones('like',p)
X = ones(n,'like',p)
X = ones(sz1,...,szN,'like',p)
X = ones(sz,'like',p)
```

Description

`X = ones('like',p)` returns a scalar `1` with the same `numerictype`, complexity (real or complex), and `fimath` as `p`.

`X = ones(n,'like',p)` returns an `n`-by-`n` array of ones like `p`.

`X = ones(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of ones like `p`.

`X = ones(sz,'like',p)` returns an array of ones like `p`. The size vector, `sz`, defines `size(X)`.

Input Arguments

n - Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If `n` is zero, `X` is an empty matrix.
- If `n` is negative, it is treated as zero.

Data Types

`double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN - Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines `X` as a sz1-by...-by-szN array.

- If the size of any dimension is zero, `X` is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, `X`, does not include those dimensions.

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

sz - Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, `X` is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, `X`, does not include those dimensions.

Example: `sz = [2,3,4]` defines `X` as a 2-by-3-by-4 array.

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

p - Prototype

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.
Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Examples

2-D Array of Ones With Fixed-Point Attributes

Create a 2-by-3 array of ones with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 2-by-3- array of ones that has the same numerictype properties as `p`.

```
X = ones(2,3,'like',p)

X =

     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Size Defined by Existing Array

Define a 3-by-2 array `A`.

```
A = [1 4 ; 2 5 ; 3 6];

sz = size(A)

sz =

     3     2
```

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create an array of ones that is the same size as `A` and has the same numerictype properties as `p`.

```
X = ones(sz,'like',p)

X =

     1     1
     1     1
     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Square Array of Ones With Fixed-Point Attributes

Create a 4-by-4 array of ones with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of ones that has the same numerictype properties as `p`.

```
X = ones(4, 'like', p)
```

```
X =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Complex Fixed-Point One

Create a scalar fixed-point `1` that is not real valued, but instead is complex like an existing array.

Define a complex `fi` object.

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

Create a scalar `1` that is complex like `p`.

```
X = ones('like',p)
```

```
X =

   1.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
```

```
a = [1.000000000000000  -0.942809041582063   0.333333333333333];

% Define floating-point types
T_float.coeffs = double([]);
T_float.data   = double([]);

% Create a step input using ones with the
% floating-point data type
t = 0:20;
x_float = ones(size(t),'like',T_float.data);

% Initialize the states using zeros with the
% floating-point data type
z_float = zeros(1,2,'like',T_float.data);

% Run the floating-point algorithm
y_float = my_filter(b,a,x_float,z_float,T_float);

% Define fixed-point types
T_fixed.coeffs = fi([],true,8,6);
T_fixed.data   = fi([],true,8,6);

% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t),'like',T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2,'like',T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf','subplot','plot','legend');
clf
subplot(211)
```

```
    plot(t,y_float,'co-',t,y_fixed,'kx-')
    legend('Floating-point output','Fixed-point output');
    title('Step response');
    subplot(212)
    plot(t,y_float - double(y_fixed),'rs-')
    legend('Error')
    figure(gcf)
end
```

See Also

zeros | cast | ones

Related Examples

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

Concepts

- “Workflow for Converting MATLAB Code to Fixed Point at the Command Line”
- “Best Practices for Converting MATLAB Code to Fixed Point at the Command Line”

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.

pcolor

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.

plotmatrix

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 Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode" in the Fixed-Point Designer documentation.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

See Also

`minus` | `mtimes` | `times` | `uminus`

polar

Purpose Plot polar coordinates

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

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 `OverflowAction` and `RoundingMethod` properties associated with `a`. If obeying the `RoundMode` property associated with `a` is not important, try using the `bitshift` function.

The `pow2` function does not support `fi` objects of data type `Boolean`.

The function also does not support the syntax `b = pow2(a)` when `a` is a `fi` object.

Examples

Example 1

In the following example, `a` is a real-valued `fi` object, and `K` is a positive integer.

The `pow2` function shifts the bits of `a` 3 places to the left, effectively multiplying `a` by 2^3.

```
a = fi(pi,1,16,8)
```

```
b = pow2(a,3)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

```
a =

    3.1406

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

b =

   25.1250

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

binary_a =

0000001100100100


binary_b =

0001100100100000
```

Example 2

In the following example, `a` is a real-valued `fi` object, and `K` is a negative integer.

The pow2 function shifts the bits of a 4 places to the right, effectively multiplying a by 2^{-4}.

```
a = fi(pi,1,16,8)
b = pow2(a,-4)
binary_a = bin(a)
binary_b = bin(b)
```

MATLAB returns:

```
a =

    3.1406

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

b =

    0.1953

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

binary_a =

0000001100100100


binary_b =

0000000000110010
```

Example 3

The following example shows the use of `pow2` with a complex `fi` object:

```
format long g
P = fipref('NumericTypeDisplay', 'short');
a = fi(57 - 2i, 1, 16, 8)

a =
                              57 -                         2i
      s16,8

pow2(a, 2)

ans =
                 127.99609375 -                        8i
      s16,8
```

See Also

bitshift | 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* has the same local fimath as the input *a*. If *a* has no local fimath, the output *c* also has no local fimath. The array power operation is performed using default fimath settings.

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 Quantize fixed-point numbers

Syntax

```
y = quantize(x)
y = quantize(x,nt)
y = quantize(x,nt,rm)
y = quantize(x,nt,rm,oa)

yBP = quantize(x,s)
yBP = quantize(x,s,wl)
yBP = quantize(x,s,wl,fl)
yBP = quantize(x,s,wl,fl,rm)
yBP = quantize(x,s,wl,fl,rm,oa)
```

Description

`y = quantize(x)` quantizes `x` using these default values:

- `numerictype (true,16,15)`
- `Floor` rounding method
- `Wrap` overflow action

The `numerictype`, rounding method, and overflow action apply only during the quantization. The resulting value, quantized `y`, does not have any fimath attached to it.

`y = quantize(x,nt)` quantizes `x` to the specified `numerictype` `nt`. The rounding method and overflow action use default values.

`y = quantize(x,nt,rm)` quantizes `x` to the specified `numerictype`, `nt` and rounding method, `rm`. The overflow action uses the default value.

`y = quantize(x,nt,rm,oa)` quantizes `x` to the specified `numerictype`, `nt`, rounding method, `rm`, and overflow action, `oa`.

`yBP = quantize(x,s)` quantizes `x` to a binary-point, scaled fixed-point number. The `s` input specifies the sign to be used in `numerictype (s,16,15)`. Unspecified properties use these default values:

- `WordLength` 16
- `FractionLength` 15
- `RoundingMethod` `Floor`
- `OverflowAction` `Wrap`

`yBP = quantize(x,s,wl)` uses the specified word length, `wl`. The fraction length defaults to `wl`–1. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl)` uses the specified fraction length, `fl`. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl,rm)` uses the specified rounding method, `rm`. Unspecified properties use default values.

`yBP = quantize(x,s,wl,fl,rm,oa)` uses the specified overflow action, `oa`.

Input Arguments

x - Input data

fi objects or built-in integers

Input data to quantize. Valid inputs are:

- Built-in signed or unsigned integers (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`)
- Binary point scaled fixed-point fi
- Slope-bias scaled fixed-point fi

Although fi doubles and fi singles are allowed as inputs, they pass through the `quantize` function without being quantized.

nt - Numerictype

`(true,16,15)` (default)

Numerictype object that defines the sign, word length, and fraction length of a fixed-point number.

rm - Rounding method
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding method to use

oa - Overflow action
`Wrap` (default) | `Saturate`

Action to take when a data overflow occurs

s - Signedness
`true` (default) | `false`

Whether the fixed-point number is signed (`true`) or unsigned (`false`)

wl - Word length
`16` (default)

Word length of the fixed-point number

fl - Fraction length
`15` (default)

Fraction length of the fixed-point number

Output Arguments

y - Quantized output

Quantized value of the input

yBP - Quantized output

Input quantized to binary-point scaled value

Examples

Quantize Binary-Point Scaled to Binary-Point Scaled Data

Create `numerictype` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numerictype(1,8,4);
```

Define the input.

```
x_BP = fi(pi)

x_BP =

    3.1416

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

Use the defined numerictype, ntBP, to quantize the input, x_BP, to a binary-point scaled data type.

```
yBP1 = quantize(x_BP,ntBP)

yBP1 =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Quantize Binary-Point Scaled to Slope-Bias Data

Create a numerictype object, ntSB, which specifies a slope-bias data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
      'SlopeAdjustmentFactor',1.8,'Bias',...
      1,'FixedExponent',-12);
```

Define the input.

```
x_BP = fi(pi)

x_BP =

    3.1416

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

Use the defined `numerictype`, `ntSB`, to quantize the input, `x_BP`, to a slope-bias data type.

```
ySB1 = quantize(x_BP, ntSB)

ySB1 =

    3.1410

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.000439453125
                  Bias: 1
```

Quantize Slope-Bias Scaled to Binary-Point Scaled Data

Create a `numerictype` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numerictype(1,8,4);
```

Define the input.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```
x_SB =

    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3.0517578125e-5
                  Bias: -0.125
```

Use the defined `numerictype`, `ntBP`, to quantize the input, `x_SB`, to a binary point scaled data type.

```
yBP2 = quantize(x_SB,ntBP,'Nearest','Saturate')

yBP2 =

    0.8125    0.1250    0.1875
    0.8750    0.2500    0.8750
    0.1250    0.5625    0.8750
    0.8750    0.8750    0.5000
    0.6250    0.8750    0.8125

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Quantize Slope-Bias Scaled to Slope-Bias Scaled Data

Create a `numerictype` object, `ntSB`, which specifies a slope-bias data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
```

```
'SlopeAdjustmentFactor',1.8,'Bias',...
1,'FixedExponent',-12);
```

Define the input.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))

x_SB =

    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3.0517578125e-5
                  Bias: -0.125
```

Use the defined `numerictype`, `ntSB`, to quantize the input, `x_SB`, to a slope-bias data type.

```
ySB2 = quantize(x_SB,ntSB,'Ceiling','Wrap')

ySB2 =

    0.8150    0.0978    0.1580
    0.8752    0.2789    0.8752
    0.1272    0.5469    0.8752
    0.8752    0.8752    0.4854
    0.6326    0.8752    0.8005

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.000439453125
```

```
                    Bias: 1
```

Quantize Built-in Integer to Binary-Point Scaled Data

Create a `numerictype` object, `ntBP`, which specifies a signed, 8-bit word length, 4-bit fraction length data type.

```
ntBP = numerictype(1,8,4);
```

Define the input.

```
xInt = int8(-16:4:16)
```

```
xInt =

  -16  -12   -8   -4    0    4    8   12   16
```

Use the defined `numerictype`, `ntBP`, to quantize the input`xInt` to a binary point scaled data type.

```
yBP3 = quantize(xInt,ntBP,'Zero')
```

```
yBP3 =

  0    4   -8   -4    0    4   -8   -4    0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Show the range of the quantized output.

```
range(yBP3)
```

```
ans =

   -8.0000    7.9375
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

The first two and last three values are wrapped because they are outside the representable range of the output type.

Quantize Built-in Integer to Slope-Bias Data

Create a `numerictype` object `ntSB`, which specifies a slope-bias data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
      'SlopeAdjustmentFactor',1.8,'Bias',...
      1,'FixedExponent',-12);
```

Define the input.

```
xInt = int8(-16:4:16)
```

```
xInt =

  -16  -12   -8   -4    0    4    8   12   16
```

Use the defined `numerictype`, `ntSB`, to quantize the input, `xInt`, to a slope-bias data type.

```
ySB3 = quantize(xInt,ntSB,'Round','Saturate')
```

```
ySB3 =

   Columns 1 through 6
  -13.4000  -11.9999   -8.0000   -4.0001   -0.0002    4.0001
  Columns 7 through 9
    8.0000   12.0000   15.3996

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
```

```
           WordLength: 16
                Slope: 0.000439453125
                 Bias: 1
```

Show the range of the quantized output.

```
range(ySB3)
```

```
ans =

  -13.4000   15.3996

         DataTypeMode: Fixed-point: slope and bias scaling
           Signedness: Signed
           WordLength: 16
                Slope: 0.000439453125
                 Bias: 1
```

The first and last values saturate because they are at the limits of he representable range of the output type.

See Also

fi | fimath | fixed.Quantizer | numerictype

Related Examples

- “Compute Quantization Error”

Purpose

Apply quantizer object to data

Syntax

```
y = quantize(q, x)
[y1,y2,...] = quantize(q,x1,x2,...)
```

Description

y = quantize(q, x) uses the quantizer object q to quantize x. When x is a numeric array, each element of x is quantized. When x is a cell array, each numeric element of the cell array is quantized. When x is a structure, each numeric field of x is quantized. Quantize does not change nonnumeric elements or fields of x, nor does it issue warnings for nonnumeric values. The output y is a built-in double. When the input x is a structure or cell array, the fields of y are built-in doubles.

[y1,y2,...] = quantize(q,x1,x2,...) is equivalent to

```
y1 = quantize(q,x1), y2 = quantize(q,x2),...
```

The quantizer object states

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

are updated during the call to quantize, and running totals are kept until a call to resetlog is made.

Examples

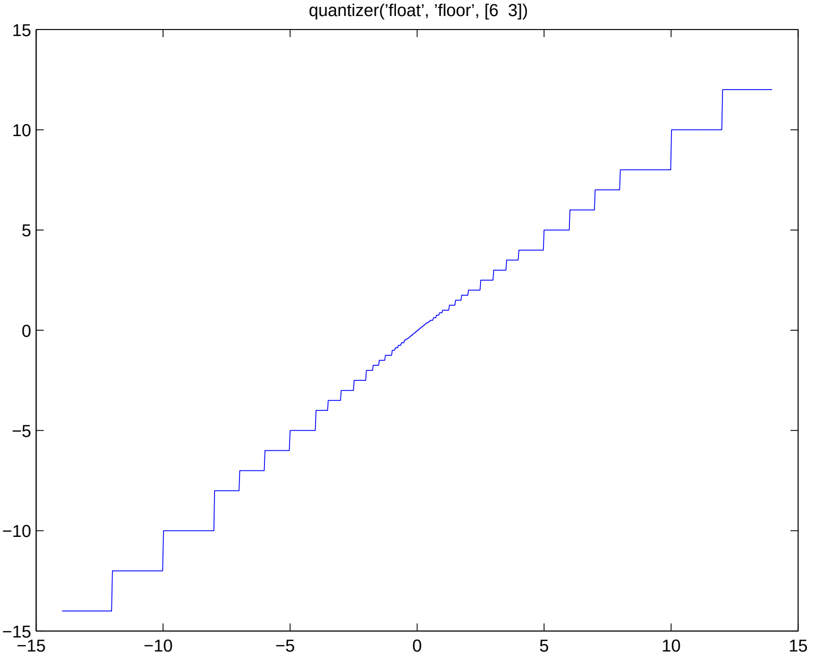
The following examples demonstrate using quantize to quantize data.

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.
```

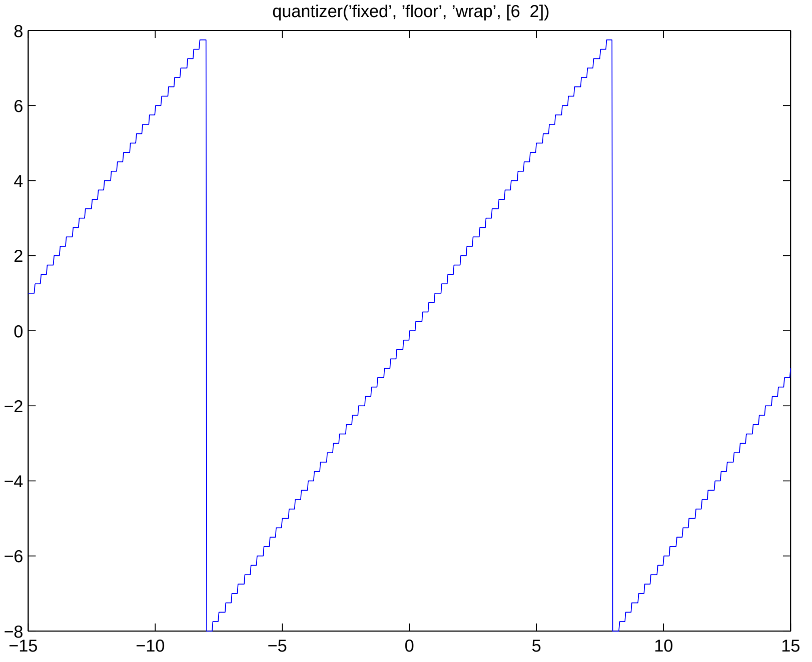


Example 2 - Fixed-Point

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

Purpose Construct `quantizer` object

Syntax

```
q = quantizer
q = quantizer('PropertyName1',PropertyValue1,...)
q = quantizer(PropertyValue1,PropertyValue2,...)
q = quantizer(struct)
q = quantizer(pn,pv)
```

Description

`q = quantizer` creates a `quantizer` object with properties set to their default values. To use this object to quantize values, use the quantize method.

`q = quantizer('PropertyName1',PropertyValue1,...)` uses property name/ property value pairs.

`q = quantizer(PropertyValue1,PropertyValue2,...)` creates a `quantizer` object with the listed property values. When two values conflict, `quantizer` sets the last property value in the list. Property values are unique; you can set the property names by specifying just the property values in the command.

`q = quantizer(struct)`, where `struct` is a structure whose field names are property names, sets the properties named in each field name with the values contained in the structure.

`q = quantizer(pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

The `quantizer` object property values are listed below. These properties are described in detail in “quantizer Object Properties” on page 1-20.

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', 'Ceiling', 'Saturate', [5 4])
```

Using a structure `struct` to set `quantizer` object properties,

```
struct.mode = 'fixed';
struct.roundmode = 'ceil';
struct.overflowmode = 'saturate';
struct.format = [5 4];
q = quantizer(struct);
```

Using property name and property value cell arrays `pn` and `pv` to set `quantizer` object properties,

```
pn = {'mode',  'roundmode', 'overflowmode', 'format'};
pv = {'fixed', 'ceil', 'saturate', [5 4]};
q = quantizer(pn, pv)
```

Using property name/property value pairs to configure a `quantizer` object,

```
q = quantizer( 'mode', 'fixed','roundingmode','ceil',...
'overflowmode', 'saturate', 'format', [5 4]);
```

See Also

`assignmentquantizer` | `fi` | `fimath` | `fipref` | `numerictype` | `quantize` | `set` | `unitquantize` | `unitquantizer`

Purpose Create quiver or velocity plot

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

quiver3

Purpose Create 3-D quiver or velocity plot

Description Refer to the MATLAB `quiver3` reference page for more information.

Purpose

Generate uniformly distributed, quantized random number using `quantizer` object

Syntax

```
randquant(q,n)
randquant(q,m,n)
randquant(q,m,n,p,...)
randquant(q,[m,n])
randquant(q,[m,n,p,...])
```

Description

`randquant(q,n)` uses `quantizer` object `q` to generate an `n`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `n`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,m,n)` uses `quantizer` object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,m,n,p,...)` uses `quantizer` object q to generate an `m`-by-`n`-by-`p`-by ... matrix with random entries whose values cover the range of `q` when `q` is fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the matrix with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,[m,n])` uses `quantizer` object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,[m,n,p,...])` uses `quantizer` object `q` to generate `p` `m`-by-`n` matrices containing random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` arrays with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant` produces pseudorandom numbers. The number sequence `randquant` generates during each call is determined by the state of the generator. Because MATLAB resets the random number generator state at startup, the sequence of random numbers generated by the function remains the same unless you change the state.

`randquant` works like `rng` in most respects.

Examples

```
q=quantizer([4 3]);
rng('default')
randquant(q,3)

ans =

    0.5000    0.6250   -0.5000
    0.6250    0.1250         0
   -0.8750   -0.8750    0.7500
```

See Also

`quantizer` | `rand` | `range` | `realmax`

Purpose

Numerical range of `fi` or `quantizer` object

Syntax

```
range(a)
[min, 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 $a \leq y \leq b$.

`[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
```

Algorithms

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	MATLAB for code generation 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`

real

Purpose Real part of complex number

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

Purpose Largest positive fixed-point value or quantized number

Syntax

```
realmax(a)
realmax(q)
```

Description

`realmax(a)` is the largest real-world value that can be represented in the data type of `fi` object `a`. Anything larger overflows.

`realmax(q)` is the largest quantized number that can be represented where `q` is a `quantizer` object. Anything larger overflows.

Examples

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

x =

    14
```

Algorithms

If `q` is a floating-point `quantizer` object, the largest positive number, x, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point `quantizer` object, the largest positive number, x, is

$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point `quantizer` object (`datamode = 'ufixed'`), the largest positive number, x, is

$$x = \frac{2^w - 1}{2^f}$$

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 positive 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
```

Algorithms

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

removefimath

Purpose

Remove fimath object from `fi` object

Syntax

```
y = removefimath(x)
```

Description

`y = removefimath(x)` returns a `fi` object `y` with `x`'s `numerictype` and value, and no fimath object attached. You can use this function as `y = removefimath(y)`, which gives you localized control over the fimath settings. This function also is useful for preventing errors about `embedded.fimath` of both operands needing to be equal.

Input Arguments

x - Input data

fi object | built-in integer | double | single

Input data, specified as a `fi` object or built-in integer, from which to copy the data type and value to the output. `x` must be a `fi` object or an integer data type (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`). If `x` is not a fi object or integer data type, then `y = x`.

Output Arguments

y - Output fi object

fi object | built-in integer | double | single

Output fi object, returned as a `fi` object with no fimath object attached. The data type and value of the output match the input. If the input, `x`, is not a `fi` object `y = x`.

Examples

Remove fimath Object from fi Object

This example shows how to define a `fi` object, define a fimath object, attach the `fimath` object to the `fi` object and then, remove the attached `fimath` object.

```
a = fi(pi)
f = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
a = setfimath(a,f)
b = removefimath(a)
```

```
a =
```

```
    3.1416

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

a =

    3.1416

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

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

b =

    3.1416

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

Set and Remove fimath for Code Generation

Use the pattern `x = setfimath(x,f)` and `y = removefimath(y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```
function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
   f = fimath('OverflowAction','Wrap',...
```

```
        'RoundingMethod','Floor',...
        'SumMode','KeepLSB',...
        'SumWordLength',32);
    a = setfimath(a,f);
    b = setfimath(b,f);
    y = a + b;
    y = removefimath(y);
end
```

If you have the MATLAB Coder product, you can generate C code. This example generates C code on a computer with 32-bit, native integer type.

```
a = fi(0,1,16,15);
b = fi(0,1,16,15);
codegen fixed_point_32bit_KeepLSB_plus_example...
       -args {a,b} -launchreport

int32_T fixed_point_32bit_KeepLSB_plus_example(int16_T a, int16_T b)
    {
        return a + b;
    }
```

See Also

fi | fimath | setfimath

Purpose Replicate and tile array

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

Purpose

Change scaling of `fi` object

Syntax

```
b = rescale(a, fractionlength)
b = rescale(a, slope, bias)
b = rescale(a, slopeadjustmentfactor, fixedexponent, bias)
b = rescale(a, ..., PropertyName, PropertyValue, ...)
```

Description

The `rescale` function acts similarly to the `fi` copy function with the following exceptions:

- The `fi` copy constructor preserves the real-world value, while `rescale` preserves the stored integer value.
- `rescale` does not allow the `Signed` and `WordLength` properties to be changed.

Examples

In the following example, `fi` object `a` is rescaled to create `fi` object `b`. The real-world values of `a` and `b` are different, while their stored integer values are the same:

```
p = fipref('FimathDisplay','none',...
  'NumericTypeDisplay','short');
a = fi(10,1,8,3)

a =

    10
      s8,3

b = rescale(a,1)

b =

    40
      s8,1

stored_integer_a = storedInteger(a);
```

```
stored_integer_b = storedInteger(b);
isequal(stored_integer_a, stored_integer_b)

ans =

1
```

See Also

fi

reset

Purpose Reset objects to initial conditions

Syntax

```
reset(P)
reset(q)
```

Description

`reset(P)` resets the `fipref` object `P` to its initial conditions.

`reset(q)` resets the following `quantizer` object properties to their initial conditions:

- `minlog`
- `maxlog`
- `noverflows`
- `nunderflows`
- `noperations`

See Also `resetlog`

Purpose

Set global fimath to MATLAB factory default

Syntax

```
resetglobalfimath
```

Description

`resetglobalfimath` sets the global fimath to the MATLAB factory default in your current MATLAB session. The MATLAB factory default has the following properties:

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

Examples

In this example, you create your own `fimath` object `F` and set it as the global fimath. Then, using the `resetglobalfimath` command, reset the global fimath to the MATLAB factory default setting.

```
F = fimath('RoundMode','Floor','OverflowMode','Wrap');
globalfimath(F);
F1 = fimath
a = fi(pi)

F1 =


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

a =

    3.1416
```

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

Now, set the global fimath back to the factory default setting using `resetglobalfimath`:

```
resetglobalfimath;
F2 = fimath
a = fi(pi)

F2 =


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

a =

    3.1416

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

You've now set the global fimath in your current MATLAB session back to the factory default setting. To use the factory default setting of the global fimath in future MATLAB sessions, you must use the `removeglobalfimathpref` command.

Alternatives

`reset(G)` — If `G` is a handle to the global fimath, `reset(G)` is equivalent to using the `resetglobalfimath` command.

See Also

`fimath` | `globalfimath` | `removeglobalfimathpref`

removeglobalfimathpref

Purpose

Remove global fimath preference

Syntax

```
removeglobalfimathpref
```

Description

`removeglobalfimathpref` removes your global fimath from the MATLAB preferences. 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.

Examples

Removing Your Global fimath from the MATLAB Preferences

Typing

```
removeglobalfimathpref;
```

at the MATLAB command line removes your global fimath from the MATLAB preferences. Using the `removeglobalfimathpref` function allows you to:

- Continue using your global fimath in the current MATLAB session
- Use the MATLAB factory default setting of the global fimath in all future MATLAB sessions

To revert back to the MATLAB factory default setting of the global fimath in both your current and future MATLAB sessions, use both the `resetglobalfimath` and the `removeglobalfimathpref` commands:

```
resetglobalfimath;
removeglobalfimath;
```

See Also

`fimath` | `globalfimath` | `resetglobalfimath`

Purpose

Clear log for `fi` or `quantizer` object

Syntax

```
resetlog(a)
resetlog(q)
```

Description

`resetlog(a)` clears the log for `fi` object a.

`resetlog(q)` clears the log for `quantizer` object q.

Turn logging on or off by setting the `fipref` property `LoggingMode`.

See Also

`fipref` | `maxlog` | `minlog` | `noperations` | `noverflows` | `nunderflows` | `reset`

reshape

Purpose Reshape array

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

Purpose Plot colormap

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

Purpose Create ribbon plot

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

Purpose Create angle histogram

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

Purpose

Round `fi` object toward nearest integer or round input data using `quantizer` object

Syntax

```
y = round(a)
y = round(q,x)
```

Description

`y = round(a)` rounds `fi` object `a` to the nearest integer. In the case of a tie, `round` rounds values to the nearest integer with greater absolute value. The rounded value is returned in `fi` object `y`.

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

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

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

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

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

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

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

Examples

Example 1

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

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

a =

    3.1250

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

y = round(a)

y =

     3

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

Example 2

The following example demonstrates how the `round` function affects the `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

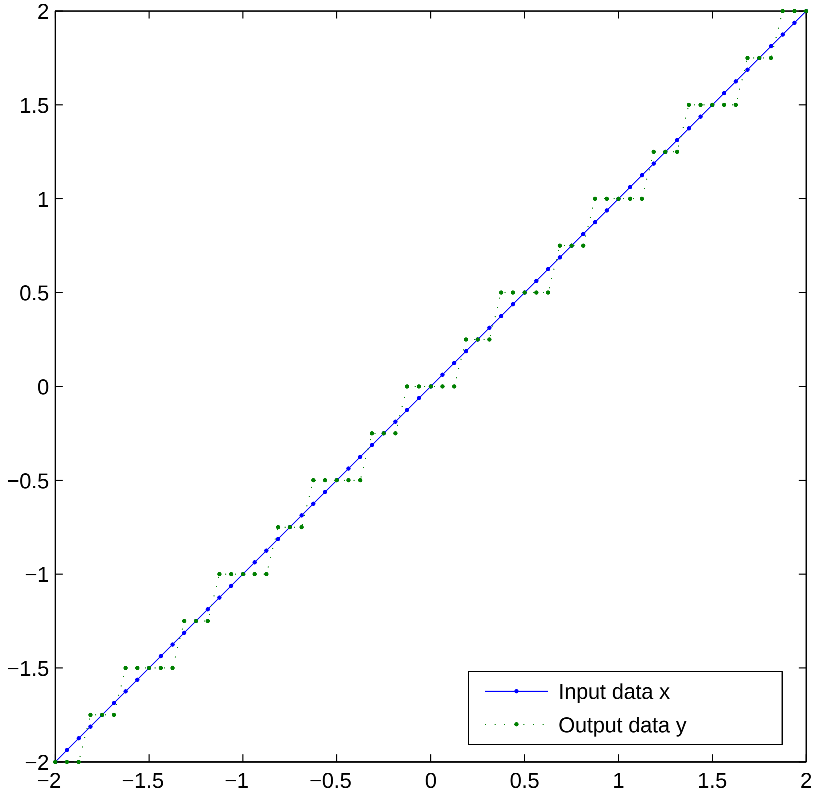
a	convergent(a)	nearest(a)	round(a)
2.5	2	3	3
3.5	4	4	4

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`

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` | `storedInteger` | `oct`

Purpose Create semilogarithmic plot with logarithmic x-axis

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

semilogy

Purpose Create semilogarithmic plot with logarithmic y-axis

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

Purpose

Set or display property values for `quantizer` objects

Syntax

```
set(q, PropertyValue1, PropertyValue2,...)
set(q,s)
set(q,pn,pv)
set(q,'PropertyName1',PropertyValue1,'PropertyName2',
PropertyValue2,...)
q.PropertyName = Value
s = set(q)
```

Description

`set(q, PropertyValue1, PropertyValue2,...)` sets the properties of `quantizer` object `q`. If two property values conflict, the last value in the list is the one that is set.

`set(q,s)`, where `s` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(q,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

`set(q,'PropertyName1',PropertyValue1,'PropertyName2', PropertyValue2,...)` sets multiple property values with a single statement.

Note You can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`q.PropertyName = Value` uses dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of `quantizer` object `q`.

`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`

Purpose

Attach fimath object to `fi` object

Syntax

```
y = setfimath(x,f)
```

Description

`y = setfimath(x,f)` returns a `fi` object, `y`, with `x`'s `numerictype` and value, and attached fimath object, `f`. This function and the related `removefimath` function are useful for preventing errors about `embedded.fimath` of both operands needing to be equal.

The `y` = setfimath(`x`,`f`) syntax does not modify the input, `x`. To modify `x`, use `x = setfimath(x,f)`. If you use `setfimath` in an expression, such as, `a*setfimath(b,f)`, the fimath object is used in the temporary variable, but `b` is not modified.

Input Arguments

x - Input data

fi object | built-in integer | double | single

Input data, specified as a `fi` object or built-in integer value, from which to copy the data type and value to the output. `x` must be a `fi` object or an integer data type (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`). Otherwise, the fimath object is not applied. If `x` is not a `fi` object or integer data type, `y` = `x`.

f - Input `fimath` object

`fimath` object

Input `fimath` object, specified as an existing fimath object to attach to the output. An error occurs if `f` is not a `fimath` object.

Output Arguments

y - Output `fi` object

`fi` object

Output `fi` object, returned as a `fi` object with the same data type and value as the `x` input. `y` also has attached `fimath` object, `f`. If the input, `x`, is not a `fi` object or integer data type, then `y` = `x`.

Examples

Add `fimath` object to `fi` Object

This examples shows how to define a `fi` object, define a `fimath` object, and use `setfimath` to attached the `fimath` object to the `fi` object.

```
a = fi(pi)

a =

    3.1416

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

f = fimath('OverflowAction','Wrap','RoundingMethod','Floor');
b = setfimath(a,f)

b =

    3.1416

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

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

Set and Remove `fimath` for Code Generation

Use the pattern `x = setfimath(x,f)` and `y = removefimath(y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```
function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
   f = fimath('OverflowAction','Wrap',...
      'RoundingMethod','Floor',...
      'SumMode','KeepLSB',...
      'SumWordLength',32);
   a = setfimath(a,f);
   b = setfimath(b,f);
   y = a + b;
   y = removefimath(y);
end
```

If you have the MATLAB Coder product, you can generate C code. This example generates C code on a computer with 32-bit, native integer type.

```
a = fi(0,1,16,15);
b = fi(0,1,16,15);
codegen fixed_point_32bit_KeepLSB_plus_example...
       -args {a,b} -launchreport

int32_T fixed_point_32bit_KeepLSB_plus_example(int16_T a, int16_T b)
    {
        return a + b;
    }
```

See Also

`fi` | `fimath` | `removefimath`

Purpose

Construct signed fixed-point numeric object

Syntax

```
a = sfi
a = sfi(v)
a = sfi(v,w)
a = sfi(v,w,f)
a = sfi(v,w,slope,bias)
a = sfi(v,w,slopeadjustmentfactor,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 2-271
- "fimath Properties" on page 2-561
- "numerictype Properties" on page 2-273

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

Note `fi` objects created by the `sfi` constructor function have no local fimath.

Data Properties

The data properties of a `fi` object are always writable.

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

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

fimath Properties

When you create a `fi` object with the `sfi` constructor function, that `fi` object does not have a local `fimath` object. You can attach a `fimath` object to that `fi` object if you do not want to use the default fimath settings. For more information, see "fimath Object Construction" in the Fixed-Point Designer documentation.

- `fimath` — fixed-point math object

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

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

---

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

- OverflowAction — Action to take on overflow
- ProductBias — Bias of the product data type
- ProductFixedExponent — Fixed exponent of the product data type
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined
- ProductSlope — Slope of the product data type
- ProductSlopeAdjustmentFactor — Slope adjustment factor of the product data type
- ProductWordLength — Word length, in bits, of the product data type
- RoundingMethod — Rounding method
- SumBias — Bias of the sum data type
- SumFixedExponent — Fixed exponent of the sum data type
- SumFractionLength — Fraction length, in bits, of the sum data type
- SumMode — Defines how the sum data type is determined
- SumSlope — Slope of the sum data type
- SumSlopeAdjustmentFactor — Slope adjustment factor of the sum data type
- SumWordLength — The word length, in bits, of the sum data type

These properties are described in detail in "fimath Object Properties" 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 “Cast fi Objects”.

Example 1

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

```
a = sfi(pi,8,3)

a =

    3.1250

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

Default `fimath` properties are associated with `a`. When a `fi` object does not have a local `fimath` object, no `fimath` object properties are displayed in its output. To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)

ans =
     0
```

A returned value of `0` means the `fi` object does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

Example 2

The value `v` can also be an array:

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

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

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

Example 3

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

```
a = sfi(pi,8)

a =

    3.1563

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

Example 4

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

```
a = sfi(pi)

a =

    3.1416

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

See Also

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

Purpose Shift data to operate on specified dimension

Syntax

```
[x,perm,nshifts] = shiftdata(x,dim)
```

Description

`[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first non-singleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples

Example 1

This example shifts `x`, a 3-x-3 magic square, permuting dimension `2` to the first column. `unshiftdata` shifts `x` back to its original shape.

1. Create a 3-x-3 magic square:

```
x = fi(magic(3))

x =

     8     1     6
     3     5     7
     4     9     2
```

2. Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`:

```
x =

     8     3     4
     1     5     9
     6     7     2


perm =

     2     1


nshifts =

     []
```

3. Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2
```

Example 2

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

1. Define `x` as a row vector:

```
x = 1:5
```

```
x =

     1     2     3     4     5
```

2. Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])
```

`x` is returned as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts:

```
x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =

     1
```

3. Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =

     1     2     3     4     5
```

See Also

permute | shiftdim | unshiftdata

Purpose Shift dimensions

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

showfixptsimerrors

Purpose Show overflows from most recent fixed-point simulation

---

Note `showfixptsimerrors` will be removed in a future release. Use `fxptdlg` instead.

Syntax

```
showfixptsimerrors
```

Description The `showfixptsimerrors` script displays any overflows from the most recent fixed-point simulation. This information is also visible in the Fixed-Point Tool.

See Also `autofixexp` | `fxptdlg`

Purpose

Show logged maximum values, minimum values, and overflow data from fixed-point simulation

---

Note `showfixptsimranges` will be removed in a future release. Use `fxptdlg` instead.

Syntax

```
showfixptsimranges
showfixptsimranges(action)
```

Description

`showfixptsimranges` displays the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the MATLAB Command Window.

`showfixptsimranges(action)` stores the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the workspace variable `FixPtSimRanges`. If `action` is `'verbose'`, the logged data also appears in the MATLAB Command Window. If `action` is `'quiet'`, no data appears.

See Also

`autofixexp` | `fxptdlg`

showInstrumentationResults

Purpose

Results logged by instrumented, compiled C code function

Syntax

```
showInstrumentationResults('mex_fcn')
showInstrumentationResults ('mex_fcn' '-options')
showInstrumentationResults mex_fcn
showInstrumentationResults mex_fcn -options
```

Description

`showInstrumentationResults('`*`mex_fcn`*`')` opens the Code Generation Report, showing results from calling the instrumented MEX function *`mex_fcn`*. Hovering over variables and expressions in the report displays the logged information. The logged information includes minimum and maximum values, proposed fraction or word lengths, percent of current range, and whether the value is always a whole number, depending on which options you specify. If you specify to include them in the `buildInstrumentedMex` function, histograms are also included. The same information is displayed in a summary table in the Variables tab.

`showInstrumentationResults ('`*`mex_fcn`*`' '`*`-options`*`')` specifies options for the instrumentation results section of the Code Generation Report.

`showInstrumentationResults` *`mex_fcn`* and `showInstrumentationResults` *`mex_fcn -options`* are alternative syntaxes for opening the Code Generation Report.

When you call `showInstrumentationResults`, a file named `instrumentation/`*`mex_fcn`*`/html/index.html` is created. *`mex_fcn`* is the name of the corresponding instrumented MEX function. Selecting this file opens a web-based version of the Code Generation Report. To open this file from within MATLAB, right-click on the file and select **Open Outside MATLAB**. `showInstrumentationResults` returns an error if the instrumented *`mex_fcn`* has not yet been called.

Input Arguments

mex_fcn

Instrumented MEX function created using `buildInstrumentedMex`.

options

Instrumentation results options.

`-browser`	Open the instrumentation results in a system web browser window. Use this option to open multiple reports so you can compare results.
`-defaultDT` *`T`*	Default data type to propose for double or single data type inputs, where *`T`* is either a numerictype object or one of these strings: `remainFloat`, `double`, `single`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`. If you specify an `int` or `uint`, the signedness and word length are that `int` or `uint` value and a fraction length is proposed. The default is `remainFloat`, which does not propose any data types.
`-optimizeWholeNumbers`	Optimize the word length of variables whose simulation min/max logs indicate that they are always whole numbers.
`-percentSafetyMargin` *`N`*	Safety margin for simulation min/max, where *`N`* is a percent value.
`-printable`	Create and open a printable HTML report.

`-proposeFL`	Propose fraction lengths for specified word lengths. This option is valid only for fi objects with scaled double data types.
`-proposeWL`	Propose word lengths for specified fraction lengths. This option is valid only for fi objects with scaled double data types.

Examples

Generate an instrumented MEX function, then run a test bench. Call `showInstrumentationResults` to open the Code Generation Report.

Note The logged results from `showInstrumentationResults` are an accumulation of all previous calls to the instrumented MEX function. To clear the log, see `clearInstrumentationResults`.

---

1 Create a temporary directory, then import an example function from Fixed-Point Designer.

```
tempdirObj=fidemo.fiTempdir('showInstrumentationResults')
copyfile(fullfile(matlabroot,'toolbox','fixedpoint',...
   'fidemos','fi_m_radix2fft_withscaling.m'),...
   'testfft.m','f')
```

2 Define prototype input arguments.

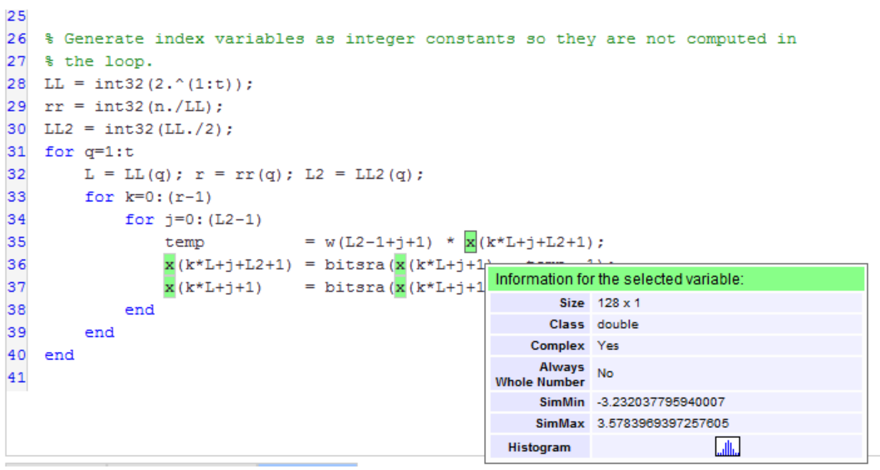
```
T = numerictype('DataType','ScaledDouble','Scaling',...
   'Unspecified');
n = 128;
x = fi(zeros(n,1),T);
W = coder.Constant(fi(fidemo.fi_radix2twiddles(n),T));
```

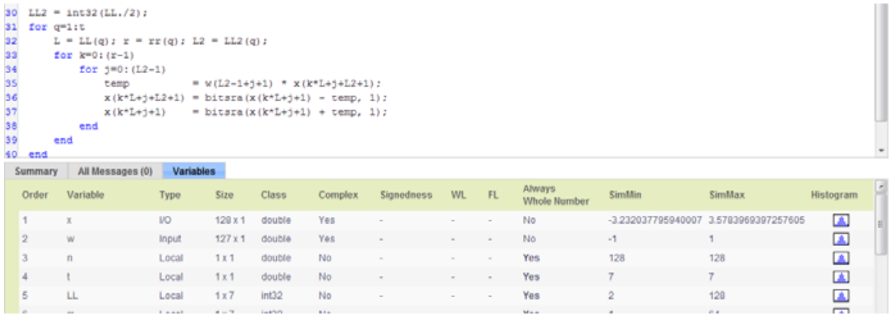
3 Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name.

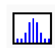
```
buildInstrumentedMex testfft -o testfft_instrumented...
   -args {x,W}
```

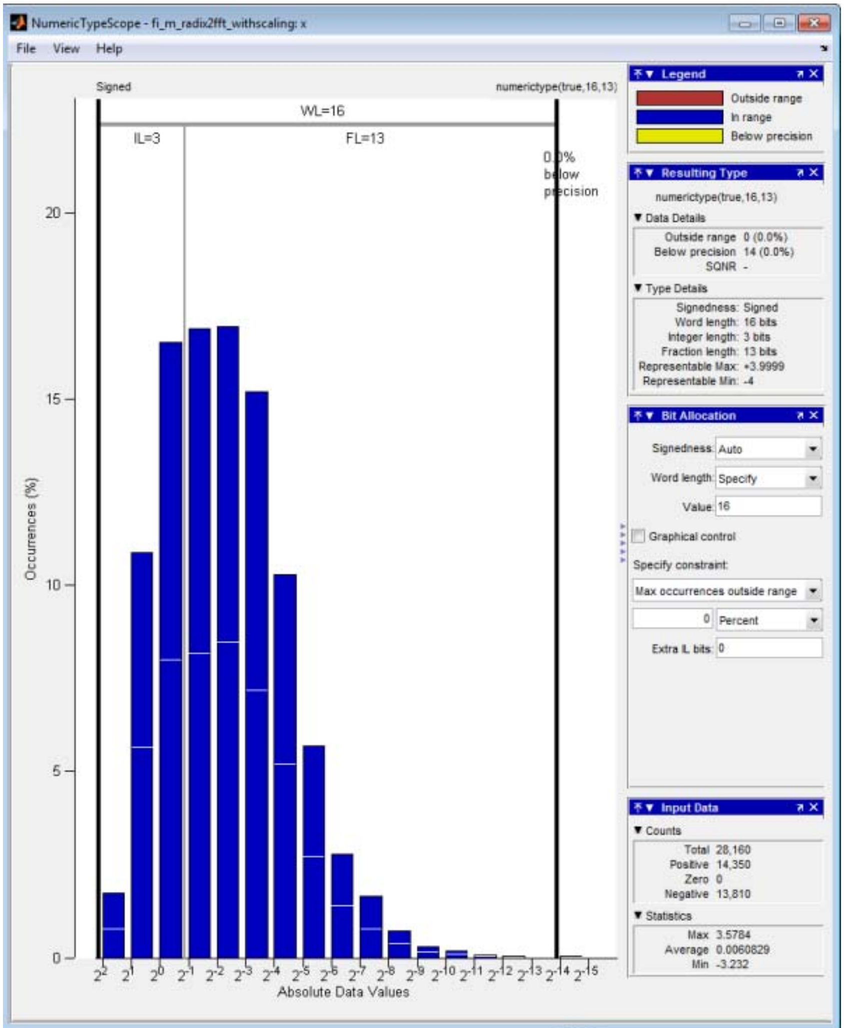
4 Run a test bench to record instrumentation results. Call `showInstrumentationResults` to open the Code Generation Report. View the simulation minimum and maximum values, proposed fraction length, percent of current range, and whole number status by hovering over a variable in the report.

```
for i=1:20
   x(:) = 2*rand(size(x))-1;
   y = testfft_instrumented(x);
end

showInstrumentationResults testfft_instrumented...
   -proposeFL -percentSafetyMargin 10
```





5 View the histogram for a variable by clicking  in the **Variables** tab.



For information on the figure, refer to the `NumericTypeScope` reference page.

6 Close the histogram display and then, clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

7 Clear the MEX function, then delete temporary files.

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

See Also

fiaccel | clearInstrumentationResults | buildInstrumentedMex | NumericTypeScope | codegen | mex

Purpose

Sine of fixed-point values

Syntax

```
y = sin(theta)
```

Description

`y = sin(theta)` returns the sine of fi input `theta` using a table-lookup algorithm.

Input Arguments

theta

`theta` can be a real-valued, signed or unsigned scalar, vector, matrix, or `N`-dimensional array containing the fixed-point angle values in radians. Valid data types of `theta` are:

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

Output Arguments

y

`y` is the sine of `theta`. `y` is a signed, fixed-point number in the range [-1,1]. It has a 16-bit word length and 15-bit fraction length (`numerictype(1,16,15)`) This sine calculation is accurate only to within the top 16 most-significant bits of the input.

Definitions

Sine

The sine of angle Θ is defined as

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

Examples

Calculate the sine of fixed-point input values.

```
theta = fi([-pi/2,-pi/3,-pi/4 0, pi/4,pi/3,pi/2])
```

```
theta =

theta =

   -1.5708  -1.0472  -0.7854  0  0.7854  1.0472  1.5708

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

y = sin(theta)

y =

   -1.0000  -0.8661  -0.7072   0  0.7070  0.8659  0.9999

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

Algorithms

The `sin` function computes the sine of fixed-point input using an 8-bit lookup table as follows:

1. Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
2. Perform a modulo 2π, so the input is in the range [0,2π) radians.
3. Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
4. Use the 8 most-significant bits to obtain the first value from the table.
5. Use the next-greater table value as the second value.

6 Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

See Also

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

Purpose

Perform signum function on array

Syntax

```
c = sign(a)
```

Description

`c = sign(a)` returns an array `c` the same size as `a`, where each element of `c` is

- `1` if the corresponding element of `a` is greater than zero
- `0` if the corresponding element of `a` is zero
- `-1` if the corresponding element of `a` is less than zero

The elements of `c` are of data type `int8`.

`sign` does not support complex `fi` inputs.

Purpose Single-precision floating-point real-world value of `fi` object

Syntax `single(a)`

Description Fixed-point numbers can be represented as

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

or, equivalently as

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

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

See Also `double`

size

Purpose Array dimensions

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

Purpose Create volumetric slice plot

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

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 2-591.

`c = sqrt(a,F)` returns the square root of `fi` object `a`. Intermediate quantities are calculated using the `fimath` object `F`. The `numerictype` object of `c` is determined automatically for you using an internal rule. When `a` is a built-in `double` or `single` data type, this syntax is equivalent to `c = sqrt(a)` and the `fimath` object `F` is ignored.

`c = sqrt(a,T,F)` returns the square root `fi` object `a` with `numerictype` object `T`. Intermediate quantities are also calculated using the `fimath` object `F`. See "Data Type Propagation Rules" on page 2-591.

`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`

squeeze

Purpose Remove singleton dimensions

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

Purpose Create stairstep graph

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

Purpose Plot discrete sequence data

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

Purpose Plot 3-D discrete sequence data

Description Refer to the MATLAB `stem3` reference page for more information.

storedInteger

Purpose

Stored integer value of `fi` object

Syntax

`st_int = storedInteger(f)`

Description

`st_int = storedInteger(f)` returns the stored integer value of fi object `f`.

Fixed-point numbers can be represented as

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

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.

Input Arguments

f - fi object

Output Arguments

st_int - Stored integer value of fi object.

integer

Data Types

`int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

The returned stored integer value is the smallest built-in integer data type in which the stored integer value `f` fits. Signed fi values return stored integers of type `int8`, `int16`, `int32`, or `int64`. Unsigned fi values return stored integers of type `uint8`, `uint16`, `uint32`, or `uint64`. The return type is determined based on the stored integer word length (WL):

- WL $\leq$ 8 bits, the return type is `int8` or `uint8`.
- 8 bits < WL $\leq$ 16 bits, the return type is `int16` or `uint16`.

- 16 bits < WL ≤ 32 bits, the return type is `int32` or `uint32`.
- 32 bits < WL ≤ 64 bits, the return type is `int64` or `uint64`.

Note When the word length is greater than 64 bits, the `storedInteger` function errors. For bit-true integer representation of very large word lengths, use `bin`, `oct`, `dec`, `hex`, or `sdec`.

Examples

Stored Integer Value of fi Objects

Find the stored integer values for two fi objects. Use the `class` function to display the stored integer data types.

```
x = fi([0.2 0.3 0.5 0.3 0.2]);
in_x = storedInteger(x);
c1 = class(in_x)

numtp = numerictype('WordLength',17);
x_n = fi([0.2 0.3 0.5 0.3 0.2],'numerictype',numtp);
in_xn = storedInteger(x_n);
c2 = class(in_xn)
```

See Also

`int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `storedIntegerToDouble`

storedIntegerToDouble

Purpose

Convert stored integer value of fi object to built-in double value

Syntax

```
d = storedIntegerToDouble(f)
```

Description

`d = storedIntegerToDouble(f)` converts the stored integer value of `fi` object, `f`, to a double-precision floating-point value, `d`.

If the input word length is greater than 52 bits, a quantization error may occur. `INF` is returned if the stored integer value of the input `fi` object is outside the representable range of built-in double values.

Input Arguments

f

fi object

Examples

Convert Stored Integer Value of fi Object to Double-Precision Value

Convert the stored integer of a `fi` value to a double-precision value. Use the `class` function to verify that the stored integer is a double-precision value.

```
f = fi(pi,1,16,12);
d = storedIntegerToDouble(f);
dtype = class(d)
```

See Also

`storedInteger` | `fi`

Purpose Create 3-D stream ribbon plot

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

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` has no local fimath.

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

Examples

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

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

c =

    0.4233


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

Algorithms

`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

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: SpecifyPrecision
         SumWordLength: 32
     SumFractionLength: 16
         CastBeforeSum: true
```

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

See Also

`add` | `divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide`

subsasgn

Purpose Subscripted assignment

Syntax

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

Description

`a(I) = b` assigns the values of `b` into the elements of `a` specified by the subscript vector `I`. `b` must have the same number of elements as `I` or be a scalar value.

`a(I,J) = b` assigns the values of `b` into the elements of the rectangular submatrix of `a` specified by the subscript vectors `I` and `J`. `b` must have `LENGTH(I)` rows and `LENGTH(J)` columns.

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

For multidimensional arrays, `a(I,J,K,...) = b` assigns `b` to the specified elements of `a`. `b` must be `length(I)`-by-`length(J)`-by-`length(K)`-... or be shiftable to that size by adding or removing singleton dimensions.

`a = subsasgn(a,S,b)` is called for the syntax `a(i)=b`, `a{i}=b`, or `a.i=b` when `a` is an object. `S` is a structure array with the following fields:

- type — String containing `'()'`, `'{}'`, or `'.'` specifying the subscript type
- subs — Cell array or string containing the actual subscripts

For instance, the syntax `a(1:2,:) = b` calls `a=subsasgn(a,S,b)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

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);
```

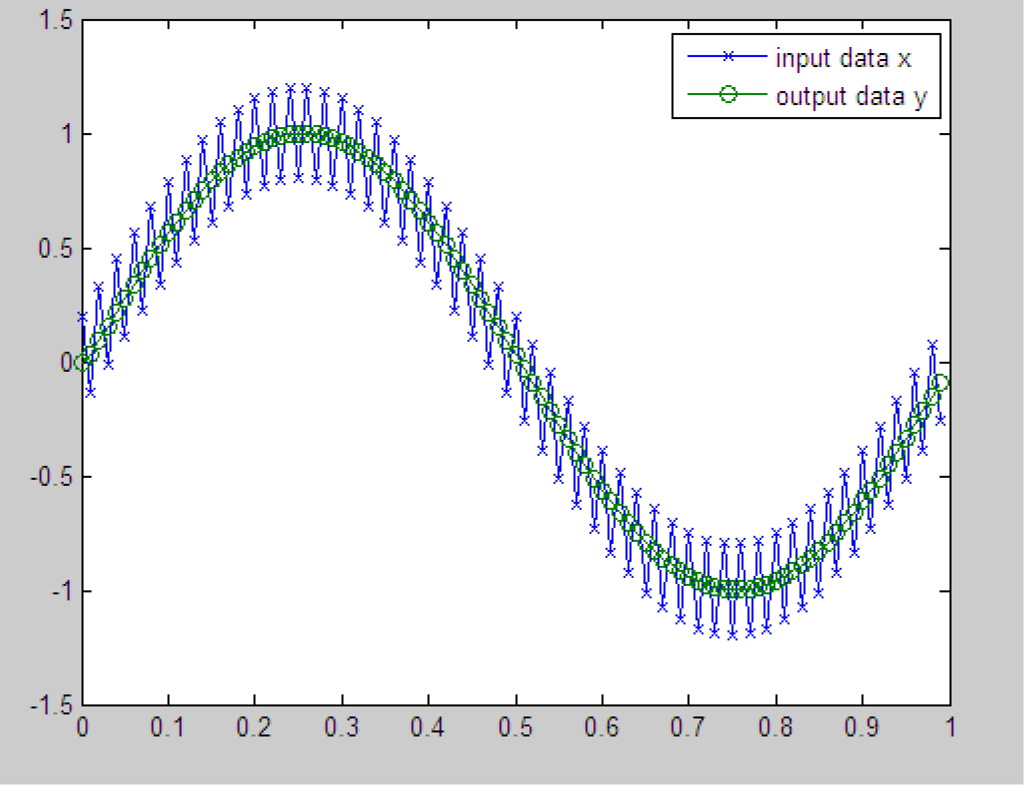
The following loop takes a running average of the input x using the coefficients in b. Notice that acc is assigned into acc(1) = ... versus using acc = ..., which would overwrite and change the data type of acc.

```
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 Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode" in the Fixed-Point Designer documentation.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

See Also `plus` | `minus` | `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` has no local 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` | `numerictype` | `quantizer`

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.

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 2-271
- "fimath Properties" on page 2-631
- "numerictype Properties" on page 2-273

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

Note `fi` objects created by the `ufi` constructor function have no local fimath.

Data Properties

The data properties of a `fi` object are always writable.

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

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

fimath Properties

When you create a `fi` object with the `ufi` constructor function, that `fi` object does not have a local `fimath` object. You can attach a `fimath` object to that `fi` object if you do not want to use the default fimath settings. For more information, see "fimath Object Construction" in the Fixed-Point Designer documentation.

- `fimath` — fixed-point math object

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

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

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

- OverflowAction — Action to take on overflow
- ProductBias — Bias of the product data type
- ProductFixedExponent — Fixed exponent of the product data type
- ProductFractionLength — Fraction length, in bits, of the product data type
- ProductMode — Defines how the product data type is determined
- ProductSlope — Slope of the product data type
- ProductSlopeAdjustmentFactor — Slope adjustment factor of the product data type
- ProductWordLength — Word length, in bits, of the product data type
- RoundingMethod — Rounding method
- SumBias — Bias of the sum data type
- SumFixedExponent — Fixed exponent of the sum data type
- SumFractionLength — Fraction length, in bits, of the sum data type
- SumMode — Defines how the sum data type is determined
- SumSlope — Slope of the sum data type
- SumSlopeAdjustmentFactor — Slope adjustment factor of the sum data type
- SumWordLength — The word length, in bits, of the sum data type

These properties are described in detail in “fimath Object Properties” 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 “View Fixed-Point Data”.

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

Example 1

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

```
a = ufi(pi,8,3)

a =

    3.1250

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

Default fimath properties are associated with a. When a fi object does not have a local fimath object, no fimath object properties are displayed in its output. To determine whether a fi object has a local fimath object, use the isfimathlocal function.

```
isfimathlocal(a)

ans =
     0
```

A returned value of 0 means the fi object does not have a local fimath object. When the isfimathlocal function returns a 1, the fi object has a local fimath object.

Example 2

The value v can also be an array:

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

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

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

Example 3

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

```
a = ufi(pi,8)

a =

    3.1406

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 6
```

Example 4

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

```
a = ufi(pi)

a =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 14
```

See Also

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

Purpose

Convert fi object to unsigned 8-bit integer

Syntax

```
c = uint8(a)
```

Description

`c = uint8(a)` returns the built-in `uint8` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint8`.

Examples

This example shows the `uint8` values of a fi object.

```
a = fi([-pi 0.5 pi],0,8);
c = uint8(a)

c =

    0    1    3
```

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint16` | `uint32` | `uint64`

uint16

Purpose

Convert `fi` object to unsigned 16-bit integer

Syntax

```
c = uint16(a)
```

Description

`c = uint16(a)` returns the built-in `uint16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint16`.

Examples

This example shows the `uint16` values of a fi object.

```
a = fi([-pi 0.5 pi],0,16);
c = uint16(a)

c =

    0    1    3
```

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint32` | `uint64`

Purpose Stored integer value of `fi` object as built-in `uint32`

Syntax

```
c = uint32(a)
```

Description `c = uint32(a)` returns the built-in `uint32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint32`.

Examples This example shows the `uint32` values of a fi object.

```
a = fi([-pi 0.5 pi],0,32);
c = uint32(a)

c =

   0    1    3
```

See Also `storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint64`

uint64

Purpose

Convert `fi` object to unsigned 64-bit integer

Syntax

```
c = uint64(a)
```

Description

`c = uint64(a)` returns the built-in `uint64` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint64`.

Examples

This example shows the `uint64` values of a fi object.

```
a = fi([-pi 0.5 pi],0,64);
c = uint64(a)

c =

    0    1    3
```

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

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

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.

xor

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.

Purpose

Create array of all zeros with fixed-point properties

Syntax

```
X = zeros('like',p)
X = zeros(n,'like',p)
X = zeros(sz1,...,szN,'like',p)
X = zeros(sz,'like',p)
```

Description

`X = zeros('like',p)` returns a scalar `0` with the same `numerictype`, complexity (real or complex), and `fimath` as `p`.

`X = zeros(n,'like',p)` returns an `n`-by-`n` array of zeros like `p`.

`X = zeros(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of zeros like `p`.

`X = zeros(sz,'like',p)` returns an array of zeros like `p`. The size vector, `sz`, defines `size(X)`.

Input Arguments

n - Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If `n` is zero, `X` is an empty matrix.
- If `n` is negative, it is treated as zero.

Data Types

`double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

sz1,...,szN - Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines `X` as a sz1-by...-by-szN array.

- If the size of any dimension is zero, `X` is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, `X`, does not include those dimensions.

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

sz - Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, `X` is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, `X`, does not include those dimensions.

Example: `sz = [2,3,4]` defines `X` as a 2-by-3-by-4 array.

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

p - Prototype

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.
Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Examples

2-D Array of Zeros With Fixed-Point Attributes

Create a 2-by-3 array of zeros with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 2-by-3 array of zeros that has the same numerictype properties as `p`.

```
X = zeros(2,3,'like',p)
```

```
X =

     0     0     0
     0     0     0

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

Size Defined by Existing Array

Define a 3-by-2 array `A`.

```
A = [1 4 ; 2 5 ; 3 6];

sz = size(A)

sz =

     3     2
```

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create an array of zeros that is the same size as `A` and has the same numerictype properties as `p`.

```
X = zeros(sz,'like',p)

X =

     0     0
     0     0
     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Square Array of Zeros With Fixed-Point Attributes

Create a 4-by-4 array of zeros with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of zeros that has the same numerictype properties as p.

```
X = zeros(4, 'like', p)
```

```
X =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Complex Fixed-Point Zero

Create a scalar fixed-point 0 that is not real valued, but instead is complex like an existing array.

Define a complex fi object.

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

Create a scalar 1 that is complex like p.

```
X = zeros('like',p)
```

```
X =

   0.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i=1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
```

```
a = [1.000000000000000  -0.942809041582063   0.333333333333333];

% Define floating-point types
T_float.coeffs = double([]);
T_float.data   = double([]);

% Create a step input using ones with the
% floating-point data type
t = 0:20;
x_float = ones(size(t),'like',T_float.data);

% Initialize the states using zeros with the
% floating-point data type
z_float = zeros(1,2,'like',T_float.data);

% Run the floating-point algorithm
y_float = my_filter(b,a,x_float,z_float,T_float);

% Define fixed-point types
T_fixed.coeffs = fi([],true,8,6);
T_fixed.data   = fi([],true,8,6);

% Create a step input using ones with the
% fixed-point data type
x_fixed = ones(size(t),'like',T_fixed.data);

% Initialize the states using zeros with the
% fixed-point data type
z_fixed = zeros(1,2,'like',T_fixed.data);

% Run the fixed-point algorithm
y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

% Compare the results
coder.extrinsic('clf','subplot','plot','legend');
clf
subplot(211)
```

```
    plot(t,y_float,'co-',t,y_fixed,'kx-')
    legend('Floating-point output','Fixed-point output');
    title('Step response');
    subplot(212)
    plot(t,y_float - double(y_fixed),'rs-')
    legend('Error')
    figure(gcf)
end
```

See Also

`cast` | `ones` | `zeros`

Related Examples

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

Concepts

- "Workflow for Converting MATLAB Code to Fixed Point at the Command Line"
- "Best Practices for Converting MATLAB Code to Fixed Point at the Command Line"

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 MathWorks products that have fixed-point support.

arithmetic shift

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias

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

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

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 Designer software.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000
0.000
00.00
000.0
0000.

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `0`.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, signedness, word length

data type override

Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

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

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

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

where the slope can be expressed as

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

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

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

full range

The broadest range available for a data type. From $-\infty$ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from `-128` to `127` for a signed 8–bit integer. Also known as representable range.

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See also binary word, bit, overflow

incorrect range

A range that is too restrictive and does not include values that can actually occur in the model element. A range that is too broad is not considered incorrect because it will not lead to overflow.

See also range analysis

integer

1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

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

model element

Entities in a model that range analysis software tracks, for example, blocks, signals, parameters, block internal data (such as accumulators, products).

See also range analysis

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Designer software.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

0000__.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$ __.

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number

of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

Qm.n

where

- *Q* indicates that the number is in Q format.
- *m* is the number of bits used to designate the two's complement integer part of the number.
- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, full range, precision, representable range

range analysis

Static analysis of model to derive minimum and maximum range values for elements in the model. The software statically analyzes the ranges of the individual computations in the model based on specified design ranges, inputs, and the semantics of the calculation.

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

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 integer

representable range

The broadest range available for a data type. From –∞ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from `-128` to `127` for a signed 8–bit integer. Also known as full range.

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaled double

A double data type that retains fixed-point scaling information. For example, in Simulink and Fixed-Point Designer software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

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

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, signedness, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, signedness, unsigned fixed-point

signedness

The signedness of a number or data type can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

See also data type, sign bit, sign/magnitude representation, signed fixed-point, unsigned fixed-point

slope

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

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]

Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer

See **integer**

trivial scaling

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$real\text{-}world\ value = stored\ integer$$

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 Designer software.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation

Selected Bibliography

[1] Burrus, C.S., J.H. McClellan, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[2] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems, Second Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[3] *Handbook For Digital Signal Processing*, edited by S.K. Mitra and J.F. Kaiser, John Wiley & Sons, Inc., New York, 1993.

[4] Hanselmann, H., "Implementation of Digital Controllers — A Survey," *Automatica*, Vol. 23, No. 1, pp. 7-32, 1987.

[5] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*, Kluwer Academic Publishers, Seventh Printing, Norwell, Massachusetts, 1993.

[6] Middleton, R. and G. Goodwin, *Digital Control and Estimation — A Unified Approach*, Prentice Hall, Englewood Cliffs, New Jersey. 1990.

[7] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/index.html`.

[8] Ogata, K., *Discrete-Time Control Systems*, *Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[9] Roberts, R.A. and C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

Index

A

abs function 2-2
accelerating fixed-point code 2-280
add function 2-17
all function 2-19
and function 2-20
any function 2-21
area function 2-22
assignmentquantizer function 2-23
autofixexp function 2-29
automatic scaling
    autofixexp 2-29
    fixpt_instrument_purge 2-310
    script 2-29
autoscaling
    autofixexp 2-29
    fixpt_instrument_purge 2-310
    script 2-29

B

bar function 2-31
barh function 2-32
Bias property 1-15
bin function 2-33
bin property 1-2
bin2num function 2-34
bitand function 2-36
bitandreduce function 2-37
bitcmp function 2-39
bitconcat function 2-40
bitget function 2-42
bitor function 2-44
bitorreduce function 2-46
bitreplicate function 2-48
bitrol function 2-50
bitror function 2-52
bitset function 2-54
bitshift function 2-55
bitsliceget function 2-58
bitsll function 2-60
bitsra function 2-62
bitsrl function 2-64
bitxor function 2-66
bitxorreduce function 2-68
buffer function 2-70

C

CastBeforeSum property 1-4
ceil function 2-87
clabel function 2-90
code generation from MATLAB
    resolving extrinsic function calls in generated
        code 2-110
coder.inline function 2-117
coder.nullcopy function 2-138
coder.target function 2-162
comet function 2-178
comet3 function 2-179
compass function 2-180
complex function 2-181
coneplot function 2-182
conj function 2-183
contour function 2-184
contour3 function 2-185
contourc function 2-186
contourf function 2-187
Controlling inlining in generated code 2-117
conv function 2-188
convergent function 2-190
copyobj function 2-193
cordicabs function 2-194
cordicangle function 2-197
cordicatan2 function 2-200
cordiccart2pol function 2-203
cordiccexp function 2-207
cordiccos function 2-210
cordicpol2cart function 2-215
cordicrotate function 2-219

cordicsin function 2-223
cordicsincos function 2-228
ctranspose function 2-234

D

data property 1-2
DataType property 1-15
DataTypeMode property 1-15
DataTypeOverride property 1-12 1-16
DataTypeOverrideAppliesTo property 1-12
dec function 2-235
dec property 1-2
Declaring uninitialized variables 2-138
denormalmax function 2-236
denormalmin function 2-237
Determining code generation target 2-162
diag function 2-238
disp function 2-239
div function 2-240
double function 2-245
double property 1-2

E

end function 2-246
eps function 2-247
eq function 2-248
errmean function 2-249
errorbar function 2-250
errpdf function 2-251
errvar function 2-254
etreeplot function 2-255
exponentbias function 2-256
exponentlength function 2-257
exponentmax function 2-258
exponentmin function 2-259
ezcontour function 2-260
ezcontourf function 2-261
ezmesh function 2-262
ezplot function 2-263
ezplot3 function 2-264
ezpolar function 2-265
ezsurf function 2-266
ezsurfc function 2-267

F

feather function 2-268
fi function 2-269
fi objects
  properties
    bin 1-2
    data 1-2
    dec 1-2
    double 1-2
    fimath 1-2
    hex 1-3
    NumericType 1-3
    oct 1-3
fiaccel function 2-280
filter function 2-286
fimath function 2-291
fimath objects
  properties
    CastBeforeSum 1-4
    MaxProductWordLength 1-4
    MaxSumWordLength 1-4
    OverflowMode 1-4
    ProductBias 1-5
    ProductFixedExponent 1-5
    ProductFractionLength 1-5
    ProductMode 1-5
    ProductSlope 1-7
    ProductSlopeAdjustmentFactor 1-7
    ProductWordLength 1-7
    RoundMode 1-8
    SumBias 1-8
    SumFixedExponent 1-8
    SumFractionLength 1-9

SumMode 1-9
SumSlope 1-11
SumSlopeAdjustmentFactor 1-11
SumWordLength 1-11
fimath property 1-2
FimathDisplay property 1-12
fipref function 2-294
fipref objects
properties
DataTypeOverride 1-12
DataTypeOverrideAppliesTo 1-12
FimathDisplay 1-12
LoggingMode 1-13
NumberDisplay 1-14
NumericTypeDisplay 1-13
fix function 2-297
FixedExponent property 1-16
fixpt_instrument_purge function 2-310
flipdim function 2-311
fliplr function 2-312
flipud function 2-313
floor function 2-314
format
rat 1-14
Format property 1-20
fplot function 2-317
fractionlength function 2-318
FractionLength property 1-17
function
line 2-390
functions
abs 2-2
add 2-17
all 2-19
and 2-20
any 2-21
area 2-22
assignmentquantizer 2-23
autofixexp 2-29
bar 2-31
barh 2-32
bin 2-33
bin2num 2-34
bitand 2-36
bitandreduce 2-37
bitcmp 2-39
bitconcat 2-40
bitget 2-42
bitor 2-44
bitorreduce 2-46
bitreplicate 2-48
bitrol 2-50
bitror 2-52
bitset 2-54
bitshift 2-55
bitsliceget 2-58
bitsll 2-60
bitsra 2-62
bitsrl 2-64
bitxor 2-66
bitxorreduce 2-68
buffer 2-70
ceil 2-87
clabel 2-90
comet 2-178
comet3 2-179
compass 2-180
complex 2-181
coneplot 2-182
conj 2-183
contour 2-184
contour3 2-185
contourc 2-186
contourf 2-187
conv 2-188
convergent 2-190
copyobj 2-193
cordicabs 2-194
cordicangle 2-197
cordicatan2 2-200

cordiccart2pol 2-203
cordiccexp 2-207
cordiccos 2-210
cordicpol2cart 2-215
cordicrotate 2-219
cordicsin 2-223
cordicsincos 2-228
ctranspose 2-234
dec 2-235
denormalmax 2-236
denormalmin 2-237
diag 2-238
disp 2-239
div 2-240
double 2-245
end 2-246
eps 2-247
eq 2-248
errmean 2-249
errorbar 2-250
errpdf 2-251
errvar 2-254
etreeplot 2-255
exponentbias 2-256
exponentlength 2-257
exponentmax 2-258
exponentmin 2-259
ezcontour 2-260
ezcontourf 2-261
ezmesh 2-262
ezplot 2-263
ezplot3 2-264
ezpolar 2-265
ezsurf 2-266
ezsurfc 2-267
feather 2-268
fi 2-269
filter 2-286
fimath 2-291
fipref 2-294
fix 2-297
fixpt_instrument_purge 2-310
flipdim 2-311
fliplr 2-312
flipud 2-313
floor 2-314
fplot 2-317
fractionlength 2-318
ge 2-319
get 2-320
getlsb 2-321
getmsb 2-322
gplot 2-325
gt 2-326
hankel 2-327
hex 2-339
hex2num 2-343
hist 2-344
histc 2-345
horzcat 2-346
imag 2-347
int16 2-351
int32 2-352
int64 2-353
int8 2-350
intmax 2-354
intmin 2-355
ipermute 2-356
isboolean 2-357
iscolumn 2-358
isdouble 2-359
isempty 2-360
isequal 2-361
isfi 2-362
isfimath 2-363
isfimathlocal 2-364
isfinite 2-365
isfipref 2-366
isfixed 2-367
isfloat 2-368

isinf 2-369
isnan 2-370
isnumeric 2-371
isnumerictype 2-372
isobject 2-373
isquantizer 2-375
isreal 2-376
isrow 2-377
isscalar 2-378
isscaleddouble 2-379
isscaledtype 2-380
isscalingbinarypoint 2-381
isscalingslopebias 2-382
isscalingunspecified 2-383
issigned 2-384
issingle 2-385
isslopebiasscaled 2-386
isvector 2-387
le 2-388
length 2-389
logical 2-391
loglog 2-392
logreport 2-393
lowerbound 2-394
lsb 2-395
lt 2-396
max 2-397
maxlog 2-398
mean 2-401
median 2-403
mesh 2-405
meshc 2-406
meshz 2-407
min 2-408
minlog 2-409
minus 2-411
mpower 2-414
mpy 2-415
mrdivide 2-417
mtimes 2-419
ndgrid 2-420
ndims 2-421
ne 2-422
nearest 2-423
noperations 2-426
not 2-427
noverflows 2-428
num2bin 2-443
num2hex 2-444
num2int 2-446
numberofelements 2-447
numerictype 2-448
nunderflows 2-470
oct 2-471
or 2-480
patch 2-481
pcolor 2-482
permute 2-483
plot 2-484
plot3 2-485
plotmatrix 2-486
plotyy 2-487
plus 2-488
polar 2-489
pow2 2-490
power 2-494
quantize 2-505
quantizer 2-508
quiver 2-513
quiver3 2-514
randquant 2-515
range 2-517
rdivide 2-519
real 2-522
realmax 2-523
realmin 2-525
reinterpretcast 2-526
removeglobalfimathpref 2-538
repmat 2-531
rescale 2-532

reset 2-534
resetglobalfimath 2-535
resetlog 2-539
reshape 2-540
rgbplot 2-541
ribbon 2-542
rose 2-543
round 2-544
savefipref 2-549
scatter 2-550
scatter3 2-551
sdec 2-552
semilogx 2-553
semilogy 2-554
set 2-555
sfi 2-560
shiftdata 2-567
shiftdim 2-571
showfixptsimerrors 2-572
showfixptsimranges 2-573
sign 2-584
single 2-585
size 2-586
slice 2-587
sort 2-588
spy 2-589
sqrt 2-590
squeeze 2-592
stairs 2-593
stem 2-594
stem3 2-595
storedInteger 2-596
streamribbon 2-599
streamslice 2-600
streamtube 2-601
stripscaling 2-602
sub 2-604
subsasgn 2-606
subsref 2-611
sum 2-612
surf 2-613
surfc 2-614
surfl 2-615
surfnorm 2-616
text 2-617
times 2-618
toeplitz 2-619
tostring 2-622
transpose 2-623
treeplot 2-624
tril 2-625
trimesh 2-626
triplot 2-627
trisurf 2-628
triu 2-629
ufi 2-630
uint16 2-638
uint32 2-639
uint64 2-640
uint8 2-637
uminus 2-641
unitquantize 2-644
unitquantizer 2-646
unshiftdata 2-647
uplus 2-650
upperbound 2-651
vertcat 2-652
voronoi 2-653
voronoin 2-654
waterfall 2-655
wordlength 2-656
xlim 2-657
xor 2-658
ylim 2-659
zlim 2-668

G

ge function 2-319
get function 2-320

getlsb function 2-321
getmsb function 2-322
gplot function 2-325
gt function 2-326

H

hankel function 2-327
hex function 2-339
hex property 1-3
hex2num function 2-343
hist function 2-344
histc function 2-345
horzcat function 2-346

I

imag function 2-347
int16 function 2-351
int32 function 2-352
int64 function 2-353
int8 function 2-350
intmax function 2-354
intmin function 2-355
ipermute function 2-356
isboolean function 2-357
iscolumn function 2-358
isdouble function 2-359
isempty function 2-360
isequal function 2-361
isfi function 2-362
isfimath function 2-363
isfimathlocal function 2-364
isfinite function 2-365
isfipref function 2-366
isfixed function 2-367
isfloat function 2-368
isinf function 2-369
isnan function 2-370
isnumeric function 2-371
isnumerictype function 2-372
isobject function 2-373
isquantizer function 2-375
isreal function 2-376
isrow function 2-377
isscalar function 2-378
isscaleddouble function 2-379
isscaledtype function 2-380
isscalingbinarypoint function 2-381
isscalingslopebias function 2-382
isscalingunspecified function 2-383
issigned function 2-384
issingle function 2-385
isslopebiasscaled function 2-386
isvector function 2-387

L

le function 2-388
length function 2-389
line function 2-390
LoggingMode property 1-13
logical function 2-391
loglog function 2-392
logreport function 2-393
lowerbound function 2-394
lsb function 2-395
lt function 2-396

M

max function 2-397
maxlog function 2-398
MaxProductWordLength property 1-4
MaxSumWordLength property 1-4
mean function 2-401
median function 2-403
mesh function 2-405
meshc function 2-406
meshz function 2-407

min function 2-408
minlog function 2-409
minus function 2-411
Mode property 1-20
mpower function 2-414
mpy function 2-415
mrdivide function 2-417
mtimes function 2-419

N

ndgrid function 2-420
ndims function 2-421
ne function 2-422
nearest function 2-423
nopnerations function 2-426
not function 2-427
noverflows function 2-428
num2bin function 2-443
num2hex function 2-444
num2int function 2-446
NumberDisplay property 1-14
numberofelements function 2-447
numerictype function 2-448
numerictype objects
    properties
        Bias 1-15
        DataType 1-15
        DataTypeMode 1-15
        DataTypeOverride 1-16
        FixedExponent 1-16
        FractionLength 1-17
        Scaling 1-17
        Signed 1-17
        Signedness 1-18
        Slope 1-18
        SlopeAdjustmentFactor 1-19
        WordLength 1-19
NumericType property 1-3
NumericTypeDisplay property 1-13
nunderflows function 2-470

O

oct function 2-471
oct property 1-3
or function 2-480
OverflowAction property
    fimath objects 1-4
    quantizers 1-21

P

patch function 2-481
pcolor function 2-482
permute function 2-483
plot function 2-484
plot3 function 2-485
plotmatrix function 2-486
plotyy function 2-487
plus function 2-488
polar function 2-489
pow2 function 2-490
power function 2-494
ProductBias property 1-5
ProductFixedExponent property 1-5
ProductFractionLength property 1-5
ProductMode property 1-5
ProductSlope property 1-7
ProductSlopeAdjustmentFactor property 1-7
ProductWordLength property 1-7
properties
    Bias, numerictype objects 1-15
    bin, fi objects 1-2
    CastBeforeSum, fimath objects 1-4
    data, fi objects 1-2
    DataType, numerictype objects 1-15
    DataTypeMode, numerictype objects 1-15
    DataTypeOverride, fipref objects 1-12

DataTypeOverride, numerictype objects 1-16
DataTypeOverrideAppliesTo, fipref objects 1-12
dec, fi objects 1-2
double, fi objects 1-2
fimath, fi objects 1-2
FimathDisplay, fipref objects 1-12
FixedExponent, numerictype objects 1-16
Format, quantizers 1-20
FractionLength, numerictype objects 1-17
hex, fi objects 1-3
LoggingMode, fipref objects 1-13
MaxProductWordLength, fimath objects 1-4
MaxSumWordLength, fimath objects 1-4
Mode, quantizers 1-20
NumberDisplay, fipref objects 1-14
NumericType, fi objects 1-3
NumericTypeDisplay, fipref objects 1-13
oct, fi objects 1-3
OverflowAction, quantizers 1-21
OverflowMode, fimath objects 1-4
ProductBias, fimath objects 1-5
ProductFixedExponent, fimath objects 1-5
ProductFractionLength, fimath objects 1-5
ProductMode, fimath objects 1-5
ProductSlope, fimath objects 1-7
ProductSlopeAdjustmentFactor, fimath objects 1-7
ProductWordLength, fimath objects 1-7
RoundingMethod, quantizers 1-22
RoundMode, fimath objects 1-8
Scaling, numerictype objects 1-17
Signed, numerictype objects 1-17
Signedness, numerictype objects 1-18
Slope, numerictype objects 1-18
SlopeAdjustmentFactor, numerictype objects 1-19
SumBias, fimath objects 1-8
SumFixedExponent, fimath objects 1-8
SumFractionLength, fimath objects 1-9
SumMode, fimath objects 1-9
SumSlope, fimath objects 1-11
SumSlopeAdjustmentFactor, fimath objects 1-11
SumWordLength, fimath objects 1-11
WordLength, numerictype objects 1-19

Q

quantize function 2-505
quantizer function 2-508
quantizers
    properties
        Format 1-20
        Mode 1-20
        OverflowAction 1-21
        RoundingMethod 1-22
quiver function 2-513
quiver3 function 2-514

R

randquant function 2-515
range function 2-517
rat format 1-14
rdivide function 2-519
real function 2-522
realmax function 2-523
realmin function 2-525
reinterpretcast function 2-526
removeglobalfimathpref function 2-538
repmat function 2-531
rescale function 2-532
reset function 2-534
resetglobalfimath function 2-535
resetlog function 2-539
reshape function 2-540
rgbplot function 2-541
ribbon function 2-542

rose function 2-543
round function 2-544
RoundingMethod property
quantizers 1-22
RoundMode property
fimath objects 1-8

S

savefipref function 2-549
Scaling property 1-17
scatter function 2-550
scatter3 function 2-551
sdec function 2-552
semilogx function 2-553
semilogy function 2-554
set function 2-555
sfi function 2-560
shiftdata function 2-567
shiftdim function 2-571
showfixptsimerrors function 2-572
showfixptsimranges function 2-573
sign function 2-584
Signed property 1-17
Signedness property 1-18
single function 2-585
size function 2-586
slice function 2-587
Slope property 1-18
SlopeAdjustmentFactor property 1-19
sort function 2-588
spy function 2-589
sqrt function 2-590
squeeze function 2-592
stairs function 2-593
stem function 2-594
stem3 function 2-595
storedInteger function 2-596
streamribbon function 2-599
streamslice function 2-600
streamtube function 2-601
stripscaling function 2-602
sub function 2-604
subsasgn function 2-606
subsref function 2-611
sum function 2-612
SumBias property 1-8
SumFixedExponent property 1-8
SumFractionLength property 1-9
SumMode property 1-9
SumSlope property 1-11
SumSlopeAdjustmentFactor property 1-11
SumWordLength property 1-11
surf function 2-613
surfc function 2-614
surfl function 2-615
surfnorm function 2-616

T

text function 2-617
times function 2-618
toeplitz function 2-619
tostring function 2-622
transpose function 2-623
treeplot function 2-624
tril function 2-625
trimesh function 2-626
triplot function 2-627
trisurf function 2-628
triu function 2-629

U

ufi function 2-630
uint16 function 2-638
uint32 function 2-639
uint64 function 2-640
uint8 function 2-637
uminus function 2-641

`unitquantize` function 2-644
`unitquantizer` function 2-646
`unshiftdata` function 2-647
`uplus` function 2-650
`upperbound` function 2-651

V

`vertcat` function 2-652
`voronoi` function 2-653
`voronoin` function 2-654

W

`waterfall` function 2-655
`wordlength` function 2-656
`WordLength` property 1-19

X

`xlim` function 2-657
`xor` function 2-658

Y

`ylim` function 2-659

Z

`zlim` function 2-668