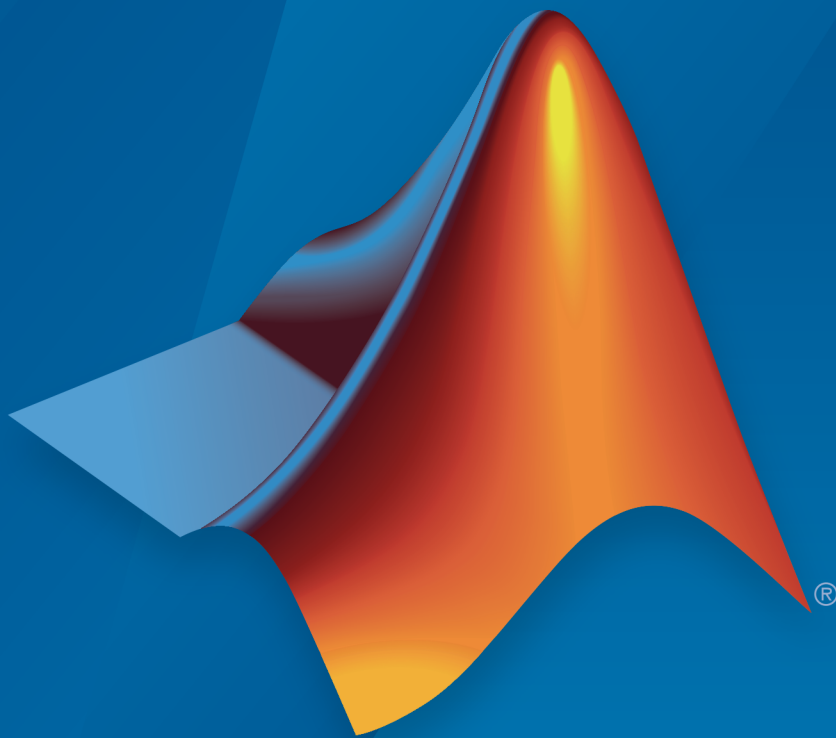


# Computer Vision System Toolbox™

## Reference



# MATLAB® & SIMULINK®

R2016a

 MathWorks®

## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Computer Vision System Toolbox™ Reference*

© COPYRIGHT 2000–2016 by The MathWorks, Inc.

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

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

### **Trademarks**

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

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.



### **Revision History**

April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.0 (Release 2012a)
September 2012	Online only	Revised for Version 5.1 (Release R2012b)
March 2013	Online only	Revised for Version 5.2 (Release R2013a)
September 2013	Online only	Revised for Version 5.3 (Release R2013b)
March 2014	Online only	Revised for Version 6.0 (Release R2014a)
October 2014	Online only	Revised for Version 6.1 (Release R2014b)
March 2015	Online only	Revised for Version 6.2 (Release R2015a)
September 2015	Online only	Revised for Version 7.0 (Release R2015b)
March 2016	Online only	Revised for Version 7.1 (Release R2016a)



<b>1</b>	<b>Blocks — Alphabetical List</b>
<b>2</b>	<b>Alphabetical List</b>
<b>3</b>	<b>Functions Alphabetical</b>



# Blocks — Alphabetical List

---

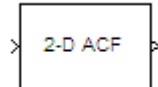
## 2-D Autocorrelation

Compute 2-D autocorrelation of input matrix

### Library

Statistics

visionstatistics



### Description 2-D Autocorrelation

The 2-D Autocorrelation block computes the two-dimensional autocorrelation of the input matrix. Assume that input matrix  $A$  has dimensions  $(Ma, Na)$ . The equation for the two-dimensional discrete autocorrelation is

$$C(i, j) = \sum_{m=0}^{(Ma-1)} \sum_{n=0}^{(Na-1)} A(m, n) \cdot \text{conj}(A(m+i, n+j))$$

where  $0 \leq i < 2Ma - 1$  and  $0 \leq j < 2Na - 1$ .

The output of this block has dimensions  $(2Ma - 1, 2Na - 1)$ .

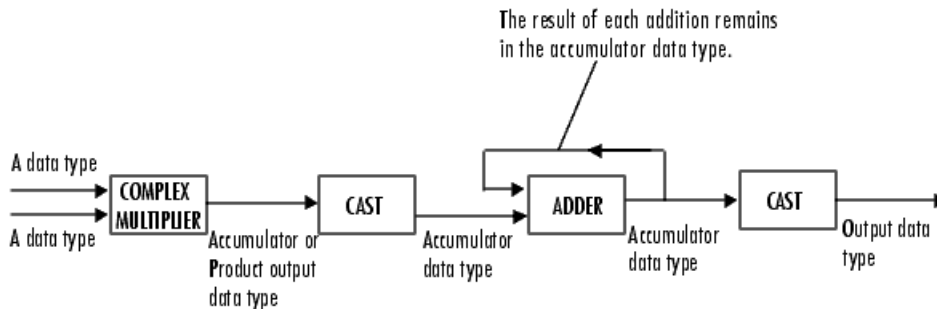
Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values or a scalar, vector, or matrix that represents one plane of the RGB video stream	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes —

Port	Input/Output	Supported Data Types	Complex Values Supported
Output	Autocorrelation of the input matrix	Same as Input port	Yes

If the data type of the input is floating point, the output of the block has the same data type.

## Fixed-Point Data Types

The following diagram shows the data types used in the 2-D Autocorrelation block for fixed-point signals.

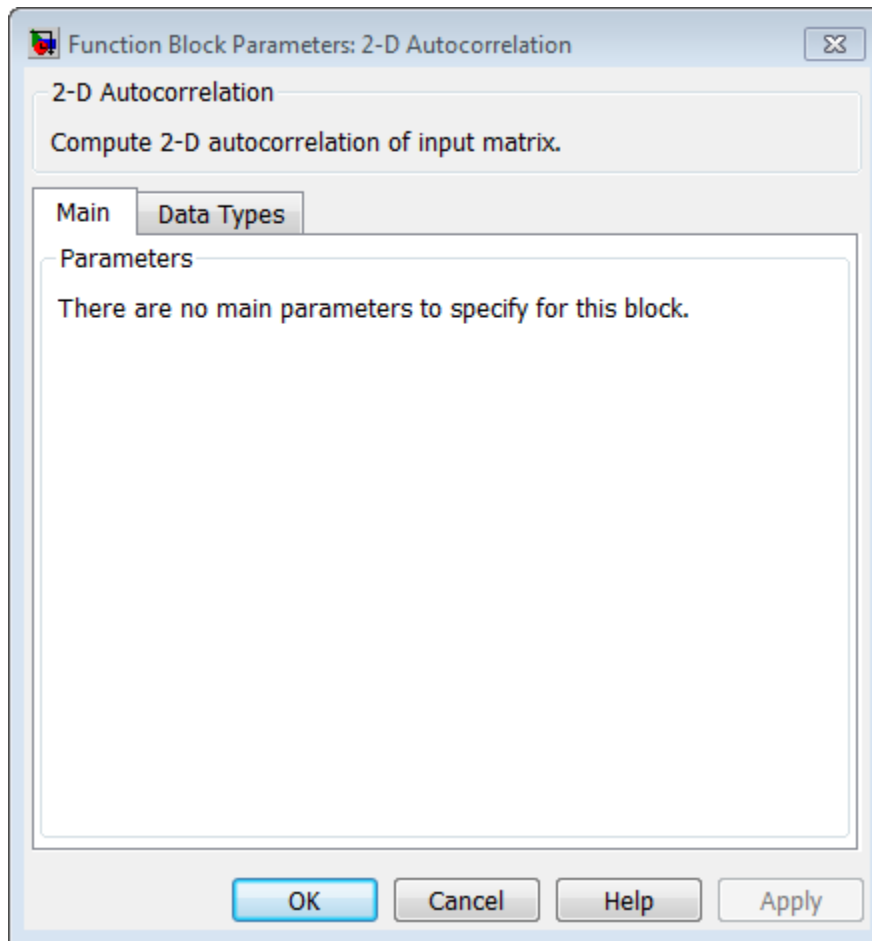


You can set the product output, accumulator, and output data types in the block mask as discussed in “Dialog Box” on page 1-3.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types”.

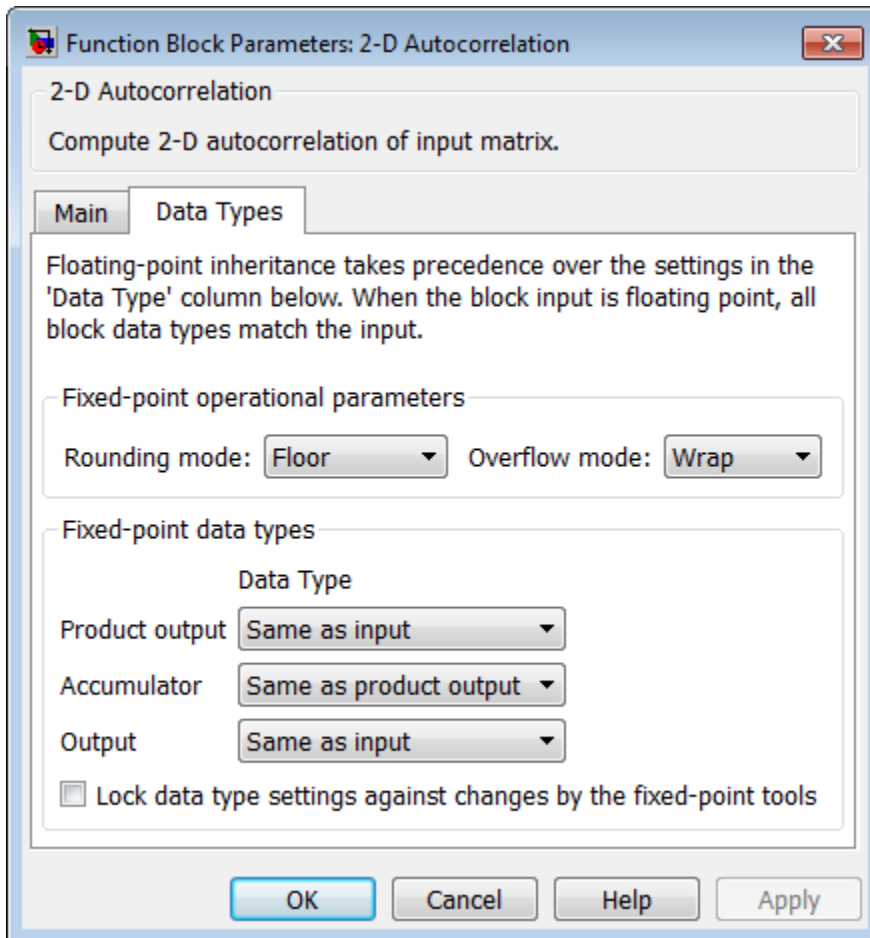
## Dialog Box

The **Main** pane of the 2-D Autocorrelation dialog box appears as shown in the following figure.



The **Data Types** pane of the 2-D Autocorrelation dialog box appears as shown in the following figure.





### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations.

### Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 1-3 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox™ software is 0.

### **Accumulator**

Use this parameter to specify how to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-3 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block. The accumulator data type is only used when both inputs to the multiplier are complex.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

### **Output**

Choose how to specify the output word length and fraction length.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### **Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink® documentation.

## See Also

2-D Correlation	Computer Vision System Toolbox
2-D Histogram	Computer Vision System Toolbox
2-D Mean	Computer Vision System Toolbox
2-D Median	Computer Vision System Toolbox
2-D Standard Deviation	Computer Vision System Toolbox
2-D Variance	Computer Vision System Toolbox
2-D Maximum	Computer Vision System Toolbox
2-D Minimum	Computer Vision System Toolbox

**Introduced before R2006a**

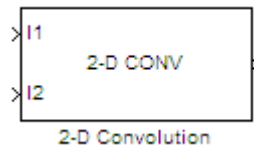
## 2-D Convolution

Compute 2-D discrete convolution of two input matrices

### Library

Filtering

visionfilter



### Description

The 2-D Convolution block computes the two-dimensional convolution of two input matrices. Assume that matrix A has dimensions  $(Ma, Na)$  and matrix B has dimensions  $(Mb, Nb)$ . When the block calculates the full output size, the equation for the 2-D discrete convolution is

$$C(i, j) = \sum_{m=0}^{(Ma-1)} \sum_{n=0}^{(Na-1)} A(m, n) * B(i-m, j-n)$$

where  $0 \leq i < Ma + Mb - 1$  and  $0 \leq j < Na + Nb - 1$ .

Port	Input/Output	Supported Data Types	Complex Values Supported
I1	Matrix of intensity values or a matrix that represents one plane of the RGB video stream	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> </ul>	Yes

Port	Input/Output	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>8-, 16-, 32-bit unsigned integer</li> </ul>	
I2	Matrix of intensity values or a matrix that represents one plane of the RGB video stream	Same as I1 port	Yes
Output	Convolution of the input matrices	Same as I1 port	Yes

If the data type of the input is floating point, the output of the block has the same data type.

The dimensions of the output are dictated by the **Output size** parameter. Assume that the input at port I1 has dimensions  $(Ma, Na)$  and the input at port I2 has dimensions  $(Mb, Nb)$ . If, for the **Output size** parameter, you choose **Full**, the output is the full two-dimensional convolution with dimensions  $(Ma+Mb-1, Na+Nb-1)$ . If, for the **Output size** parameter, you choose **Same as input port I1**, the output is the central part of the convolution with the same dimensions as the input at port I1. If, for the **Output size** parameter, you choose **Valid**, the output is only those parts of the convolution that are computed without the zero-padded edges of any input. This output has dimensions  $(Ma-Mb+1, Na-Nb+1)$ . However, if  $\text{all}(\text{size}(\text{I1}) < \text{size}(\text{I2}))$ , the block errors out.

If you select the **Output normalized convolution** check box, the block's output is divided by  $\sqrt{\text{sum}(\text{dot}(\text{I1p}, \text{I1p})) * \text{sum}(\text{dot}(\text{I2}, \text{I2}))}$ , where **I1p** is the portion of the I1 matrix that aligns with the I2 matrix. See “Example 2” on page 1-12 for more information.

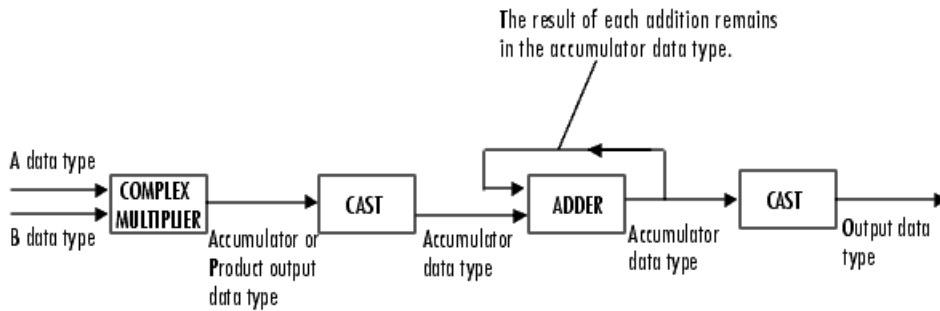
---

**Note:** When you select the **Output normalized convolution** check box, the block input cannot be fixed point.

---

## Fixed-Point Data Types

The following diagram shows the data types used in the 2-D Convolution block for fixed-point signals.



You can set the product output, accumulator, and output data types in the block mask as discussed in “Dialog Box” on page 1-14.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types”.

## Examples

### Example 1

Suppose  $I1$ , the first input matrix, has dimensions (4,3) and  $I2$ , the second input matrix, has dimensions (2,2). If, for the **Output size** parameter, you choose **Full**, the block uses the following equations to determine the number of rows and columns of the output matrix:

$$C_{\text{full}_{\text{rows}}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 5$$

$$C_{\text{full}_{\text{columns}}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 4$$

The resulting matrix is

$$C_{\text{full}} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \\ c_{40} & c_{41} & c_{42} & c_{43} \end{bmatrix}$$

If, for the **Output size** parameter, you choose **Same** as input port I1, the output is the central part of  $C_{\text{full}}$  with the same dimensions as the input at port I1, (4,3). However, since a 4-by-3 matrix cannot be extracted from the exact center of  $C_{\text{full}}$ , the block leaves more rows and columns on the top and left side of the  $C_{\text{full}}$  matrix and outputs:

$$C_{\text{same}} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \\ c_{41} & c_{42} & c_{43} \end{bmatrix}$$

If, for the **Output size** parameter, you choose **Valid**, the block uses the following equations to determine the number of rows and columns of the output matrix:

$$C_{\text{valid}_{\text{rows}}} = I1_{\text{rows}} - I2_{\text{rows}} + 1 = 3$$

$$C_{\text{valid}_{\text{columns}}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

In this case, it is always possible to extract the exact center of  $C_{\text{full}}$ . Therefore, the block outputs

$$C_{\text{full}} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

## Example 2

In convolution, the value of an output element is computed as a weighted sum of neighboring elements.

For example, suppose the first input matrix represents an image and is defined as

$$I1 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The second input matrix also represents an image and is defined as

$$I2 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

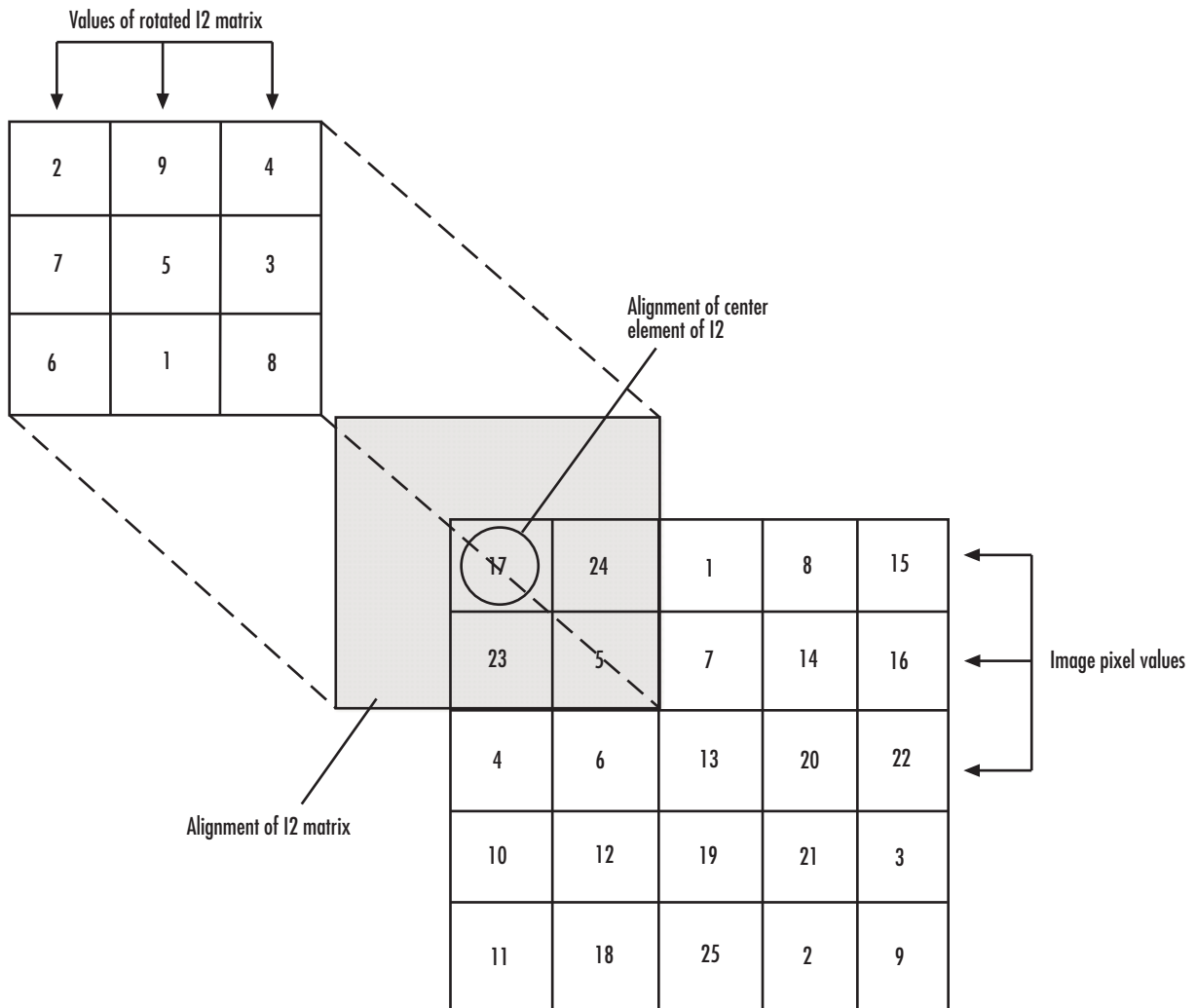
The following figure shows how to compute the (1,1) output element (zero-based indexing) using these steps:

- 1 Rotate the second input matrix, I2, 180 degrees about its center element.
- 2 Slide the center element of I2 so that it lies on top of the (0,0) element of I1.
- 3 Multiply each element of the rotated I2 matrix by the element of I1 underneath.
- 4 Sum the individual products from step 3.

Hence the (1,1) output element is

$$0 \cdot 2 + 0 \cdot 9 + 0 \cdot 4 + 0 \cdot 7 + 17 \cdot 5 + 24 \cdot 3 + 0 \cdot 6 + 23 \cdot 1 + 5 \cdot 8 = 220 .$$



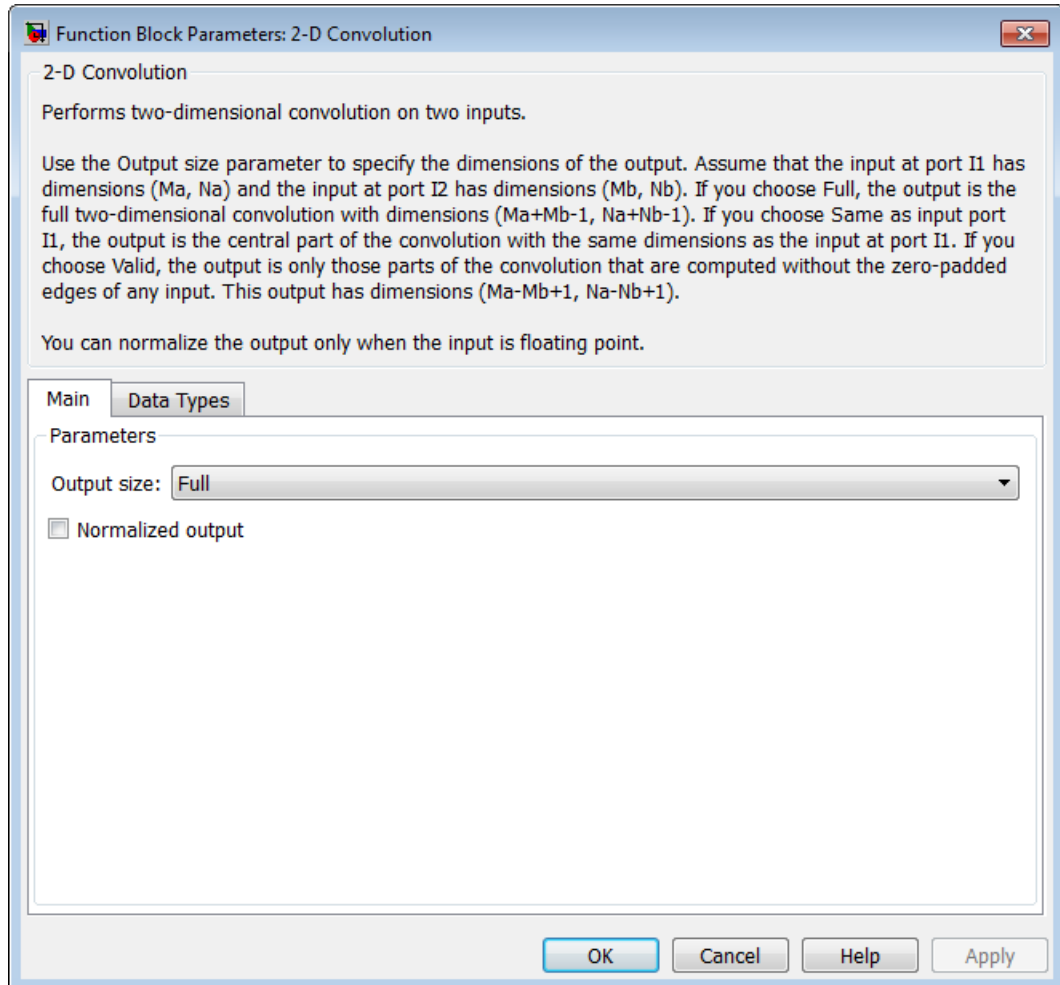


### Computing the (1,1) Output of Convolution

The normalized convolution of the (1,1) output element is  $220 / \sqrt{(\text{sum}(\text{dot}(I1p, I1p)) * \text{sum}(\text{dot}(I2, I2)))} = 0.3459$ , where  $I1p = [0 \ 0 \ 0; 0 \ 17 \ 24; 0 \ 23 \ 5]$ .

## Dialog Box

The **Main** pane of the 2-D Convolution dialog box appears as shown in the following figure.



### Output size

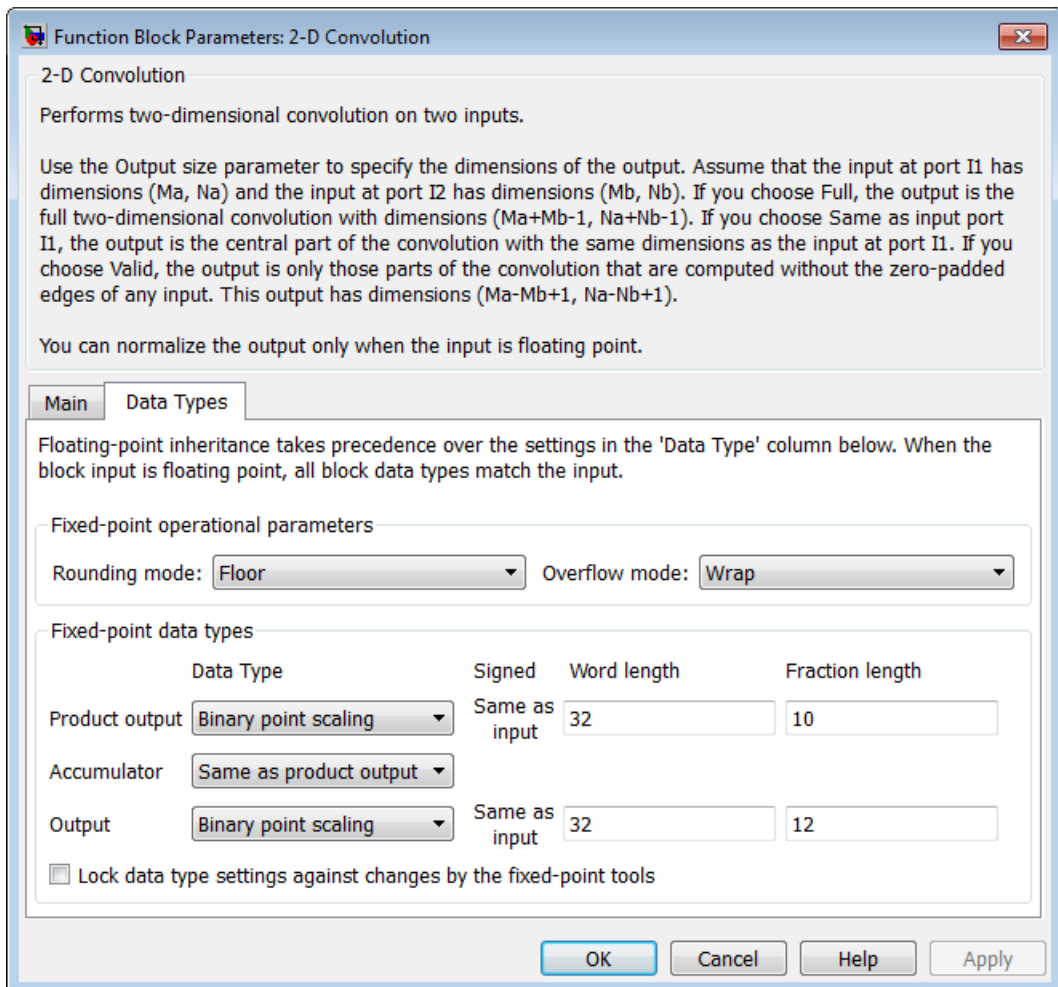
This parameter controls the size of the output scalar, vector, or matrix produced as a result of the convolution between the two inputs. If you choose **Full**, the output

has dimensions  $(Ma+Mb-1, Na+Nb-1)$ . If you choose **Same** as input port I1, the output has the same dimensions as the input at port I1. If you choose **Valid**, output has dimensions  $(Ma-Mb+1, Na-Nb+1)$ .

### Output normalized convolution

If you select this check box, the block's output is normalized.

The **Data Types** pane of the 2-D Convolution dialog box appears as shown in the following figure.



**Rounding mode**

Select the rounding mode for fixed-point operations.

**Overflow mode**

Select the Overflow mode for fixed-point operations.

**Product output**

Use this parameter to specify how to designate the product output word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-9 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

The Product Output inherits its sign according to the inputs. If either or both input **I1** and **I2** are signed, the Product Output will be signed. Otherwise, the Product Output is unsigned. The following table shows all cases.

Sign of Input I1	Sign of Input I2	Sign of Product Output
unsigned	unsigned	unsigned
unsigned	signed	signed
signed	unsigned	signed
signed	signed	signed

**Accumulator**

Use this parameter to specify how to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-9 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block. The accumulator data type is only used when both inputs to the multiplier are complex:

- When you select **Same as product output**, these characteristics match those of the product output.

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

2-D FIR Filter	Computer Vision System Toolbox software
----------------	---

Introduced before R2006a

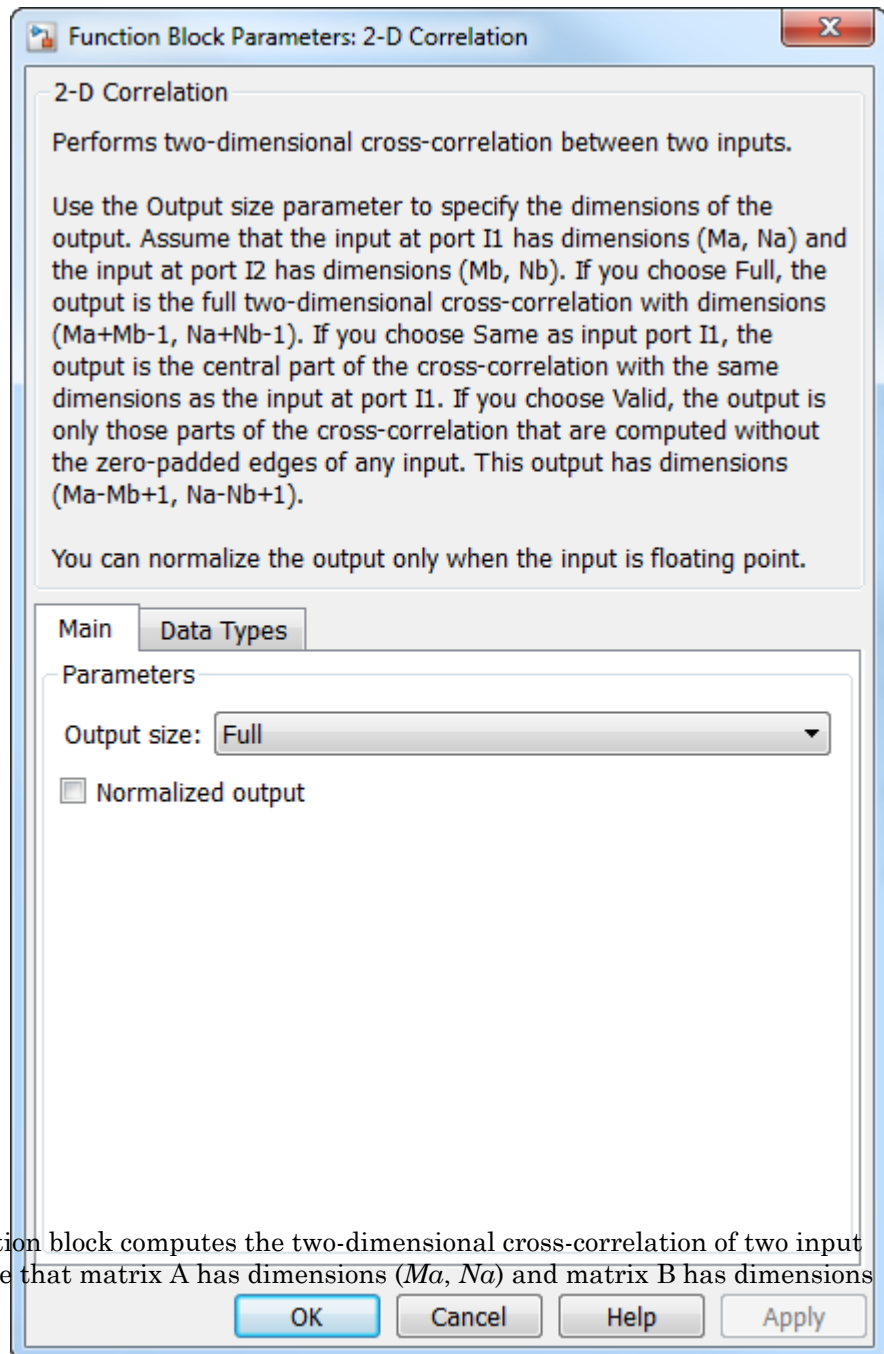
## **2-D Correlation**

Compute 2-D cross-correlation of two input matrices

### **Library**

Statistics

`visionstatistics`



The 2-D Correlation block computes the two-dimensional cross-correlation of two input matrices. Assume that matrix A has dimensions  $(M_a, N_a)$  and matrix B has dimensions

## Description

$(Mb, Nb)$ . When the block calculates the full output size, the equation for the two-dimensional discrete cross-correlation is

$$C(i, j) = \sum_{m=0}^{(Ma-1)(Na-1)} \sum_{n=0}^{(Na-1)} A(m, n) \cdot \text{conj}(B(m+i, n+j))$$

where  $0 \leq i < Ma + Mb - 1$  and  $0 \leq j < Na + Nb - 1$ .

Port	Input/Output	Supported Data Types	Complex Values Supported
I1	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes
I2	Scalar, vector, or matrix of intensity values or a scalar, vector, or matrix that represents one plane of the RGB video stream	Same as I1 port	Yes
Output	Convolution of the input matrices	Same as I1 port	Yes

If the data type of the input is floating point, the output of the block is the same data type.

The dimensions of the output are dictated by the **Output size** parameter and the sizes of the inputs at ports I1 and I2. For example, assume that the input at port I1 has dimensions  $(Ma, Na)$  and the input at port I2 has dimensions  $(Mb, Nb)$ . If, for the **Output size** parameter, you choose **Full**, the output is the full two-dimensional cross-correlation with dimensions  $(Ma+Mb-1, Na+Nb-1)$ . If, for the **Output size** parameter, you choose **Same as input port I1**, the output is the central part of the cross-correlation with the same dimensions as the input at port I1. If, for the **Output size** parameter, you choose **Valid**, the output is only those parts of the cross-correlation that are computed without the zero-padded edges of any input. This output has dimensions  $(Ma-Mb+1, Na-Nb+1)$ . However, if  $\text{all}(\text{size}(\text{I1}) < \text{size}(\text{I2}))$ , the block errors out.



If you select the **Normalized output** check box, the block's output is divided by  $\sqrt{\text{sum}(\text{dot}(I1p, I1p)) * \text{sum}(\text{dot}(I2, I2))}$ , where  $I1p$  is the portion of the  $I1$  matrix that aligns with the  $I2$  matrix. See “Example 2” on page 1-23 for more information.

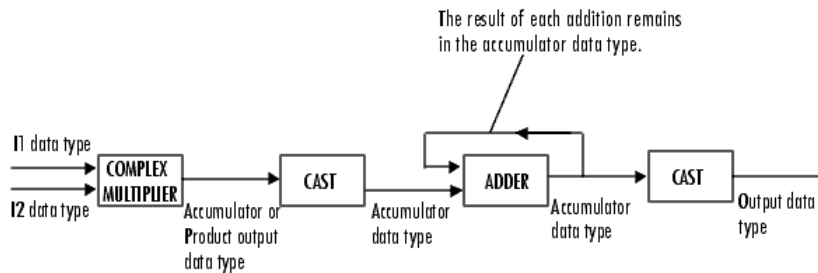
---

**Note:** When you select the **Normalized output** check box, the block input cannot be fixed point.

---

## Fixed-Point Data Types

The following diagram shows the data types used in the 2-D Correlation block for fixed-point signals.



You can set the product output, accumulator, and output data types in the block mask as discussed in “Dialog Box” on page 1-26.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types”.

## Examples

### Example 1

Suppose I1, the first input matrix, has dimensions (4,3). I2, the second input matrix, has dimensions (2,2). If, for the **Output size** parameter, you choose Full, the block uses the following equations to determine the number of rows and columns of the output matrix:

$$C_{\text{full}_{\text{rows}}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 4 + 2 - 1 = 5$$

$$C_{\text{full}_{\text{columns}}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 3 + 2 - 1 = 4$$

The resulting matrix is

$$C_{\text{full}} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \\ c_{40} & c_{41} & c_{42} & c_{43} \end{bmatrix}$$

If, for the **Output size** parameter, you choose Same as input port I1, the output is the central part of  $C_{\text{full}}$  with the same dimensions as the input at port I1, (4,3). However, since a 4-by-3 matrix cannot be extracted from the exact center of  $C_{\text{full}}$ , the block leaves more rows and columns on the top and left side of the  $C_{\text{full}}$  matrix and outputs:

$$C_{\text{same}} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \\ c_{41} & c_{42} & c_{43} \end{bmatrix}$$

If, for the **Output size** parameter, you choose **Valid**, the block uses the following equations to determine the number of rows and columns of the output matrix:

$$C_{\text{valid}_{\text{rows}}} = I1_{\text{rows}} - I2_{\text{rows}} + 1 = 3$$

$$C_{\text{valid}_{\text{columns}}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

In this case, it is always possible to extract the exact center of  $C_{\text{full}}$ . Therefore, the block outputs

$$C_{\text{full}} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

## Example 2

In cross-correlation, the value of an output element is computed as a weighted sum of neighboring elements.

For example, suppose the first input matrix represents an image and is defined as

$$I1 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The second input matrix also represents an image and is defined as

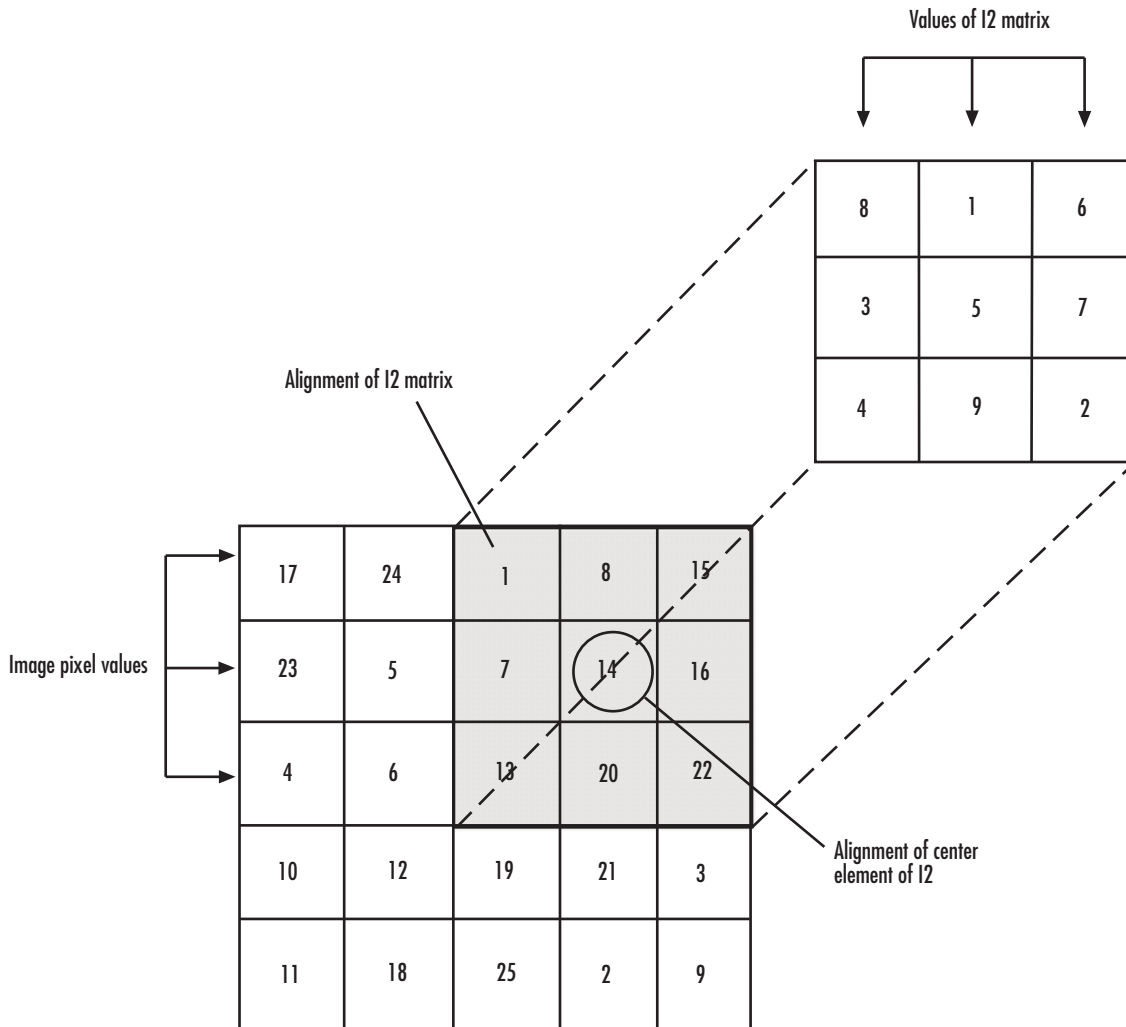
$$I2 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

The following figure shows how to compute the (2,4) output element (zero-based indexing) using these steps:

- 1** Slide the center element of I2 so that lies on top of the (1,3) element of I1.
- 2** Multiply each weight in I2 by the element of I1 underneath.
- 3** Sum the individual products from step 2.

The (2,4) output element from the cross-correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585 .$$

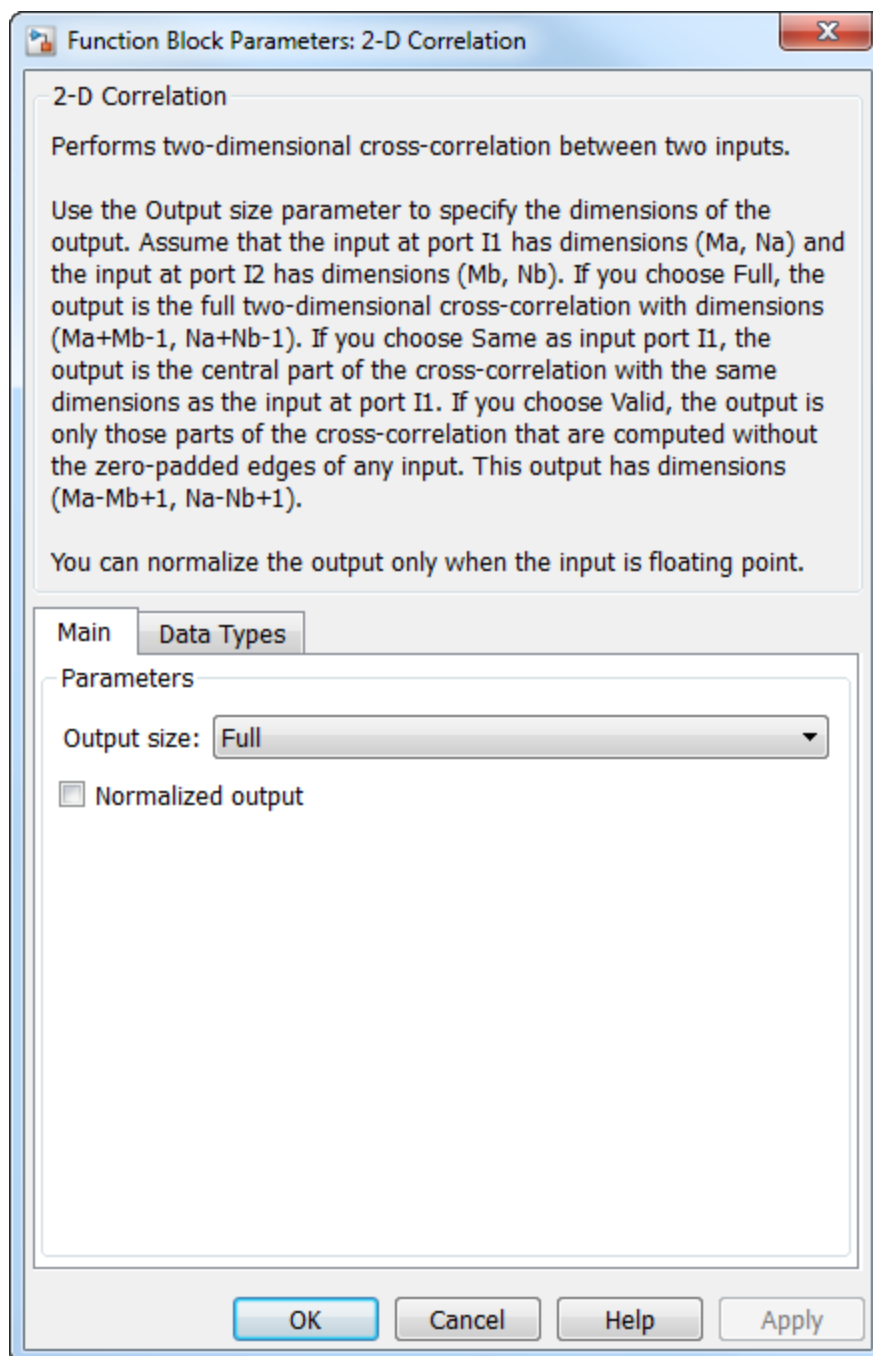


### Computing the (2,4) Output of Cross-Correlation

The normalized cross-correlation of the (2,4) output element is  $585 / \sqrt{(\text{sum}(\text{dot}(I1p, I1p)) * \text{sum}(\text{dot}(I2, I2)))} = 0.8070$ , where  $I1p = [1 \ 8 \ 15; 7 \ 14 \ 16; 13 \ 20 \ 22]$ .

## Dialog Box

The **Main** pane of the 2-D Correlation dialog box appears as shown in the following figure.



**Output size**

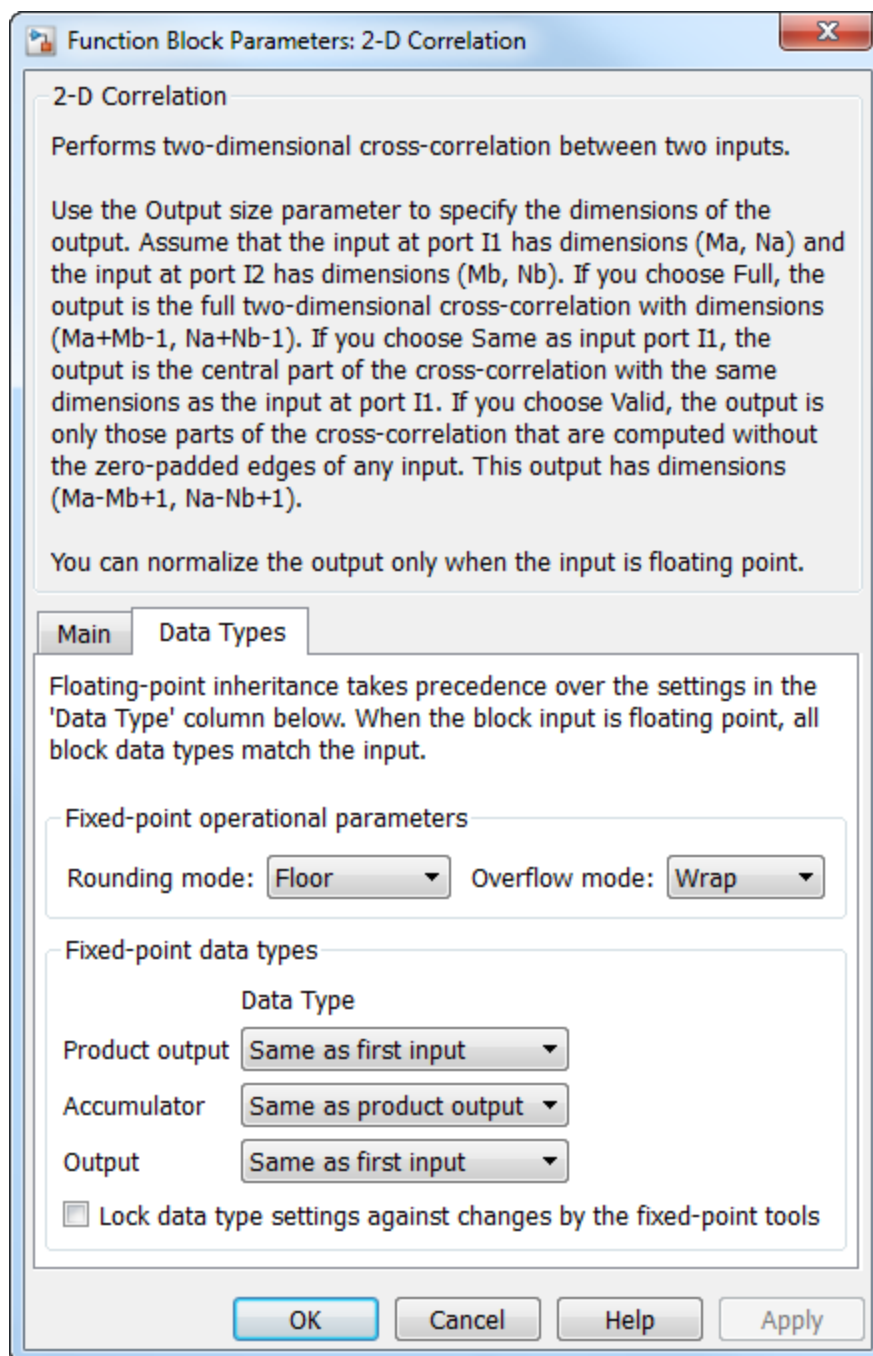
This parameter controls the size of the output scalar, vector, or matrix produced as a result of the cross-correlation between the two inputs. If you choose **Full**, the output has dimensions  $(Ma+Mb-1, Na+Nb-1)$ . If you choose **Same as input port I1**, the output has the same dimensions as the input at port I1. If you choose **Valid**, output has dimensions  $(Ma-Mb+1, Na-Nb+1)$ .

**Normalized output**

If you select this check box, the block's output is normalized.

The **Data Types** pane of the 2-D Correlation dialog box appears as shown in the following figure.





**Rounding mode**

Select the “Rounding Modes” for fixed-point operations.

**Overflow mode**

Select the Overflow mode for fixed-point operations.

**Product output**

Specify the product output data type. See “Fixed-Point Data Types” on page 1-21 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

The Product Output inherits its sign according to the inputs. If either or both input **I1** and **I2** are signed, the Product Output will be signed. Otherwise, the Product Output is unsigned. The table below show all cases.

Sign of Input I1	Sign of Input I2	Sign of Product Output
unsigned	unsigned	unsigned
unsigned	signed	signed
signed	unsigned	signed
signed	signed	signed

**Accumulator**

Use this parameter to specify how to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-21 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block. The accumulator data type is only used when both inputs to the multiplier are complex:

- When you select **Same as product output**, these characteristics match those of the product output.

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

2-D Autocorrelation	Computer Vision System Toolbox
2-D Histogram	Computer Vision System Toolbox
2-D Mean	Computer Vision System Toolbox
2-D Median	Computer Vision System Toolbox
2-D Standard Deviation	Computer Vision System Toolbox
2-D Variance	Computer Vision System Toolbox
2-D Maximum	Computer Vision System Toolbox
2-D Minimum	Computer Vision System Toolbox

**Introduced before R2006a**

## 2-D DCT

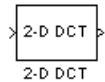
Compute 2-D discrete cosine transform (DCT)

### Library

Transforms

visiontransforms

### Description



The 2-D DCT block calculates the two-dimensional discrete cosine transform of the input signal. The equation for the two-dimensional DCT is

$$F(m,n) = \frac{2}{\sqrt{MN}} C(m)C(n) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos \frac{(2x+1)m\pi}{2M} \cos \frac{(2y+1)n\pi}{2N}$$

where  $C(m), C(n) = 1 / \sqrt{2}$  for  $m, n = 0$  and  $C(m), C(n) = 1$  otherwise.

The number of rows and columns of the input signal must be powers of two. The output of this block has dimensions the same dimensions as the input.

Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No

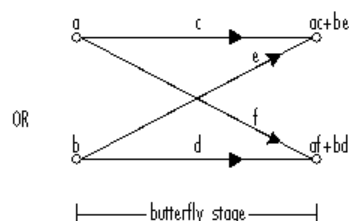
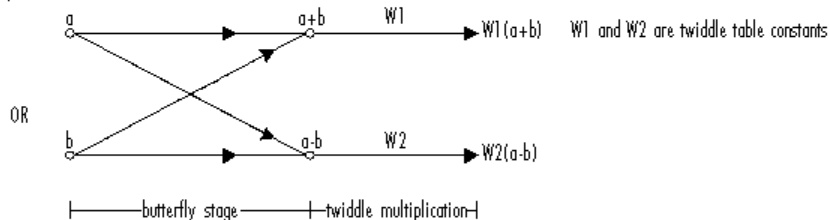
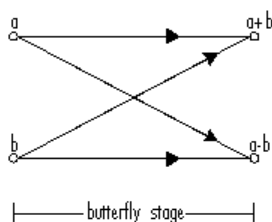
Port	Input/Output	Supported Data Types	Complex Values Supported
Output	2-D DCT of the input	Same as Input port	No

If the data type of the input signal is floating point, the output of the block is the same data type.

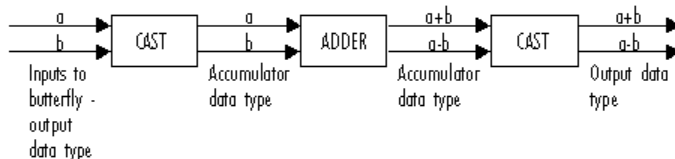
Use the **Sine and cosine computation** parameter to specify how the block computes the sine and cosine terms in the DCT algorithm. If you select **Trigonometric fcn**, the block computes the sine and cosine values during the simulation. If you select **Table lookup**, the block computes and stores the trigonometric values before the simulation starts. In this case, the block requires extra memory.

### Fixed-Point Data Types

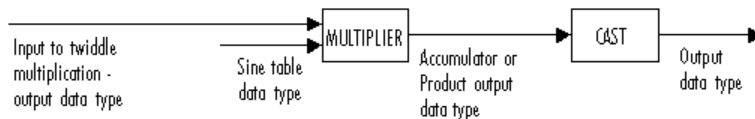
The following diagram shows the data types used in the 2-D DCT block for fixed-point signals. Inputs are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



### Butterfly Stage Data Types



### Twiddle Multiplication Data Types

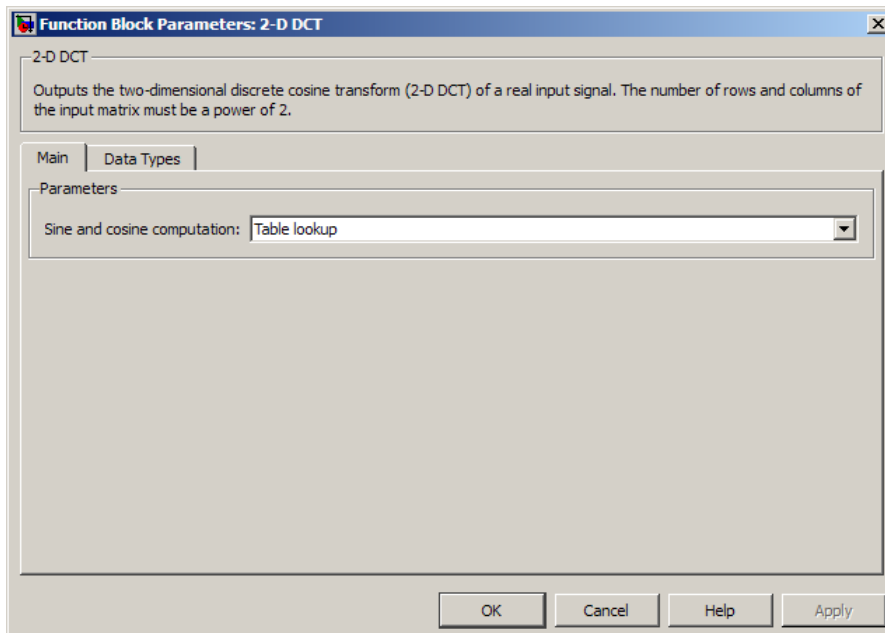


The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex

multiplication performed, refer to “Multiplication Data Types”. You can set the sine table, product output, accumulator, and output data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the 2-D DCT dialog box appears as shown in the following figure.

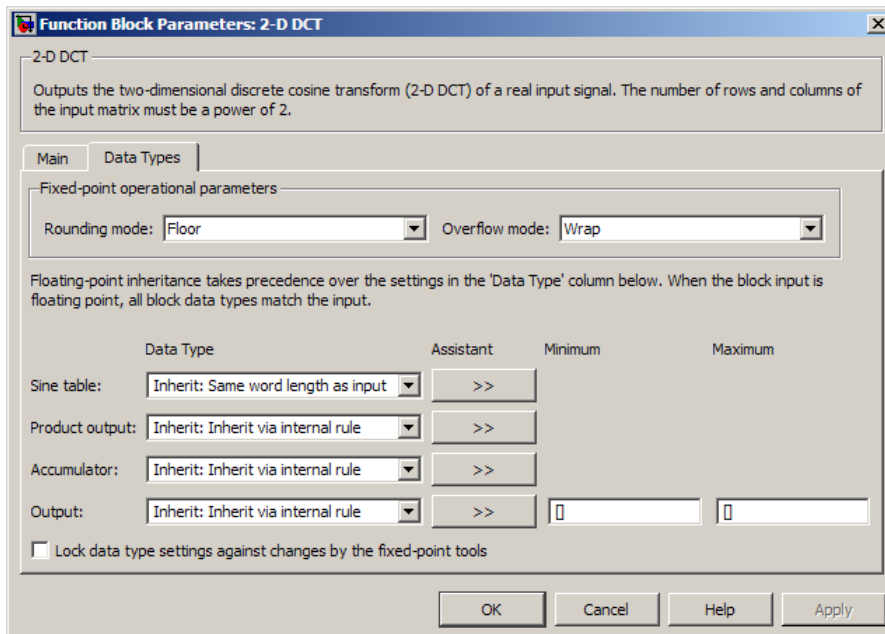


### Sine and cosine computation

Specify how the block computes the sine and cosine terms in the DCT algorithm. If you select **Trigonometric fcn**, the block computes the sine and cosine values during the simulation. If you select **Table lookup**, the block computes and stores the trigonometric values before the simulation starts. In this case, the block requires extra memory.

The **Data Types** pane of the 2-D DCT dialog box appears as shown in the following figure.





## Rounding mode

Select the “Rounding Modes” for fixed-point operations. The sine table values do not obey this parameter; they always round to **Nearest**.

## Overflow mode

Select the Overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

## Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Same word length as input**
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to **Nearest**.

### Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 1-34 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-34 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 1-34 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.


When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{ideal\ output} = WL_{input} + \text{floor}(\log_2(DCT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Chen, W.H, C.H. Smith, and S.C. Fralick, “A fast computational algorithm for the discrete cosine transform,” *IEEE Trans. Commun.*, vol. COM-25, pp. 1004-1009. 1977.
- [2] Wang, Z. “Fast algorithms for the discrete W transform and for the discrete Fourier transform,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 803-816, Aug. 1984.

## See Also

2-D IDCT	Computer Vision System Toolbox software
2-D FFT	Computer Vision System Toolbox software
2-D IFFT	Computer Vision System Toolbox software

**Introduced before R2006a**

## 2-D FFT

Compute two-dimensional fast Fourier transform of input

### Library

Transforms

visiontransforms



The 2-D FFT block computes the fast Fourier transform (FFT). The block does the computation of a two-dimensional  $M$ -by- $N$  input matrix in two steps. First it computes the one-dimensional FFT along one dimension (row or column). Then it computes the FFT of the output of the first step along the other dimension (column or row).

The output of the 2-D FFT block is equivalent to the MATLAB<sup>®</sup> `fft2` function:

```
y = fft2(A) % Equivalent MATLAB code
```

Computing the FFT of each dimension of the input matrix is equivalent to calculating the two-dimensional discrete Fourier transform (DFT), which is defined by the following equation:

$$F(m,n) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j\frac{2\pi mx}{M}} e^{-j\frac{2\pi ny}{N}}$$

where  $0 \leq m \leq M-1$  and  $0 \leq n \leq N-1$ .

The output of this block has the same dimensions as the input. If the input signal has a floating-point data type, the data type of the output signal uses the same floating-point data type. Otherwise, the output can be any fixed-point data type. The block computes scaled and unscaled versions of the FFT.

The input to this block can be floating-point or fixed-point, real or complex, and conjugate symmetric. The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library [1], [2], or an implementation based on a collection of Radix-2 algorithms. You can select **Auto** to allow the block to choose the implementation.

### Port Description

Port	Description	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes
Output	2-D FFT of the input	Same as Input port	Yes

### FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to those computers which are capable of running MATLAB. The input data type must be floating-point.

### Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the “Simulink

Coder”. The dimensions of the input matrix,  $M$  and  $N$ , must be powers of two. To work with other input sizes, use the **Image Pad** block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

### Radix-2 Algorithms for Real or Complex Input Complexity Floating-Point Signals

Other Parameter Settings	Algorithms Used for IFFT Computation
<input type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT
<input checked="" type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF
<input type="checkbox"/> Input is in bit-reversed order <input checked="" type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
<input checked="" type="checkbox"/> Input is in bit-reversed order <input checked="" type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

### Radix-2 Algorithms for Real or Complex Input Complexity Fixed-Point Signals

Other Parameter Settings	Algorithms Used for IFFT Computation
<input type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT
<input checked="" type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF

---

**Note:** The **Input is conjugate symmetric** parameter cannot be used for fixed-point signals.

---

## Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix-2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where  $K$  is the greater value of either  $M$  or  $N$  and  $k = 0, \dots, K - 1$ . The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

Number of Table Entries for N-Point FFT	
floating-point	$3N/4$
fixed-point	$N$

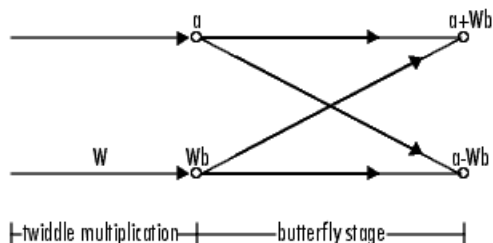
### Fixed-Point Data Types

The following diagrams show the data types used in the FFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the FFT dialog box as discussed in "Dialog Box" on page 1-46.

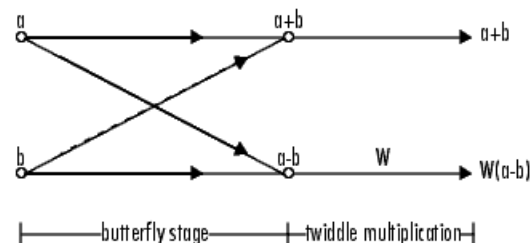
Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time FFT and after each butterfly stage in a decimation-in-frequency FFT.



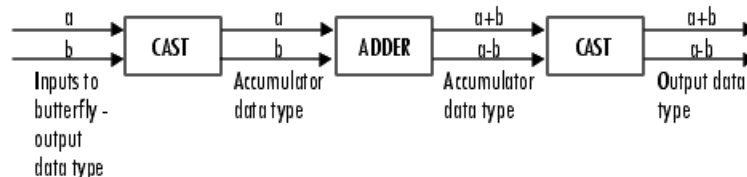
## Decimation-in-time IFFT



## Decimation-in-frequency IFFT



## Butterfly stage data types



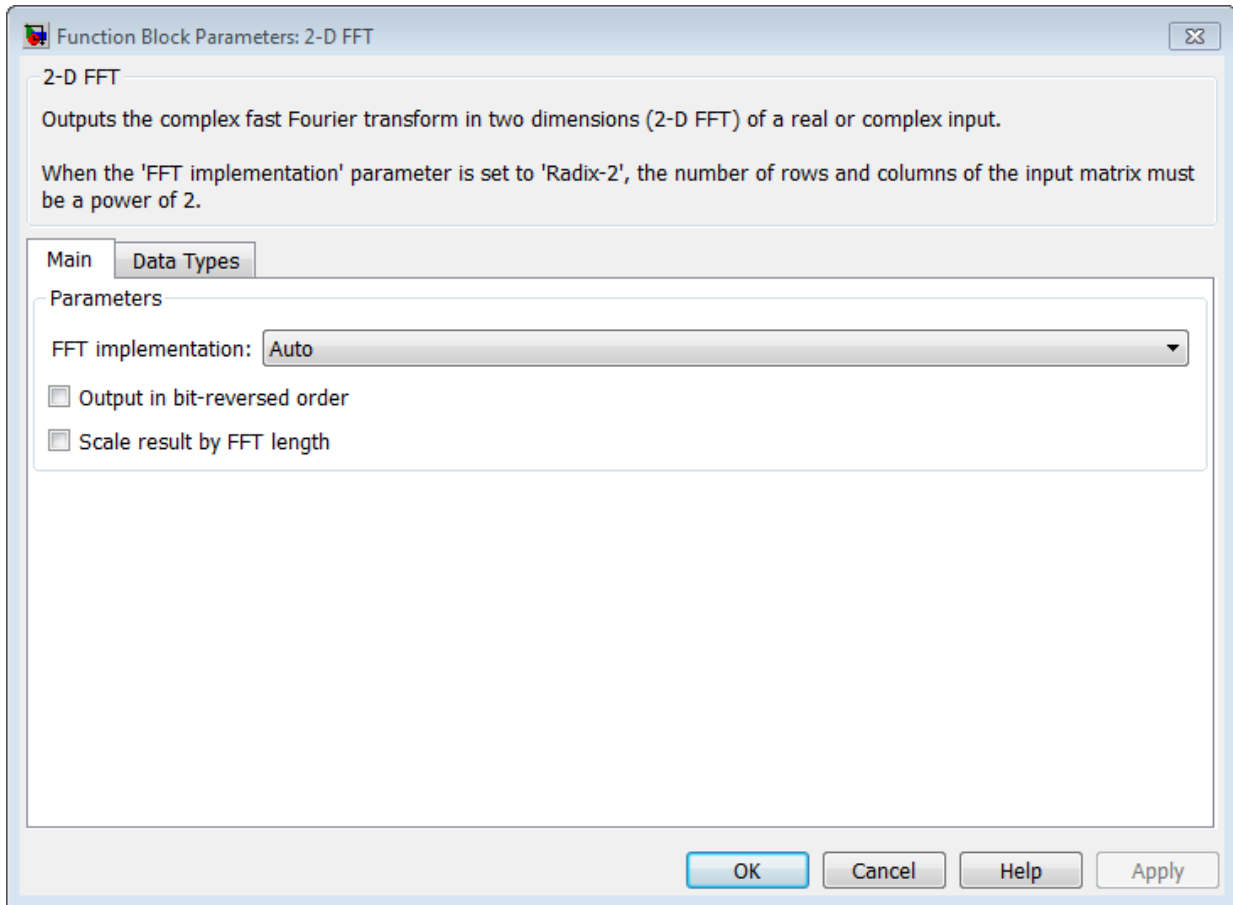
## Twiddle multiplication data types



The multiplier output appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to "Multiplication Data Types".

## Dialog Box

The **Main** pane of the 2-D FFT dialog box appears as shown in the following figure.



### FFT implementation

Set this parameter to `FFTW [1], [2]` to support an arbitrary length input signal. The block restricts generated code with `FFTW` implementation to host computers capable of running `MATLAB`.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the “Simulink Coder”. The dimensions

of the input matrix,  $M$  and  $N$ , must be powers of two. To work with other input sizes, use the **Image Pad** block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. See “Radix-2 Implementation” on page 1-42.

Set this parameter to **Auto** to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

### **Output in bit-reversed order**

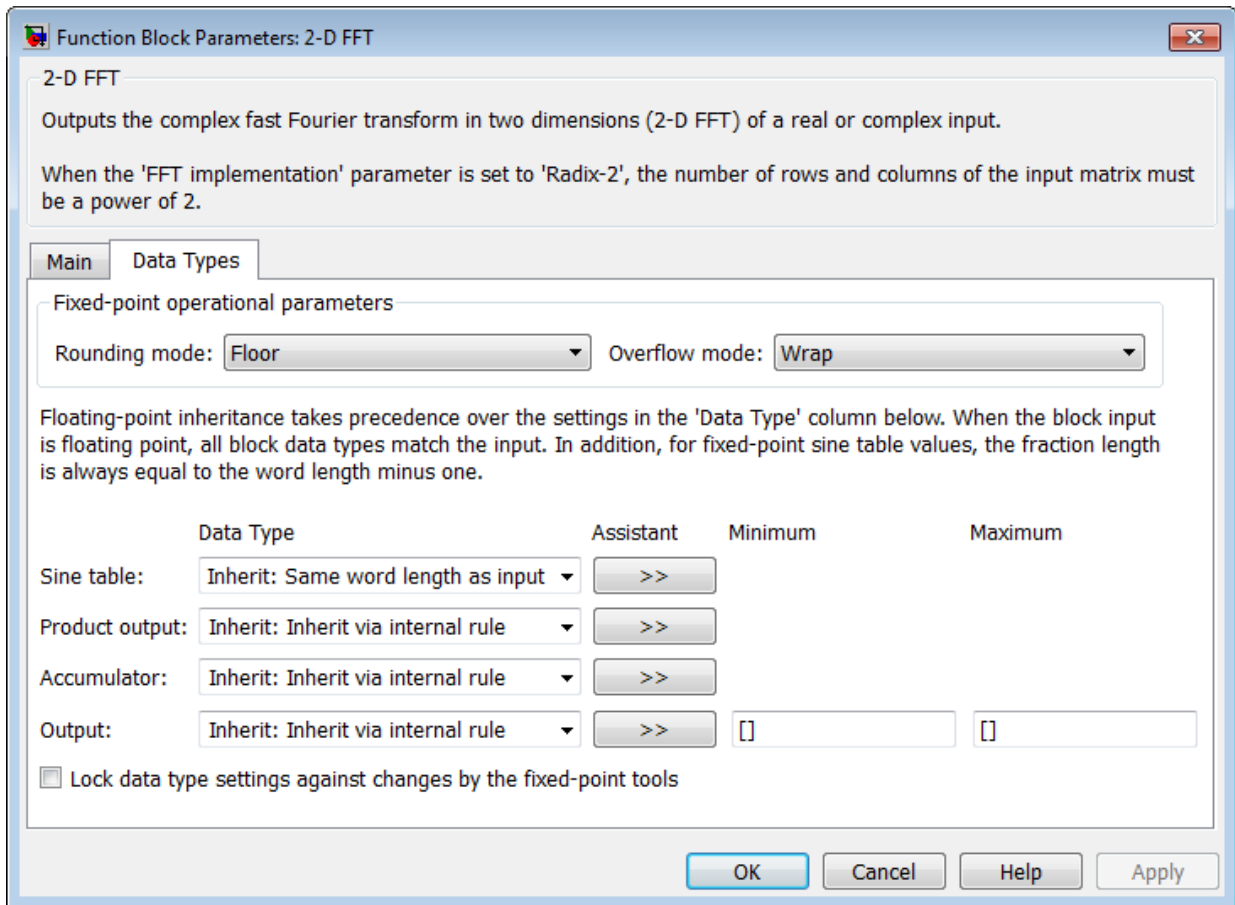
Designate the order of the output channel elements relative to the ordering of the input elements. When you select this check box, the output channel elements appear in bit-reversed order relative to the input ordering. If you clear this check box, the output channel elements appear in linear order relative to the input ordering.

Linearly ordering the output requires extra data sorting manipulation. For more information, see “Bit-Reversed Order” on page 1-51.

### **Scale result by FFT length**

When you select this parameter, the block divides the output of the FFT by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

The **Data Types** pane of the 2-D FFT dialog box appears as shown in the following figure.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to **Nearest**.

### Overflow mode

Select the Overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

### Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to **Nearest**.

### Product output data type

Specify the product output data type. See and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

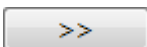
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**.

When you select **Inherit: Inherit via internal rule**, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

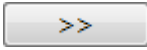
- When you select the **Divide butterfly outputs by two** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
- When you clear the **Divide butterfly outputs by two** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + \text{floor}(\log_2(FFT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### **Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

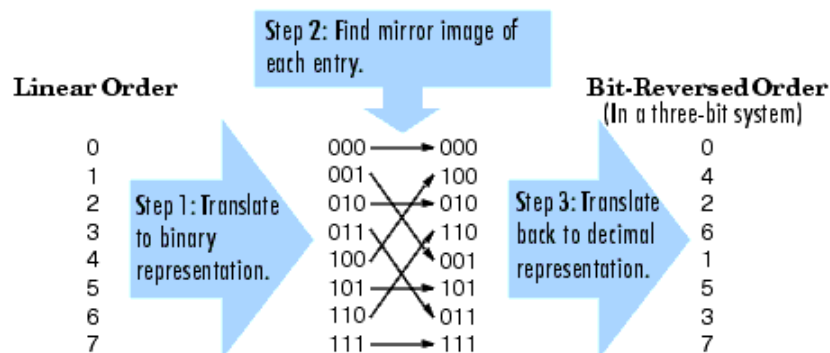
## Example

### Bit-Reversed Order

Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other because the three-bit binary representation of one, 001, is the mirror image of the three-bit binary representation of four, 100. The following diagram shows the row indices in linear order. To put them in bit-reversed order

- 1 Translate the indices into their binary representation with the minimum number of bits. In this example, the minimum number of bits is three because the binary representation of 7 is 111.
- 2 Find the mirror image of each binary entry, and write it beside the original binary representation.
- 3 Translate the indices back to their decimal representation.

The row indices now appear in bit-reversed order.



If, on the 2-D FFT block parameters dialog box, you select the **Output in bit-reversed order** check box, the block bit-reverses the order of both the columns and the rows. The next diagram illustrates the linear and bit-reversed outputs of the 2-D FFT block. The output values are the same, but they appear in different order.

245	-13	$10 - 5i$	$10 + 5i$	$13.9 - 0.4i$	$-15.9 - 21.6i$	$-15.9 + 21.6i$	13.9
-9	1	$14 - 31i$	$14 + 31i$	$16.3 + 5.9i$	$17.7 - 23.9i$	$17.7 + 23.9i$	$16.3 - 5.9i$
$18 - 5i$	$6 - 3i$	$19 - 24i$	$5 + 4i$	$-4.3 - 10.4i$	$-5.7 + 16.4i$	$12.4 - 11.4i$	$5.5 + 1.4i$
$18 + 5i$	$6 + 3i$	$5 - 4i$	$19 + 24i$	$5.5 - 1.4i$	$12.5 + 11.3i$	$-5.7 - 16.4i$	$34 + 0.5i$
$-4.3 - 10.3i$	$1.1 - i$	$-5.6 + 13.1i$	$-11.5 - 11i$	$-27.6 - 6.6i$	$-2.6i$	$-3.4 + 8.7i$	$6.2 + 13i$
$8.4 + 2.4i$	$11 + 9i$	$-18.4 - 25.1i$	$-4.5 - 1.1i$	$3.4 - 5.4i$	$17.6 - 9.4i$	$-2.2 + 13i$	$-1 - 2.7i$
$8.4 - 2.4i$	$11 - 9i$	$-4.5 + 1.1i$	$-18.4 + 25.1i$	$-0.6 + 2.7i$	$-2.2 - 13i$	$17.6 + 9.4i$	$34 + 0.5i$
$-4.4 + 10.3i$	$1.1 + i$	$-11.5 + 11i$	$-5.6 - 13.1$	$6.2 - 13i$	$-3.4 - 8.7i$	$2.6i$	$-27.6 + 6.6i$

## References

- [1] FFTW (<http://www.fftw.org>)
- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## See Also

2-D DCT	Computer Vision System Toolbox software
2-D IDCT	Computer Vision System Toolbox software
2-D IFFT	Computer Vision System Toolbox software
2-D IFFT	Computer Vision System Toolbox software
bitrevorder	Signal Processing Toolbox software
fft	MATLAB
ifft	MATLAB
"Simulink Coder"	Simulink Coder™

Introduced before R2006a



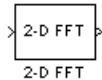
## 2-D FFT (To Be Removed)

Compute two-dimensional fast Fourier transform of input

### Library

Transforms

### Description



---

**Note:** The 2-D FFT block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox software. Use the replacement block 2-D FFT.

Refer to “FFT and IFFT Support for Non-Power-of-Two Transform Length with FFTW Library” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2012a**

## 2-D FIR Filter

Perform 2-D FIR filtering on input matrix

### Library

Filtering

visionfilter



### Description

The 2-D Finite Impulse Response (FIR) filter block filters the input matrix **I** using the coefficient matrix **H** or the coefficient vectors **HH** and **HV**.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes
H	Matrix of filter coefficients	Same as I port.	Yes
HH	Vector of filter coefficients	Same as I port. The input to ports HH and HV must be the same data type.	Yes
HV	Vector of filter coefficients	Same as I port. The input to ports HH and HV must be the same data type.	Yes

Port	Input/Output	Supported Data Types	Complex Values Supported
PVal	Scalar value that represents the constant pad value	Input must have the same data type as the input to I port.	Yes
Output	Scalar, vector, or matrix of filtered values	Same as I port.	Yes

If the input has a floating-point data type, then the output uses the same data type. Otherwise, the output can be any fixed-point data type.

Select the **Separable filter coefficients** check box if your filter coefficients are separable. Using separable filter coefficients reduces the amount of calculations the block must perform to compute the output. For example, suppose your input image is  $M$ -by- $N$  and your filter coefficient matrix is  $x$ -by- $y$ . For a nonseparable filter with the **Output size** parameter set to **Same as input port I**, it would take

$$x \cdot y \cdot M \cdot N$$

multiply-accumulate (MAC) operations for the block to calculate the output. For a separable filter, it would only take

$$(x + y) \cdot M \cdot N$$

MAC operations. If you do not know whether or not your filter coefficients are separable, use the `isfilterseparable` function.

Here is an example of the function syntax, `[S, HCOL, HROW] = isfilterseparable(H)`. The `isfilterseparable` function takes the filter kernel, `H`, and returns `S`, `HCOL` and `HROW`. Here, `S` is a Boolean variable that is 1 if the filter is separable and 0 if it is not. `HCOL` is a vector of vertical filter coefficients, and `HROW` is a vector of horizontal filter coefficients.

Use the **Coefficient source** parameter to specify how to define your filter coefficients. If you select the **Separable filter coefficients** check box and then select a **Coefficient source** of **Specify via dialog**, the **Vertical coefficients (across height)** and **Horizontal coefficients (across width)** parameters appear in the dialog box. You can use these parameters to enter vectors of vertical and horizontal filter coefficients, respectively.

You can also use the variables `HCOL` and `HROW`, the output of the `isfilterseparable` function, for these parameters. If you select the **Separable filter coefficients** check box and then select a **Coefficient source** of `Input port`, ports `HV` and `HH` appear on the block. Use these ports to specify vectors of vertical and horizontal filter coefficients.

If you clear the **Separable filter coefficients** check box and select a **Coefficient source** of `Specify via dialog`, the **Coefficients** parameter appears in the dialog box. Use this parameter to enter your matrix of filter coefficients.

If you clear the **Separable filter coefficients** check box and select a **Coefficient source** of `Input port`, port `H` appears on the block. Use this port to specify your filter coefficient matrix.

The block outputs the result of the filtering operation at the `Output` port. The **Output size** parameter and the sizes of the inputs at ports `I` and `H` dictate the dimensions of the output. For example, assume that the input at port `I` has dimensions  $(M_i, N_i)$  and the input at port `H` has dimensions  $(M_h, N_h)$ . If you select an **Output size** of `Full`, the output has dimensions  $(M_i+M_h-1, N_i+N_h-1)$ . If you select an **Output size** of `Same as input port I`, the output has the same dimensions as the input at port `I`. If you select an **Output size** of `Valid`, the block filters the input image only where the coefficient matrix fits entirely within it, so no padding is required. The output has dimensions  $(M_i-M_h+1, N_i-N_h+1)$ . However, if  $\text{all}(\text{size}(I) < \text{size}(H))$ , the block errors out.

Use the **Padding options** parameter to specify how to pad the boundary of your input matrix. To pad your matrix with a constant value, select `Constant`. To pad your input matrix by repeating its border values, select `Replicate`. To pad your input matrix with its mirror image, select `Symmetric`. To pad your input matrix using a circular repetition of its elements, select `Circular`. For more information on padding, see the `Image Pad` block reference page.

If, for the **Padding options** parameter, you select `Constant`, the **Pad value source** parameter appears in the dialog box. If you select `Specify via dialog`, the **Pad value** parameter appears in the dialog box. Use this parameter to enter the constant value with which to pad your matrix. If you select **Pad value source** of `Input port`, the `PVal` port appears on the block. Use this port to specify the constant value with which to pad your matrix. The pad value must be real if the input image is real. You will get an error message if the pad value is complex when the input image is real.

Use the **Filtering based on** parameter to specify the algorithm by which the block filters the input matrix. If you select `Convolution` and set the **Output size** parameter to `Full`, the block filters your input using the following algorithm

$$C(i, j) = \sum_{m=0}^{(Ma-1)(Na-1)} \sum_{n=0}^{(Na-1)} A(m, n) * H(i-m, j-n)$$

where  $0 \leq i < Ma + Mh - 1$  and  $0 \leq j < Na + Nh - 1$ . If you select **Correlation** and set the **Output size** parameter to **Full**, the block filters your input using the following algorithm

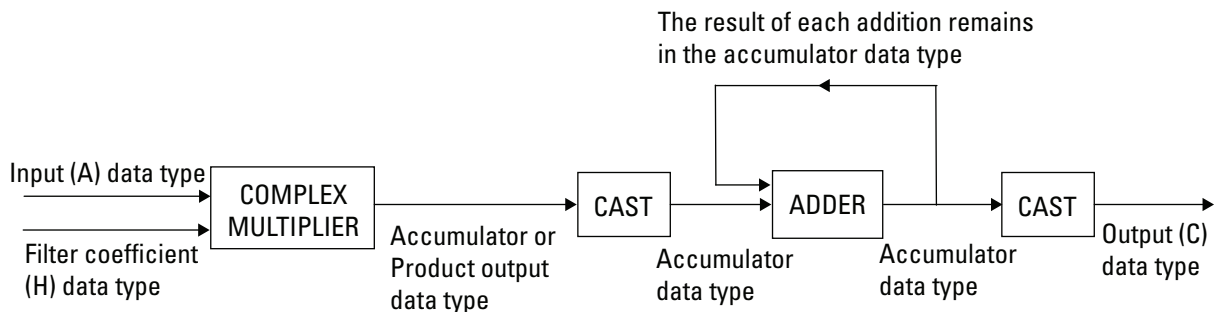
$$C(i, j) = \sum_{m=0}^{(Ma-1)(Na-1)} \sum_{n=0}^{(Na-1)} A(m, n) \cdot \text{conj}(H(m+i, n+j))$$

where  $0 \leq i < Ma + Mh - 1$  and  $0 \leq j < Na + Nh - 1$ .

The `imfilter` function from the Image Processing Toolbox™ product similarly performs N-D filtering of multidimensional images.

## Fixed-Point Data Types

The following diagram shows the data types used in the 2-D FIR Filter block for fixed-point signals.



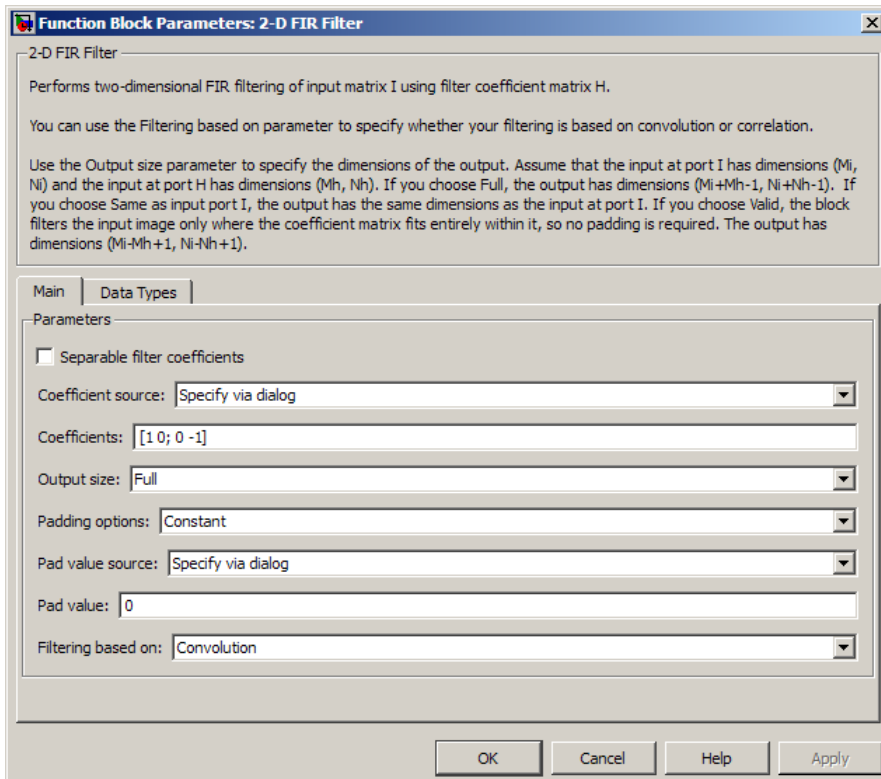
You can set the coefficient, product output, accumulator, and output data types in the block mask as discussed in “Dialog Box” on page 1-58.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the

result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types”.

## Dialog Box

The **Main** pane of the 2-D FIR Filter dialog box appears as shown in the following figure.



### Separable filter coefficients

Select this check box if your filter coefficients are separable. Using separable filter coefficients reduces the amount of calculations the block must perform to compute the output.

### Coefficient source

Specify how to define your filter coefficients. Select **Specify via dialog** to enter your coefficients in the block parameters dialog box. Select **Input port** to specify your filter coefficient matrix using port H or ports HH and HV.

### **Coefficients**

Enter your real or complex-valued filter coefficient matrix. This parameter appears if you clear the **Separable filter coefficients** check box and then select a **Coefficient source** of **Specify via dialog**. Tunable.

### **Vertical coefficients (across height)**

Enter the vector of vertical filter coefficients for your separable filter. This parameter appears if you select the **Separable filter coefficients** check box and then select a **Coefficient source** of **Specify via dialog**.

### **Horizontal coefficients (across width)**

Enter the vector of horizontal filter coefficients for your separable filter. This parameter appears if you select the **Separable filter coefficients** check box and then select a **Coefficient source** of **Specify via dialog**.

### **Output size**

This parameter controls the size of the filtered output. If you choose **Full**, the output has dimensions  $(Ma+Mh-1, Na+Nh-1)$ . If you choose **Same as input port I**, the output has the same dimensions as the input at port I. If you choose **Valid**, output has dimensions  $(Ma-Mh+1, Na-Nh+1)$ .

### **Padding options**

Specify how to pad the boundary of your input matrix. Select **Constant** to pad your matrix with a constant value. Select **Replicate** to pad your input matrix by repeating its border values. Select **Symmetric** to pad your input matrix with its mirror image. Select **Circular** to pad your input matrix using a circular repetition of its elements. This parameter appears if you select an **Output size** of **Full** or **Same as input port I**.

### **Pad value source**

Use this parameter to specify how to define your constant boundary value. Select **Specify via dialog** to enter your value in the block parameters dialog box. Select **Input port** to specify your constant value using the PVal port. This parameter appears if you select a **Padding options** of **Constant**.

### **Pad value**

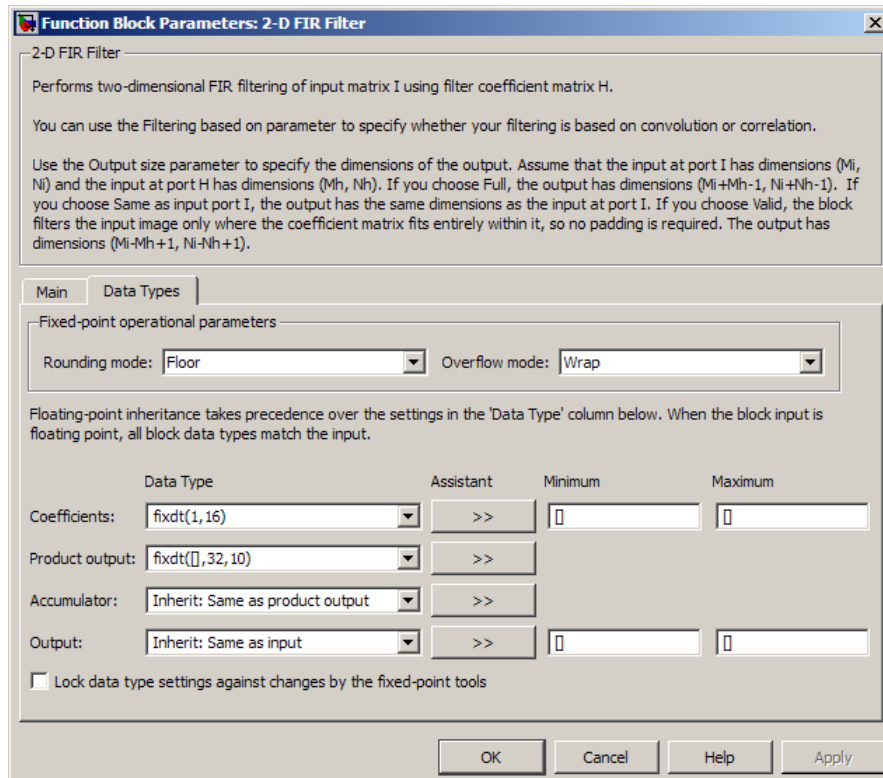
Enter the constant value with which to pad your matrix. This parameter is visible if, for the **Pad value source** parameter, you select **Specify via dialog**. Tunable.

The pad value must be real if the input image is real. You will get an error message if the pad value is complex when the input image is real.

### Filtering based on

Specify the algorithm by which the block filters the input matrix. You can select Convolution or Correlation.

The **Data Types** pane of the 2-D FIR Filter dialog box appears as shown in the following figure.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations.



## Coefficients

Choose how to specify the word length and the fraction length of the filter coefficients.

- When you select **Inherit: Same word length as input**, the word length of the filter coefficients match that of the input to the block. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select **fixdt(1,16)**, you can enter the word length of the coefficients, in bits. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select **fixdt(1,16,0)**, you can enter the word length and the fraction length of the coefficients, in bits.
- When you select **<data type expression>**, you can enter the data type expression.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; instead, they always saturated and rounded to **Nearest**.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.


## Product output

Use this parameter to specify how to designate the product output word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-57 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select **Inherit: Same as input**, these characteristics match those of the input to the block.
- When you select **fixdt([],16,0)**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **<data type expression>**, you can enter the data type expression.

If you set the **Coefficient source** (on the **Main** tab) to **Input port** the Product Output will inherit its sign according to the inputs. If either or both input **I1** and **I2** are signed, the Product Output will be signed. Otherwise, the Product Output is unsigned. The following table shows all cases.

Sign of Input I1	Sign of Input I2	Sign of Product Output
unsigned	unsigned	unsigned
unsigned	signed	signed
signed	unsigned	signed
signed	signed	signed


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator

Use this parameter to specify how to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 1-57 and “Multiplication Data Types” in the DSP System Toolbox™ documentation for illustrations depicting the use of the accumulator data type in this block. The accumulator data type is only used when both inputs to the multiplier are complex:

- When you select **Inherit: Same as input**, these characteristics match those of the input to the block.
- When you select **Inherit: Same as product output**, these characteristics match those of the product output.
- When you select `fixdt([], 16, 0)`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. All signals in the Computer Vision System Toolbox software have a bias of 0.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.


## Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Inherit: Same as input**, these characteristics match those of the input to the block.
- When you select `fixdt([],16,0)`, you can enter the word length and the fraction length of the output, in bits.

You can choose to set signedness of the output to **Auto**, **Signed** or **Unsigned**.

- When you select `<data type expression>`, you can enter the a data type expression.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

<code>imfilter</code>	Image Processing Toolbox
-----------------------	--------------------------

**Introduced before R2006a**

## 2-D Histogram

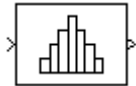
Generate histogram of input or sequence of inputs

### Library

Statistics

visionstatistics

### Description



The 2-D Histogram block computes the frequency distribution of the elements in the input. You must use the **Find the histogram over** parameter to specify whether the block computes the histogram for **Each column** of the input or of the **Entire input**. The **Running histogram** check box allows you to select between basic operation and running operation, as described below.

The block distributes the elements of the input into the number of discrete bins specified by the **Number of bins** parameter,  $n$ .

```
y = hist(u,n)      % Equivalent MATLAB code
```

The 2-D Histogram block sorts all complex input values into bins according to their magnitude.

The histogram value for a given bin represents the frequency of occurrence of the input values bracketed by that bin. You specify the upper boundary of the highest-valued bin in the **Upper limit of histogram** parameter,  $B_M$ , and the lower boundary of the lowest-valued bin in the **Lower limit of histogram** parameter,  $B_m$ . The bins have equal width of

$$\Delta = \frac{B_M - B_m}{n}$$

and centers located at

$$B_m + \left(k + \frac{1}{2}\right)\Delta \quad k = 0, 1, 2, \dots, n - 1$$

Input values that fall on the border between two bins are placed into the lower valued bin; that is, each bin includes its upper boundary. For example, a bin of width 4 centered on the value 5 contains the input value 7, but not the input value 3. Input values greater than the **Upper limit of histogram** parameter or less than **Lower limit of histogram** parameter are placed into the highest valued or lowest valued bin, respectively.

The values you enter for the **Upper limit of histogram** and **Lower limit of histogram** parameters must be real-valued scalars. NaN and inf are not valid values for the **Upper limit of histogram** and **Lower limit of histogram** parameters.

## Basic Operation

When the **Running histogram** check box is not selected, the 2-D Histogram block computes the frequency distribution of the current input.

When you set the **Find the histogram over** parameter to **Each column**, the 2-D Histogram block computes a histogram for each column of the  $M$ -by- $N$  matrix independently. The block outputs an  $n$ -by- $N$  matrix, where  $n$  is the **Number of bins** you specify. The  $j$ th column of the output matrix contains the histogram for the data in the  $j$ th column of the  $M$ -by- $N$  input matrix.

When you set the **Find the histogram over** parameter to **Entire input**, the 2-D Histogram block computes the frequency distribution for the entire input vector, matrix or N-D array. The block outputs an  $n$ -by-1 vector, where  $n$  is the **Number of bins** you specify.

## Running Operation

When you select the **Running histogram** check box, the 2-D Histogram block computes the frequency distribution of both the past and present data for successive inputs. The block resets the histogram (by emptying all of the bins) when it detects a reset event at the optional Rst port. See “Resetting the Running Histogram” on page 1-66 for more information on how to trigger a reset.

When you set the **Find the histogram over** parameter to **Each column**, the 2-D Histogram block computes a running histogram for each column of the  $M$ -by- $N$  matrix. The block outputs an  $n$ -by- $N$  matrix, where  $n$  is the **Number of bins** you specify. The  $j$ th column of the output matrix contains the running histogram for the  $j$ th column of the  $M$ -by- $N$  input matrix.

When you set the **Find the histogram over** parameter to **Entire input**, the 2-D Histogram block computes a running histogram for the data in the first dimension of the input. The block outputs an  $n$ -by-1 vector, where  $n$  is the **Number of bins** you specify.

---

**Note:** When the 2-D Histogram block is used in running mode and the input data type is non-floating point, the output of the histogram is stored as a `uint32` data type. The largest number that can be represented by this data type is  $2^{32} - 1$ . If the range of the `uint32` data type is exceeded, the output data will wrap back to 0.

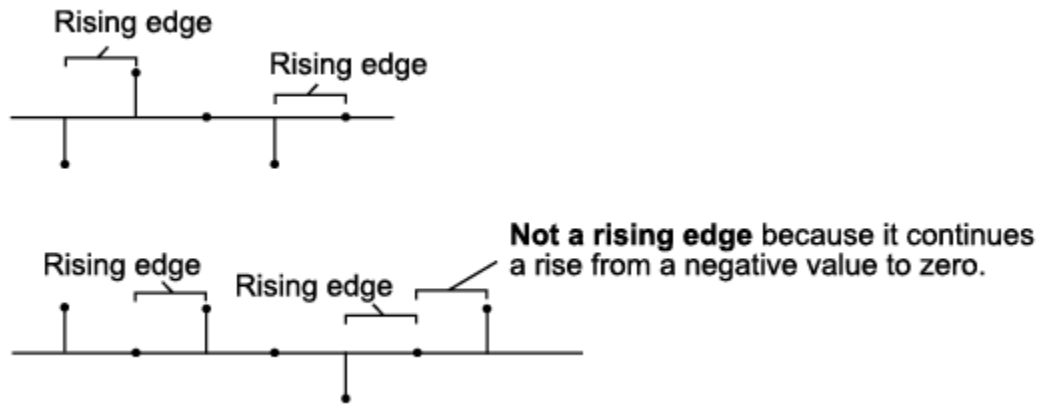
---

## Resetting the Running Histogram

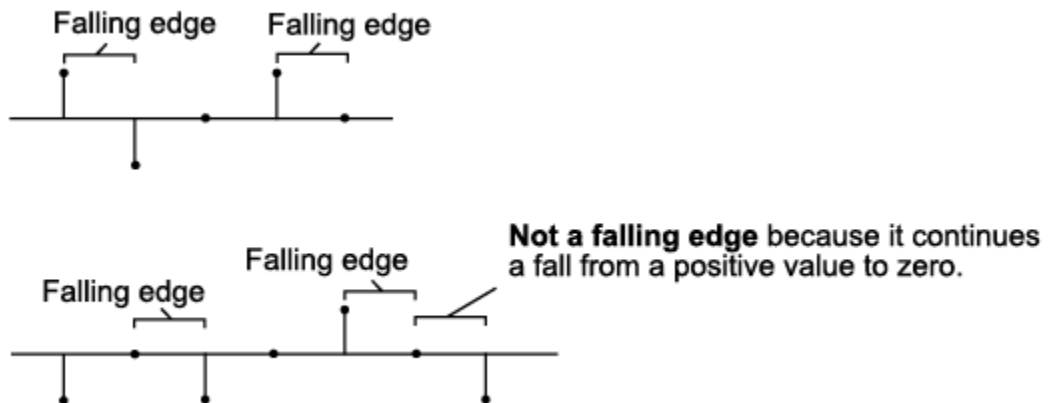
The block resets the running histogram whenever a reset event is detected at the optional Rst port. The reset signal and the input data signal must be the same rate.

You specify the reset event using the **Reset port** menu:

- **None** — Disables the Rst port
- **Rising edge** — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the RSt input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

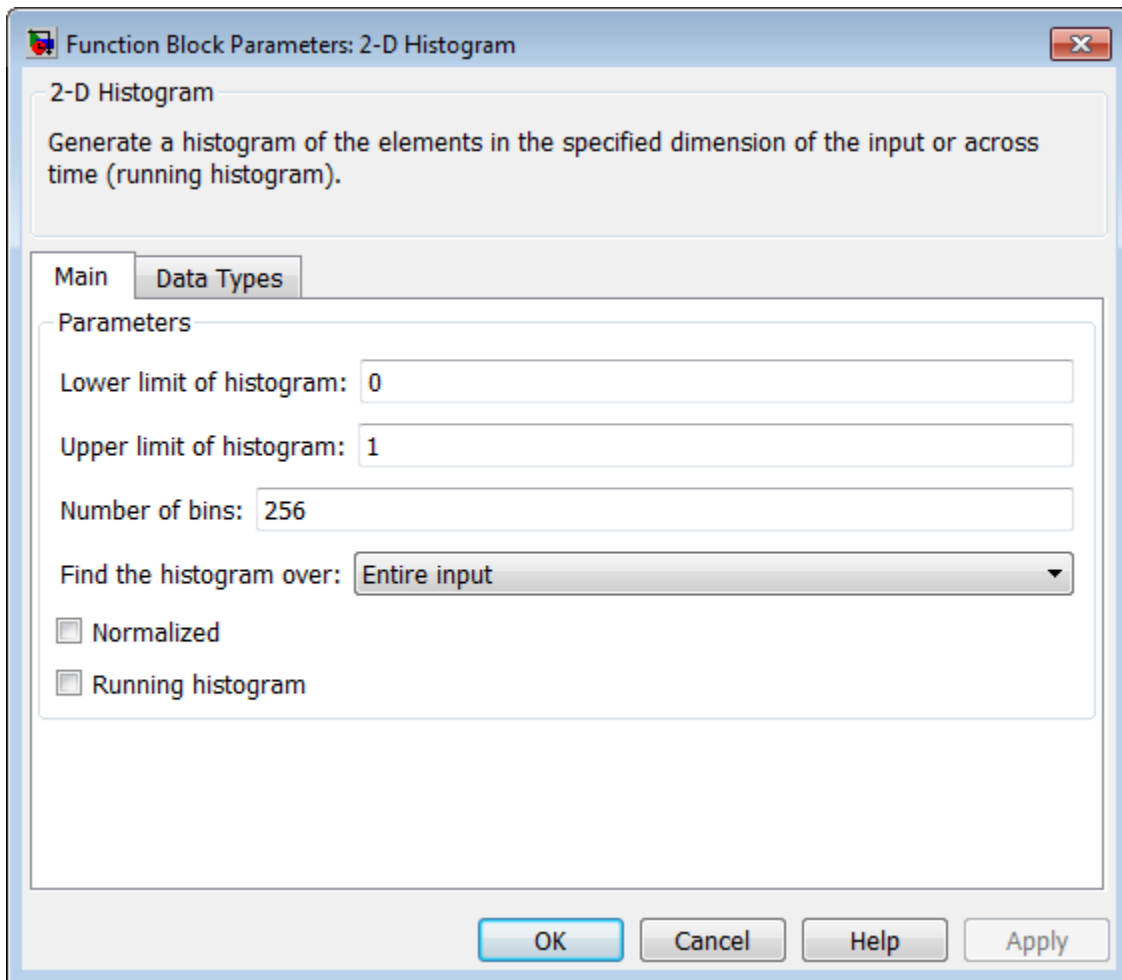


- **Either edge** — Triggers a reset operation when the RSt input is a **Rising edge** or **Falling edge** (as described earlier)

- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero

## Dialog Box

The **Main** pane of the 2-D Histogram block dialog appears as follows.





**Lower limit of histogram**

Enter a real-valued scalar for the lower boundary,  $B_m$ , of the lowest-valued bin. NaN and `inf` are not valid values for  $B_m$ . Tunable.

**Upper limit of histogram**

Enter a real-valued scalar for the upper boundary,  $B_M$ , of the highest-valued bin. NaN and `inf` are not valid values for  $B_M$ . Tunable.

**Number of bins**

The number of bins,  $n$ , in the histogram.

**Find the histogram over**

Specify whether the block finds the histogram over the entire input or along each column of the input.

---

**Note:** The option will be removed in a future release.

---

**Normalized**

When selected, the output vector,  $v$ , is normalized such that  $\text{sum}(v) = 1$ .

Use of this parameter is not supported for fixed-point signals.

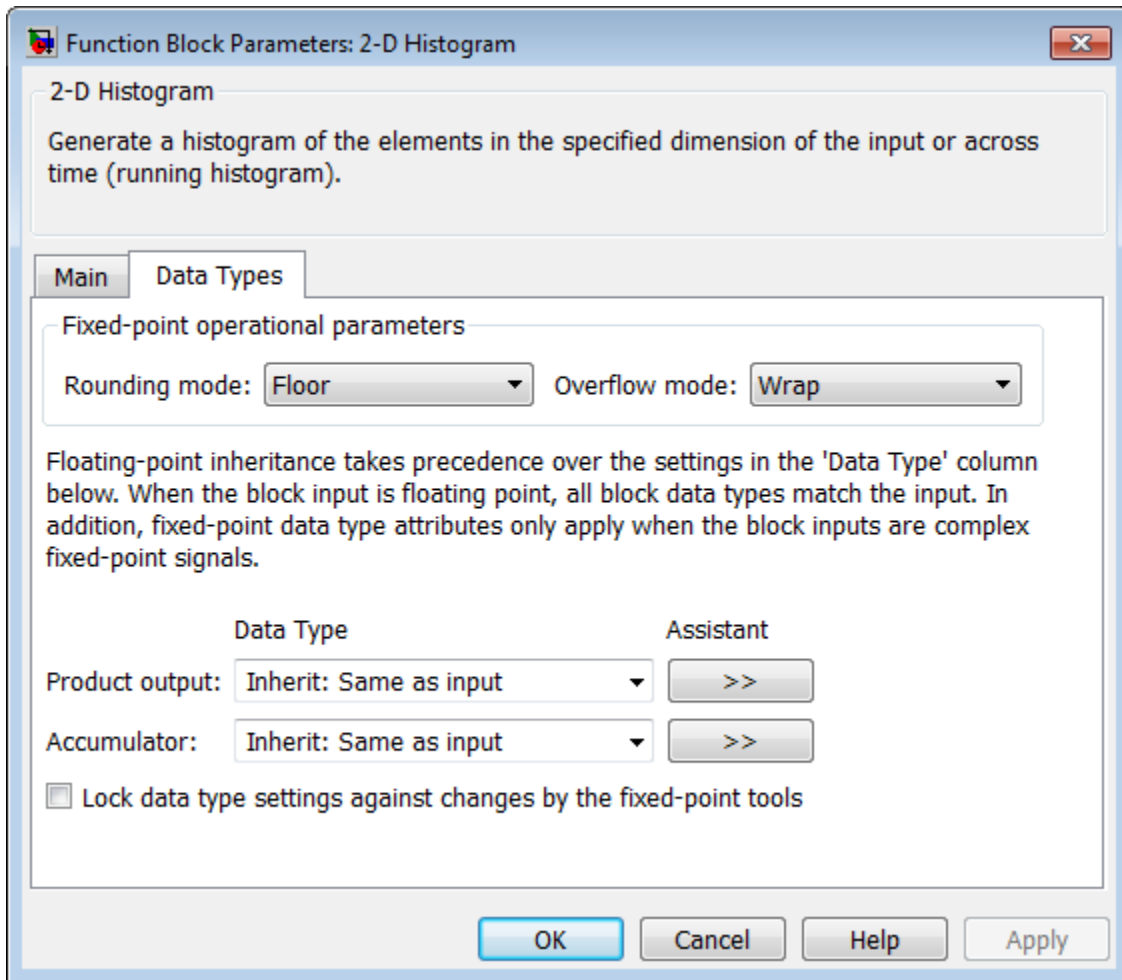
**Running histogram**

Set to enable the running histogram operation, and clear to enable basic histogram operation. For more information, see “Basic Operation” on page 1-65 and “Running Operation” on page 1-65.

**Reset port**

The type of event that resets the running histogram. For more information, see “Resetting the Running Histogram” on page 1-66. The reset signal and the input data signal must be the same rate. This parameter is enabled only when you select the **Running histogram** check box. For more information, see “Running Operation” on page 1-65.

The **Data Types** pane of the 2-D Histogram block dialog appears as follows.



---

**Note:** The fixed-point parameters listed are only used for fixed-point complex inputs, which are distributed by squared magnitude.

---

### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations.

### Product output data type

Specify the product output data type. See “Multiplication Data Types” for illustrations depicting the use of the product output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

## Supported Data Types

Port	Supported Data Types
In	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>

<b>Port</b>	<b>Supported Data Types</b>
Output	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• 32-bit unsigned integers</li></ul>
Rst	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• Boolean</li><li>• 8-, 16-, and 32-bit signed integers</li><li>• 8-, 16-, and 32-bit unsigned integers</li></ul>

## See Also

histogram

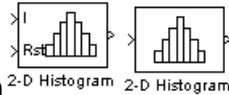
MATLAB

## 2-D Histogram (To Be Removed)

Generate histogram of each input matrix

### Library

Statistics



### Description

---

**Note:** The 2-D Histogram block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox software. Use the replacement block `Histogram`.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

## 2-D IDCT

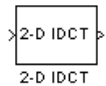
Compute 2-D inverse discrete cosine transform (IDCT)

### Library

Transforms

visiontransforms

### Description



The 2-D IDCT block calculates the two-dimensional inverse discrete cosine transform of the input signal. The equation for the two-dimensional IDCT is

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} C(m)C(n)F(m,n) \cos \frac{(2x+1)m\pi}{2M} \cos \frac{(2y+1)n\pi}{2N}$$

where  $F(m,n)$  is the DCT of the signal  $f(x,y)$  and  $C(m), C(n) = \frac{1}{\sqrt{2}}$  for  $m, n = 0$  and  $C(m), C(n) = 1$  otherwise.

The number of rows and columns of the input signal must be powers of two. The output of this block has dimensions the same dimensions as the input.

Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> </ul>	No

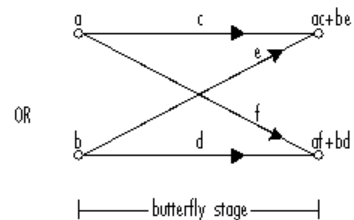
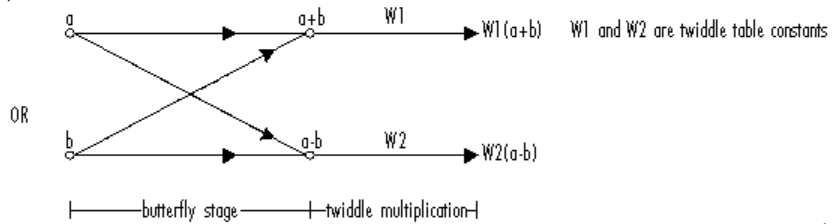
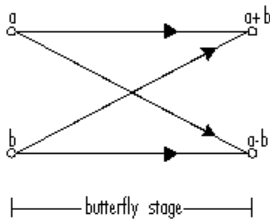
Port	Input/Output	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	
Output	2-D IDCT of the input	Same as Input port	No

If the data type of the input signal is floating point, the output of the block is the same data type.

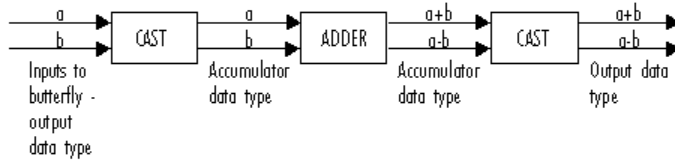
Use the **Sine and cosine computation** parameter to specify how the block computes the sine and cosine terms in the IDCT algorithm. If you select `Trigonometric fcn`, the block computes the sine and cosine values during the simulation. If you select `Table lookup`, the block computes and stores the trigonometric values before the simulation starts. In this case, the block requires extra memory.

## Fixed-Point Data Types

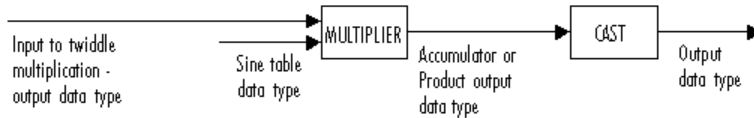
The following diagram shows the data types used in the 2-D IDCT block for fixed-point signals. Inputs are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



**Butterfly Stage Data Types**



**Twiddle Multiplication Data Types**



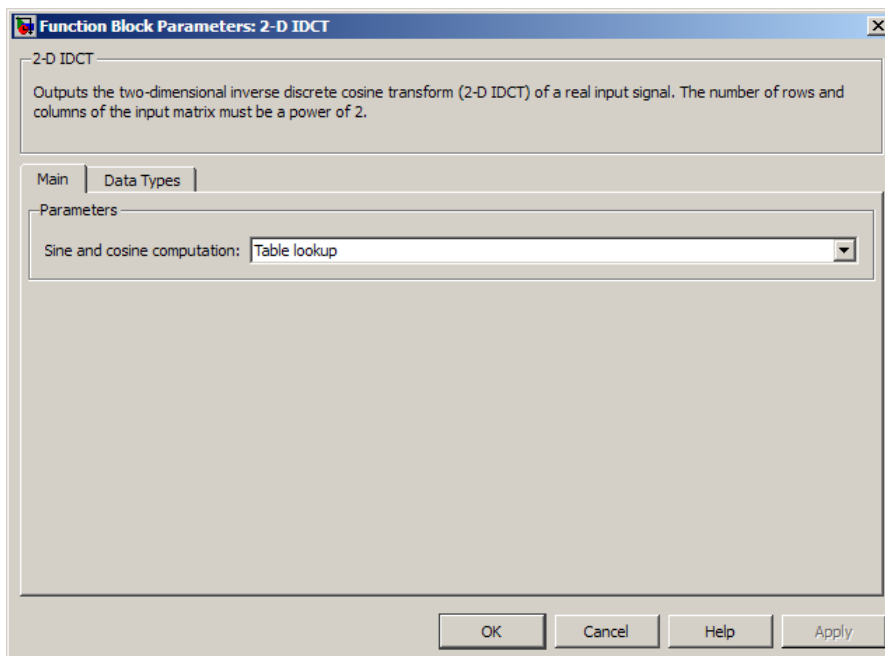
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the



complex multiplication performed, refer to “Multiplication Data Types”. You can set the sine table, product output, accumulator, and output data types in the block mask as discussed in the next section.

## Dialog Box

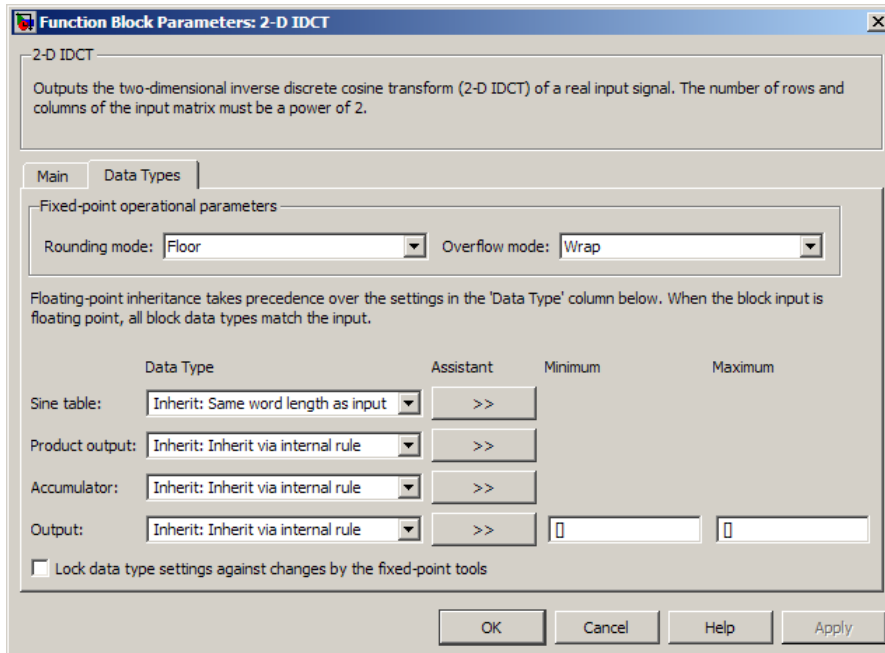
The **Main** pane of the 2-D IDCT dialog box appears as shown in the following figure.



### Sine and cosine computation

Specify how the block computes the sine and cosine terms in the IDCT algorithm. If you select **Trigonometric fcn**, the block computes the sine and cosine values during the simulation. If you select **Table lookup**, the block computes and stores the trigonometric values before the simulation starts. In this case, the block requires extra memory.

The **Data Types** pane of the 2-D IDCT dialog box appears as shown in the following figure.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

### Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:


- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to **Nearest**.

### Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 1-75 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

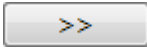
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-75 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 1-75 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.


When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{ideal\ output} = WL_{input} + \text{floor}(\log_2(DCT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### **Lock scaling against changes by the autoscaling tool**

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

### **Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## **References**

- [1] Chen, W.H, C.H. Smith, and S.C. Fralick, “A fast computational algorithm for the discrete cosine transform,” *IEEE Trans. Commun.*, vol. COM-25, pp. 1004-1009. 1977.
- [2] Wang, Z. “Fast algorithms for the discrete W transform and for the discrete Fourier transform,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-32, pp. 803-816, Aug. 1984.

## See Also

2-D DCT	Computer Vision System Toolbox software
2-D FFT	Computer Vision System Toolbox software
2-D IFFT	Computer Vision System Toolbox software

**Introduced before R2006a**

## 2-D IFFT

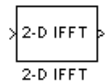
2-D Inverse fast Fourier transform of input

### Library

Transforms

visiontransforms

### Description



The 2-D IFFT block computes the inverse fast Fourier transform (IFFT) of an  $M$ -by- $N$  input matrix in two steps. First, it computes the one-dimensional IFFT along one dimension (row or column). Next, it computes the IFFT of the output of the first step along the other dimension (column or row).

The output of the IFFT block is equivalent to the MATLAB `ifft2` function:

```
y = ifft2(A) % Equivalent MATLAB code
```

Computing the IFFT of each dimension of the input matrix is equivalent to calculating the two-dimensional inverse discrete Fourier transform (IDFT), which is defined by the following equation:

$$f(x,y) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(m,n) e^{j\frac{2\pi mx}{M}} e^{j\frac{2\pi ny}{N}}$$

where  $0 \leq x \leq M-1$  and  $0 \leq y \leq N-1$ .

The output of this block has the same dimensions as the input. If the input signal has a floating-point data type, the data type of the output signal uses the same floating-point

data type. Otherwise, the output can be any fixed-point data type. The block computes scaled and unscaled versions of the IFFT.

The input to this block can be floating-point or fixed-point, real or complex, and conjugate symmetric. The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library [1], [2], or an implementation based on a collection of Radix-2 algorithms. You can select **Auto** to allow the block to choose the implementation.

## Port Description

Port	Description	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes
Output	2-D IFFT of the input	Same as Input port	Yes

## FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to MATLAB host computers. The data type must be floating-point. Refer to “Simulink Coder” for more details on generating code.

## Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the “Simulink Coder”. The dimensions of the input matrix,  $M$  and  $N$ , must be powers of two. To work

with other input sizes, use the **Image Pad** block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

**Radix-2 Algorithms for Real or Complex Input Complexity Floating-Point Signals**

Parameter Settings	Algorithms Used for IFFT Computation
<input type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT
<input checked="" type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF
<input type="checkbox"/> Input is in bit-reversed order <input checked="" type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
<input checked="" type="checkbox"/> Input is in bit-reversed order <input checked="" type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

**Radix-2 Algorithms for Real or Complex Input Complexity Fixed-Point Signals**

Other Parameter Settings	Algorithms Used for IFFT Computation
<input type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Butterfly operation and radix-2 DIT
<input checked="" type="checkbox"/> Input is in bit-reversed order <input type="checkbox"/> Input is conjugate symmetric	Radix-2 DIF

---

**Note:** The **Input is conjugate symmetric** parameter cannot be used for fixed-point signals.

---



### Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix-2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where  $K$  is the greater value of either  $M$  or  $N$  and  $k = 0, \dots, K - 1$ . The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

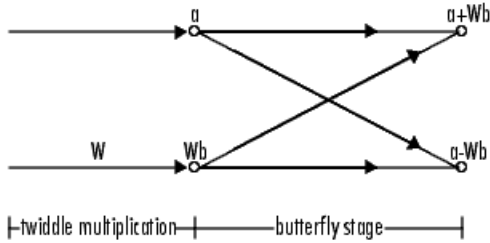
Number of Table Entries for N-Point FFT	
floating-point	3 $N/4$
fixed-point	$N$

### Fixed-Point Data Types

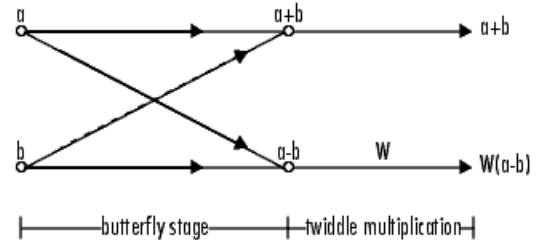
The following diagrams show the data types used in the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT dialog box as discussed in "Dialog Box" on page 1-87.

Inputs to the IFFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT and after each butterfly stage in a decimation-in-frequency IFFT.

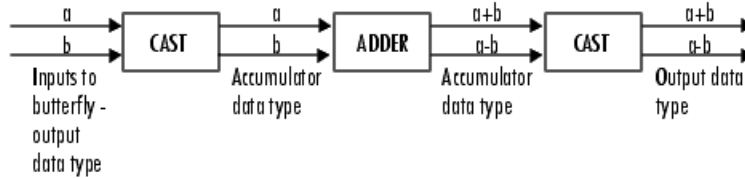
**Decimation-in-time IFFT**



**Decimation-in-frequency IFFT**



**Butterfly stage data types**



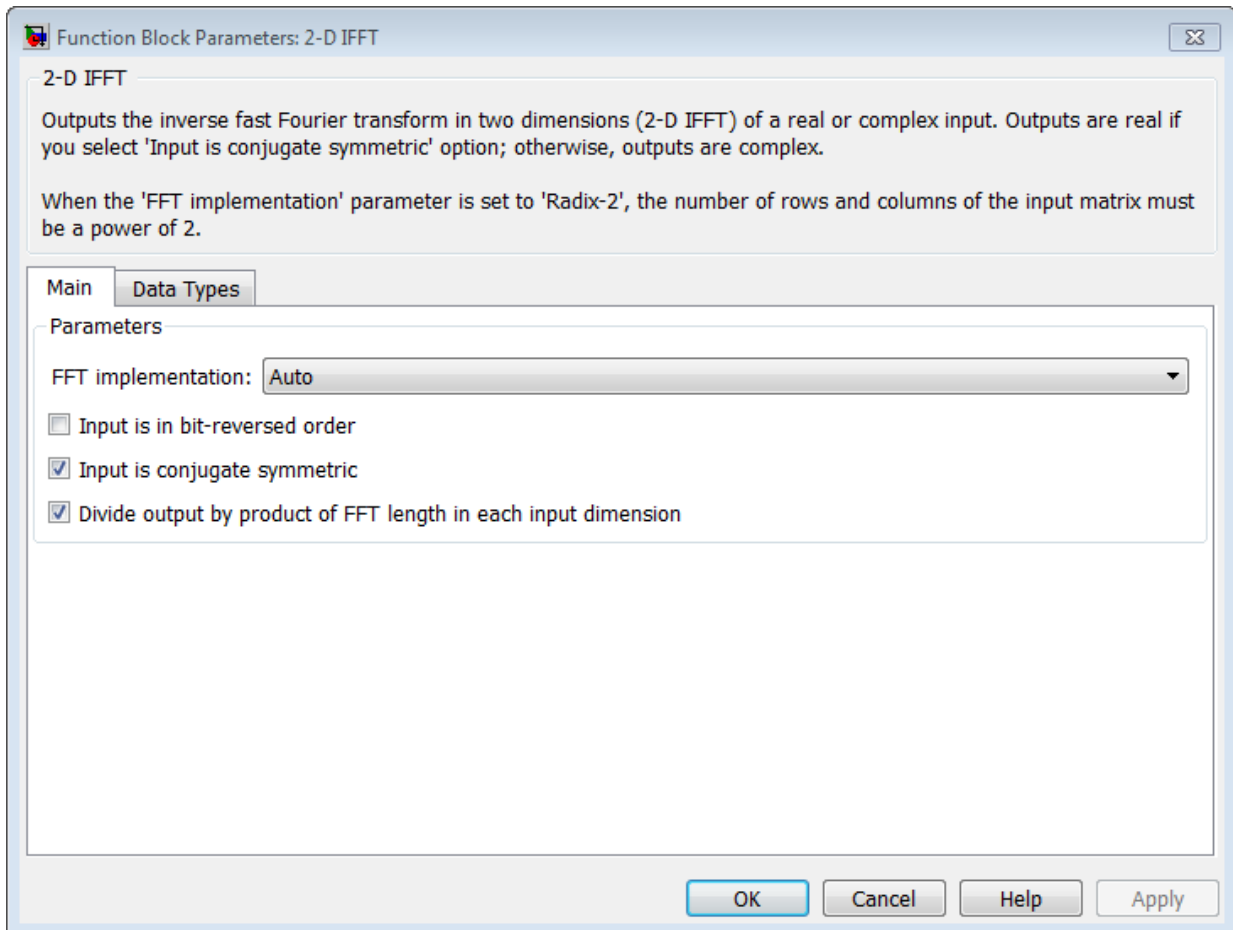
**Twiddle multiplication data types**



The multiplier output appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to “Multiplication Data Types” in the DSP System Toolbox documentation.

## Dialog Box

The **Main** pane of the 2-D IFFT dialog box appears as shown in the following figure.



### FFT implementation

Set this parameter to FFTW [1], [2] to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the “Simulink Coder”. The dimensions of the input matrix,  $M$  and  $N$ , must be powers of two. To work with other input sizes, use the `Image Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the `FFTW` implementation. See “Radix-2 Implementation” on page 1-83.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

#### **Input is in bit-reversed order**

Select or clear this check box to designate the order of the input channel elements. Select this check box when the input should appear in reversed order, and clear it when the input should appear in linear order. The block yields invalid outputs when you do not set this parameter correctly. This check box only appears when you set the **FFT implementation** parameter to `Radix-2` or `Auto`.

For more information ordering of the output, see “Bit-Reversed Order” on page 1-51. The 2-D FFT block bit-reverses the order of both the columns and the rows.

#### **Input is conjugate symmetric**

Select this option when the block inputs both floating point and conjugate symmetric, and you want real-valued outputs. This parameter cannot be used for fixed-point signals. Selecting this check box optimizes the block's computation method.

The FFT block yields conjugate symmetric output when you input real-valued data. Taking the IFFT of a conjugate symmetric input matrix produces real-valued output. Therefore, if the input to the block is both floating point and conjugate symmetric, and you select the this check box, the block produces real-valued outputs.

If the IFFT block inputs conjugate symmetric data and you do not select this check box, the IFFT block outputs a complex-valued signal with small imaginary parts. The block outputs invalid data if you select this option with non conjugate symmetric input data.

#### **Divide output by product of FFT length in each input dimension**

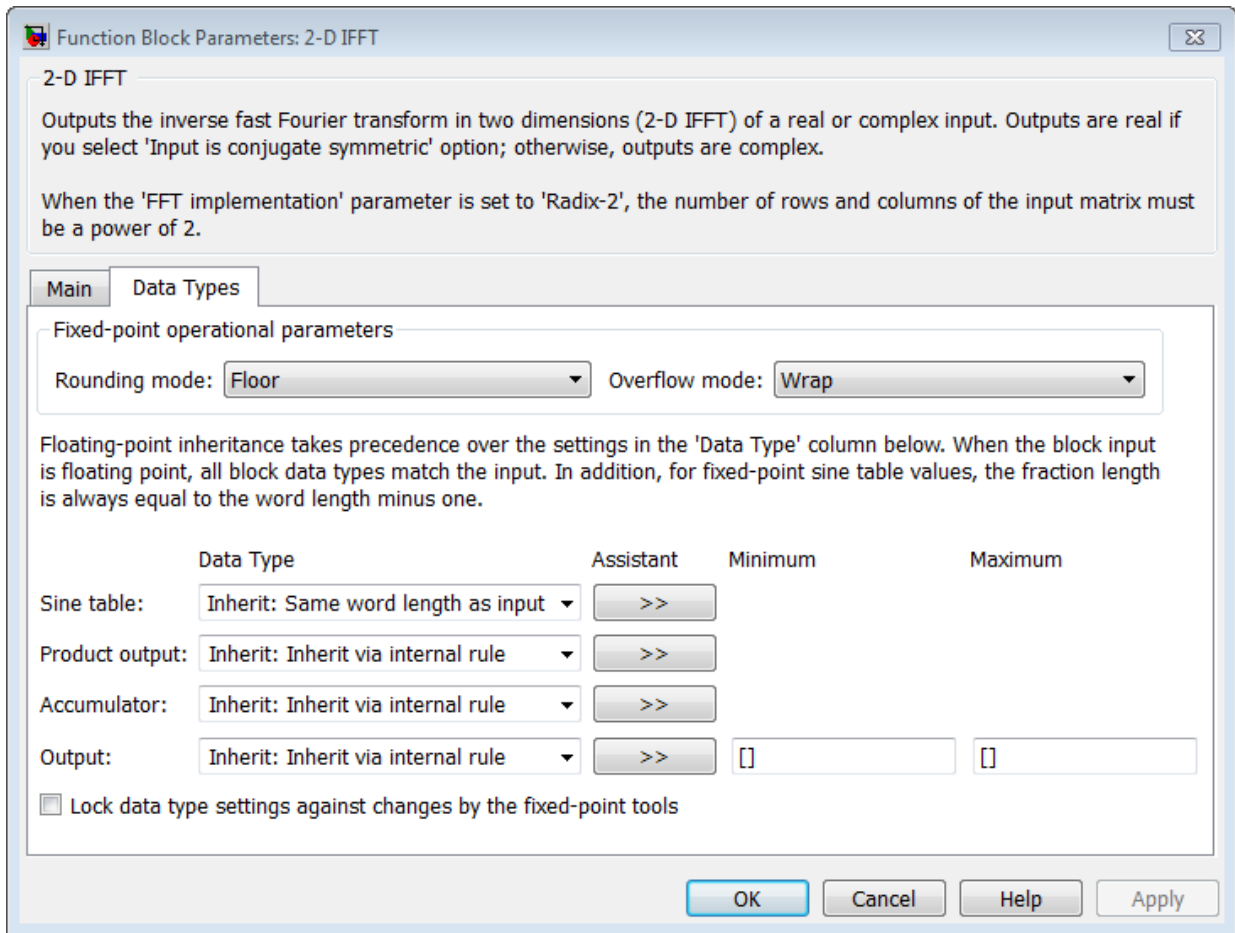
Select this check box to compute the scaled IFFT. The block computes scaled and unscaled versions of the IFFT. If you select this option, the block computes the scaled version of the IFFT. The unscaled IFFT is defined by the following equation:

$$f(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(m,n) e^{j\frac{2\pi mx}{M}} e^{j\frac{2\pi ny}{N}}$$

where  $0 \leq x \leq M-1$  and  $0 \leq y \leq N-1$ .

The scaled version of the IFFT multiplies the above unscaled version by  $\frac{1}{MN}$ .

The **Data Types** pane of the 2-D IFFT dialog box appears as shown in the following figure.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to Nearest.

### Overflow mode

Select the Overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

### Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Same word length as input**
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to **Nearest**.

### Product output data type

Specify the product output data type. See and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**.

When you select **Inherit: Inherit via internal rule**, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

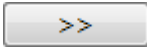
- When you select the **Divide butterfly outputs by two** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
- When you clear the **Divide butterfly outputs by two** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + \text{floor}(\log_2(\text{FFT length} - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### **Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] FFTW (<http://www.fftw.org>)



- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## See Also

2-D DCT	Computer Vision System Toolbox software
2-D FFT	Computer Vision System Toolbox software
2-D IDCT	Computer Vision System Toolbox software
2-D FFT	Computer Vision System Toolbox software
2-D IFFT	Computer Vision System Toolbox software
bitrevorder	Signal Processing Toolbox software
fft	MATLAB
ifft	MATLAB
"Simulink Coder"	Simulink

**Introduced before R2006a**

## 2-D IFFT (To Be Removed)

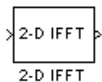
Compute 2-D IFFT of input

### Library

Transforms

viptransforms

### Description



---

**Note:** The 2-D IFFT block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox software. Use the replacement block `2-D IFFT`.

Refer to “FFT and IFFT Support for Non-Power-of-Two Transform Length with FFTW Library” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

## 2-D Maximum

Find maximum values in input or sequence of inputs

### Library

Statistics

visionstatistics



The 2-D Maximum block identifies the value and/or position of the smallest element in each row or column of the input, or along a specified dimension of the input. The 2-D Maximum block can also track the maximum values in a sequence of inputs over a period of time.

The 2-D Maximum block supports real and complex floating-point, fixed-point, and Boolean inputs. Real fixed-point inputs can be either signed or unsigned, while complex fixed-point inputs must be signed. The output data type of the maximum values match the data type of the input. The block outputs `double` index values, when the input is `double`, and `uint32` otherwise.

### Port Descriptions

Port	Input/Output	Supported Data Types
Input	Scalar, vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul>

Port	Input/Output	Supported Data Types
		<ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Rst	Scalar value	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Val	Maximum value output based on the “Value Mode” on page 1-96	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Idx	One-based output location of the maximum value based on the “Index Mode” on page 1-97	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• 32-bit unsigned integers</li> </ul>

## Value Mode

When you set the **Mode** parameter to **Value**, the block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each sample time, and outputs the array  $y$ . Each element in  $y$  is the maximum value in the corresponding column, row, vector, or entire input. The output  $y$  depends on the setting of the **Find the maximum value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the maximum value of each vector over the second dimension

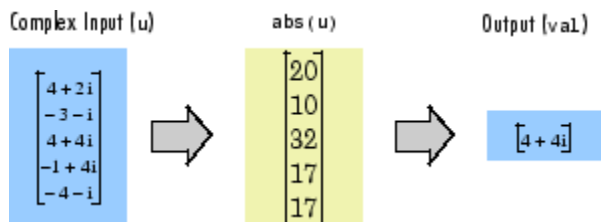
of the input. For an  $M$ -by- $N$  input matrix, the block outputs an  $M$ -by-1 column vector at each sample time.

- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the maximum value of each vector over the first dimension of the input. For an  $M$ -by- $N$  input matrix, the block outputs a 1-by- $N$  row vector at each sample time.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Entire input** — The output at each sample time is a scalar that contains the maximum value in the  $M$ -by- $N$ -by- $P$  input matrix.
- **Specified dimension** — The output at each sample time depends on **Dimension**. When you set **Dimension** to 1, the block output is the same as when you select **Each column**. When you set **Dimension** to 2, the block output is the same as when you select **Each row**. When you set **Dimension** to 3, the block outputs an  $M$ -by- $N$  matrix containing the maximum value of each vector over the third dimension of the input, at each sample time.

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the maximum magnitude squared as shown below. For complex value  $u = a + bi$ , the magnitude squared is  $a^2 + b^2$ .



## Index Mode

When you set the **Mode** parameter to **Index**, the block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array  $I$ . Each element in  $I$  is an integer indexing the maximum value in the corresponding column, row, vector, or entire input. The output  $I$  depends on the setting of the **Find the maximum value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the index of the maximum value of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the index of the maximum value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Entire input** — The output at each sample time is a 1-by-3 vector that contains the location of the maximum value in the  $M$ -by- $N$ -by- $P$  input matrix. For an input that is an  $M$ -by- $N$  matrix, the output will be a 1-by-2 vector of one-based [x y] location coordinates for the maximum value.
- **Specified dimension** — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the indices of the maximum values of each vector over the third dimension of the input.

When a maximum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector [3 2 1 2 3]', the computed one-based index of the maximum value is 1 rather than 5 when **Each column** is selected.

When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

## Value and Index Mode

When you set the **Mode** parameter to **Value** and **Index**, the block outputs both the maxima and the indices.

## Running Mode

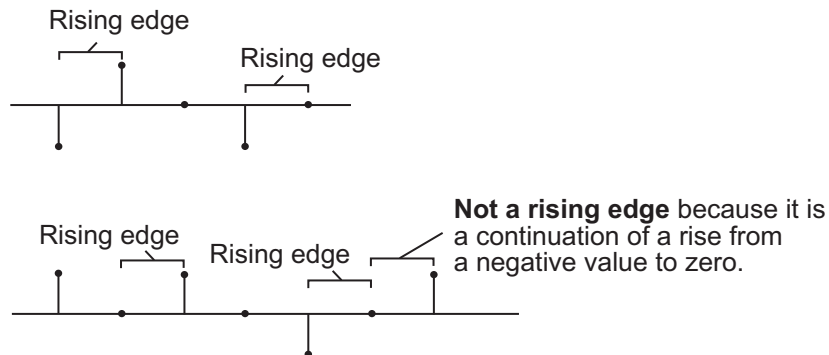
When you set the **Mode** parameter to **Running**, the block tracks the maximum value of each channel in a time sequence of  $M$ -by- $N$  inputs. In this mode, the block treats each element as a channel.

## Resetting the Running Maximum

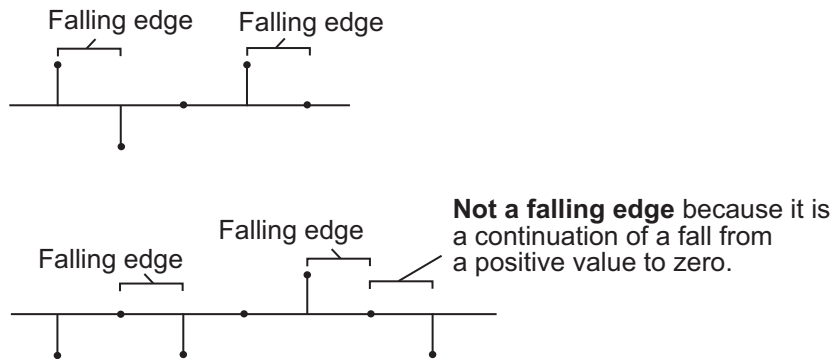
The block resets the running maximum whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** menu:

- **None** — Disables the Rst port.
- **Rising edge** — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the **Rst** input is a **Rising edge** or **Falling edge** (as described above)
- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero

---

**Note:** When running simulations in the Simulink **MultiTasking** mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset.

---

## ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This applies to any mode other than running mode and when you set the **Find the maximum value over** parameter to **Entire input** and you select the **Enable ROI processing** check box. ROI processing applies only for 2-D inputs.

You can specify a rectangle, line, label matrix, or binary mask ROI type.

Use the binary mask to specify which pixels to highlight or select.

Use the label matrix to label regions. Pixels set to **0** represent the background. Pixels set to **1** represent the first object, pixels set to **2**, represent the second object, and so on. Use the **Label Numbers** port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix.



For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter. For more information about the format of the input to the ROI port when you set the ROI to a rectangle or a line, see the Draw Shapes block reference page.

### ROI Output Statistics

#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

#### Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

#### Output = Single statistic for all ROIs

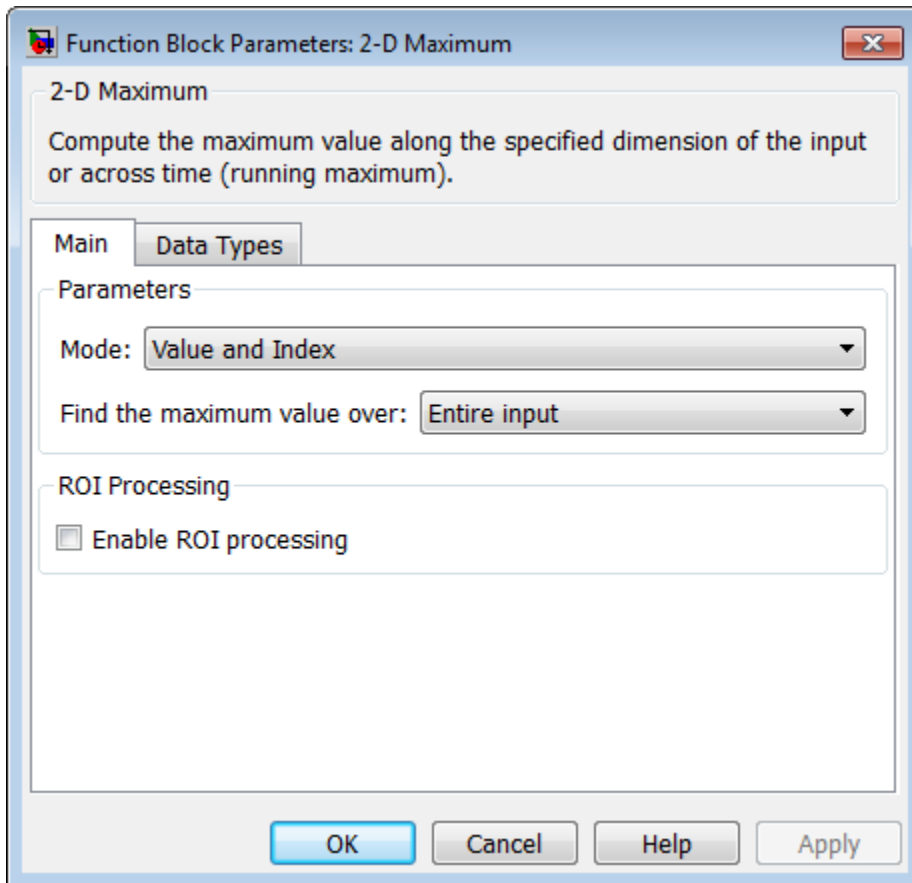
Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

## **Fixed-Point Data Types**

The parameters on the **Data Types** pane of the block dialog are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in “Value Mode” on page 1-96. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

## **Dialog Box**

The **Main** pane of the Maximum block dialog appears as follows.



## Mode

Specify the block's mode of operation:

- **Value and Index** — Output both the value and the index location.
- **Value** — Output the maximum value of each input matrix. For more information, see “Value Mode” on page 1-96.
- **Index** — Output the one-based index location of the maximum value. For more information, see “Index Mode” on page 1-97.
- **Running** — Track the maximum value of the input sequence over time. For more information, see “Running Mode” on page 1-98.

For the **Value**, **Index**, and **Value and Index** modes, the 2-D Maximum block produces identical results as the MATLAB `max` function when it is called as `[y I] = max(u, [], D)`, where  $u$  and  $y$  are the input and output, respectively,  $D$  is the dimension, and  $I$  is the index.

### Find the maximum value over

Specify whether the block should find the maximum of the entire input each row, each column, or dimensions specified by the **Dimension** parameter.

### Reset port

Specify the reset event that causes the block to reset the running maximum. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter appears only when you set the **Mode** parameter to **Running**. For information about the possible values of this parameter, see “Resetting the Running Maximum” on page 1-99.

### Dimension

Specify the dimension (one-based value) of the input signal, over which the maximum is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter applies only when you set the **Find the maximum value over** parameter to **Specified dimension**.

### Enable ROI processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter applies only when you set the **Find the maximum value over** parameter to **Entire input**, and the block is not in running mode.

### ROI type

Specify the type of ROI you want to use. Your choices are **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**.

When you set this parameter to **Rectangles** or **Lines**, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the **Flag** port appears on the block.

When you set this parameter to **Label matrix**, the **Label** and **Label Numbers** ports appear on the block and the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the **Flag** port appears on the block.

See for details.

**ROI portion to process**

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter applies only when you set the **ROI type** parameter to **Rectangles**.

**Output**

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter does not apply when you set the **ROI type** parameter, to **Binary mask**.

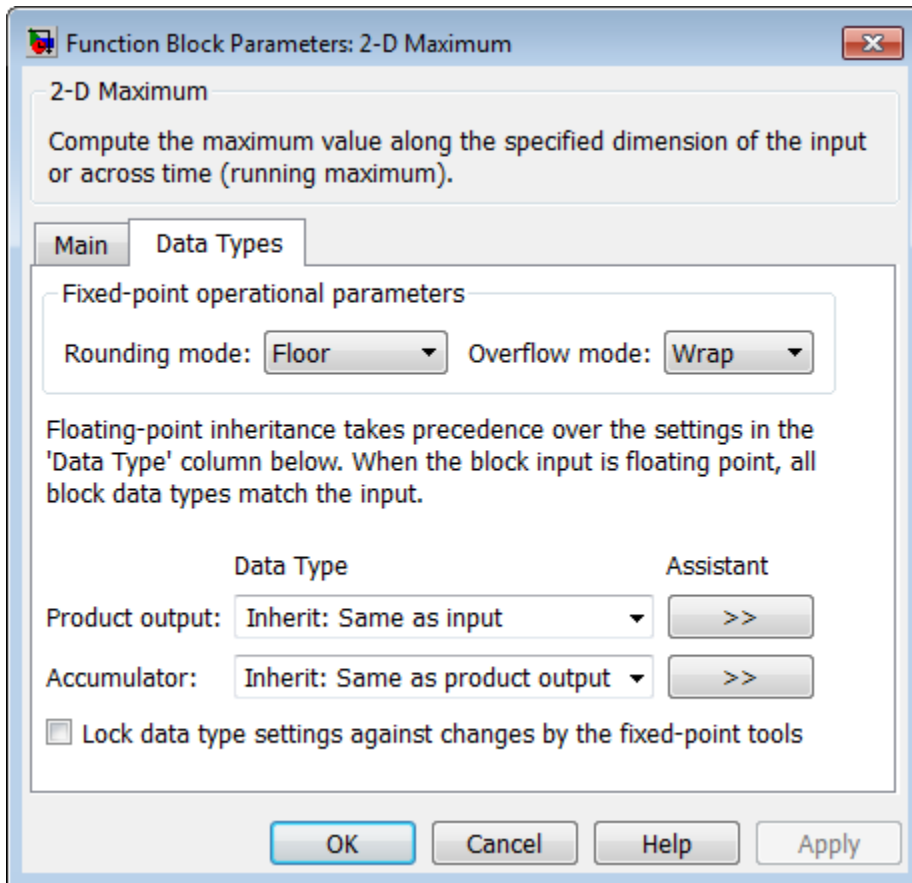
**Output flag indicating if ROI is within image bounds**

When you select this check box, the **Flag** port appears on the block. This check box applies only when you set the **ROI type** parameter to **Rectangles** or **Lines**. For a description of the **Flag** port output, see the tables in “ROI Processing” on page 1-100.

**Output flag indicating if label numbers are valid**

When you select this check box, the **Flag** port appears on the block. This check box applies only when you set the **ROI type** parameter to **Label matrix**. For a description of the **Flag** port output, see the tables in “ROI Processing” on page 1-100.

The **Data Types** pane of the Maximum block dialog appears as follows.




---

**Note:** The parameters on the **Data Types** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in “Value Mode” on page 1-96. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

---

### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations.

### Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 1-102 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-102 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

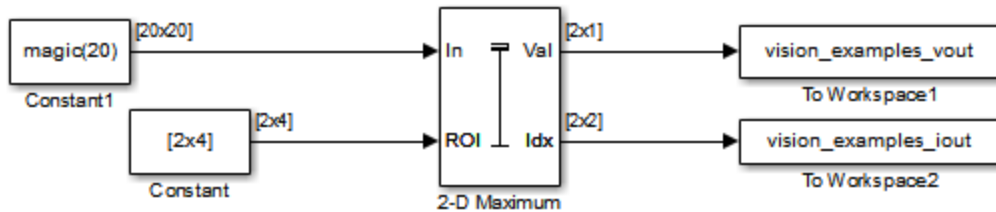
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

## Examples



The `ex_vision_2dmaximum` example finds the maximum value within two ROIs. The model outputs the maximum values and their one-based [x y] coordinate locations.

## See Also

2-D Mean

2-D Minimum

MinMax

max

Computer Vision  
System Toolbox

Computer Vision  
System Toolbox

Simulink

MATLAB

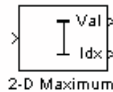


## 2-D Maximum (To Be Removed)

Find maximum values in an input or sequence of inputs

### Library

vipobslib



### Description

---

**Note:** This 2-D Maximum block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated 2-D Maximum block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

## 2-D Mean

Find mean value of each input matrix

### Library

Statistics

visionstatistics



### Description 2-D Mean

The 2-D Mean block computes the mean of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The 2-D Mean block can also track the mean value in a sequence of inputs over a period of time. To track the mean value in a sequence of inputs, select the **Running mean** check box.

### Port Description

Port	Supported Data Types
Input	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Reset	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
ROI	Rectangles and lines:

Port	Supported Data Types
	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> Binary Mask: <ul style="list-style-type: none"> <li>• Boolean</li> </ul>
Label	<ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Label Numbers	<ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Output	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Flag	<ul style="list-style-type: none"> <li>• Boolean</li> </ul>

## Basic Operation

When you do not select the **Running mean** check box, the block computes the mean value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each individual sample time. Each element in the output array  $y$  is the mean value of the corresponding column, row, vector, or entire input. The output array,  $y$ , depends on the setting of the **Find the mean value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Entire input** — The output at each sample time is a scalar that contains the mean value of the  $M$ -by- $N$ -by- $P$  input matrix.

```
y = mean(u(:))      % Equivalent MATLAB code
```

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the mean value of each vector over the second dimension of the

input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.

```
y = mean(u,2) % Equivalent MATLAB code
```

- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the mean value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

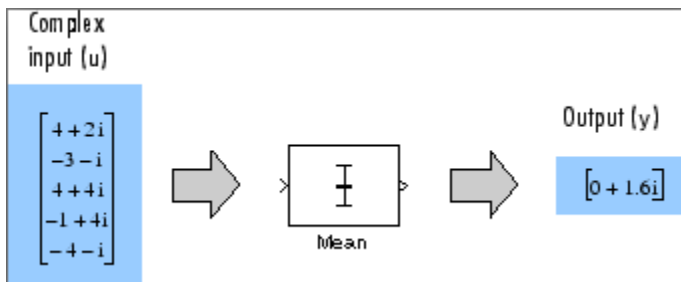
```
y = mean(u) % Equivalent MATLAB code
```

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the mean value of each vector over the third dimension of the input.

```
y = mean(u,Dimension) % Equivalent MATLAB code
```

The mean of a complex input is computed independently for the real and imaginary components, as shown in the following figure.



## Running Operation

When you select the **Running mean** check box, the block tracks the mean value of each channel in a time sequence of inputs. In this mode, the block treats each element as a channel.

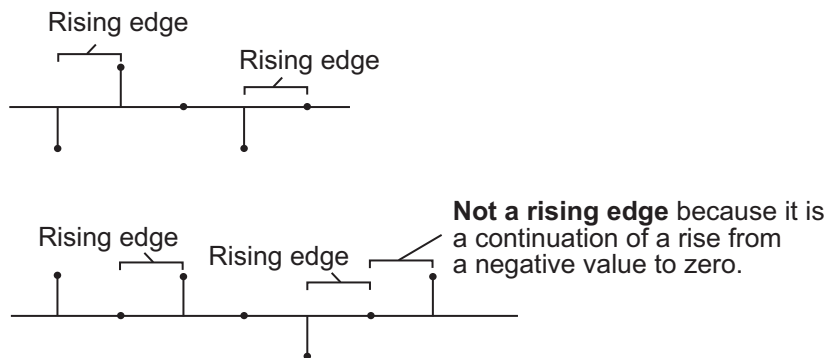
## Resetting the Running Mean

The block resets the running mean whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

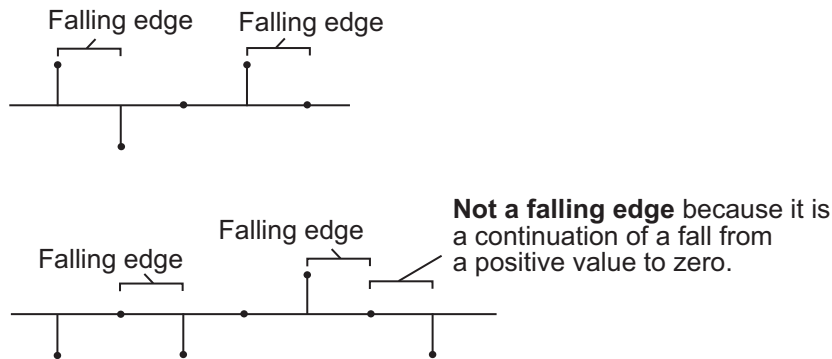
When a reset event occurs, the running mean for each channel is initialized to the value in the corresponding channel of the current input.

You specify the reset event by the **Reset port** parameter:

- **None** disables the Rst port.
- **Rising edge** — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a **Rising edge** or **Falling edge** (as described earlier)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero

---

**Note:** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset.

---

## ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This option is only available when the **Find the mean value over** parameter is set to **Entire input** and the **Running mean** check box is not selected. ROI processing is only supported for 2-D inputs.

- A binary mask is a binary image that enables you to specify which pixels to highlight, or select.
- In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to **Label matrix**, the **Label** and **Label Numbers** ports appear on the block. Use the **Label Numbers** port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix.

- For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the [Draw Shapes](#) reference page.

For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the [Draw Shapes](#) block reference page.

---

**Note:** For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

---

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

If, for the **ROI type** parameter, you select **Rectangles** or **Lines**, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

#### Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select **Label matrix**, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this

check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

**Output = Individual statistics for each ROI**

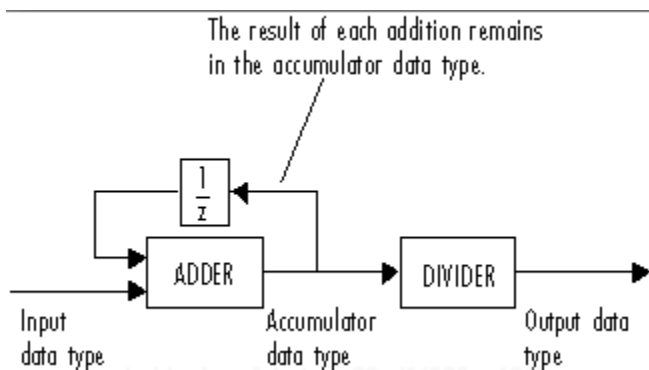
Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

**Output = Single statistic for all ROIs**

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

**Fixed-Point Data Types**

The following diagram shows the data types used within the Mean block for fixed-point signals.

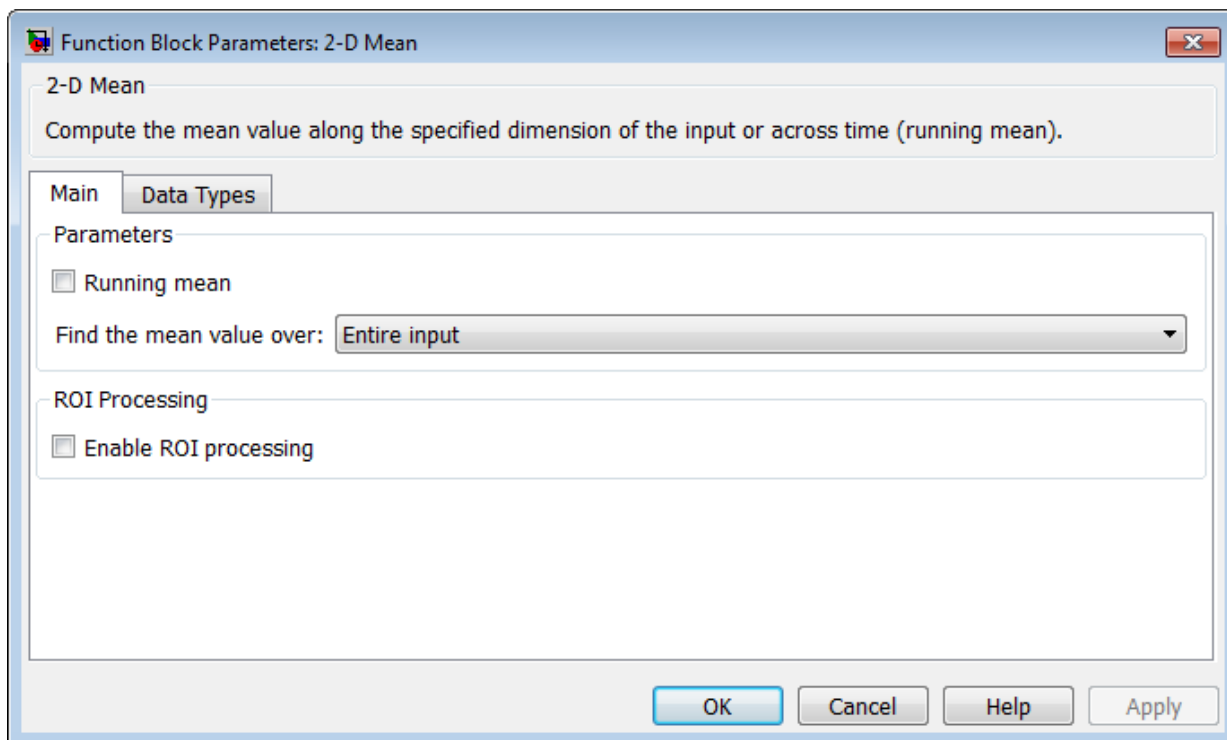


You can set the accumulator and output data types in the block dialog, as discussed in “Dialog Box” on page 1-117.



## Dialog Box

The **Main** pane of the Mean block dialog appears as follows.



### Running mean

Enables running operation when selected.

### Reset port

Specify the reset event that causes the block to reset the running mean. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running mean** check box. For more information, see “Resetting the Running Mean” on page 1-113.

### Find the mean value over

Specify whether to find the mean value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see “Basic Operation” on page 1-111.

### **Dimension**

Specify the dimension (one-based value) of the input signal, over which the mean is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the mean value over** parameter is set to **Specified dimension**.

### **Enable ROI Processing**

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the mean value over** parameter is set to **Entire input**, and the block is not in running mode.

### **ROI type**

Specify the type of ROI you want to use. Your choices are **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**.

### **ROI portion to process**

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify **Rectangles**.

### **Output**

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

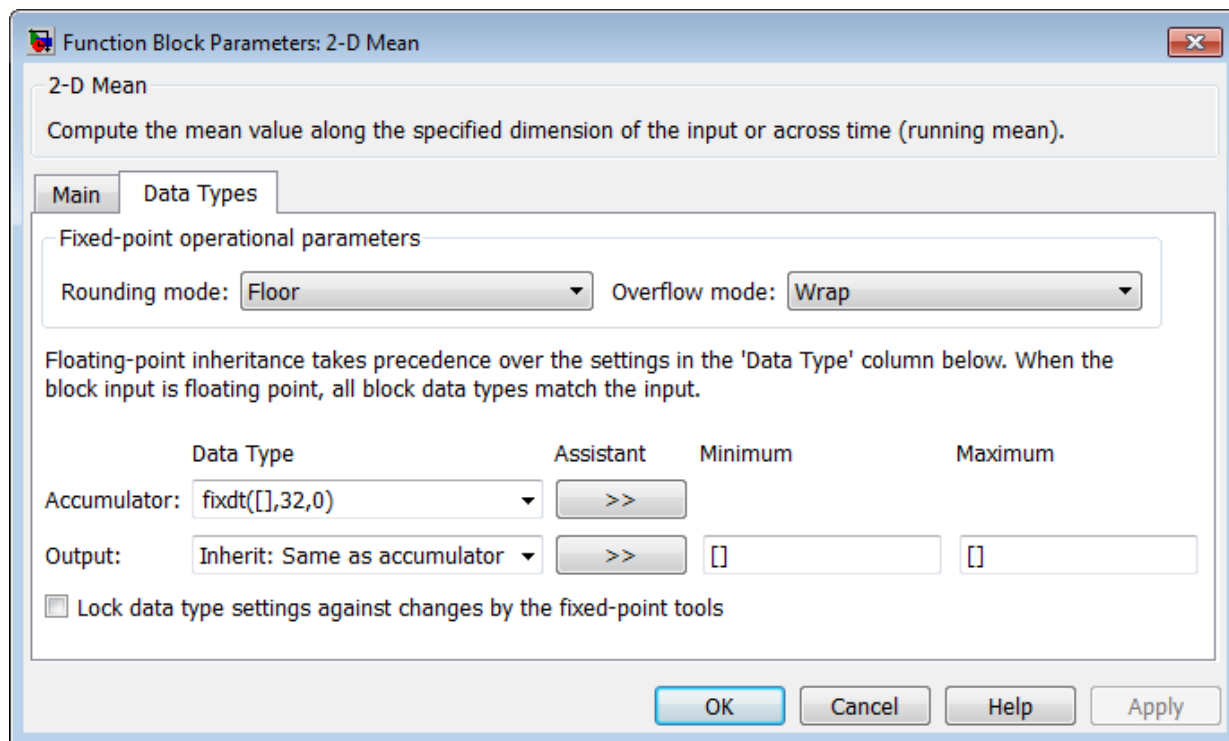
### **Output flag indicating if ROI is within image bounds**

When you select this check box, a Flag port appears on the block. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-114.

### **Output flag indicating if label numbers are valid**

When you select this check box, a Flag port appears on the block. This check box is visible only when you select **Label matrix** for the **ROI type** parameter. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-114.

The **Data Types** pane of the Mean block dialog appears as follows.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations.


### Overflow mode

Select the Overflow mode for fixed-point operations.

### Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-116 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

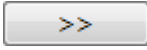
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 1-116 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Minimum

Specify the minimum value that the block should output. The default value, `[]`, is equivalent to `-Inf`. Simulink software uses this value to perform:

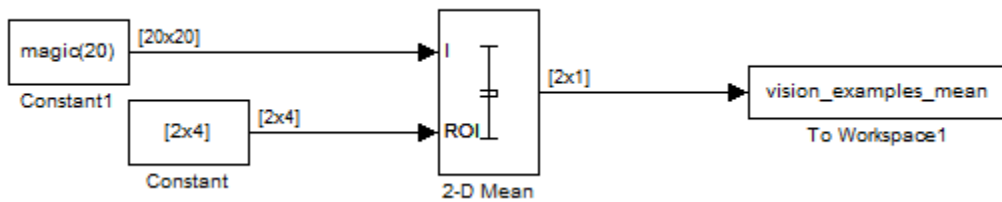
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

### Maximum

Specify the maximum value that the block should output. The default value, `[]`, is equivalent to `Inf`. Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

## Example



The `ex_vision_2dmean` calculates the mean value within two ROIs.

## See Also

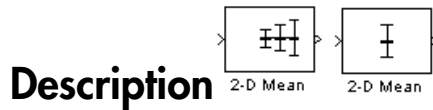
2-D Maximum	Computer Vision System Toolbox
2D-Median	Computer Vision System Toolbox
2-D Minimum	Computer Vision System Toolbox
2-D Standard Deviation	Computer Vision System Toolbox
mean	MATLAB

## 2-D Mean (To Be Removed)

Find mean value of each input matrix

### Library

Statistics



---

**Note:** This 2-D Mean block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated 2-D Mean block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

## 2-D Median

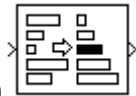
Find 2-D Median value of each input matrix

### Library

Statistics

visionstatistics

### Description



The 2-D Median block computes the median value of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The median of a set of input values is calculated as follows:

- 1 The values are sorted.
- 2 If the number of values is odd, the median is the middle value.
- 3 If the number of values is even, the median is the average of the two middle values.

For a given input  $u$ , the size of the output array  $y$  depends on the setting of the **Find the median value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Entire input** — The output at each sample time is a scalar that contains the median value of the  $M$ -by- $N$ -by- $P$  input matrix.

```
y = median(u(:)) % Equivalent MATLAB code
```

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the median value of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output is an  $M$ -by-1 column vector.

```
y = median(u,2) % Equivalent MATLAB code
```

- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the median value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

```
y = median(u) % Equivalent MATLAB code
```

For convenience, length- $M$  1-D vector inputs are treated as  $M$ -by-1 column vectors when the block is in this mode. Sample-based length- $M$  row vector inputs are also treated as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the median value of each vector over the third dimension of the input.

```
y = median(u,Dimension) % Equivalent MATLAB code
```

The block sorts complex inputs according to their magnitude.

## Fixed-Point Data Types

For fixed-point inputs, you can specify accumulator, product output, and output data types as discussed in “Dialog Box” on page 1-125. Not all these fixed-point parameters are applicable for all types of fixed-point inputs. The following table shows when each kind of data type and scaling is used.

	Output data type	Accumulator data type	Product output data type
<b>Even <math>M</math></b>	X	X	
<b>Odd <math>M</math></b>	X		
<b>Odd <math>M</math> and complex</b>	X	X	X
<b>Even <math>M</math> and complex</b>	X	X	X

The accumulator and output data types and scalings are used for fixed-point signals when  $M$  is even. The result of the sum performed while calculating the average of the two central rows of the input matrix is stored in the accumulator data type and scaling. The total result of the average is then put into the output data type and scaling.

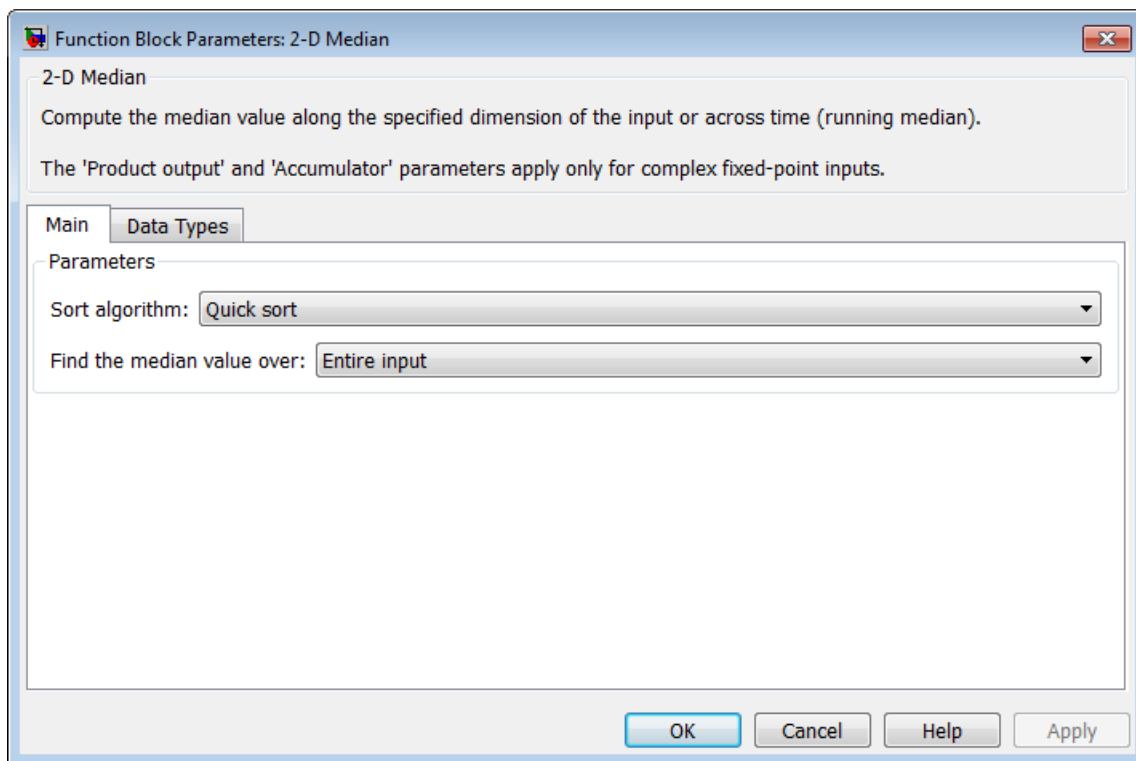


The accumulator and product output parameters are used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before the input elements are sorted, as described in Description. The results of the squares of the real and imaginary parts are placed into the product output data type and scaling. The result of the sum of the squares is placed into the accumulator data type and scaling.

For fixed-point inputs that are both complex and have even  $M$ , the data types are used in all of the ways described. Therefore, in such cases, the accumulator type is used in two different ways.

## Dialog Box

The **Main** pane of the 2-D Median block dialog appears as follows.



### Sort algorithm

Specify whether to sort the elements of the input using a **Quick sort** or an **Insertion sort** algorithm.

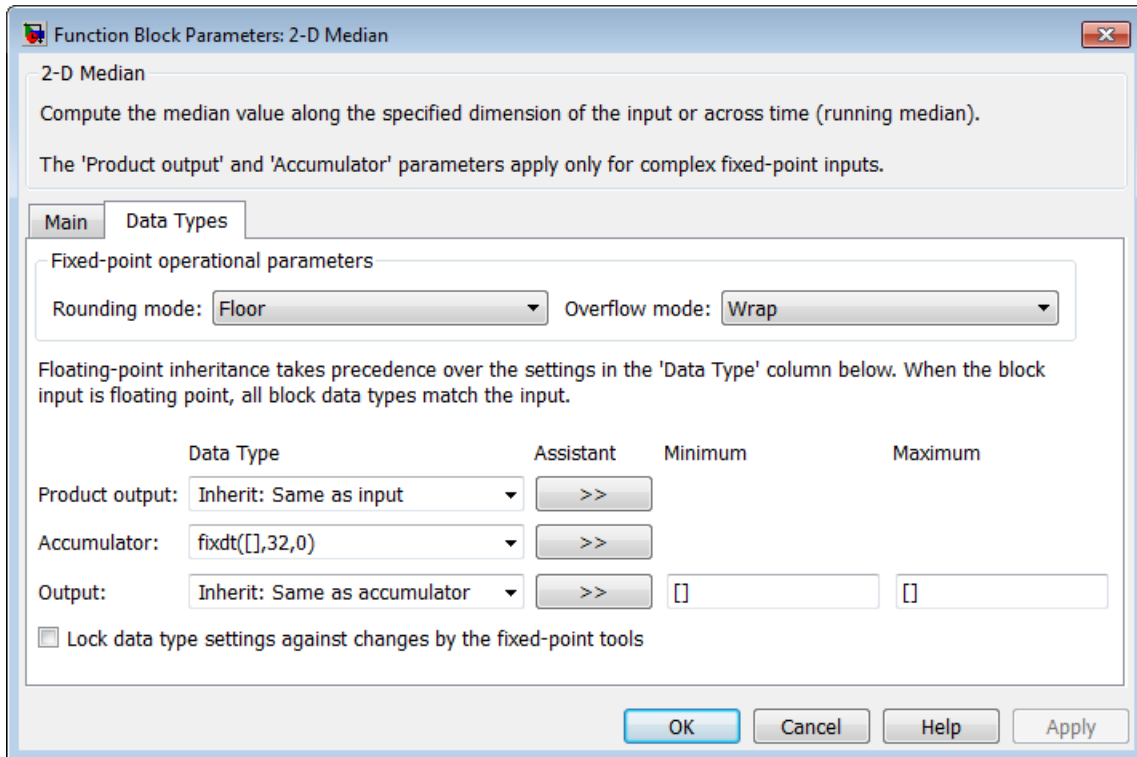
### Find the median value over

Specify whether to find the median value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see [Description](#).

### Dimension

Specify the dimension (one-based value) of the input signal, over which the median is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the median value over** parameter is set to **Specified dimension**.

The **Data Types** pane of the 2-D Median block dialog appears as follows.



---

**Note:** Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

---

### Rounding mode

Select the Rounding mode for fixed-point operations.

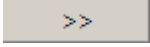
### Overflow mode

Select the Overflow mode for fixed-point operations.

### Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 1-124 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`

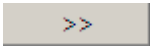
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-124 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`

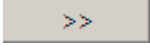
Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 1-124 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

### Minimum

Specify the minimum value that the block should output. The default value, `[]`, is equivalent to `-Inf`. Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

### Maximum

Specify the maximum value that the block should output. The default value, `[]`, is equivalent to `Inf`. Simulink software uses this value to perform:

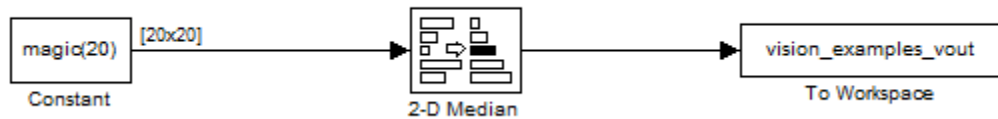
- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

## Supported Data Types

Port	Supported Data Types
Input	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• 8-, 16-, 32-, and 128-bit signed integers</li> <li>• 8-, 16-, 32-, and 128-bit unsigned integers</li> </ul>
Output	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• 8-, 16-, 32-, and 128-bit signed integers</li> <li>• 8-, 16-, 32-, and 128-bit unsigned integers</li> </ul>

## Examples

### Calculate Median Value Over Entire Input



The `ex_vision_2dmedian` calculates the median value over the entire input.

## See Also

2-D Maximum	Computer Vision System Toolbox
2-D Mean	Computer Vision System Toolbox
2-D Minimum	Computer Vision System Toolbox
2-D Standard Deviation	Computer Vision System Toolbox
2-D Variance	Computer Vision System Toolbox
median	MATLAB

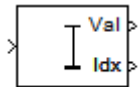
## 2-D Minimum

Find minimum values in input or sequence of inputs

### Library

Statistics

visionstatistics



### Description 2-D Minimum

The 2-D Minimum block identifies the value and/or position of the smallest element in each row or column of the input, or along a specified dimension of the input. The 2-D Minimum block can also track the minimum values in a sequence of inputs over a period of time.

The 2-D Minimum block supports real and complex floating-point, fixed-point, and Boolean inputs. Real fixed-point inputs can be either signed or unsigned, while complex fixed-point inputs must be signed. The output data type of the minimum values match the data type of the input. The block outputs `double` index values, when the input is `double`, and `uint32` otherwise.

### Port Description

Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Scalar, vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> </ul>	Yes

Port	Input/Output	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>8-, 16-, 32-bit signed integer</li> <li>8-, 16-, 32-bit unsigned integer</li> </ul>	
Rst	Scalar value	Boolean	No
Val	Minimum value output based on the “Value Mode” on page 1-131	Same as Input port	Yes
Idx	One-based output location of the minimum value based on the “Index Mode” on page 1-132	Same as Input port	No

Length- $M$  1-D vector inputs are treated as  $M$ -by-1 column vectors.

## Value Mode

When you set the **Mode** parameter to **Value**, the block computes the minimum value in each row, column, entire input, or over a specified dimension. The block outputs each element as the minimum value in the corresponding column, row, vector, or entire input. The output depends on the setting of the **Find the minimum value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

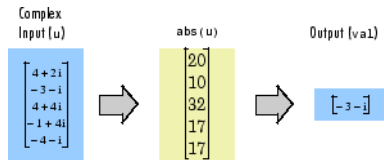
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the minimum value of each vector over the second dimension of the input. For an  $M$ -by- $N$  input matrix, the block outputs an  $M$ -by-1 column vector at each sample time.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the minimum value of each vector over the first dimension of the input. For an  $M$ -by- $N$  input matrix, the block outputs a 1-by- $N$  row vector at each sample time.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Entire input** — The output at each sample time is a scalar that contains the minimum value in the  $M$ -by- $N$ -by- $P$  input matrix.

- **Specified dimension** — The output at each sample time depends on **Dimension**. When you set **Dimension** to 1, the block output is the same as when you select **Each column**. When you set **Dimension** to 2, the block output is the same as when you select **Each row**. When you set **Dimension** to 3, the block outputs an  $M$ -by- $N$  matrix containing the minimum value of each vector over the third dimension of the input, at each sample time.

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the minimum magnitude squared as shown below. For complex value  $u = a + bi$ , the magnitude squared is  $a^2 + b^2$ .



## Index Mode

When you set the **Mode** parameter to **Index**, the block computes the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array  $I$ . Each element in  $I$  is an integer indexing the minimum value in the corresponding column, row, vector, or entire input. The output  $I$  depends on the setting of the **Find the minimum value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the index of the minimum value of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the index of the minimum value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.



- **Entire input** — The output at each sample time is a 1-by-3 vector that contains the location of the minimum value in the  $M$ -by- $N$ -by- $P$  input matrix. For an input that is an  $M$ -by- $N$  matrix, the output will be a 1-by-2 vector of one-based [x y] location coordinates for the minimum value.
- **Specified dimension** — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the indices of the minimum values of each vector over the third dimension of the input.

When a minimum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector  $[-1 \ 2 \ 3 \ 2 \ -1]'$ , the computed one-based index of the minimum value is 1 rather than 5 when **Each column** is selected.

## Value and Index Mode

When you set the **Mode** parameter to **Value** and **Index**, the block outputs both the minima, and the indices.

## Running Mode

When you set the **Mode** parameter to **Running**, the block tracks the minimum value of each channel in a time sequence of  $M$ -by- $N$  inputs. In this mode, the block treats each element as a channel.

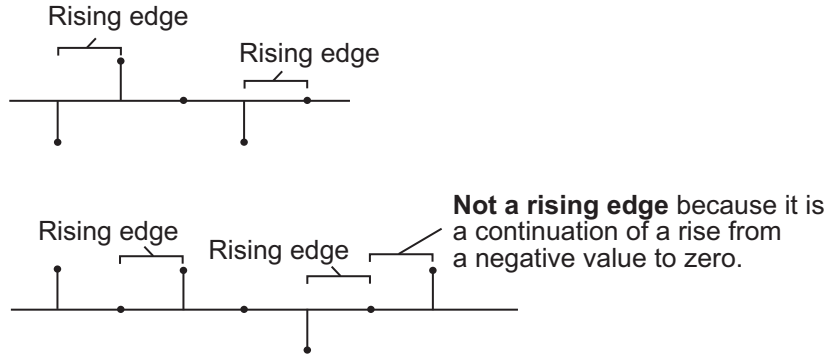
## Resetting the Running Minimum

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

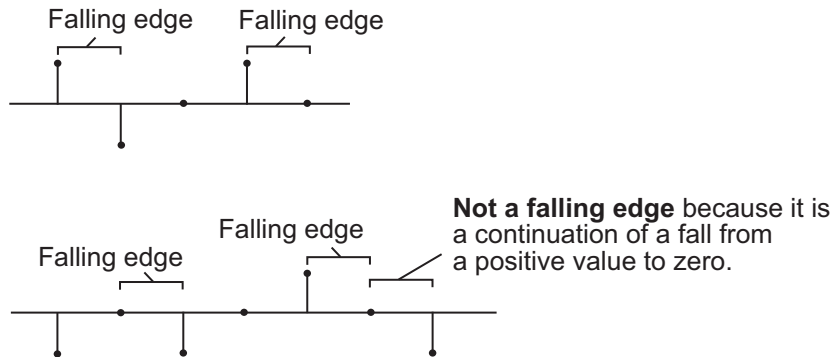
You specify the reset event by the **Reset port** parameter:

- **None** — Disables the **Rst** port
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:

- Rises from a negative value to a positive value or zero
- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a **Rising edge** or **Falling edge** (as described above)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero

---

**Note:** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset.

---

## ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This applies to any mode other than the running mode and when you set the **Find the minimum value over** parameter to Entire input and you select the **Enable ROI processing** check box. ROI processing applies only for 2-D inputs.

You can specify a rectangle, line, label matrix, or binary mask ROI type.

Use the binary mask to specify which pixels to highlight or select.

Use the label matrix to label regions. Pixels set to 0 represent the background. Pixels set to 1 represent the first object, pixels set to 2, represent the second object, and so on. Use the **Label Numbers** port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter. For more information about the format of the input to the ROI port when you set the ROI to a rectangle or a line, see the Draw Shapes block reference page.

### ROI Output Statistics

#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

#### Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.

Flag Port Output	Description
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

**Output = Individual statistics for each ROI**

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

**Output = Single statistic for all ROIs**

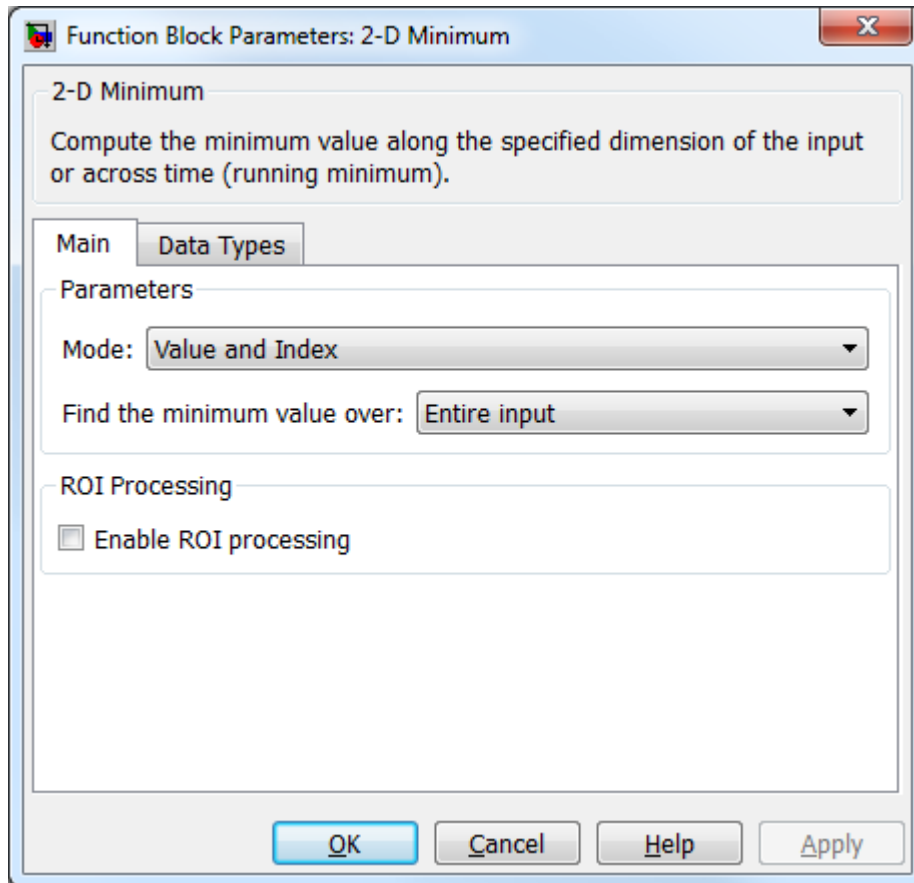
Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

## Fixed-Point Data Types

The parameters on the **Fixed-point** pane of the dialog box are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in “Value Mode” on page 1-131. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

## Dialog Box

The **Main** pane of the 2-D Minimum dialog box appears as shown in the following figure.



## Mode

Specify the block's mode of operation:

- **Value and Index** — Output both the value and the index location.
- **Value** — Output the minimum value of each input matrix. For more information, see “Value Mode” on page 1-131
- **Index** — Output the one-based index location of the minimum value. For more information, see “Index Mode” on page 1-132
- **Running** — Track the minimum value of the input sequence over time. For more information, see “Running Mode” on page 1-133.

For the **Value**, **Index**, and **Value and Index** modes, the 2-D Minimum block produces identical results as the MATLAB `min` function when it is called as `[y I] = min(u, [], D)`, where  $u$  and  $y$  are the input and output, respectively,  $D$  is the dimension, and  $I$  is the index.

### Find the minimum value over

Specify whether the block should find the minimum of the entire input each row, each column, or dimensions specified by the **Dimension** parameter.

### Reset port

Specify the reset event that causes the block to reset the running minimum. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter appears only when you set the **Mode** parameter to **Running**. For information about the possible values of this parameter, see “Resetting the Running Minimum” on page 1-133.

### Dimension

Specify the dimension (one-based value) of the input signal, over which the minimum is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter applies only when you set the **Find the minimum value over** parameter to **Specified dimension**.

### Enable ROI processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter applies only when you set the **Find the minimum value over** parameter to **Entire input**, and the block is not in running mode.

### ROI type

Specify the type of ROI you want to use. Your choices are **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**.

When you set this parameter to **Rectangles** or **Lines**, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the **Flag** port appears on the block.

When you set this parameter to **Label matrix**, the **Label** and **Label Numbers** ports appear on the block and the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the **Flag** port appears on the block.

See **Output = Individual statistics for each ROI** for details.

**ROI portion to process**

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter applies only when you set the **ROI type** parameter to **Rectangles**.

**Output**

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter does not apply when you set the **ROI type** parameter, to **Binary mask**.

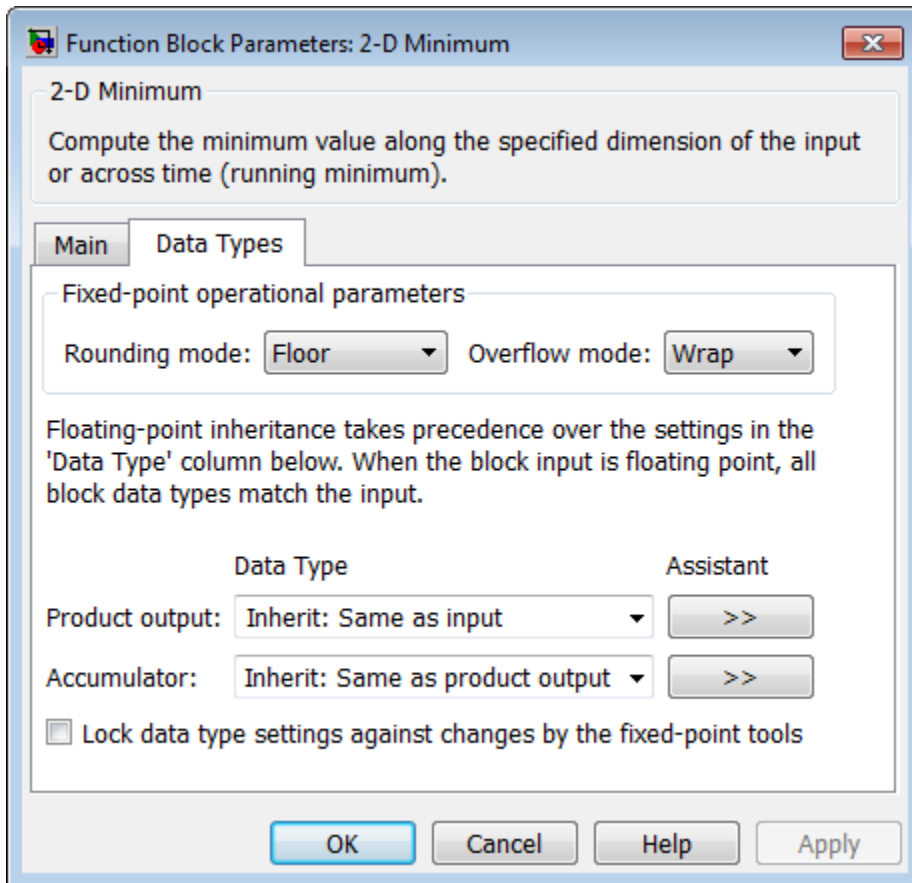
**Output flag indicating if ROI is within image bounds**

When you select this check box, the **Flag** port appears on the block. This check box applies only when you set the **ROI type** parameter to **Rectangles** or **Lines**. For a description of the **Flag** port output, see the tables in “ROI Processing” on page 1-135.

**Output flag indicating if label numbers are valid**

When you select this check box, the **Flag** port appears on the block. This check box applies only when you set the **ROI type** parameter to **Label matrix**. For a description of the **Flag** port output, see the tables in “ROI Processing” on page 1-135.

The **Fixed-point** pane of the 2-D Minimum dialog box appears as shown in the following figure.



---

**Note:** The parameters on the **Data Types** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in “Value Mode” on page 1-131. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

---

### Rounding mode

Select the “Rounding Modes” for fixed-point operations.




## Overflow mode

Select the Overflow mode for fixed-point operations.

## Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 1-136 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

## Accumulator data type

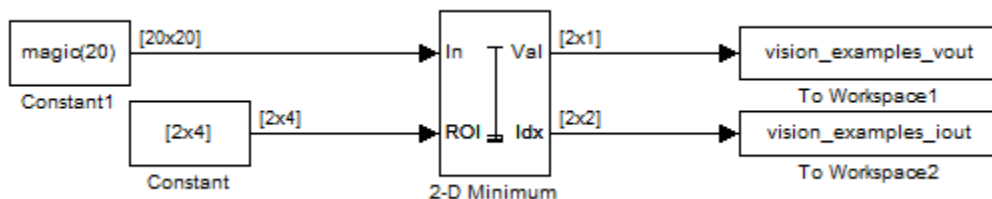
Specify the accumulator data type. See “Fixed-Point Data Types” on page 1-136 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” for more information.

## Examples



The `ex_vision_2dminimum` example finds the minimum value within two ROIs. The model outputs the minimum values and their one-based [x y] coordinate locations.

## See Also

2-D Maximum

2-D Mean

MinMax

2D-Histogram

min

Computer Vision System Toolbox

Computer Vision System Toolbox

Simulink

Computer Vision System Toolbox

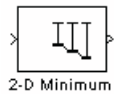
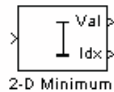
MATLAB

## 2-D Minimum (To Be Removed)

Find minimum values in an input or sequence of inputs

### Library

vipobslib



### Description

---

**Note:** This 2-D Minimum block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated 2-D Minimum block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

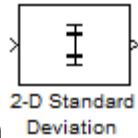
## 2-D Standard Deviation

Find standard deviation of each input matrix

### Library

Statistics

visionstatistics



### Description

The Standard Deviation block computes the standard deviation of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The Standard Deviation block can also track the standard deviation of a sequence of inputs over a period of time. The **Running standard deviation** parameter selects between basic operation and running operation.

### Port Description

Port	Supported Data Types
Input	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>
Reset	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
ROI	Rectangles and lines:

Port	Supported Data Types
	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> Binary Mask: <ul style="list-style-type: none"> <li>• Boolean</li> </ul>
Label	<ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Label Numbers	<ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Output	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>
Flag	<ul style="list-style-type: none"> <li>• Boolean</li> </ul>

## Basic Operation

When you do not select the **Running standard deviation** check box, the block computes the standard deviation of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each individual sample time, and outputs the array  $y$ . Each element in  $y$  contains the standard deviation of the corresponding column, row, vector, or entire input. The output  $y$  depends on the setting of the **Find the standard deviation value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Entire input** — The output at each sample time is a scalar that contains the standard deviation of the entire input.

```
y = std(u(:))      % Equivalent MATLAB code
```

- **Each Row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the standard deviation of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.

```
y = std(u,0,2)    % Equivalent MATLAB code
```

- **Each Column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the standard deviation of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

```
y = std(u,0,1)      % Equivalent MATLAB code
```

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified Dimension** — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the standard deviation of each vector over the third dimension of the input.

```
y = std(u,0,Dimension)  % Equivalent MATLAB code
```

For purely real or purely imaginary inputs, the standard deviation of the  $j$ th column of an  $M$ -by- $N$  input matrix is the square root of its variance:

$$y_j = \sigma_j = \sqrt{\frac{\sum_{i=1}^M |u_{ij} - \mu_j|^2}{M-1}} \quad 1 \leq j \leq N$$

For complex inputs, the output is the *total standard deviation*, which equals the square root of the *total variance*, or the square root of the sum of the variances of the real and imaginary parts. The standard deviation of each column in an  $M$ -by- $N$  input matrix is given by:

$$\sigma_j = \sqrt{\sigma_{j,\text{Re}}^2 + \sigma_{j,\text{Im}}^2}$$

---

**Note:** The total standard deviation does *not* equal the sum of the real and imaginary standard deviations.

---

## Running Operation

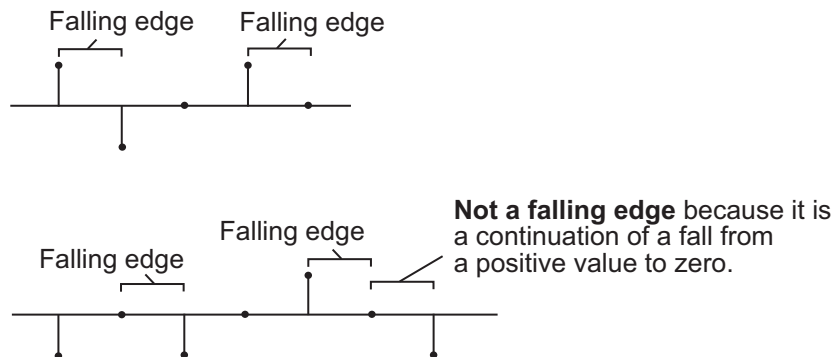
When you select the **Running standard deviation** check box, the block tracks the standard deviation of successive inputs to the block. In this mode, the block treats each element as a channel.

## Resetting the Running Standard Deviation

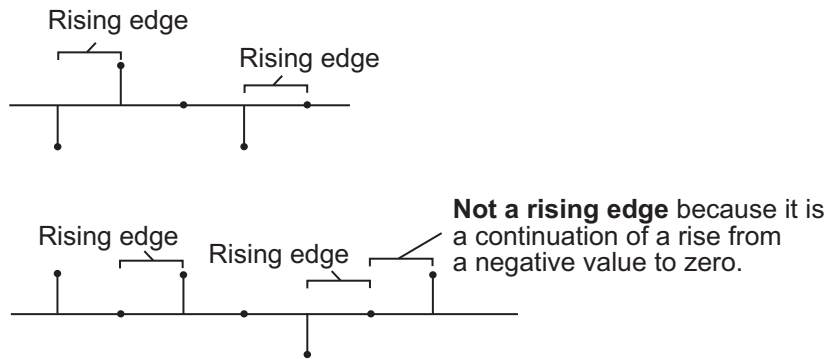
The block resets the running standard deviation whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** parameter:

- **None** disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a **Rising edge** or **Falling edge** (as described earlier)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero

---

**Note:** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset.

---

## ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This option is only available when the **Find the standard deviation value over** parameter is set to **Entire input** and the **Running standard deviation** check box is not selected. ROI processing is only supported for 2-D inputs.

Use the **ROI type** parameter to specify whether the ROI is a rectangle, line, label matrix, or binary mask. A binary mask is a binary image that enables you to specify which pixels to highlight, or select. In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to **Label matrix**, the **Label** and **Label Numbers** ports appear on the block. Use the **Label Numbers** port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the



label matrix. For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the **Draw Shapes** block reference page.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

If, for the **ROI type** parameter, you select **Rectangles** or **Lines**, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

#### Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select **Label matrix**, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

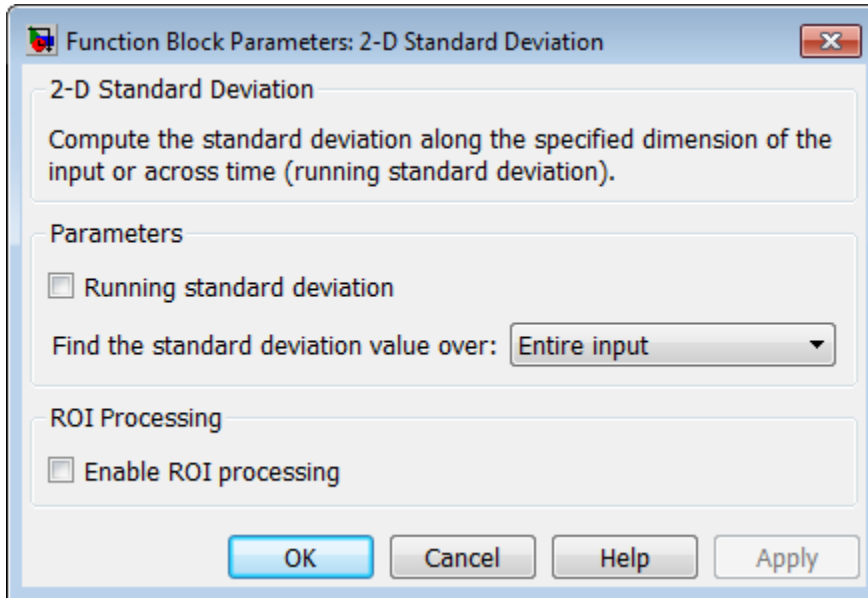
#### Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

**Output = Single statistic for all ROIs**

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

## Dialog Box



### Running standard deviation

Enables running operation when selected.

### Reset port

Specify the reset event that causes the block to reset the running standard deviation. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running standard deviation** check box. For more information, see “Resetting the Running Standard Deviation” on page 1-147.

### **Find the standard deviation value over**

Specify whether to find the standard deviation value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see “Basic Operation” on page 1-145.

### **Dimension**

Specify the dimension (one-based value) of the input signal, over which the standard deviation is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the standard deviation value over** parameter is set to **Specified dimension**.

### **Enable ROI Processing**

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the standard deviation value over** parameter is set to **Entire input**, and the block is not in running mode.

### **ROI type**

Specify the type of ROI you want to use. Your choices are **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**.

### **ROI portion to process**

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify **Rectangles**.

### **Output**

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

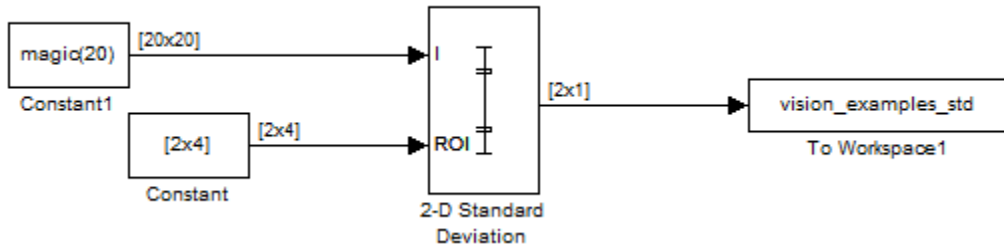
### **Output flag indicating if ROI is within image bounds**

When you select this check box, a Flag port appears on the block. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-148.

### **Output flag indicating if label numbers are valid**

When you select this check box, a Flag port appears on the block. This check box is visible only when you select **Label** matrix for the **ROI type** parameter. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-148.

## Example



The `ex_vision_2dstd` calculates the standard deviation value within two ROIs.

## See Also

2-D Mean

2-D Variance

std

Computer Vision System Toolbox

Computer Vision System Toolbox

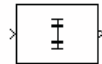
MATLAB

# 2-D Standard Deviation (To Be Removed)

Find standard deviation of each input matrix

## Library

Statistics



2-D Standard Deviation



2-D Standard Deviation

## Description

---

**Note:** This 2-D Standard Deviation block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated 2-D Standard Deviation block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

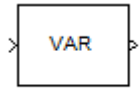
## 2-D Variance

Compute variance of input or sequence of inputs

### Library

Statistics

visionstatistics



### Description 2-D Variance

The 2-D Variance block computes the unbiased variance of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The 2-D Variance block can also track the variance of a sequence of inputs over a period of time. The **Running variance** parameter selects between basic operation and running operation.

### Port Description

Port	Supported Data Types
Input	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
Reset	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul>

Port	Supported Data Types
	<ul style="list-style-type: none"> <li>8-, 16-, and 32-bit unsigned integers</li> </ul>
ROI	Rectangles and lines: <ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> <li>Boolean</li> <li>8-, 16-, and 32-bit signed integers</li> <li>8-, 16-, and 32-bit unsigned integers</li> </ul> Binary Mask: <ul style="list-style-type: none"> <li>Boolean</li> </ul>
Label	<ul style="list-style-type: none"> <li>8-, 16-, and 32-bit unsigned integers</li> </ul>
Label Numbers	<ul style="list-style-type: none"> <li>8-, 16-, and 32-bit unsigned integers</li> </ul>
Output	<ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> <li>Fixed point</li> <li>8-, 16-, and 32-bit signed integers</li> <li>8-, 16-, and 32-bit unsigned integers</li> </ul>
Flag	<ul style="list-style-type: none"> <li>Boolean</li> </ul>

## Basic Operation

When you do not select the **Running variance** check box, the block computes the variance of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each individual sample time, and outputs the array  $y$ . Each element in  $y$  is the variance of the corresponding column, row, vector, or entire input. The output  $y$  depends on the setting of the **Find the variance value over** parameter. For example, consider a 3-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :

- **Entire input** — The output at each sample time is a scalar that contains the variance of the entire input.

```
y = var(u(:))      % Equivalent MATLAB code
```

- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the variance of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.

```
y = var(u,0,2)    % Equivalent MATLAB code
```

- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the variance of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

```
y = var(u,0,1)    % Equivalent MATLAB code
```

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as that when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the variance of each vector over the third dimension of the input.

```
y = var(u,0,Dimension)    % Equivalent MATLAB code
```

For purely real or purely imaginary inputs, the variance of an  $M$ -by- $N$  matrix is the square of the standard deviation:

$$y = \sigma^2 = \frac{\sum_{i=1}^M \sum_{j=1}^N |u_{ij}|^2 - \frac{\left| \sum_{i=1}^M \sum_{j=1}^N u_{ij} \right|^2}{M * N}}{M * N - 1}$$



For complex inputs, the variance is given by the following equation:

$$\sigma^2 = \sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2$$

## Running Operation

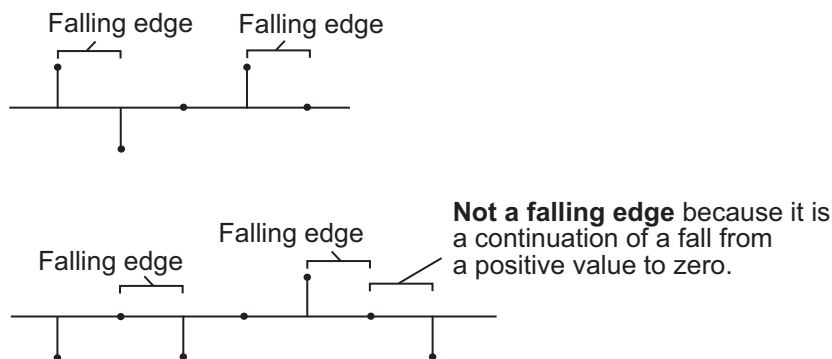
When you select the **Running variance** check box, the block tracks the variance of successive inputs to the block. In this mode, the block treats each element as a channel.

## Resetting the Running Variance

The block resets the running variance whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

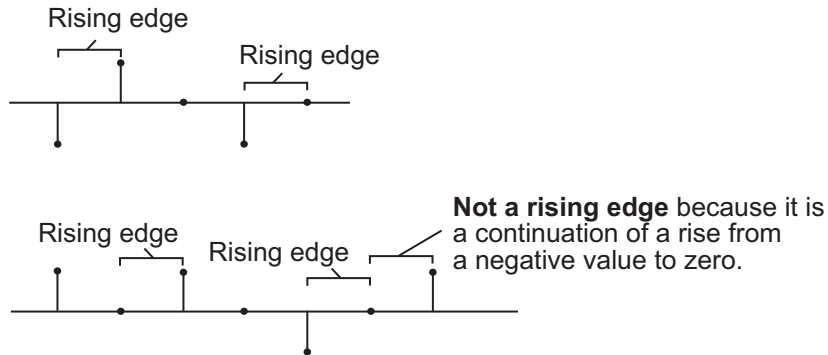
You specify the reset event in the **Reset port** parameter:

- **None** disables the Rst port.
- **Rising edge** — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Rst input does one of the following:

- Falls from a positive value to a negative value or zero
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a **Rising edge** or **Falling edge** (as described earlier)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero

---

**Note:** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset.

---

## ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This option is only available when the **Find the variance value over** parameter is set to **Entire input** and the **Running variance** check box is not selected. ROI processing is only supported for 2-D inputs.

Use the **ROI type** parameter to specify whether the ROI is a binary mask, label matrix, rectangle, or line. ROI processing is only supported for 2-D inputs.

- A binary mask is a binary image that enables you to specify which pixels to highlight, or select.

- In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to **Label matrix**, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix.
- For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the **Draw Shapes** reference page.

---

**Note:** For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

---

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

If, for the **ROI type** parameter, you select **Rectangles** or **Lines**, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

#### Output = Individual Statistics for Each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

#### Output = Single Statistic for All ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select `Label matrix`, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

**Output = Individual Statistics for Each ROI**

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

**Output = Single Statistic for All ROIs**

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

## Fixed-Point Data Types

The parameters on the **Data Types** pane of the block dialog are only used for fixed-point inputs. For purely real or purely imaginary inputs, the variance of the input is the square of its standard deviation. For complex inputs, the output is the sum of the variance of the real and imaginary parts of the input.

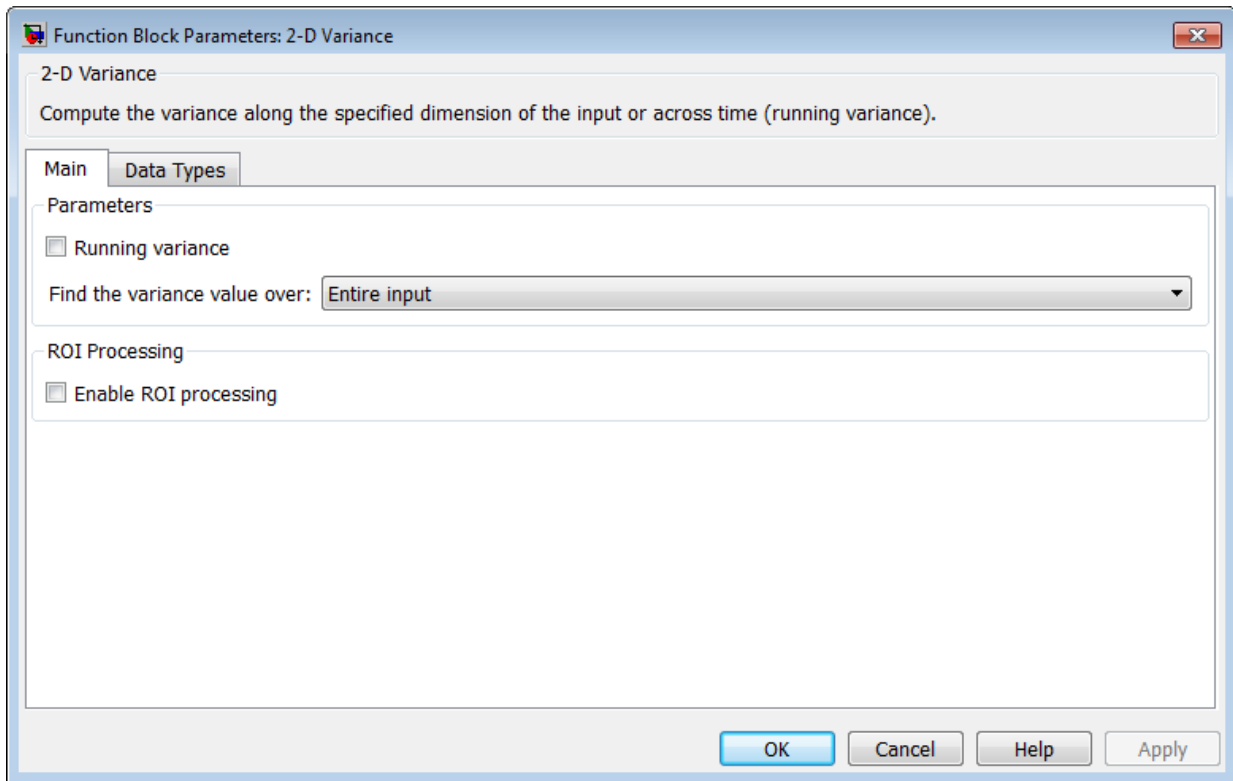
The following diagram shows the data types used within the Variance block for fixed-point signals.

$u_{ij}$

The results of the magnitude-squared calculations in the figure are in the product output data type. You can set the accumulator, product output, and output data types in the block dialog as discussed in “Dialog Box” on page 1-160.

## Dialog Box

The **Main** pane of the Variance block dialog appears as follows.



### Running variance

Enables running operation when selected.

### Reset port

Specify the reset event that causes the block to reset the running variance. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running variance** check box. For more information, see “Resetting the Running Variance” on page 1-157

### Find the variance value over

Specify whether to find the variance along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see “Basic Operation” on page 1-155.

### Dimension

Specify the dimension (one-based value) of the input signal, over which the variance is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the variance value over** parameter is set to **Specified dimension**.

### Enable ROI Processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the variance value over** parameter is set to **Entire input**, and the block is not in running mode.

---

**Note:** Full ROI processing is available only if you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type Rectangles**.

---

### ROI type

Specify the type of ROI you want to use. Your choices are **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**.

### ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify **Rectangles**.

### Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select **Binary mask**.

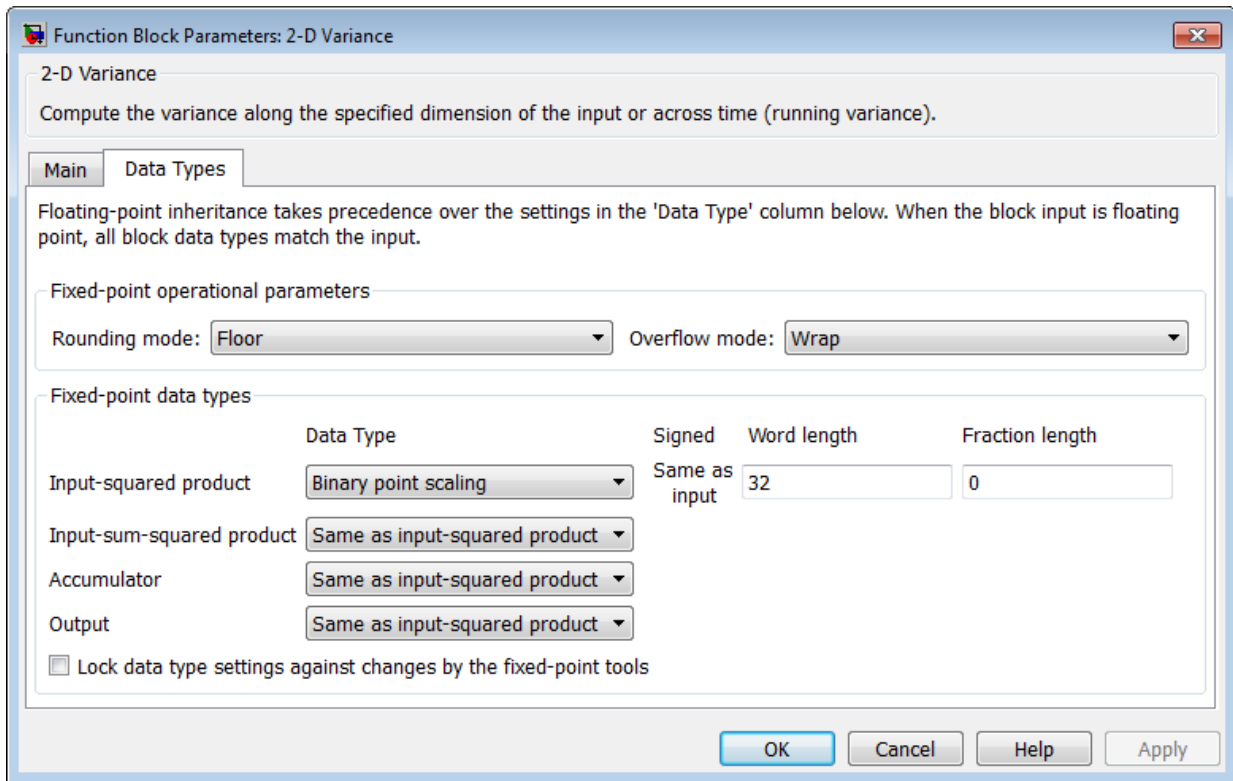
### Output flag indicating if ROI is within image bounds

When you select this check box, a Flag port appears on the block. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-158.

### Output flag indicating if label numbers are valid

When you select this check box, a Flag port appears on the block. This check box is visible only when you select **Label matrix** for the **ROI type** parameter. For a description of the Flag port output, see the tables in “ROI Processing” on page 1-158.

The **Data Types** pane of the Variance block dialog appears as follows.



### Rounding mode

Select the “Rounding Modes” for fixed-point operations.

### Overflow mode

Select the Overflow mode for fixed-point operations.

---

**Note:** See “Fixed-Point Data Types” on page 1-160 for more information on how the product output, accumulator, and output data types are used in this block.

---

### Input-squared product

Use this parameter to specify how to designate the input-squared product word and fraction lengths:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the input-squared product, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the input-squared product. This block requires power-of-two slope and a bias of zero.

### **Input-sum-squared product**

Use this parameter to specify how to designate the input-sum-squared product word and fraction lengths:

- When you select **Same as input-squared product**, these characteristics match those of the input-squared product.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the input-sum-squared product, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the input-sum-squared product. This block requires power-of-two slope and a bias of zero.

### **Accumulator**

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block:

- When you select **Same as input-squared product**, these characteristics match those of the input-squared product.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

### **Output**

Choose how you specify the output word length and fraction length:

- When you select **Same as accumulator**, these characteristics match those of the accumulator.

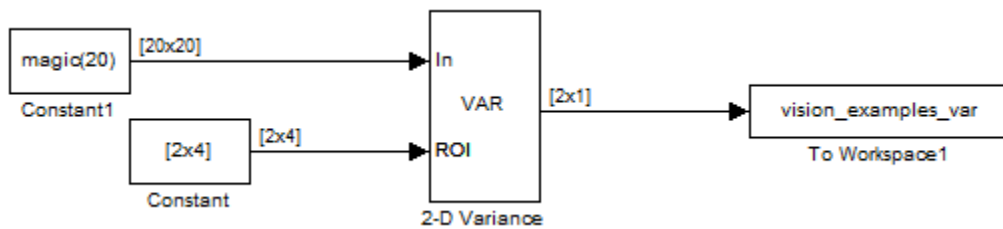


- When you select **Same as input-squared product**, these characteristics match those of the input-squared product.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

### Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

## Example



The `ex_vision_2dvar` calculates the variance value within two ROIs.

## See Also

2-D Mean

2-D Standard Deviation

`var`

Computer Vision System Toolbox

Computer Vision System Toolbox

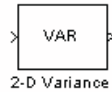
MATLAB

## 2-D Variance (To Be Removed)

Compute variance of each input matrix

### Library

Statistics



### Description

---

**Note:** This 2-D Variance block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated 2-D Variance block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

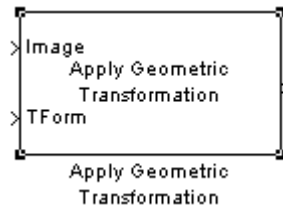
# Apply Geometric Transformation

Apply projective or affine transformation to an image

## Library

Geometric Transformations

visiongeotforms



## Description

---

**Note:** The Apply Geometric Transformation will be removed in a future release. Use the Warp block instead.

---

Use the Apply Geometric Transformation block to apply “Transformations” on page 1-337 or “Transformations” on page 1-337 transform to an image. You can use this block to transform the entire image or portions of the image with either polygon or rectangle Regions of Interest (ROIs).

## Port Descriptions

Port	Description
Image	<p><math>M</math>-by-<math>N</math> or <math>M</math>-by-<math>N</math>-by-<math>P</math> input matrix.</p> <p><math>M</math>: Number of rows in the image.</p> <p><math>N</math>: Number of columns in the image.</p> <p><math>P</math>: Number of color planes in the image.</p>

Port	Description
<b>TForm</b>	<p>When you set the <b>Transformation matrix source</b> parameter to Input port, the <b>TForm</b> input port accepts:</p> <ul style="list-style-type: none"> <li>• 3-by-2 matrix (affine transform) or a <math>Q</math>-by-6 matrix (multiple affine transforms).</li> <li>• 3-by-3 matrix (projective transform) or a <math>Q</math>-by-9 matrix (multiple projective transforms).</li> </ul> <p><math>Q</math>: Number of transformations.            When you specify multiple transforms in a single matrix, each transform is applied separately to the original image. If the individual transforms produce an overlapping area, the result of the transform in the last row of the matrix is overlaid on top.</p>
<b>ROI</b>	<p>When you set the ROI source parameter to Input port, the ROI input port accepts:</p> <ul style="list-style-type: none"> <li>• <math>4</math>-element vector rectangle ROI.</li> <li>• <math>2L</math>-element vector polygon ROI.</li> <li>• <math>R</math>-by-4 matrix for multiple rectangle ROIs.</li> <li>• <math>R</math>-by-<math>2L</math> matrix for multiple polygon ROIs.</li> </ul> <p><math>R</math>: Number of Region of Interests (ROIs).  <math>L(L \geq 3)</math>: Number of vertices in a polygon ROI.</p>

## Transformations

The size of the transformation matrix will dictate the transformation type.

### Affine Transformation

For affine transformation, the value of the pixel located at  $[\hat{x}, \hat{y}]$  in the output image, is determined by the value of the pixel located at  $[x, y]$  in the input image. The relationship between the input and the output point locations is defined by the following equations:

$$\begin{cases} \hat{x} = xh_1 + yh_2 + h_3 \\ \hat{y} = xh_4 + yh_5 + h_6 \end{cases}$$

where  $h_1, h_2, \dots, h_6$  are transformation coefficients.

If you use one transformation, the transformation coefficients must be arranged as a 3-by-2 matrix as in:

$$H = \begin{bmatrix} h_1 & h_4 \\ h_2 & h_5 \\ h_3 & h_6 \end{bmatrix}$$

or in a 1-by-6 vector as in  $H = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6]$  .

If you use more than one transformation, the transformation coefficients must be arranged as a  $Q$ -by-6 matrix, where each row has the format of

$H = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6]$  , and  $Q$  is the number of transformations as in:

$$H = \begin{bmatrix} h_{11} & h_{12} & \dots & h_{16} \\ h_{21} & h_{22} & \dots & h_{26} \\ \vdots & \vdots & \dots & \vdots \\ h_{Q1} & h_{Q2} & \dots & h_{Q6} \end{bmatrix}$$

When you specify multiple transforms in a single matrix, each transform is applied separately to the original image. If the individual transforms produce an overlapping area, the result of the transform in the last row of the matrix is overlaid on top.

### Projective Transformation

For projective transformation, the relationship between the input and the output points is defined by the following equations:

$$\begin{cases} \hat{x} = \frac{xh_1 + yh_2 + h_3}{xh_7 + yh_8 + h_9} \\ \hat{y} = \frac{xh_4 + yh_5 + h_6}{xh_7 + yh_8 + h_9} \end{cases}$$

where  $h_1, h_2, \dots, h_9$  are transformation coefficients.

If you use one transformation, the transformation coefficients must be arranged as a 3-by-3 matrix as in:

$$H = \begin{bmatrix} h_1 & h_4 & h_7 \\ h_2 & h_5 & h_8 \\ h_3 & h_6 & h_9 \end{bmatrix}$$

or in a 1-by-9 vector as in,  $H = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6 \ h_7 \ h_8 \ h_9]$  .

If you use more than one transformation, the transformation coefficients must be arranged as a  $Q$ -by-9 matrix, where each row has the format of

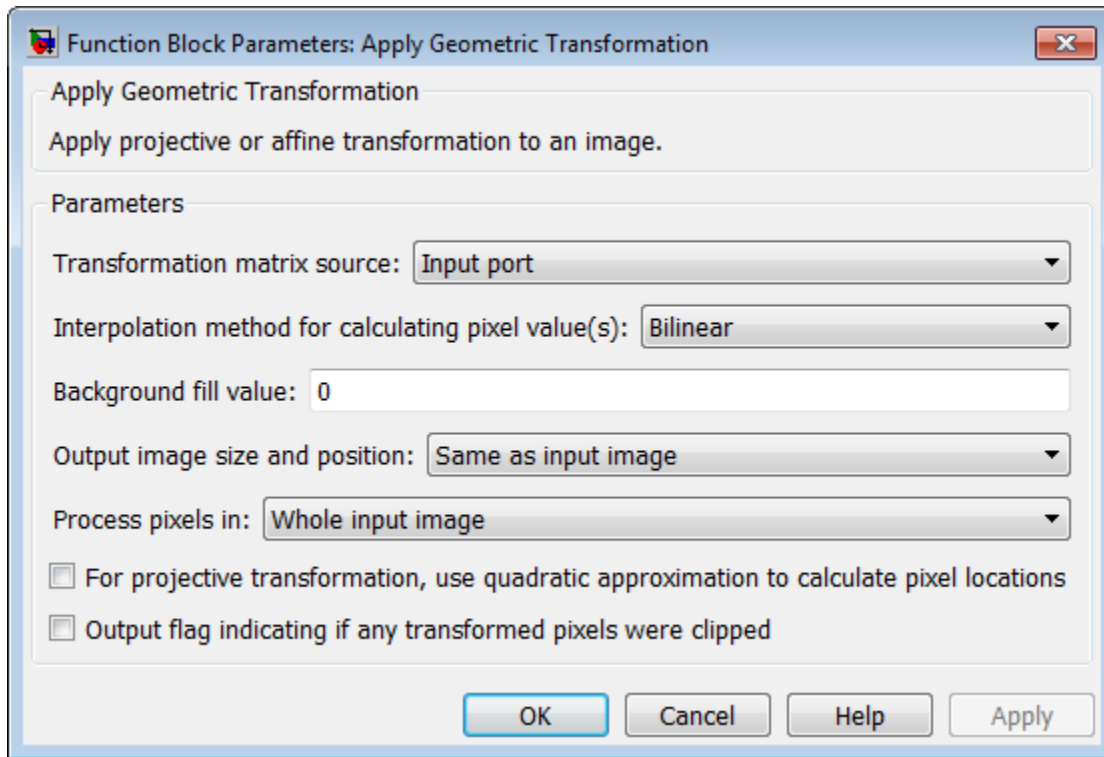
$H = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6 \ h_7 \ h_8 \ h_9]$  , and  $Q$  is the number of transformations.

For example,

$$H = \begin{bmatrix} h_{11} & h_{12} & \dots & h_{19} \\ h_{21} & h_{22} & \dots & h_{29} \\ \vdots & \vdots & \dots & \vdots \\ h_{Q1} & h_{Q2} & \dots & h_{Q9} \end{bmatrix}$$

## Dialog Box

The **Main** pane of the **Apply Geometric Transformation** dialog box appears as shown in the following figure.



### Transformation matrix source

Specify input matrix source, either **Specified via dialog**, or **Input port**. If you select **Specify via dialog**, you can enter the transformation matrix parameters in the parameter that appear with this selection.

### Transformation matrix

Specify a 3-by-2, 3-by-3,  $Q$ -by-6, or a  $Q$ -by-9 matrix. This option appears when you set the **Transformation matrix source** parameter to **Specified via dialog**.

### Interpolation method for calculating pixel value(s)

Specify interpolation method, either **Nearest neighbor**, **Bilinear**, or **Bicubic** interpolation to calculate output pixel values. See **Geometric Transformation Interpolation Methods** for an overview of these methods.

**Background fill value**

Specify the value of the pixels that are outside of the input image. Use either a scalar value of P-element vector.

**Output image size and position**

Specify the output image size to be either `Same as input image`, or `Specify via dialog`. If you select to `Specify via dialog`, you can specify the bounding box in the size and location parameters that appear with this selection.

**Size [height width]**

Specify the height and width for the output image size as `[height width]`. You can specify this parameter, along with the **Location of the upper left corner [x y]** parameter when you set the **Output image size and position** parameter to `Specify via dialog`.

**Location of the upper left corner [x y]**

Specify the `[x y]` location for the upper left corner of the output image. You can specify this parameter, along with the **Size [height width]** parameter, when you set the **Output image size and position** parameter to `Specify via dialog`.

**Process pixels in**

Specify the region in which to process pixels. Specify `Whole input image`, `Rectangle ROI`, or `Polygon ROI`. If you select `Rectangle ROI`, or `Polygon ROI` the **ROI source** parameter becomes available.

The transformations will be applied on the whole image, or on specified multiple ROIs. The table below outlines how transformation matrices are handled with an entire image and with single and multiple ROIs.

<b>Number of Transformation Matrices</b>	<b>Region of Interest</b>
One transformation matrix	You can apply the transformation on the entire image, single ROI or multiple ROIs.



Number of Transformation Matrices	Region of Interest
Multiple transformation matrices	<ul style="list-style-type: none"> <li>You can apply multiple transformation matrices on one ROI or on the entire image. The transformations are done in the order they are entered in the <b>TForm</b>.</li> <li>You can apply multiple transformation matrices on multiple ROIs. Each transformation matrix is applied to one ROI. The first transformation matrix specified is applied to the first ROI specified. The second transformation matrix is applied to the second ROI specified, and so on. The number of transformation matrices must be equal to the number of ROIs.</li> </ul>

### ROI source

Specify the source for the region of interest (ROI), either **Specify via dialog** or **Input port**. This appears when you set the **Process pixels in** parameter to either **Rectangle ROI**, or **Polygon ROI**.

### Location and size of rectangle ROI [x y width height]

Specify a *4-element* vector or an *R-by-4* matrix, (where *R* represents the number of ROIs). This parameter appears when you set the **Process pixels in** parameter to **Rectangle ROI**.

Specify the rectangle by its top-left corner and size in width and height. If you specify one ROI, it must be a *4-element* vector of format [x y width height]. If you specify more than one ROI, it must be an *R-by-4* matrix, such that each row's format is [x y width height].

### Vertices of polygon ROI [x1 y1 x2 y2 ... xL yL]

Specify a *2L-element* vector or an *R-by-2L* matrix, (where *R* represents the number of ROIs and *L* is the number of vertices in a polygon). This parameter appears when you set the **Process pixels in** parameter to **Polygon ROI**.

Specify the polygon by its vertices in clockwise or counter-clockwise order, with at least three or more vertices. If you specify one ROI of *L* vertices, it must be a *2L-element* vector of format [x<sub>1</sub> y<sub>1</sub> x<sub>2</sub> y<sub>2</sub> ... x<sub>L</sub> y<sub>L</sub>]. If you specify more than one ROI, it must be an *R-by-2L* matrix, where *L* is the maximum number of vertices in the ROIs. For ROI with vertices fewer than *L*, its last vertex can be repeated to form a vector.

**Output flag indicating if any part of ROI is outside input image**

Select the **Output flag indicating if any part of ROI is outside input image** check box to enable this output port on the Apply Geometric Transformation block.

**For projective transformation, use quadratic approximation to calculate pixel locations**

Specify whether to use an exact computation or an approximation for the projective transformation. If you select this option, you can enter an error tolerance in the **Error tolerance (in pixels)** parameter.

**Error tolerance (in pixels)**

Specify the maximum error tolerance in pixels. This appears when you select the **For projective transformation, use quadratic approximation to calculate pixel locations** check box.

**Output flag indicating if any transformed pixels were clipped**

Enable output port for flag, which indicates clipping. Clipping occurs when any of the transformed pixels fall outside of the output image.

## References

[1] George Wolberg, “Digital Image Warping”, IEEE Computer Society Press, 3<sup>rd</sup> edition, 1994.

Richard Hartley and Andrew Zisserman, “Multiple View Geometry in Computer Vision“, Cambridge University Press, 2<sup>nd</sup> edition, 2003.

## Supported Data Types

Port	Supported Data Types
Image	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul>
TForm	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul>

Port	Supported Data Types
<b>ROI</b>	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
<b>Output</b>	Same as input
<b>Err_roi</b>	Boolean
<b>Err_clip</b>	Boolean

## See Also

imtransform	Image Processing Toolbox
Estimate Geometric Transformation	Computer Vision System Toolbox
Trace Boundary	Computer Vision System Toolbox
Blob Analysis	Computer Vision System Toolbox
Video and Image Processing Demos	Computer Vision System Toolbox

**Introduced in R2008b**

# Apply Geometric Transformation (To Be Removed)

Apply projective or affine transformation to an image

## Library

Geometric Transformations

## Description

---

**Note:** This Apply Geometric Transformation block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Apply Geometric Transformation block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

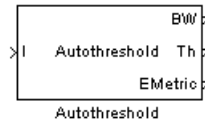
# Autothreshold

Convert intensity image to binary image

## Library

Conversions

visionconversions



## Description

The Autothreshold block converts an intensity image to a binary image using a threshold value computed using Otsu's method.

This block computes this threshold value by splitting the histogram of the input image such that the variance of each pixel group is minimized.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
BW	Scalar, vector, or matrix that represents a binary image	Boolean	No
Th	Threshold value	Same as I port	No
EMetric	Effectiveness metric	Same as I port	No

Use the **Thresholding operator** parameter to specify the condition the block places on the input values. If you select **>** and the input value is greater than the threshold value, the block outputs 1 at the BW port; otherwise, it outputs 0. If you select **<=** and the input value is less than or equal to the threshold value, the block outputs 1; otherwise, it outputs 0.

Select the **Output threshold** check box to output the calculated threshold values at the Th port.

Select the **Output effectiveness metric** check box to output values that represent the effectiveness of the thresholding at the EMetric port. This metric ranges from 0 to 1. If every pixel has the same value, the effectiveness metric is 0. If the image has two pixel values or the histogram of the image pixels is symmetric, the effectiveness metric is 1.

If you clear the **Specify data range** check box, the block assumes that floating-point input values range from 0 to 1. To specify a different data range, select this check box. The **Minimum value of input** and **Maximum value of input** parameters appear in the dialog box. Use these parameters to enter the minimum and maximum values of your input signal.

Use the **When data range is exceeded** parameter to specify the block's behavior when the input values are outside the expected range. The following options are available:

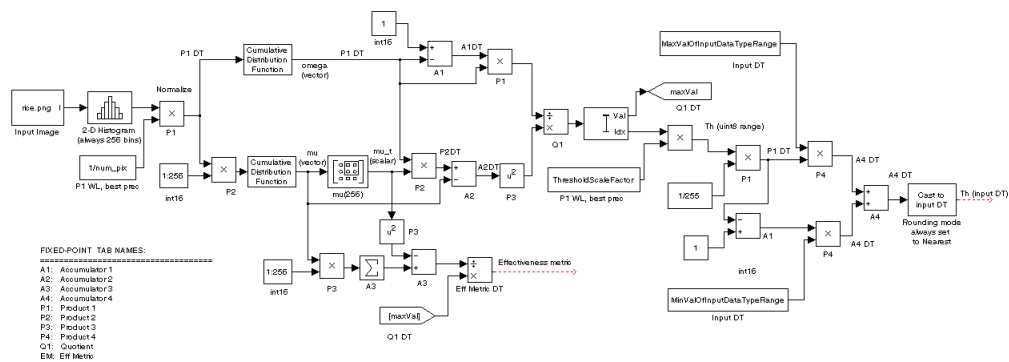
- **Ignore** — Proceed with the computation and do not issue a warning message. If you choose this option, the block performs the most efficient computation. However, if the input values exceed the expected range, the block produces incorrect results.
- **Saturate** — Change any input values outside the range to the minimum or maximum value of the range and proceed with the computation.
- **Warn and saturate** — Display a warning message in the MATLAB Command Window, saturate values, and proceed with the computation.
- **Error** — Display an error dialog box and terminate the simulation.

If you clear the **Scale threshold** check box, the block uses the threshold value computed by Otsu's method to convert intensity images into binary images. If you select the **Scale threshold** check box, the **Threshold scaling factor** appears in the dialog box. Enter a scalar value. The block multiplies this scalar value with the threshold value computed by Otsu's method and uses the result as the new threshold value.

## Fixed-Point Data Types

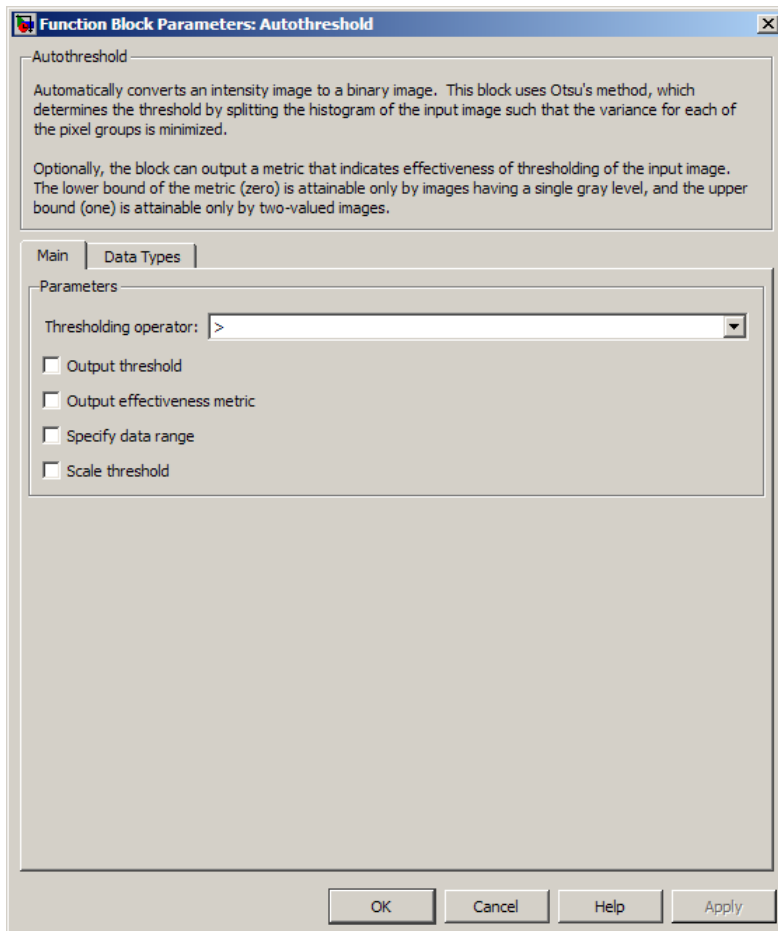
The following diagram shows the data types used in the Autothreshold block for fixed-point signals. You can use the default fixed-point parameters if your input has a word length less than or equal to 16.

In this diagram, DT means data type. You can set the product, accumulator, quotient, and effectiveness metric data types in the block mask.



## Dialog Box

The **Main** pane of the Autothreshold dialog box appears as shown in the following figure.



## Thresholding operator

Specify the condition the block places on the input matrix values. If you select > or <=, the block outputs 0 or 1 depending on whether the input matrix values are above, below, or equal to the threshold value.

## Output threshold

Select this check box to output the calculated threshold values at the Th port.

## Output effectiveness metric

Select this check box to output values that represent the effectiveness of the thresholding at the EMetric port.



**Specify data range**

If you clear this check box, the block assumes that floating-point input values range from 0 to 1. To specify a different data range, select this check box.

**Minimum value of input**

Enter the minimum value of your input data. This parameter is visible if you select the **Specify data range** check box.

**Maximum value of input**

Enter the maximum value of your input data. This parameter is visible if you select the **Specify data range** check box.

**When data range is exceeded**

Specify the block's behavior when the input values are outside the expected range. Your options are Ignore, Saturate, Warn and saturate, or Error. This parameter is visible if you select the **Specify data range** check box.

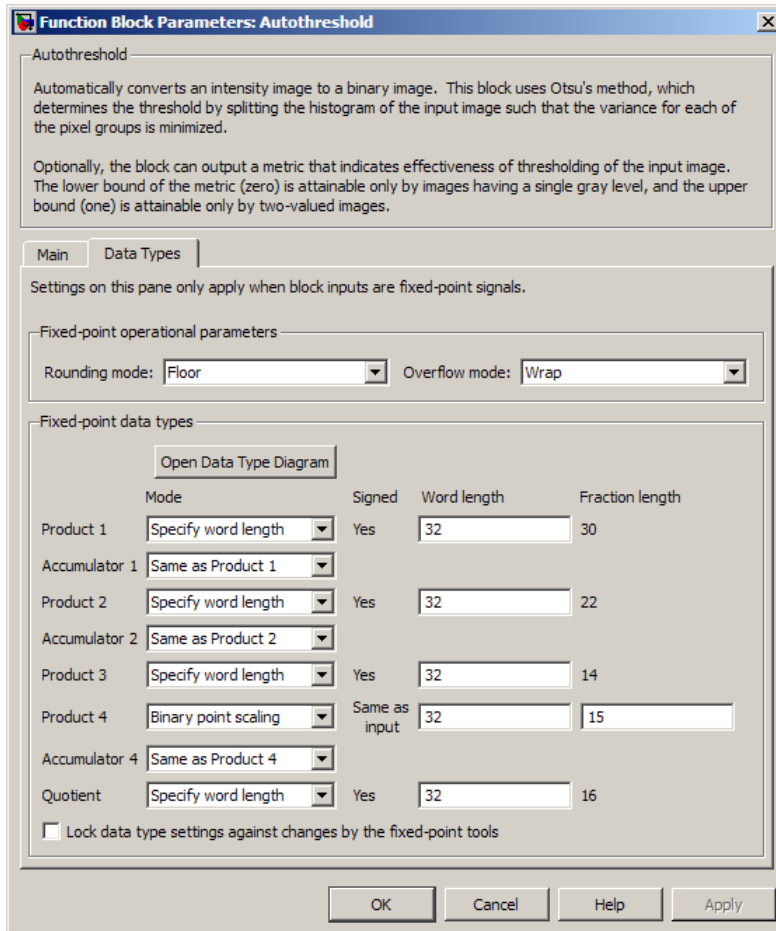
**Scale threshold**

Select this check box to scale the threshold value computed by Otsu's method.

**Threshold scaling factor**

Enter a scalar value. The block multiplies this scalar value with the threshold value computed by Otsu's method and uses the result as the new threshold value. This parameter is visible if you select the **Scale threshold** check box.

The **Data Types pane** pane of the Autothreshold dialog box appears as follows. You can use the default fixed-point parameters if your input has a word length less than or equal to 16.



## Rounding mode

Select the rounding mode for fixed-point operations. This parameter does not apply to the Cast to input DT step shown in “Fixed-Point Data Types” on page 1-179. For this step, **Rounding mode** is always set to Nearest.

## Overflow mode

Select the overflow mode for fixed-point operations.

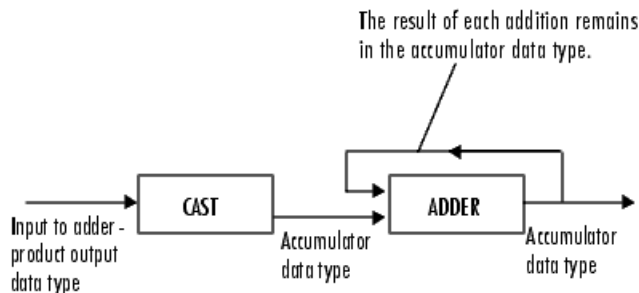
## Product 1, 2, 3, 4



As shown previously, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate the product output word and fraction lengths.

- When you select **Specify word length**, you can enter the word length of the product values in bits. The block sets the fraction length to give you the best precision.
- When you select **Same as input**, the characteristics match those of the input to the block. This choice is only available for the **Product 4** parameter.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Accumulator 1, 2, 3, 4



As shown previously, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element

of the input is added to it. Use this parameter to specify how to designate the accumulator word and fraction lengths.

- When you select **Same as Product**, these characteristics match those of the product output.
- When you select **Specify word length**, you can enter the word length of the accumulator values in bits. The block sets the fraction length to give you the best precision. This choice is not available for the **Accumulator 4** parameter because it is dependent on the input data type.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

The **Accumulator 3** parameter is only visible if, on the **Main** pane, you select the **Output effectiveness metric** check box.

### Quotient

Choose how to specify the word length and fraction length of the quotient data type:

- When you select **Specify word length**, you can enter the word length of the quotient values in bits. The block sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the quotient, in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the quotient. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Eff Metric

Choose how to specify the word length and fraction length of the effectiveness metric data type. This parameter is only visible if, on the **Main** tab, you select the **Output effectiveness metric** check box.

- When you select **Specify word length**, you can enter the word length of the effectiveness metric values, in bits. The block sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the effectiveness metric in bits.

- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the effectiveness metric. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## Example

### Thresholding Intensity Images Using the Autothreshold Block

Convert an intensity image into a binary image. Use the Autothreshold block when lighting conditions vary and the threshold needs to change for each video frame.

You can open the example model by typing

```
ex_vision_autothreshold
```

on the MATLAB command line.

## See Also

Compare To Constant	Simulink
Relational Operator	Simulink
graythresh	Image Processing Toolbox

**Introduced before R2006a**

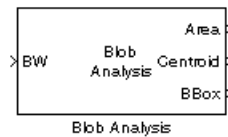
# Blob Analysis

Compute statistics for labeled regions

## Library

Statistics

visionstatistics



## Description

Use the Blob Analysis block to calculate statistics for labeled regions in a binary image. The block returns quantities such as the centroid, bounding box, label matrix, and blob count. The Blob Analysis block supports input and output variable size signals. You can also use the **Selector** block from Simulink, to select certain blobs based on their statistics.

For information on pixel and spatial coordinate system definitions, see “Expressing Image Locations” and “Coordinate Systems”.

## Port Descriptions

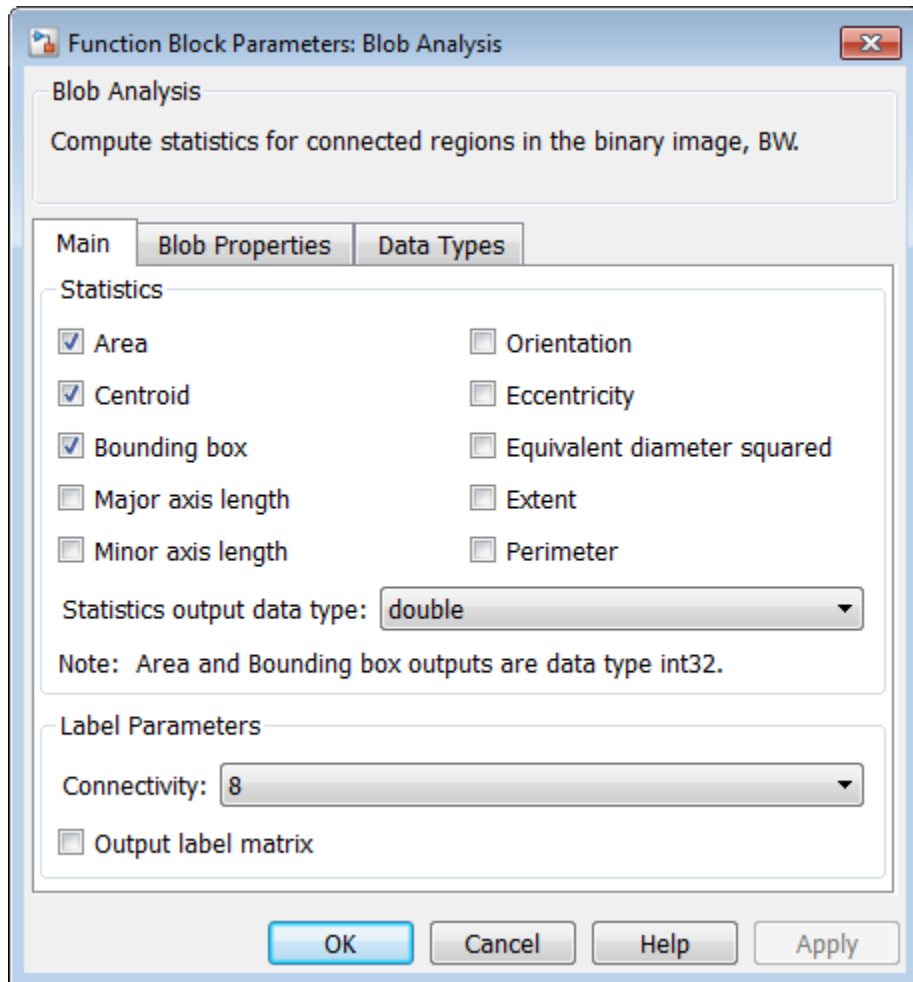
Port	Input/Output	Supported Data Types
BW	Vector or matrix that represents a binary image	Boolean
Area	Vector that represents the number of pixels in labeled regions	32-bit signed integer
Centroid	$M$ -by-2 matrix of centroid coordinates, where $M$ represents the number of blobs	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> </ul>

Port	Input/Output	Supported Data Types
BBox	$M$ -by-4 matrix of [x y width height] bounding box coordinates, where $M$ represents the number of blobs and [x y] represents the upper left corner of the bounding box.	32-bit signed integer
MajorAxis	Vector that represents the lengths of major axes of ellipses	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>
MinorAxis	Vector that represents the lengths of minor axes of ellipses	Same as MajorAxis port
Orientation	Vector that represents the angles between the major axes of the ellipses and the $x$ -axis.	Same as MajorAxis port
Eccentricity	Vector that represents the eccentricities of the ellipses	Same as MajorAxis port
Diameter ^2	Vector that represents the equivalent diameters squared	Same as Centroid port
Extent	Vector that represents the results of dividing the areas of the blobs by the area of their bounding boxes	Same as Centroid port
Perimeter	Vector containing an estimate of the perimeter length, in pixels, for each blob	Same as Centroid port
Label	Label matrix	8-, 16-, or 32-bit unsigned integer
Count	Scalar value that represents the actual number of labeled regions in each image	Same as Label port

## Dialog Box

The **Main** pane of the Blob Analysis dialog box appears as shown in the following figure. Use the check boxes to specify the statistics values you want the block to output. For a

full description of each of these statistics, see the `regionprops` function reference page in the Image Processing Toolbox documentation.



vision

### Area

Select this check box to output a vector that represents the number of pixels in labeled regions



**Centroid**

Select this check box to output an  $M$ -by-2 matrix of [x y] centroid coordinates. The rows represent the coordinates of the centroid of each region, where  $M$  represents the number of blobs.

*Example:* Suppose there are two blobs, where the row and column coordinates of their centroids are  $x_1, y_1$  and  $x_2, y_2$ , respectively. The block outputs:

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix}$$

at the Centroid port.

**Bounding box**

Select this check box to output an  $M$ -by-4 matrix of [x y width height] bounding boxes. The rows represent the coordinates of each bounding box, where  $M$  represents the number of blobs.

*Example:* Suppose there are two blobs, where  $x$  and  $y$  define the location of the upper-left corner of the bounding box, and  $w, h$  define the width and height of the bounding box. The block outputs

$$\begin{bmatrix} x_1 & y_1 & w_1 & h_1 \\ x_2 & y_2 & w_2 & h_2 \end{bmatrix}$$

at the BBox port.

**Major axis length**

Select this check box to output a vector with the following characteristics:

- Represents the lengths of the major axes of ellipses
- Has the same normalized second central moments as the labeled regions

**Minor axis length**

Select this check box to output a vector with the following characteristics:

- Represents the lengths of the minor axes of ellipses
- Has the same normalized second central moments as the labeled regions

**Orientation**

Select this check box to output a vector that represents the angles between the major axes of the ellipses and the  $x$ -axis. The angle values are in radians and range between:

$$-\frac{\pi}{2} \text{ and } \frac{\pi}{2}$$

### **Eccentricity**

Select this check box to output a vector that represents the eccentricities of ellipses that have the same second moments as the region

### **Equivalent diameter squared**

Select this check box to output a vector that represents the equivalent diameters squared

### **Extent**

Select this check box to output a vector that represents the results of dividing the areas of the blobs by the area of their bounding boxes

### **Perimeter**

Select this check box to output an  $N$ -by-1 vector of the perimeter lengths, in pixels, of each blob, where  $N$  is the number of blobs.

### **Statistics output data type**

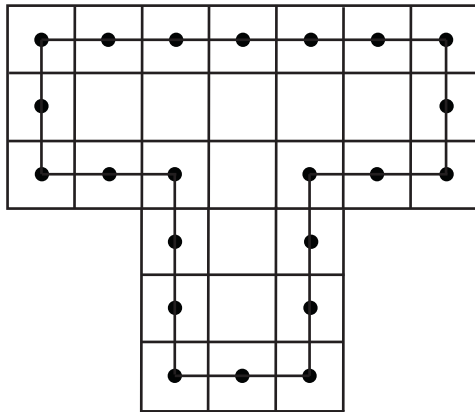
Specify the data type of the outputs as `double`, `single`, or to `Specify via Data Types` tab. The fields on the **Data Types** tab appear when you set the output data type to `Specify via Data Types` tab.

### **Connectivity**

Define which pixels connect to each other. If you want to connect pixels located on the top, bottom, left, and right, select **4**. If you want to connect pixels to the other pixels on the top, bottom, left, right, and diagonally, select **8**. For more information about this parameter, see the `Label` block reference page.

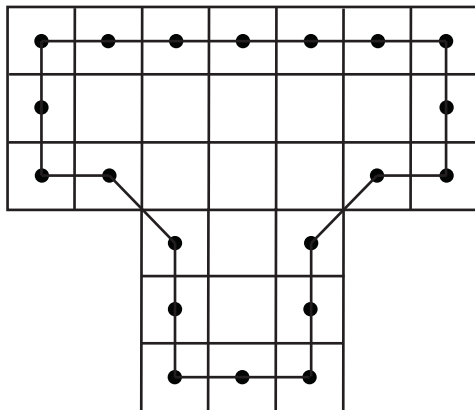
The **Connectivity** parameter also affects how the block calculates the perimeter of a blob. For example:

The following figure illustrates how the block calculates the perimeter when you set the **Connectivity** parameter to **4**.



The block calculates the distance between the center of each pixel (marked by the black dots) and estimates the perimeter to be 22.

The next figure illustrates how the block calculates the perimeter of a blob when you set the **Connectivity** parameter to 8.

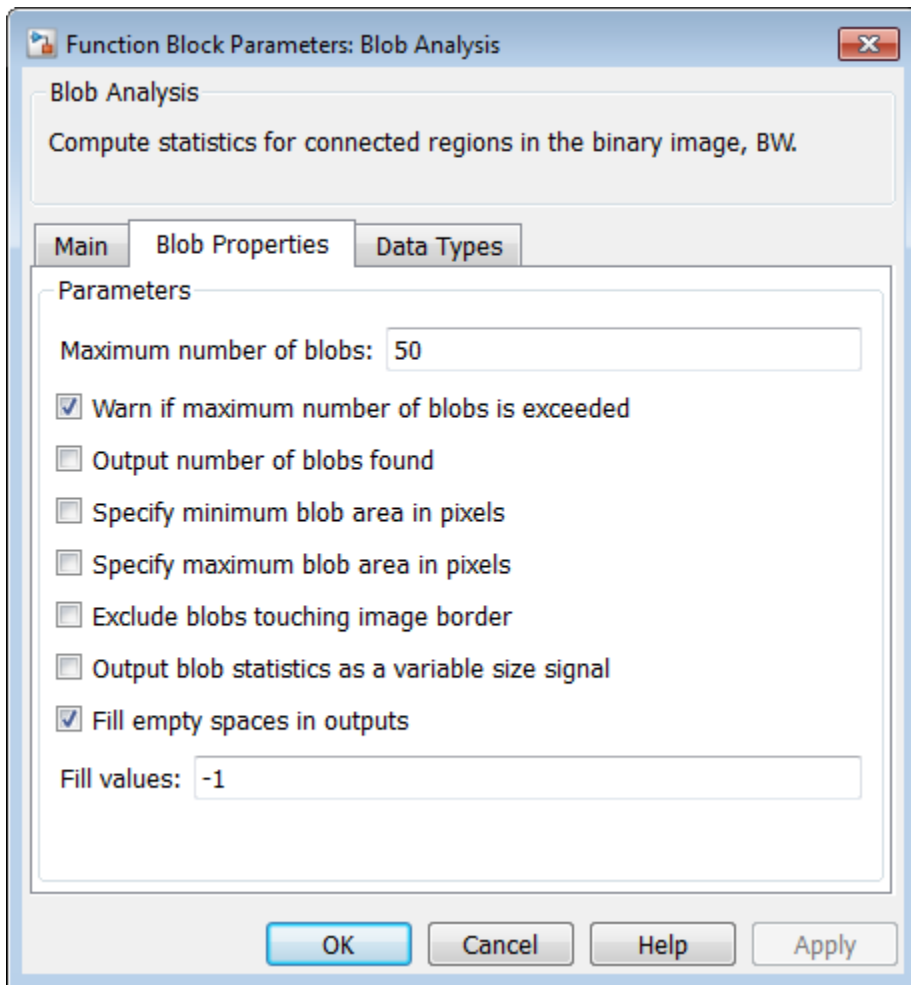


The block takes a different path around the blob and estimates the perimeter to be  $18 + 2\sqrt{2}$ .

**Output label matrix**

Select this check box, to output the label matrix at the **Label** port. The pixels equal to 0 represent the background. The pixels equal to 1 represent the first object. The pixels equal to 2 represent the second object, and so on.

The **Blob Properties** pane of the Blob Analysis dialog box appears as shown in the following figure.



### Maximum number of blobs

Specify the maximum number of labeled regions in each input image. The block uses this value to preallocate vectors and matrices to ensure that they are long enough to hold the statistical values. The maximum number of blobs the block outputs depends on both the value of this parameter, and on the size of the input image. The number of blobs the block outputs may be limited by the input image size.

**Warn if maximum number of blobs is exceeded**

Select this check box to output a warning when the number of blobs in an image is greater than the value of **Maximum number of blobs** parameter.

**Output number of blobs found**

Select this check box to output a scalar value that represents the actual number of connected regions in each image at the **Count** port.

**Specify maximum blob area in pixels**

Select this check box to enter the minimum blob area in the edit box that appears beside the check box. The blob gets a label if the number of pixels meets the minimum size specified. The maximum allowable value is the maximum of `uint32` data type. This parameter is tunable.

**Exclude blobs touching image border**

Select this check box to exclude a labeled blob that contains at least one border pixel.

**Output blob statistics as a variable-size signal**

Select this check box to output blob statistics as a variable-size signal. Selecting this check box means that you do not need to specify fill values.

**Fill empty spaces in outputs**

Select this check box to fill empty spaces in the statistical vectors with the values you specify in the **Fill values** parameter.

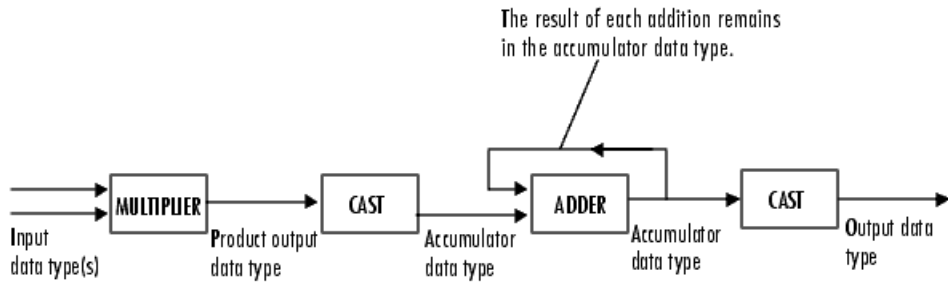
The **Fill empty spaces in outputs** check box does not appear when you select the **Output blob statistics as a variable-size signal** check box.

**Fill values**

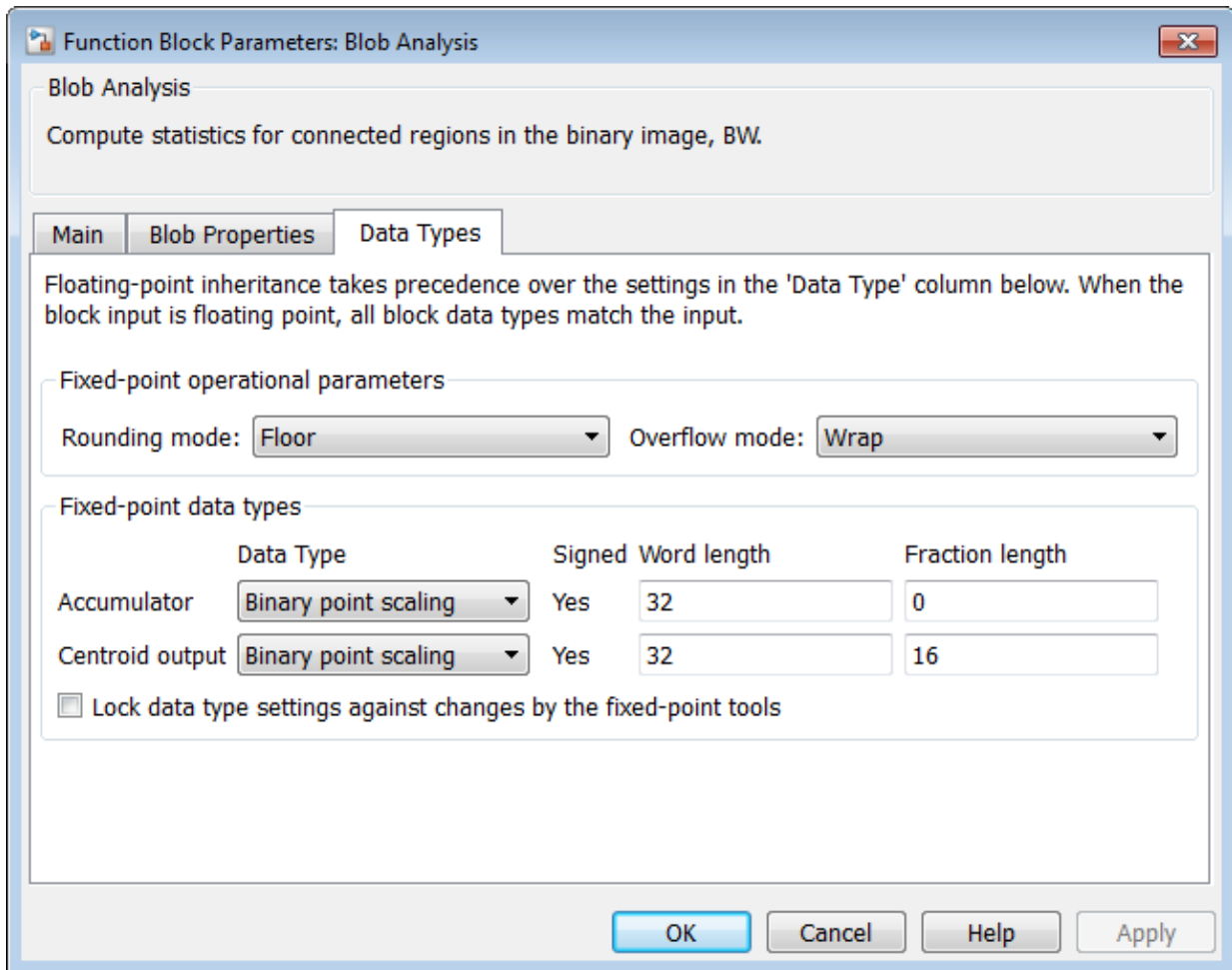
If you enter a scalar value, the block fills all the empty spaces in the statistical vectors with this value. If you enter a vector, it must have the same length as the number of selected statistics. The block uses each vector element to fill a different statistics vector. If the empty spaces do not affect your computation, you can deselect the **Fill empty spaces in outputs** check box. As a best practice, leave this check box selected.

The **Fill values** parameter is not visible when you select the **Output blob statistics as a variable-size signal** check box.

The **Data Types** pane of the Blob Analysis dialog box appears as shown in the following figure.



The parameters on the **Data Types** tab apply only when you set the **Statistics output data type** parameter to **Specify via Data Types** tab.



### Rounding mode

Select the rounding mode Floor, Ceiling, Nearest or Zero for fixed-point operations.

### Overflow mode

Select the overflow mode, Wrap or Saturate for fixed-point operations.

### Product output

When you select **Binary point scaling**, you can enter the **Word length** and the **Fraction length** of the product output, in bits.

When you select **Slope and bias scaling**, you can enter the **Word length** in bits, and the **Slope** of the product output. All signals in the Computer Vision System Toolbox software have a bias of 0.



The block places the output of the multiplier into the **Product output** data type and scaling. The computation of the equivalent diameter squared uses the product output data type. During this computation, the block multiplies the blob area (stored in the accumulator) by the  $4/\pi$  factor. This factor has a word length that equals the value of **Equivalent diameter squared** output data type **Word length**. The value of the **Fraction length** equals its word length minus two. Use this parameter to specify how to designate this product output word and fraction lengths.

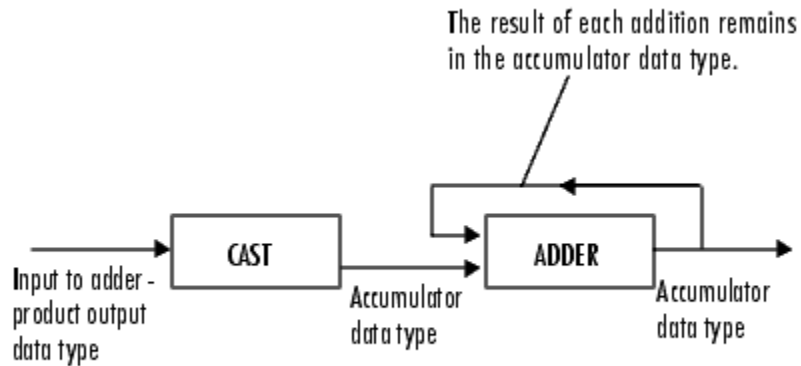
### Accumulator

When you select **Same as product output** the characteristics match the characteristics of the product output.

When you select **Binary point scaling**, you can enter the **Word length** and the **Fraction length** of the accumulator, in bits.

When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the **Accumulator**. All signals in the Computer Vision System Toolbox software have a bias of 0.





Inputs to the **Accumulator** get cast to the accumulator data type. Each element of the input gets added to the output of the adder, which remains in the accumulator data type. Use this parameter to specify how to designate this accumulator word and fraction lengths.

### Centroid output

Choose how to specify the **Word length** and **Fraction length** of the output at the **Centroid** port:

- When you select **Same as accumulator**, these characteristics match the characteristics of the accumulator.
- When you select **Binary point scaling**, you can enter the **Word length** and **Fraction length** of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the output. All signals in the Computer Vision System Toolbox software have a bias of 0.

### Equiv Diam<sup>2</sup> output

Choose how to specify the **Word length** and **Fraction length** of the output at the **Diameter <sup>2</sup>** port:

- When you select **Same as accumulator**, these characteristics match the characteristics of the **Accumulator**.
- When you select **Same as product output**, these characteristics match the characteristics of the **Product output**.

- When you select **Binary point scaling**, you can enter the **Word length** and **Fraction length** of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the output. All signals in the Computer Vision System Toolbox software have a bias of 0.

### **Extent output**

Choose how to specify the **Word length** and **Fraction length** of the output at the **Extent** port:

- When you select **Same as accumulator**, these characteristics match the characteristics of the accumulator.
- When you select **Binary point scaling**, you can enter the **Word length** and **Fraction length** of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the output. All signals in the Computer Vision System Toolbox software have a bias of 0.

### **Perimeter output**

Choose how to specify the **Word length** and **Fraction length** of the output at the **Perimeter** port:

- When you select **Same as accumulator**, these characteristics match the characteristics of the accumulator.
- When you select **Binary point scaling**, you can enter the **Word length** and **Fraction length** of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the **Slope** of the output. All signals in the Computer Vision System Toolbox software have a bias of 0.

### **Lock data type settings against changes by the fixed-point tools**

Select this parameter to prevent the autoscaling tool in the Fixed-Point Tool from overriding any fixed-point scaling you specify in this block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

Label	Computer Vision System Toolbox
regionprops	Image Processing Toolbox

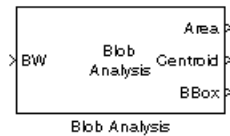
**Introduced before R2006a**

# Blob Analysis (To Be Removed)

Compute statistics for labeled regions

## Library

Statistics



## Description

---

**Note:** This Blob Analysis block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated **Blob Analysis** block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

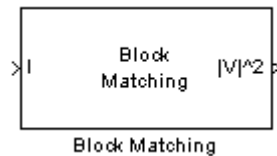
# Block Matching

Estimate motion between images or video frames

## Library

Analysis & Enhancement

visionanalysis



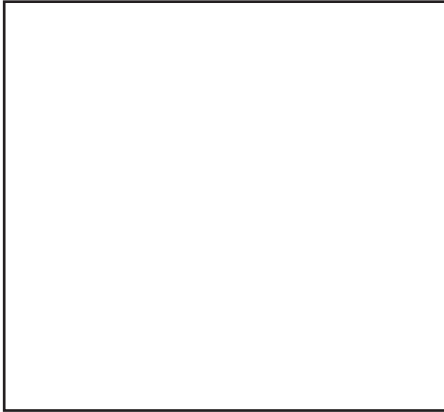
## Description

The Block Matching block estimates motion between two images or two video frames using “blocks” of pixels. The Block Matching block matches the block of pixels in frame  $k$  to a block of pixels in frame  $k+1$  by moving the block of pixels over a search region.

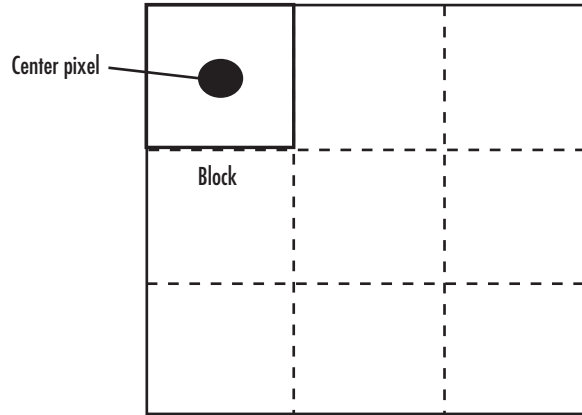
Suppose the input to the block is frame  $k$ . The Block Matching block performs the following steps:

- 1 The block subdivides this frame using the values you enter for the **Block size [height width]** and **Overlap [r c]** parameters. In the following example, the **Overlap [r c]** parameter is [0 0].
- 2 For each subdivision or block in frame  $k+1$ , the Block Matching block establishes a search region based on the value you enter for the **Maximum displacement [r c]** parameter.
- 3 The block searches for the new block location using either the **Exhaustive** or **Three-step** search method.

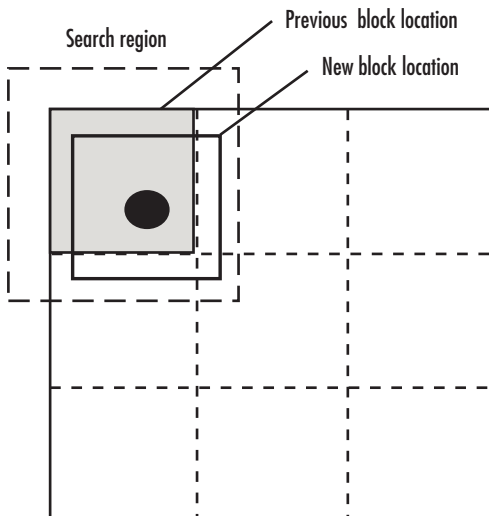
Input image = frame k



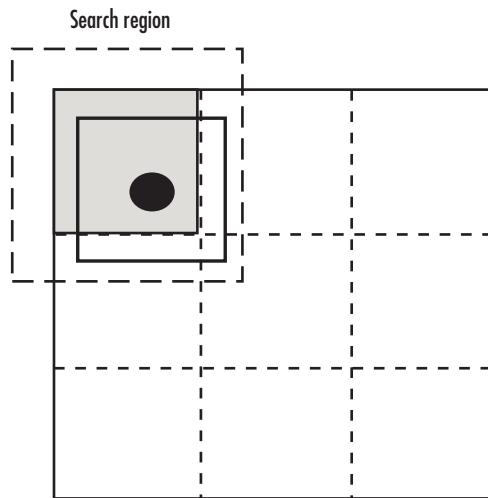
STEP 1: Subdivide the image in frame k.



STEP 2: Establish the search region in frame k+1.



STEP 3: Search for the new block location in frame k+1.



Port	Output	Supported Data Types	Complex Values Supported
I/I1	Scalar, vector, or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
I2	Scalar, vector, or matrix of intensity values	Same as I port	No
V ^2	Matrix of velocity magnitudes	Same as I port	No
V	Matrix of velocity components in complex form	Same as I port	Yes

Use the **Estimate motion between** parameter to specify whether to estimate the motion between two images or two video frames. If you select **Current frame and N-th frame back**, the **N** parameter appears in the dialog box. Enter a scalar value that represents the number of frames between the reference frame and the current frame.

Use the **Search method** parameter to specify how the block locates the block of pixels in frame  $k+1$  that best matches the block of pixels in frame  $k$ .

- If you select **Exhaustive**, the block selects the location of the block of pixels in frame  $k+1$  by moving the block over the search region 1 pixel at a time. This process is computationally expensive.
- If you select **Three - step**, the block searches for the block of pixels in frame  $k+1$  that best matches the block of pixels in frame  $k$  using a steadily decreasing step size. The block begins with a step size approximately equal to half the maximum search range. In each step, the block compares the central point of the search region to eight search points located on the boundaries of the region and moves the central point to the search point whose values is the closest to that of the central point. The block then reduces the step size by half, and begins the process again. This option is less computationally expensive, though it might not find the optimal solution.

Use the **Block matching criteria** parameter to specify how the block measures the similarity of the block of pixels in frame  $k$  to the block of pixels in frame  $k+1$ . If you select

Mean square error (MSE), the Block Matching block estimates the displacement of the center pixel of the block as the  $(d_1, d_2)$  values that minimize the following MSE equation:

$$MSE(d_1, d_2) = \frac{1}{N_1 \times N_2} \sum_{(n_1, n_2) \in B} [s(n_1, n_2, k) - s(n_1 + d_1, n_2 + d_2, k + 1)]^2$$

In the previous equation,  $B$  is an  $N_1 \times N_2$  block of pixels, and  $s(x, y, k)$  denotes a pixel location at  $(x, y)$  in frame  $k$ . If you select **Mean absolute difference (MAD)**, the Block Matching block estimates the displacement of the center pixel of the block as the  $(d_1, d_2)$  values that minimize the following MAD equation:

$$MAD(d_1, d_2) = \frac{1}{N_1 \times N_2} \sum_{(n_1, n_2) \in B} |s(n_1, n_2, k) - s(n_1 + d_1, n_2 + d_2, k + 1)|$$

Use the **Block size [height width]** and **Overlap [r c]** parameters to specify how the block subdivides the input image. For a graphical description of these parameters, see the first figure in this reference page. If the **Overlap [r c]** parameter is not [0 0], the blocks would overlap each other by the number of pixels you specify.

Use the **Maximum displacement [r c]** parameter to specify the maximum number of pixels any center pixel in a block of pixels might move from image to image or frame to frame. The block uses this value to determine the size of the search region.

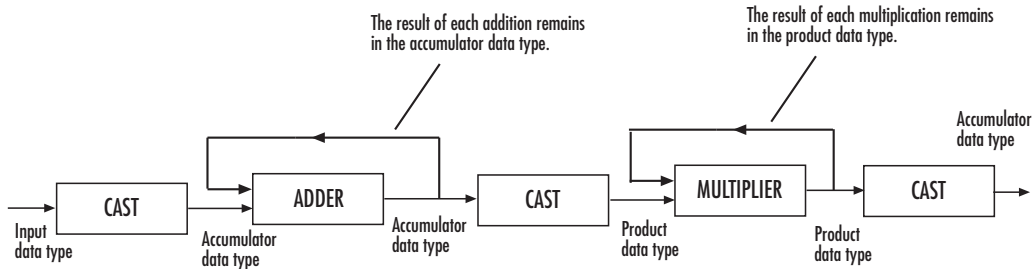
Use the **Velocity output** parameter to specify the block's output. If you select **Magnitude-squared**, the block outputs the optical flow matrix where each element is of the form  $u^2 + v^2$ . If you select **Horizontal and vertical components in complex form**, the block outputs the optical flow matrix where each element is of the form  $u + jv$ . The real part of each value is the horizontal velocity component and the imaginary part of each value is the vertical velocity component.

## Fixed-Point Data Types

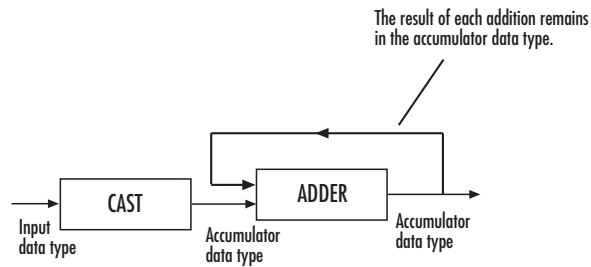
The following diagram shows the data types used in the Block Matching block for fixed-point signals.



MSE Block Matching



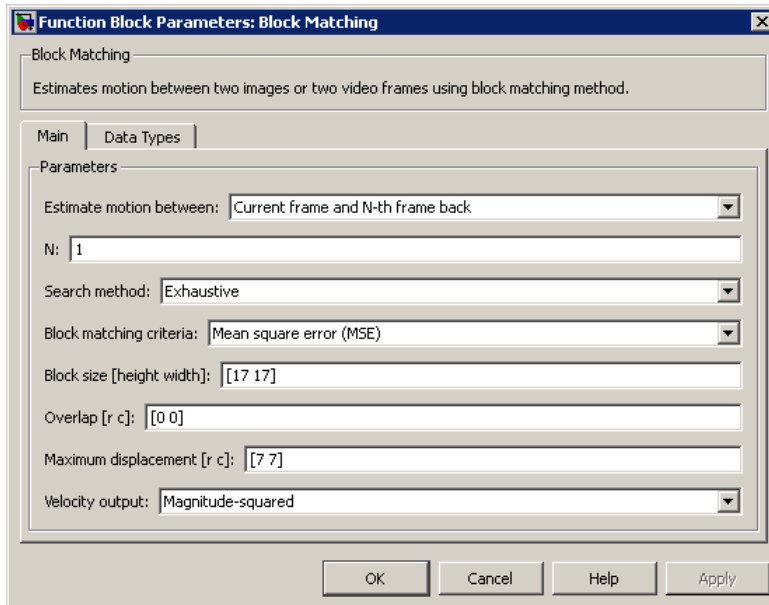
MAD Block Matching



You can set the accumulator and output data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Block Matching dialog box appears as shown in the following figure.



## Estimate motion between

Select **Two images** to estimate the motion between two images. Select **Current frame and N-th frame back** to estimate the motion between two video frames that are N frames apart.

## N

Enter a scalar value that represents the number of frames between the reference frame and the current frame. This parameter is only visible if, for the **Estimate motion between** parameter, you select **Current frame and N-th frame back**.

## Search method

Specify how the block searches for the block of pixels in the next image or frame. Your choices are **Exhaustive** or **Three-step**.

## Block matching criteria

Specify how the block measures the similarity of the block of pixels in frame k to the block of pixels in frame k+1. Your choices are **Mean square error (MSE)** or **Mean absolute difference (MAD)**.

## Block size [height width]

Specify the size of the block of pixels.

### Overlap [r c]

Specify the overlap (in pixels) of two subdivisions of the input image.

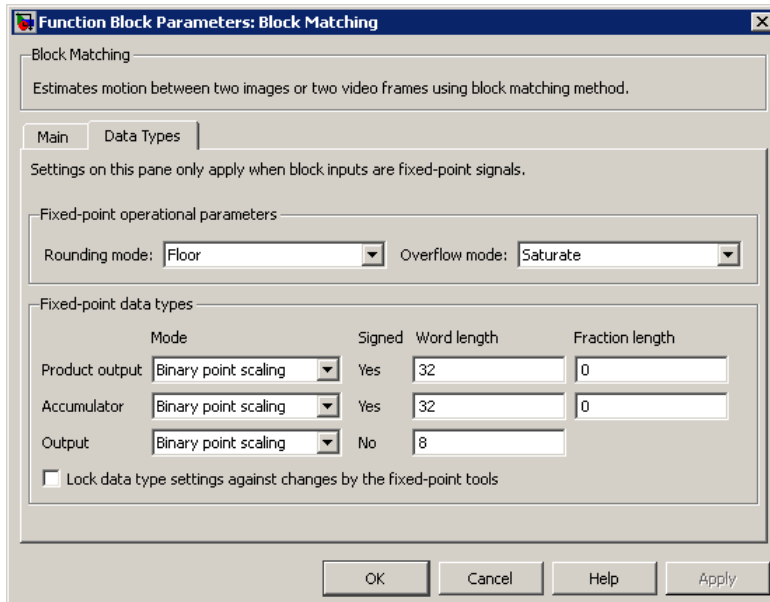
### Maximum displacement [r c]

Specify the maximum number of pixels any center pixel in a block of pixels might move from image to image or frame to frame. The block uses this value to determine the size of the search region.

### Velocity output

If you select **Magnitude-squared**, the block outputs the optical flow matrix where each element is of the form  $u^2 + v^2$ . If you select **Horizontal and vertical components in complex form**, the block outputs the optical flow matrix where each element is of the form  $u + jv$ .

The **Data Types** pane of the Block Matching dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

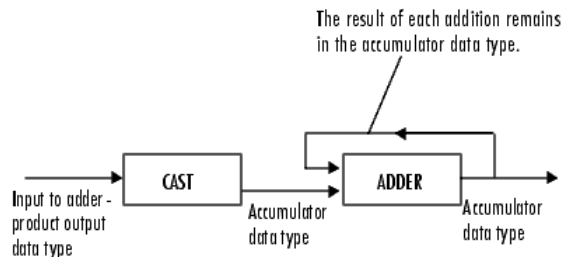
### Product output



As shown previously, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate the product output word and fraction lengths.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Accumulator



As depicted previously, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Binary point scaling**, you can enter the word length of the output, in bits. The fractional length is always 0.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, of the output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

Optical Flow	Computer Vision System Toolbox software
--------------	---

**Introduced before R2006a**

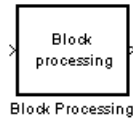
# Block Processing

Repeat user-specified operation on submatrices of input matrix

## Library

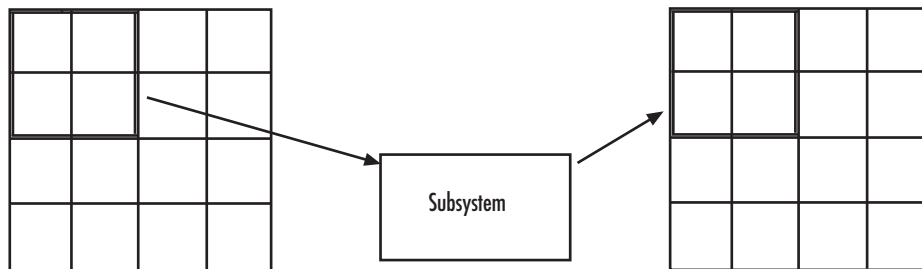
Utilities

visionutilities



## Description

The Block Processing block extracts submatrices of a user-specified size from each input matrix. It sends each submatrix to a subsystem for processing, and then reassembles each subsystem output into the output matrix.



---

**Note** Because you modify the Block Processing block's subsystem, the link between this block and the block library is broken when you click-and-drag a Block Processing block into your model. As a result, this block will not be automatically updated if you upgrade to a newer version of the Computer Vision System Toolbox software. If you right-click on the block and select **Mask>Look Under Mask**, you can delete blocks from this subsystem without triggering a warning. Lastly, if you search for library blocks in a model, this block will not be part of the results.

---

The blocks inside the subsystem dictate the frame status of the input and output signals, whether single channel or multichannel signals are supported, and which data types are supported by this block.

Use the **Number of inputs** and **Number of outputs** parameters to specify the number of input and output ports on the Block Processing block.

Use the **Block size** parameter to specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input; the block uses the vectors in the order you enter them. If you have one input port, enter one vector. If you have more than one input port, you can enter one vector that is used for all inputs or you can specify a different vector for each input. For example, if you want each submatrix to be 2-by-3, enter `{[2 3]}`.

Use the **Overlap** parameter to specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input; the block uses the vectors in the order they are specified. If you enter one vector, each overlap is the same size. For example, if you want each 3-by-3 submatrix to overlap by 1 row and 2 columns, enter `{[1 2]}`.

The **Traverse order** parameter determines how the block extracts submatrices from the input matrix. If you select **Row-wise**, the block extracts submatrices by moving across the rows. If you select **Column-wise**, the block extracts submatrices by moving down the columns.

Click the **Open Subsystem** button to open the block's subsystem. Click-and-drag blocks into this subsystem to define the processing operation(s) the block performs on the submatrices. The input to this subsystem are the submatrices whose size is determined by the **Block size** parameter.

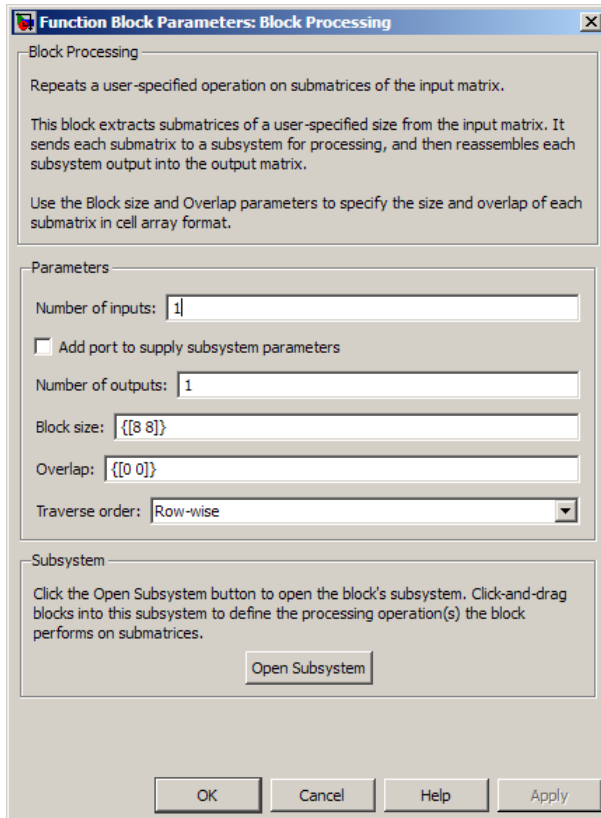
---

**Note:** When you place an Assignment block inside a Block Processing block's subsystem, the Assignment block behaves as though it is inside a **For Iterator** block. For a description of this behavior, see the “Iterated Assignment” section of the **Assignment** block reference page.

---

## Dialog Box

The Block Processing dialog box appears as shown in the following figure.



## Number of inputs

Enter the number of input ports on the Block Processing block.

## Add port to supply subsystem parameters

Add an input port to the block to supply subsystem parameters.

## Number of outputs

Enter the number of output ports on the Block Processing block.

## Block size

Specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input.

## Overlap



Specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input.

**Traverse order**

Determines how the block extracts submatrices from the input matrix. If you select **Row-wise**, the block extracts submatrices by moving across the rows. If you select **Column-wise**, the block extracts submatrices by moving down the columns.

**Open Subsystem**

Click this button to open the block's subsystem. Click-and-drag blocks into this subsystem to define the processing operation(s) the block performs on the submatrices.

## See Also

For Iterator	Simulink
blockproc	Image Processing Toolbox

**Introduced before R2006a**

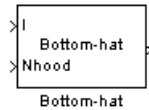
# Bottom-hat

Perform bottom-hat filtering on intensity or binary images

## Library

Morphological Operations

visionmorphops



## Description

Use the Bottom-hat block to perform bottom-hat filtering on an intensity or binary image using a predefined neighborhood or structuring element. Bottom-hat filtering is the equivalent of subtracting the input image from the result of performing a morphological closing operation on the input image. This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of ones and zeros that represents the neighborhood values	Boolean	No
Output	Scalar, vector, or matrix that represents the filtered image	Same as I port	No

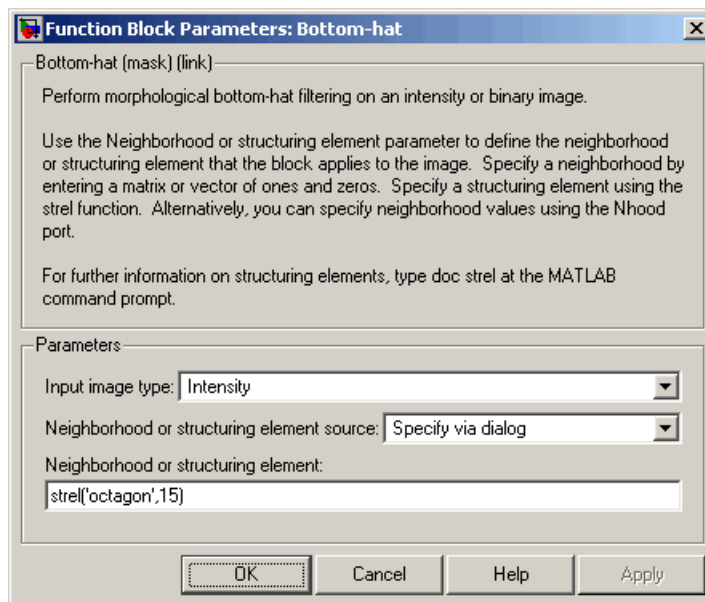
If your input image is a binary image, for the **Input image type** parameter, select **Binary**. If your input image is an intensity image, select **Intensity**.

Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the **Nhood** port appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. You can only specify a structuring element using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the region the block moves throughout the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm.

## Dialog Box

The Bottom-hat dialog box appears as shown in the following figure.



### Input image type

If your input image is a binary image, select **Binary**. If your input image is an intensity image, select **Intensity**.

### Neighborhood or structuring element source

Specify how to enter your neighborhood or structuring element values. Select **Specify via dialog** to enter the values in the dialog box. Select **Input port** to use the Nhood port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

### Neighborhood or structuring element

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the **strel** function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select **Specify via dialog**.

## See Also

Closing	Computer Vision System Toolbox software
Dilation	Computer Vision System Toolbox software
Erosion	Video and Image Processing Blockset software
Label	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
imbothat	Image Processing Toolbox software
strel	Image Processing Toolbox software

Introduced before R2006a

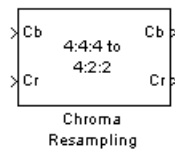
# Chroma Resampling

Downsample or upsample chrominance components of images

## Library

Conversions

visionconversions



## Description

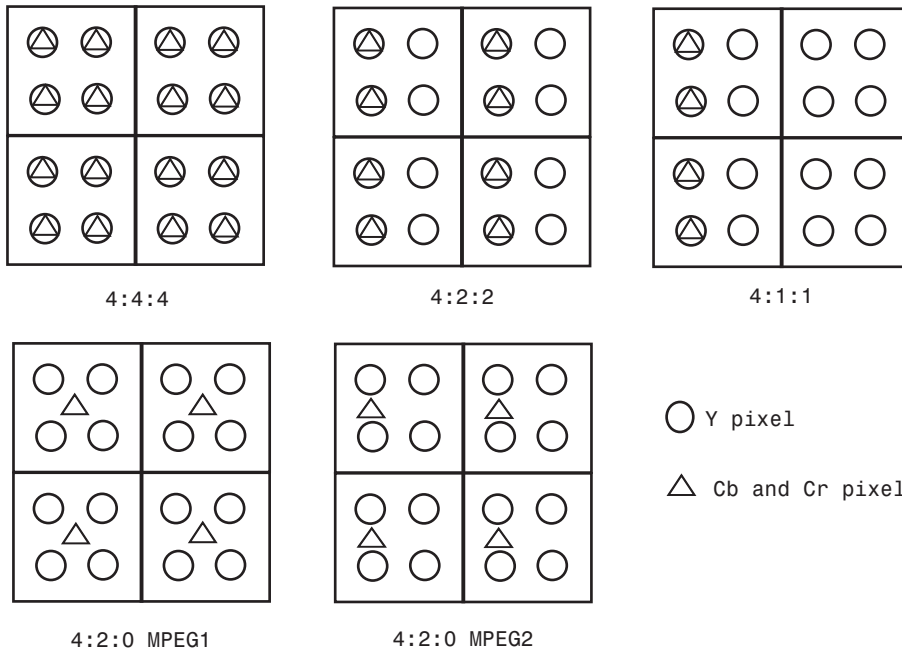
The Chroma Resampling block downsamples or upsamples chrominance components of pixels to reduce the bandwidth required for transmission or storage of a signal.

Port	Input/Output	Supported Data Types	Complex Values Supported
Cb	Matrix that represents one chrominance component of an image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-bit unsigned integer</li> </ul>	No
Cr	Matrix that represents one chrominance component of an image	Same as Cb port	No

The data type of the output signals is the same as the data type of the input signals.

## Chroma Resampling Formats

The Chroma Resampling block supports the formats shown in the following diagram.



## Downsampling

If, for the **Resampling** parameter, you select **4:4:4 to 4:2:2**, **4:4:4 to 4:2:0 (MPEG1)**, **4:4:4 to 4:2:0 (MPEG2)**, **4:4:4 to 4:1:1**, **4:2:2 to 4:2:0 (MPEG1)**, or **4:2:2 to 4:2:0 (MPEG2)**, the block performs a downsampling operation. When the block downsamples from one format to another, it can bandlimit the input signal by applying a lowpass filter to prevent aliasing.

If, for the **Antialiasing filter** parameter, you select **Default**, the block uses a built-in lowpass filter to prevent aliasing.

If, for the **Resampling** parameter, you select **4:4:4 to 4:2:2**, **4:4:4 to 4:2:0 (MPEG1)**, **4:4:4 to 4:2:0 (MPEG2)**, or **4:4:4 to 4:1:1** and, for the **Antialiasing filter** parameter, you select **User-defined**, the **Horizontal filter coefficients** parameter appears on the dialog box. Enter the filter coefficients to apply to your input.

If, for the **Resampling** parameter, you select **4:4:4 to 4:2:0 (MPEG1)**, **4:4:4 to 4:2:0 (MPEG2)**, **4:2:2 to 4:2:0 (MPEG1)**, or

4:2:2 to 4:2:0 (MPEG2) and, for the **Antialiasing filter** parameter, you select **User-defined**. **Vertical filter coefficients** parameters appear on the dialog box. Enter an even number of filter coefficients to apply to your input signal.

If, for the **Antialiasing filter** parameter, you select **None**, the block does not filter the input signal.

## Upsampling

If, for the **Resampling** parameter, you select 4:2:2 to 4:4:4, 4:2:0 (MPEG1) to 4:2:2, 4:2:0 (MPEG1) to 4:4:4, 4:2:0 (MPEG2) to 4:2:2, 4:2:0 (MPEG2) to 4:4:4, or 4:1:1 to 4:4:4, the block performs an upsampling operation.

When the block upsamples from one format to another, it uses interpolation to approximate the missing chrominance values. If, for the **Interpolation** parameter, you select **Linear**, the block uses linear interpolation to calculate the missing values. If, for the **Interpolation** parameter, you select **Pixel replication**, the block replicates the chrominance values of the neighboring pixels to create the upsampled image.

## Row-Major Data Format

The MATLAB environment and the Computer Vision System Toolbox software use column-major data organization. However, the Chroma Resampling block gives you the option to process data that is stored in row-major format. When you select the **Input image is transposed (data order is row major)** check box, the block assumes that the input buffer contains contiguous data elements from the first row first, then data elements from the second row second, and so on through the last row. Use this functionality only when you meet all the following criteria:

- You are developing algorithms to run on an embedded target that uses the row-major format.
- You want to limit the additional processing required to take the transpose of signals at the interfaces of the row-major and column-major systems.

When you use the row-major functionality, you must consider the following issues:

- When you select this check box, the signal dimensions of the Chroma Resampling block's input are swapped.

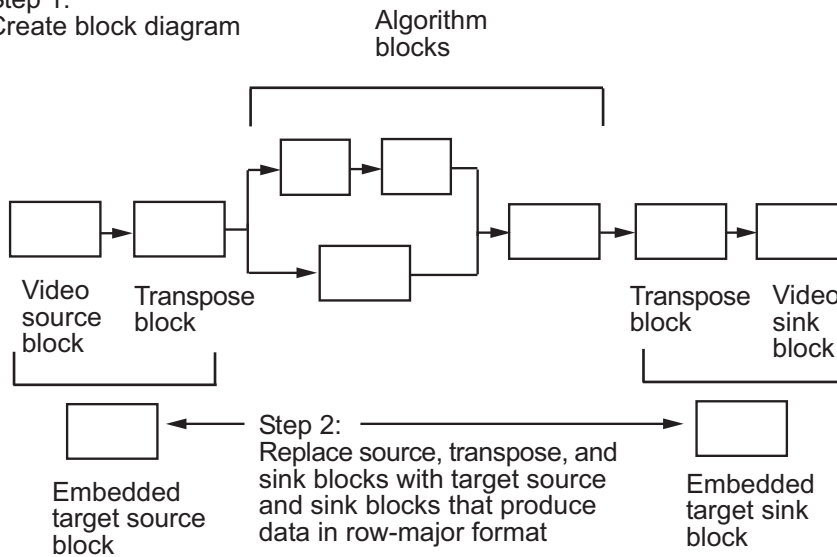
- All the Computer Vision System Toolbox blocks can be used to process data that is in the row-major format, but you need to know the image dimensions when you develop your algorithms.

For example, if you use the 2-D FIR Filter block, you need to verify that your filter coefficients are transposed. If you are using the Rotate block, you need to use negative rotation angles, etc.

- Only three blocks have the **Input image is transposed (data order is row major)** check box. They are the Chroma Resampling, Deinterlacing, and Insert Text blocks. You need to select this check box to enable row-major functionality in these blocks. All other blocks must be properly configured to process data in row-major format.

Use the following two-step workflow to develop algorithms in row-major format to run on an embedded target.

Step 1:  
Create block diagram

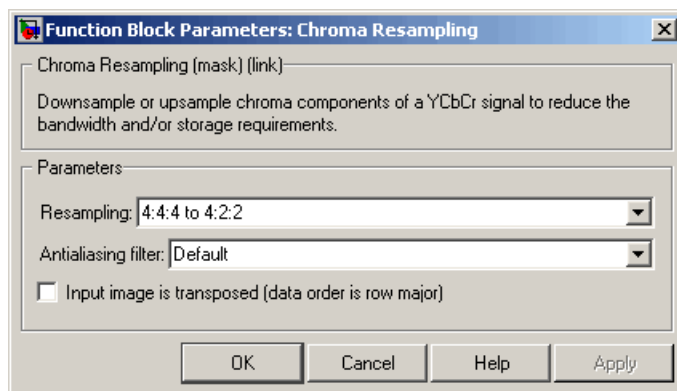


See the DM642 EVM Video ADC and DM642 EVM Video DAC reference pages.

## Dialog Box

The Chroma Resampling dialog box appears as shown in the following figure.





## Resampling

Specify the resampling format.

## Antialiasing filter

Specify the lowpass filter that the block uses to prevent aliasing. If you select **Default**, the block uses a built-in lowpass filter. If you select **User-defined**, the **Horizontal filter coefficients** and/or **Vertical filter coefficients** parameters appear on the dialog box. If you select **None**, the block does not filter the input signal. This parameter is visible when you are downsampling the chrominance values.

## Horizontal filter coefficients

Enter the filter coefficients to apply to your input signal. This parameter is visible if, for the **Resampling** parameter, you select **4:4:4 to 4:2:2**, **4:4:4 to 4:2:0 (MPEG1)**, **4:4:4 to 4:2:0 (MPEG2)**, or **4:4:4 to 4:1:1** and, for the **Antialiasing filter** parameter, you select **User-defined**.

## Vertical filter coefficients

Enter the filter coefficients to apply to your input signal. This parameter is visible if, for the **Resampling** parameter, you select **4:4:4 to 4:2:0 (MPEG1)**, **4:4:4 to 4:2:0 (MPEG2)**, **4:2:2 to 4:2:0 (MPEG1)**, or **4:2:2 to 4:2:0 (MPEG2)** and, for the **Antialiasing filter** parameter, you select **User-defined**.

## Interpolation

Specify the interpolation method that the block uses to approximate the missing chrominance values. If you select **Linear**, the block uses linear interpolation to calculate the missing values. If you select **Pixel replication**, the block replicates the chrominance values of the neighboring pixels to create the upsampled image.

This parameter is visible when you are upsampling the chrominance values. This parameter is visible if the **Resampling** parameter is set to 4:2:2 to 4:4:4 , 4:2:0 (MPEG1) to 4:4:4 , 4:2:0 (MPEG2) to 4:4:4 , 4:1:1 to 4:4:4 , 4:2:0 (MPEG1) to 4:2:2 , or 4:2:0 (MPEG2) to 4:2:2 .

**Input image is transposed (data order is row major)**

When you select this check box, the block assumes that the input buffer contains data elements from the first row first, then data elements from the second row second, and so on through the last row.

## References

- [1] Haskell, Barry G., Atul Puri, and Arun N. Netravali. *Digital Video: An Introduction to MPEG-2*. New York: Chapman & Hall, 1996.
- [2] Recommendation ITU-R BT.601-5, Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide Screen 16:9 Aspect Ratios.
- [3] Wang, Yao, Jorn Ostermann, Ya-Qin Zhang. *Video Processing and Communications*. Upper Saddle River, NJ: Prentice Hall, 2002.

## See Also

Autothreshold	Computer Vision System Toolbox software
Color Space Conversion	Computer Vision System Toolbox software
Image Complement	Computer Vision System Toolbox software

**Introduced before R2006a**

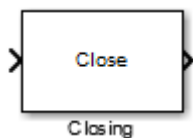
# Closing

Perform morphological closing on binary or intensity images

## Library

Morphological Operations

visionmorphops



## Description

The Closing block performs a dilation operation followed by an erosion operation using a predefined neighborhood or structuring element. This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of ones and zeros that represents the neighborhood values	Boolean	No
Output	Vector or matrix of intensity values that represents the closed image	Same as I port	No

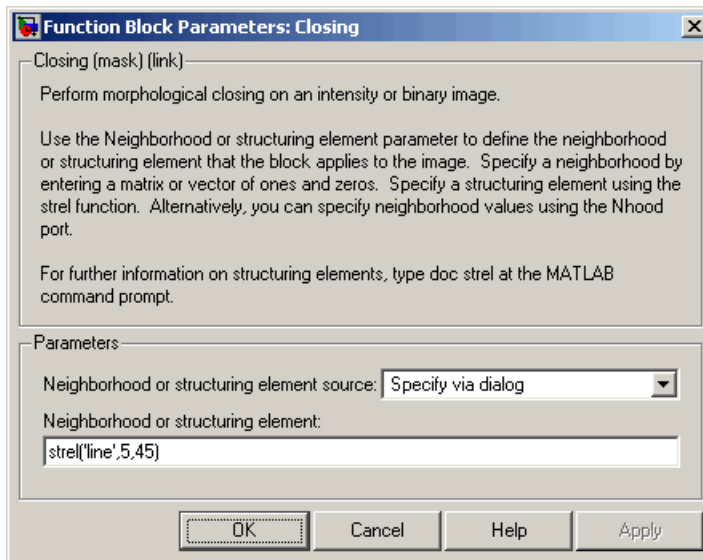
The output signal has the same data type as the input to the I port.

Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the Nhood port appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. You can only specify a structuring element using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the region the block moves throughout the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm.

## Dialog Box

The Closing dialog box appears as shown in the following figure.



### Neighborhood or structuring element source

Specify how to enter your neighborhood or structuring element values. Select `Specify via dialog` to enter the values in the dialog box. Select `Input port` to use the `Nhood` port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

### Neighborhood or structuring element

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select `Specify via dialog`.

## References

[1] Soille, Pierre. *Morphological Image Analysis*. 2nd ed. New York: Springer, 2003.

## See Also

Bottom-hat	Computer Vision System Toolbox software
Dilation	Computer Vision System Toolbox software
Erosion	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
<code>imclose</code>	Image Processing Toolbox software
<code>strel</code>	Image Processing Toolbox software

**Introduced before R2006a**

# Color Space Conversion

Convert color information between color spaces

## Library

Conversions

visionconversions



## Description

The Color Space Conversion block converts color information between color spaces. Use the **Conversion** parameter to specify the color spaces you are converting between. Your choices are R'G'B' to Y'CbCr, Y'CbCr to R'G'B', R'G'B' to intensity, R'G'B' to HSV, HSV to R'G'B', sR'G'B' to XYZ, XYZ to sR'G'B', sR'G'B' to L\*a\*b\*, and L\*a\*b\* to sR'G'B'.

- If the input is `uint8`, YCbCr is `uint8`, where Y is in the range [16 235], and Cb and Cr are in the range [16 240].
- If the input is a `double`, Y is in the range [16/255 235/255] and Cb and Cr are in the range [16/255 240/255].

Port	Input/Output	Supported Data Types	Complex Values Supported
Input / Output	M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-bit unsigned integer</li> </ul>	No
R'	Matrix that represents one plane of the input RGB video stream	Same as the Input port	No

Port	Input/Output	Supported Data Types	Complex Values Supported
G'	Matrix that represents one plane of the input RGB video stream	Same as the Input port	No
B'	Matrix that represents one plane of the input RGB video stream	Same as the Input port	No
Y'	Matrix that represents the luma portion of an image	Same as the Input port	No
Cb	Matrix that represents one chrominance component of an image	Same as the Input port	No
Cr	Matrix that represents one chrominance component of an image	Same as the Input port	No
I'	Matrix of intensity values	Same as the Input port	No
H	Matrix that represents the hue component of an image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>	No
S	Matrix that represents represent the saturation component of an image	Same as the H port	No
V	Matrix that represents the value (brightness) component of an image	Same as the H port	No
X	Matrix that represents the X component of an image	Same as the H port	No
Y	Matrix that represents the Y component of an image	Same as the H port	No
Z	Matrix that represents the Z component of an image	Same as the H port	No
L*	Matrix that represents the luminance portion of an image	Same as the H port	No
a*	Matrix that represents the a* component of an image	Same as the H port	No
b*	Matrix that represents the b* component of an image	Same as the H port	No

The data type of the output signal is the same as the data type of the input signal.

Use the **Image signal** parameter to specify how to input and output a color video signal. If you select **One multidimensional signal**, the block accepts an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port accepts one M-by-N plane of an RGB video stream.

---

**Note** The prime notation indicates that the signals are gamma corrected.

---

## Conversion Between R'G'B' and Y'CbCr Color Spaces

The following equations define R'G'B' to Y'CbCr conversion and Y'CbCr to R'G'B' conversion:

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + A \times \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = B \times \left( \begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right)$$

The values in matrices A and B are based on your choices for the **Use conversion specified by** and **Scanning standard** parameters.

Matrix	Use conversion specified by = Rec. 601 (SDTV)	Use conversion specified by = Rec. 709 (HDTV)	
		Scanning standard = 1125/60/2:1	Scanning standard = 1250/50/2:1
A	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.07142737 \end{bmatrix}$	$\begin{bmatrix} 0.18258588 & 0.61423059 & 0.06200706 \\ -0.10064373 & -0.33857195 & 0.43921569 \\ 0.43921569 & -0.39894216 & -0.04027352 \end{bmatrix}$	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.07142737 \end{bmatrix}$
B	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.16438356 & 0 & 1.79274107 \\ 1.16438356 & -0.21324861 & -0.53290933 \\ 1.16438356 & 2.11240179 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$



## Conversion from R'G'B' to Intensity

The following equation defines conversion from R'G'B' color space to intensity:

$$\text{intensity} = [0.299 \quad 0.587 \quad 0.114] \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

## Conversion Between R'G'B' and HSV Color Spaces

The R'G'B' to HSV conversion is defined by the following equations. In these equations, *MAX* and *MIN* represent the maximum and minimum values of each R'G'B' triplet, respectively. *H*, *S*, and *V* vary from 0 to 1, where 1 represents the greatest saturation and value.

$$H = \begin{cases} \left( \frac{G' - B'}{MAX - MIN} \right) / 6, & \text{if } R' = MAX \\ \left( 2 + \frac{B' - R'}{MAX - MIN} \right) / 6, & \text{if } G' = MAX \\ \left( 4 + \frac{R' - G'}{MAX - MIN} \right) / 6, & \text{if } B' = MAX \end{cases}$$

$$S = \frac{MAX - MIN}{MAX}$$

$$V = MAX$$

The HSV to R'G'B' conversion is defined by the following equations:

$$\begin{aligned}H_i &= \lfloor 6H \rfloor \\f &= 6H - H_i \\p &= 1 - S \\q &= 1 - fS \\t &= 1 - (1 - f)S \\ \text{if } H_i = 0, & \quad R_{tmp} = 1, \quad G_{tmp} = t, \quad B_{tmp} = p \\ \text{if } H_i = 1, & \quad R_{tmp} = q, \quad G_{tmp} = 1, \quad B_{tmp} = p \\ \text{if } H_i = 2, & \quad R_{tmp} = p, \quad G_{tmp} = 1, \quad B_{tmp} = t \\ \text{if } H_i = 3, & \quad R_{tmp} = p, \quad G_{tmp} = q, \quad B_{tmp} = 1 \\ \text{if } H_i = 4, & \quad R_{tmp} = t, \quad G_{tmp} = p, \quad B_{tmp} = 1 \\ \text{if } H_i = 5, & \quad R_{tmp} = 1, \quad G_{tmp} = p, \quad B_{tmp} = q \\u &= V / \max(R_{tmp}, G_{tmp}, B_{tmp}) \\R' &= uR_{tmp} \\G' &= uG_{tmp} \\B' &= uB_{tmp}\end{aligned}$$

For more information about the HSV color space, see “Convert from HSV to RGB Color Space” in the Image Processing Toolbox documentation.

## Conversion Between sR'G'B' and XYZ Color Spaces

The sR'G'B' to XYZ conversion is a two-step process. First, the block converts the gamma-corrected sR'G'B' values to linear sRGB values using the following equations:

$$\begin{aligned}
&\text{If } R'_{sRGB}, G'_{sRGB}, B'_{sRGB} \leq 0.03928 \\
&R_{sRGB} = R'_{sRGB} / 12.92 \\
&G_{sRGB} = G'_{sRGB} / 12.92 \\
&B_{sRGB} = B'_{sRGB} / 12.92 \\
&\text{otherwise, if } R'_{sRGB}, G'_{sRGB}, B'_{sRGB} > 0.03928 \\
&R_{sRGB} = \left[ \frac{(R'_{sRGB} + 0.055)}{1.055} \right]^{2.4} \\
&G_{sRGB} = \left[ \frac{(G'_{sRGB} + 0.055)}{1.055} \right]^{2.4} \\
&B_{sRGB} = \left[ \frac{(B'_{sRGB} + 0.055)}{1.055} \right]^{2.4}
\end{aligned}$$

Then the block converts the sRGB values to XYZ values using the following equation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.41239079926596 & 0.35758433938388 & 0.18048078840183 \\ 0.21263900587151 & 0.71516867876776 & 0.07219231536073 \\ 0.01933081871559 & 0.11919477979463 & 0.95053215224966 \end{bmatrix} \times \begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix}$$

The XYZ to sR'G'B' conversion is also a two-step process. First, the block converts the XYZ values to linear sRGB values using the following equation:

$$\begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix} = \begin{bmatrix} 0.41239079926596 & 0.35758433938388 & 0.18048078840183 \\ 0.21263900587151 & 0.71516867876776 & 0.07219231536073 \\ 0.01933081871559 & 0.11919477979463 & 0.95053215224966 \end{bmatrix}^{-1} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Then the block applies gamma correction to obtain the sR'G'B' values. This process is described by the following equations:

$$\begin{aligned}
 &\text{If } R_{sRGB}, G_{sRGB}, B_{sRGB} \leq 0.00304 \\
 &R'_{sRGB} = 12.92R_{sRGB} \\
 &G'_{sRGB} = 12.92G_{sRGB} \\
 &B'_{sRGB} = 12.92B_{sRGB} \\
 &\text{otherwise, if } R_{sRGB}, G_{sRGB}, B_{sRGB} > 0.00304 \\
 &R'_{sRGB} = 1.055R_{sRGB}^{(1.0/2.4)} - 0.055 \\
 &G'_{sRGB} = 1.055G_{sRGB}^{(1.0/2.4)} - 0.055 \\
 &B'_{sRGB} = 1.055B_{sRGB}^{(1.0/2.4)} - 0.055
 \end{aligned}$$

---

**Note:** Computer Vision System Toolbox software uses a D65 white point, which is specified in Recommendation ITU-R BT.709, for this conversion. In contrast, the Image Processing Toolbox conversion is based on ICC profiles, and it uses a D65 to D50 Bradford adaptation transformation to the D50 white point. If you are using these two products and comparing results, you must account for this difference.

---

## Conversion Between sR'G'B' and L\*a\*b\* Color Spaces

The Color Space Conversion block converts sR'G'B' values to L\*a\*b\* values in two steps. First it converts sR'G'B' to XYZ values using the equations described in “Conversion Between sR'G'B' and XYZ Color Spaces” on page 1-230. Then it uses the following equations to transform the XYZ values to L\*a\*b\* values. Here,  $X_n$ ,  $Y_n$ , and  $Z_n$  are the tristimulus values of the reference white point you specify using the **White point** parameter:

$$\begin{aligned}
 L^* &= 116(Y/Y_n)^{1/3} - 16, \text{ for } Y/Y_n > 0.008856 \\
 L^* &= 903.3 Y/Y_n, \quad \text{otherwise}
 \end{aligned}$$

$$\begin{aligned}
 a^* &= 500(f(X/X_n) - f(Y/Y_n)) \\
 b^* &= 200(f(Y/Y_n) - f(Z/Z_n)), \\
 &\text{where } f(t) = t^{1/3}, \text{ for } t > 0.008856 \\
 &f(t) = 7.787t + 16/166, \quad \text{otherwise}
 \end{aligned}$$

The block converts L\*a\*b\* values to sR'G'B' values in two steps as well. The block transforms the L\*a\*b\* values to XYZ values using these equations:

For  $Y/Y_n > 0.008856$

$$X = X_n (P + a^*/500)^3$$

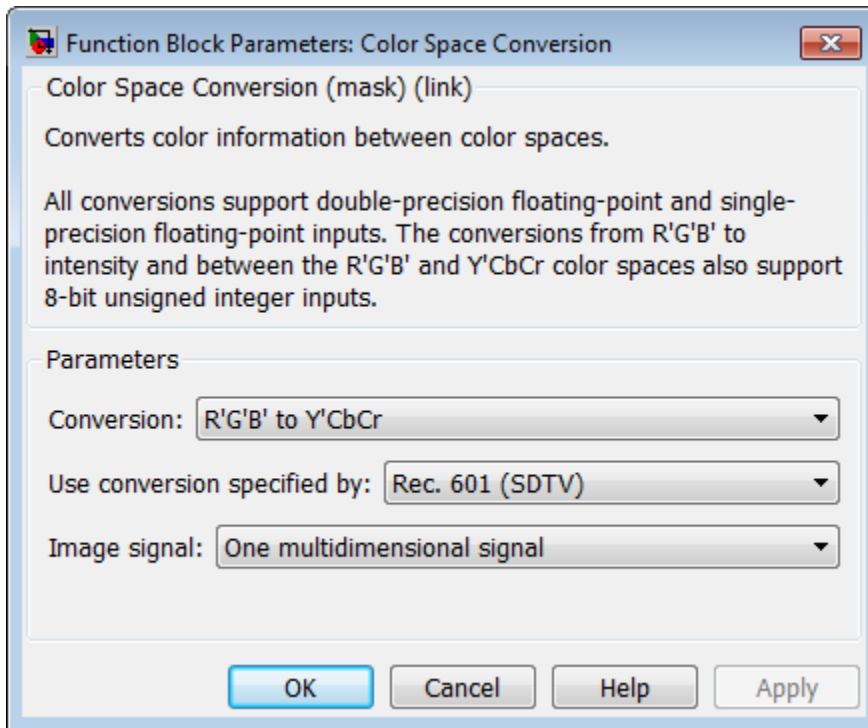
$$Y = Y_n P^3$$

$$Z = Z_n (P - b^*/200)^3,$$

where  $P = (L^* + 16) / 116$

## Dialog Box

The Color Space Conversion dialog box appears as shown in the following figure.



### Conversion

Specify the color spaces you are converting between. Your choices are R'G'B' to Y'CbCr, Y'CbCr to R'G'B', R'G'B' to intensity, R'G'B' to HSV, HSV to R'G'B', sR'G'B' to XYZ, XYZ to sR'G'B', sR'G'B' to L\*a\*b\*, and L\*a\*b\* to sR'G'B'.

### Use conversion specified by

Specify the standard to use to convert your values between the R'G'B' and Y'CbCr color spaces. Your choices are Rec. 601 (SDTV) or Rec. 709 (HDTV). This parameter is only available if, for the **Conversion** parameter, you select R'G'B' to Y'CbCr or Y'CbCr to R'G'B'.

### Scanning standard

Specify the scanning standard to use to convert your values between the R'G'B' and Y'CbCr color spaces. Your choices are 1125/60/2:1 or 1250/50/2:1. This parameter is only available if, for the **Use conversion specified by** parameter, you select Rec. 709 (HDTV).

**White point**

Specify the reference white point. This parameter is visible if, for the **Conversion** parameter, you select **SR'G'B'** to **L\*a\*b\*** or **L\*a\*b\*** to **SR'G'B'**.

**Image signal**

Specify how to input and output a color video signal. If you select **One multidimensional signal**, the block accepts an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port accepts one M-by-N plane of an RGB video stream.

**References**

- [1] Poynton, Charles A. *A Technical Introduction to Digital Video*. New York: John Wiley & Sons, 1996.
- [2] Recommendation ITU-R BT.601-5, Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide Screen 16:9 Aspect Ratios.
- [3] Recommendation ITU-R BT.709-5. Parameter values for the HDTV standards for production and international programme exchange.
- [4] Stokes, Michael, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta, "A Standard Default Color Space for the Internet - sRGB." November 5, 1996.
- [5] Berns, Roy S. *Principles of Color Technology, 3rd ed.* New York: John Wiley & Sons, 2000.

**See Also**

Chroma Resampling	Computer Vision System Toolbox software
rgb2hsv	MATLAB software
hsv2rgb	MATLAB software
rgb2ycbcr	Image Processing Toolbox software
ycbcr2rgb	Image Processing Toolbox software
rgb2gray	Image Processing Toolbox software

makecform	Image Processing Toolbox software
applycform	Image Processing Toolbox software

**Introduced before R2006a**



# Compositing

Combine pixel values of two images, overlay one image over another, or highlight selected pixels

## Library

Text & Graphics

visiontextngfix



## Description

You can use the Compositing block to combine two images. Each pixel of the output image is a linear combination of the pixels in each input image. This process is defined by the following equation:

$$O(i, j) = (1 - X) * I1(i, j) + X * I2(i, j)$$

You can define the amount by which to scale each pixel value before combining them using the opacity factor,  $X$ , where ,  $0 \leq X \leq 1$ .

You can use the **Compositing** block to overlay one image over another image. The masking factor and the location determine which pixels are overwritten. Masking factors can be 0 or 1, where 0 corresponds to not overwriting pixels and 1 corresponds to overwriting pixels.

You can also use this block to highlight selected pixels in the input image. The block uses a binary input image at the **Mask** port, to specify which pixels to highlight.

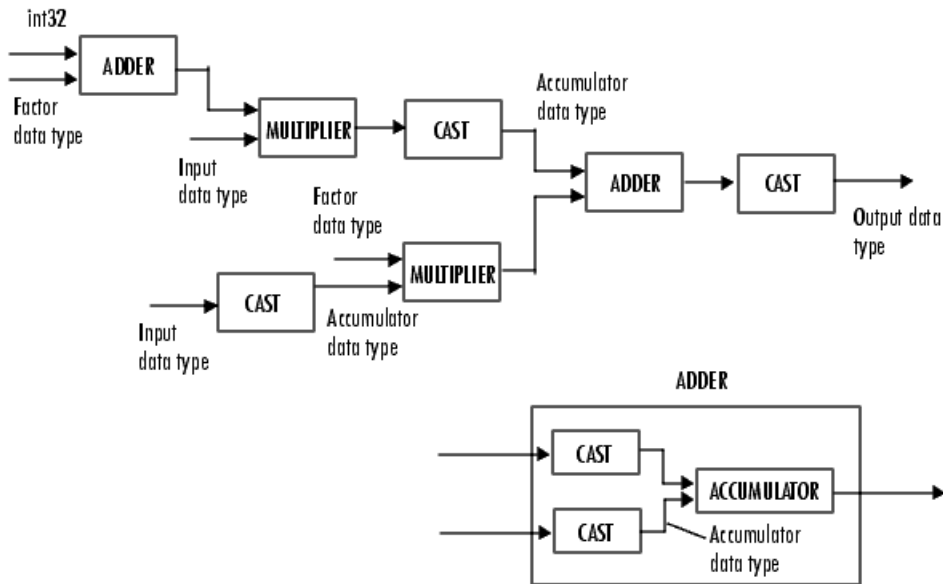
---

**Note:** This block supports intensity and color images.

---

## Fixed-Point Data Types

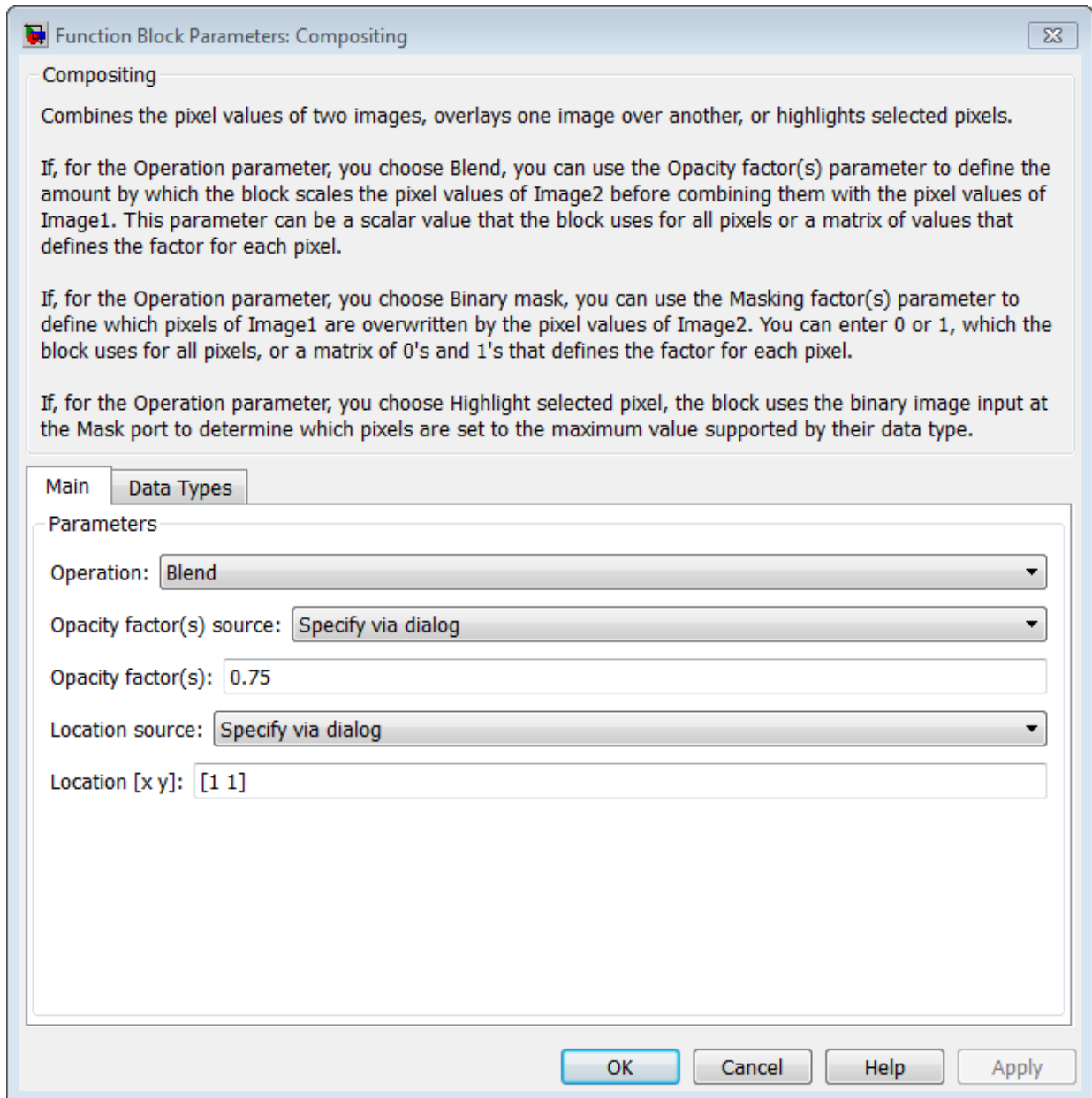
The following diagram shows the data types used in the Compositing block for fixed-point signals. These data types applies when the **Operation** parameter is set to **Blend**.



You can set the product output, accumulator, and output data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Compositing dialog box appears as shown in the following figure.



## Operation

Specify the operation you want the block to perform. If you choose **Blend**, the block linearly combines the pixels of one image with another image. If you choose **Binary mask**, the block overwrites the pixel values of one image with the pixel values of another image. If you choose **Highlight selected pixels**, the block uses the binary image input at the **Mask** port. Using this image, the block then determines which pixels are set to the maximum value supported by their data type.

### Blend

If, for the **Operation** parameter, you choose **Blend**, the **Opacity factor(s) source** parameter appears on the dialog box. Use this parameter to indicate where to specify the opacity factor(s).

- If you choose **Specify via dialog**, the **Opacity factor(s)** parameter appears on the dialog box. Use this parameter to define the amount by which the block scales each pixel values for input image at the **Image2** port before combining them with the pixel values of the input image at **Image1** port. You can enter a scalar value used for all pixels or a matrix of values that is the same size as the input image at the **Image2** port.
- If you choose **Input port**, the **Factor** port appears on the block. The input to this port must be a scalar or matrix of values as described for the **Opacity factor(s)** parameter. If the input to the **Image1** and **Image2** ports is floating point, the input to this port must be the same floating-point data type.

### Binary mask

If, for the **Operation** parameter, you choose **Binary mask**, the **Mask source** parameter appears on the dialog box. Use this parameter to indicate where to specify the masking factor(s).

- If you choose **Specify via dialog**, the **Mask** parameter appears on the dialog box. Use this parameter and the location source of the image to define which pixels are overwritten. You can enter 0 or 1 to use for all pixels in the image, or a matrix of 0s and 1s that defines the factor for each pixel.
- If you choose **Input port**, the **Factor** port appears on the block. The input to this port must be a 0 or 1 whose data type is Boolean. Or, a matrix of 0s or 1s whose data type is Boolean, as described for the **Mask** parameter.

### Highlight selected pixels

If, for the **Operation** parameter, you choose **Highlight selected pixels**, the block uses the binary image input at the **Mask** port to determine which pixels are set to the maximum value supported by their data type. For example, for every pixel value set to 1 in the binary image, the block sets the corresponding pixel in the input image to the maximum value supported by its data type. For every 0 in the binary image, the block leaves the corresponding pixel value alone.

### Opacity factor(s) source

Indicate where to specify any opacity factors. Your choices are **Specify via dialog** and **Input port**. This parameter is visible if, for the **Operation** parameter, you choose **Blend**.

### Opacity factor(s)

Define the amount by which the block scales each pixel value before combining them. You can enter a scalar value used for all pixels or a matrix of values that defines the factor for each pixel. This parameter is visible if, for the **Opacity factor(s) source** parameter, you choose **Specify via dialog**. Tunable.

### Mask source

Indicate where to specify any masking factors. Your choices are **Specify via dialog** and **Input port**. This parameter is visible if, for the **Operation** parameter, you choose **Binary mask**.

### Mask

Define which pixels are overwritten. You can enter 0 or 1, which is used for all pixels, or a matrix of 0s and 1s that defines the factor for each pixel. This parameter is visible if, for the **Mask source** parameter, you choose **Specify via dialog**. Tunable.

### Location source

Use this parameter to specify where to enter the location of the upper-left corner of the image input at input port **Image2**. You can choose either **Specify via dialog** or **Input port**.

When you choose **Specify via dialog**, you can set the **Location [x y]** parameter.

When you choose **Input port**, the **Location** port appears on the block. The input to this port must be a two-element vector as described for the **Location [x y]** parameter.

### **Location [x y]**

Enter a two-element vector that specifies the row and column position of the upper-left corner of the image input at **Image2** port. The position is relative to the upper-left corner of the image input at **Image1** port. This parameter is visible if, for the **Location source** parameter, you choose `Specify via dialog`. Tunable.

Positive values move the image down and to the right; negative values move the image up and to the left. If the first element is greater than the number of rows in the **Image1** matrix, the value is clipped to the total number of rows. If the second element is greater than the number of columns in the input **Image1** matrix, the value is clipped to the total number of columns.

The **Data Types** pane of the Compositing dialog box appears as follows. These parameters apply only when the **Operation** parameter is set to **Blend**.

Function Block Parameters: Compositing

**Compositing**

Combines the pixel values of two images, overlays one image over another, or highlights selected pixels.

If, for the Operation parameter, you choose Blend, you can use the Opacity factor(s) parameter to define the amount by which the block scales the pixel values of Image2 before combining them with the pixel values of Image1. This parameter can be a scalar value that the block uses for all pixels or a matrix of values that defines the factor for each pixel.

If, for the Operation parameter, you choose Binary mask, you can use the Masking factor(s) parameter to define which pixels of Image1 are overwritten by the pixel values of Image2. You can enter 0 or 1, which the block uses for all pixels, or a matrix of 0's and 1's that defines the factor for each pixel.

If, for the Operation parameter, you choose Highlight selected pixel, the block uses the binary image input at the Mask port to determine which pixels are set to the maximum value supported by their data type.

**Main**   **Data Types**

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

Fixed-point operational parameters

Rounding mode: **Floor**   Overflow mode: **Wrap**

Fixed-point data types

	Data Type	Signed	Word length	Fraction length
Opacity factor	Same word length as input			
Product output	Binary point scaling	Yes	32	10
Accumulator	Same as product output			
Output	Same as first input			

Lock data type settings against changes by the fixed-point tools

OK   Cancel   Help   Apply

### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Opacity factor

Choose how to specify the word length and fraction length of the opacity factor:

- When you select **Same word length as input**, these characteristics match those of the input to the block.
- When you select **Specify word length**, enter the word length of the opacity factor.
- When you select **Binary point scaling**, you can enter the word length of the opacity factor, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, of the opacity factor. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Product output

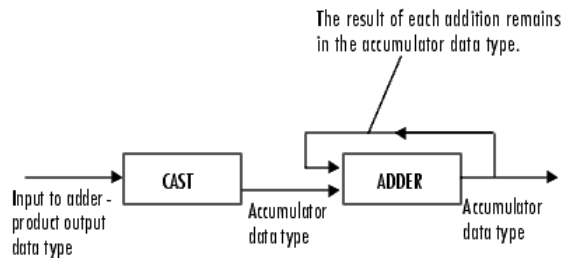


As the previous figure shows, the block places the output of the multiplier into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same as first input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.



## Accumulator



As the previous figure shows, the block takes inputs to the accumulator and casts them to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

## Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox software is 0.

## Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## Supported Data Types

Port	Input/Output	Supported Data Types	Complex Values Supported
Image 1	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Image 2	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes	Same as Image 1 port	No
Factor	Scalar or matrix of opacity or masking factor	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Mask	Binary image that specifies which pixels to highlight	<p>Same as Factor port</p> <p>When the <b>Operation</b> parameter is set to <b>Highlight selected pixel</b>, the input to the Mask port must be a Boolean data type.</p>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
Location	Two-element vector $[x\ y]$ , that specifies the position of the upper-left corner of the image input at port I2	<ul style="list-style-type: none"> <li>• Double-precision floating point. (Only supported if the input to the Image 1 and Image 2 ports is a floating-point data type.)</li> <li>• Single-precision floating point. (Only supported if the input to the Image 1 and Image 2 ports is a floating-point data type.)</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Output	Vector or matrix of intensity or color values	Same as Image 1 port	No

## See Also

Insert Text	Computer Vision System Toolbox
Draw Markers	Computer Vision System Toolbox
Draw Shapes	Computer Vision System Toolbox

**Introduced before R2006a**

## Compositing (To Be Removed)

Combine pixel values of two images, overlay one image over another, or highlight selected pixels

### Library

Text & Graphics

`viptextngfix`



### Description

Compositing

---

**Note:** This Compositing block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated `Compositing` block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

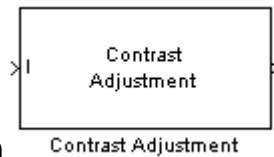
# Contrast Adjustment

Adjust image contrast by linearly scaling pixel values

## Library

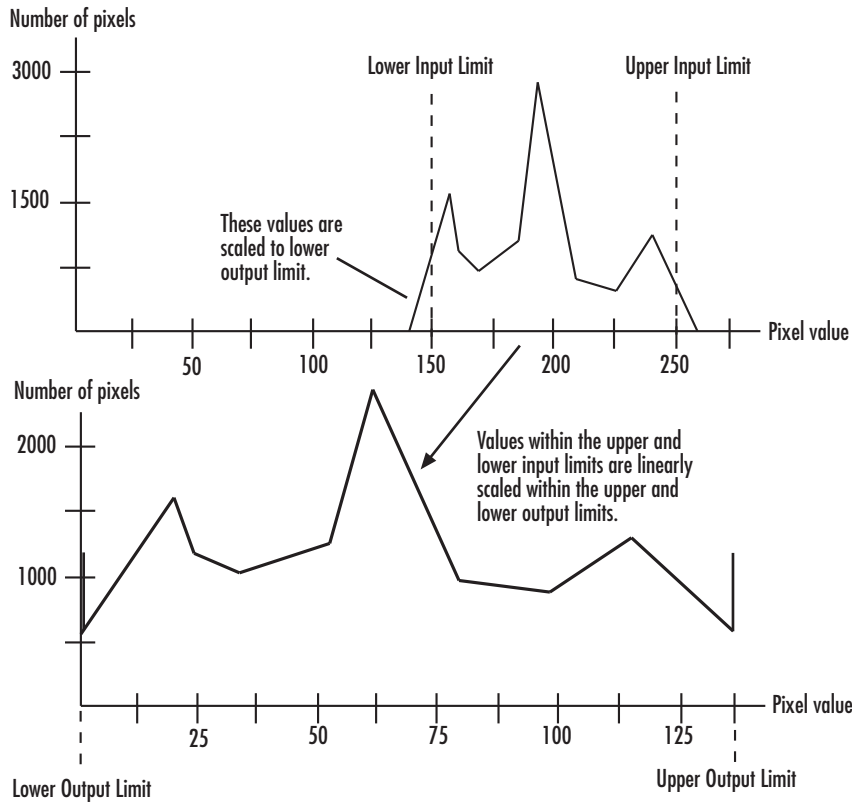
Analysis & Enhancement

visionanalysis



## Description

The Contrast Adjustment block adjusts the contrast of an image by linearly scaling the pixel values between upper and lower limits. Pixel values that are above or below this range are saturated to the upper or lower limit value, respectively.



Mathematically, the contrast adjustment operation is described by the following equation, where the input limits are  $[low\_in\ high\_in]$  and the output limits are  $[low\_out\ high\_out]$ :

$$Output = \left\{ \begin{array}{l} low\_out, \quad Input \leq low\_in \\ low\_out + (Input - low\_in) \frac{high\_out - low\_out}{high\_in - low\_in}, \quad low\_in < Input < high\_in \\ high\_out, \quad Input \geq high\_in \end{array} \right\}$$

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Output	Scalar, vector, or matrix of intensity values or a scalar, vector, or matrix that represents one plane of the RGB video stream	Same as I port	No

## Specifying upper and lower limits

Use the **Adjust pixel values from** and **Adjust pixel values to** parameters to specify the upper and lower input and output limits. All options are described below.

### Input limits

Use the **Adjust pixel values from** parameter to specify the upper and lower input limits.

If you select **Full input data range [min max]**, uses the minimum input value as the lower input limit and the maximum input value as the upper input limit.

If you select **User-defined**, the **Range [low high]** parameter associated with this option appears. Enter a two-element vector of scalar values, where the first element corresponds to the lower input limit and the second element corresponds to the upper input limit.

If you select **Range determined by saturating outlier pixels**, the **Percentage of pixels to saturate [low high] (in %)**, **Specify number of histogram bins (used to calculate the range when outliers are eliminated)**, and **Number of histogram bins** parameters appear on the block. The block uses these parameter values to calculate the input limits in this three-step process:

- 1 Find the minimum and maximum input values,  $[min\_in\ max\_in]$ .
- 2 Scale the pixel values from  $[min\_in\ max\_in]$  to  $[0\ num\_bins-1]$ , where  $num\_bins$  is the scalar value you specify in the **Number of histogram bins** parameter. This

parameter always displays the value used by the block. Then the block calculates the histogram of the scaled input. For additional information about histograms, see the 2D-Histogram block reference page.

- Find the lower input limit such that the percentage of pixels with values smaller than the lower limit is at most the value of the first element of the **Percentage of pixels to saturate [low high] (in %)** parameter. Similarly, find the upper input limit such that the percentage of pixels with values greater than the upper limit is at least the value of the second element of the parameter.

### Output limits

Use the **Adjust pixel values to** parameter to specify the upper and lower output limits. If you select **Full data type range**, the block uses the minimum value of the input data type as the lower output limit and the maximum value of the input data type as the upper out

If you select **User-defined range**, the **Range [low high]** parameter appears on the block. Enter a two-element vector of scalar values, where the first element corresponds to the lower output limit and the second element corresponds to the upper output limit.

### For INF, -INF and NAN Input Values

If any input pixel value is either INF or -INF, the Contrast Adjustment block will change the pixel value according to how the parameters are set. The following table shows how the block handles these pixel values.

If Adjust pixel values from parameter is set to...	Contrast Adjustment block will:
Full data range [min,max]	Set the entire output image to the lower limit of the <b>Adjust pixel values to</b> parameter setting.
Range determined by saturating outlier pixels	
User defined range	Lower and higher limits of the <b>Adjust pixel values to</b> parameter set to -INF and INF , respectively.

If any input pixel has a NAN value, the block maps the pixels with valid numerical values according to the user-specified method. It maps the NAN pixels to the lower limit of the **Adjust pixels values to** parameter.

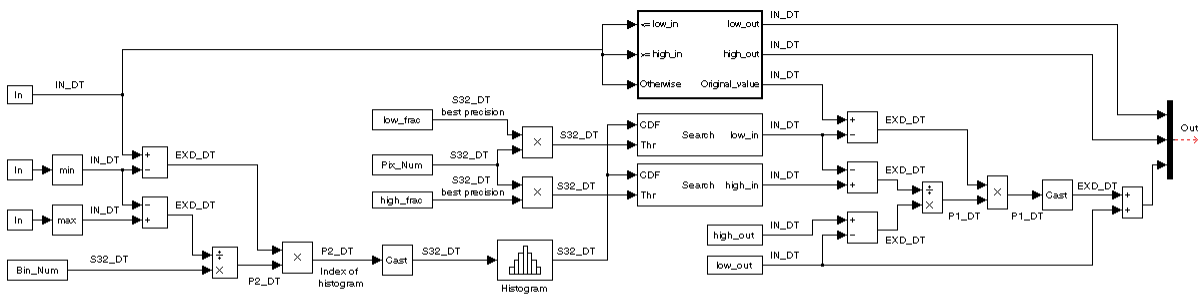


## Examples

See “Adjust the Contrast of Intensity Images” in the *Computer Vision System Toolbox User's Guide*.

## Fixed-Point Data Types

The following diagram shows the data types used in the Contrast Adjustment block for fixed-point signals:

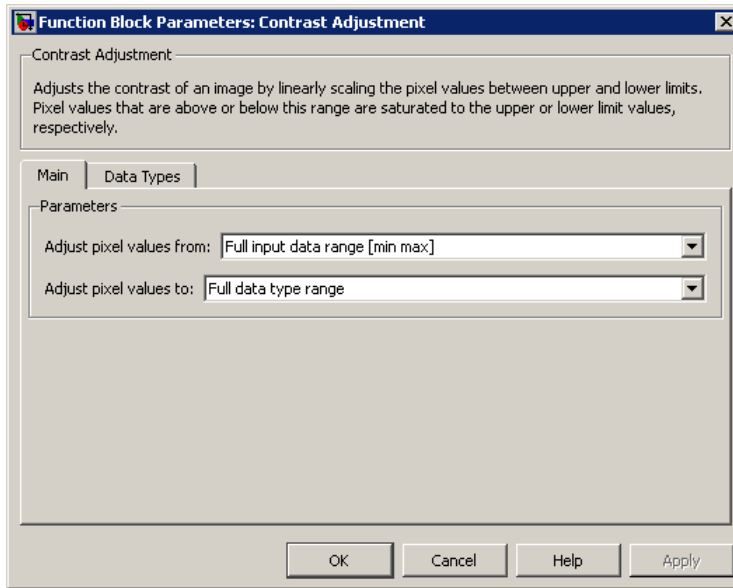


FIXED-POINT TAB NAMES:  
 =====  
 IN\_DT: Datatype of the input image  
 EXD\_DT: IN\_DT with extra bit for sign  
 P1\_DT: Datatype of product 1  
 P2\_DT: Datatype of product 2

SUB-SYSTEMS:  
 =====  
 Search:  
 Calculate the pixel value that corresponds  
 to Thr in the cumulative histogram (CDF).

## Dialog Box

The Contrast Adjustment dialog box appears as shown in the following figure.



## Adjust pixel values from

Specify how to enter the upper and lower input limits. Your choices are **Full input data range [min max]**, **User-defined**, and **Range determined by saturating outlier pixels**.

## Range [low high]

Enter a two-element vector of scalar values. The first element corresponds to the lower input limit, and the second element corresponds to the upper input limit. This parameter is visible if, for the **Adjust pixel values from** parameter, you select **User-defined**.

## Percentage of pixels to saturate [low high] (in %)

Enter a two-element vector. The block calculates the lower input limit such that the percentage of pixels with values smaller than the lower limit is at most the value of the first element. It calculates the upper input limit similarly. This parameter is visible if, for the **Adjust pixel values from** parameter, you select **Range determined by saturating outlier pixels**.

**Specify number of histogram bins (used to calculate the range when outliers are eliminated)**

Select this check box to change the number of histogram bins. This parameter is editable if, for the **Adjust pixel values from** parameter, you select **Range determined by saturating outlier pixels**.

**Number of histogram bins**

Enter the number of histogram bins to use to calculate the scaled input values. This parameter is available if you select the **Specify number of histogram bins (used to calculate the range when outliers are eliminated)** check box.

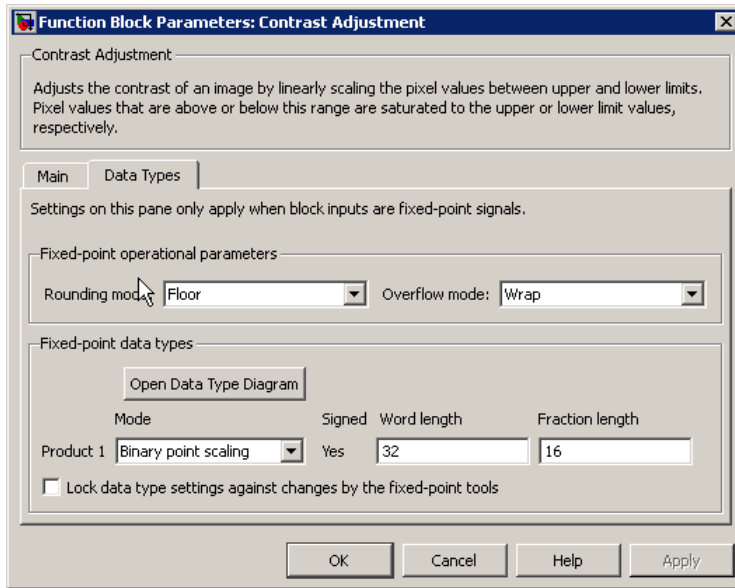
**Adjust pixel values to**

Specify the upper and lower output limits. If you select **Full data type range**, the block uses the minimum value of the input data type as the lower output limit and the maximum value of the input data type as the upper output limit. If you select **User-defined range**, the **Range [low high]** parameter appears on the block.

**Range [low high]**

Enter a two-element vector of scalar values. The first element corresponds to the lower output limit and the second element corresponds to the upper output limit. This parameter is visible if, for the **Adjust pixel values to** parameter, you select **User-defined range**

The **Data Types** pane of the Contrast Adjustment dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Product 1

The product output type when the block calculates the ratio between the input data range and the number of histogram bins.



As shown in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths:

When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.

When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Product 2

The product output type when the block calculates the bin location of each input value.



As shown in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths:

When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.

When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

This parameter is visible if, for the **Adjust pixel values from** parameter, you select **Range determined by saturating outlier pixels**.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

2D-Histogram	Computer Vision System Toolbox software
Histogram Equalization	Computer Vision System Toolbox software

Introduced in R2006b

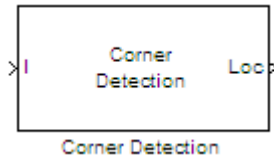
# Corner Detection

Calculate corner metric matrix and find corners in images

## Library

Analysis & Enhancement

visionanalysis



## Description

The Corner Detection block finds corners in an image using the Harris corner detection (by Harris & Stephens), minimum eigenvalue (by Shi & Tomasi), or local intensity comparison (Features from Accelerated Segment Test, FAST by Rosten & Drummond) method. The block finds the corners in the image based on the pixels that have the largest corner metric values.

For the most accurate results, use the “Minimum Eigenvalue Method” on page 1-259. For the fastest computation, use the “Local Intensity Comparison” on page 1-260. For the trade-off between accuracy and computation, use the “Harris Corner Detection Method” on page 1-260.

## Port Description

Port	Description	Supported Data Types
I	Matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>

Port	Description	Supported Data Types
Loc	$M$ -by-2 matrix of [x y] coordinates, that represents the locations of the corners. $M$ represents the number of corners and is less than or equal to the <b>Maximum number of corners</b> parameter	32-bit unsigned integer
Count	Scalar value that represents the number of detected corners	32-bit unsigned integer
Metric	Matrix of corner metric values that is the same size as the input image	Same as I port

## Minimum Eigenvalue Method

This method is more computationally expensive than the Harris corner detection algorithm because it directly calculates the eigenvalues of the sum of the squared difference matrix,  $M$ .

The sum of the squared difference matrix,  $M$ , is defined as follows:

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

The previous equation is based on the following values:

$$A = (I_x)^2 \otimes w$$

$$B = (I_y)^2 \otimes w$$

$$C = (I_x I_y)^2 \otimes w$$

where  $I_x$  and  $I_y$  are the gradients of the input image,  $I$ , in the  $x$  and  $y$  direction, respectively. The  $\otimes$  symbol denotes a convolution operation.

Use the **Coefficients for separable smoothing filter** parameter to define a vector of filter coefficients. The block multiplies this vector of coefficients by its transpose to create a matrix of filter coefficients,  $w$ .

The block calculates the smaller eigenvalue of the sum of the squared difference matrix. This minimum eigenvalue corresponds to the corner metric matrix.

## Harris Corner Detection Method

The Harris corner detection method avoids the explicit computation of the eigenvalues of the sum of squared differences matrix by solving for the following corner metric matrix,  $R$ :

$$R = AB - C^2 - k(A + B)^2$$

$A$ ,  $B$ ,  $C$  are defined in the previous section, “Minimum Eigenvalue Method” on page 1-259.

The variable  $k$  corresponds to the sensitivity factor. You can specify its value using the **Sensitivity factor ( $0 < k < 0.25$ )** parameter. The smaller the value of  $k$ , the more likely it is that the algorithm can detect sharp corners.

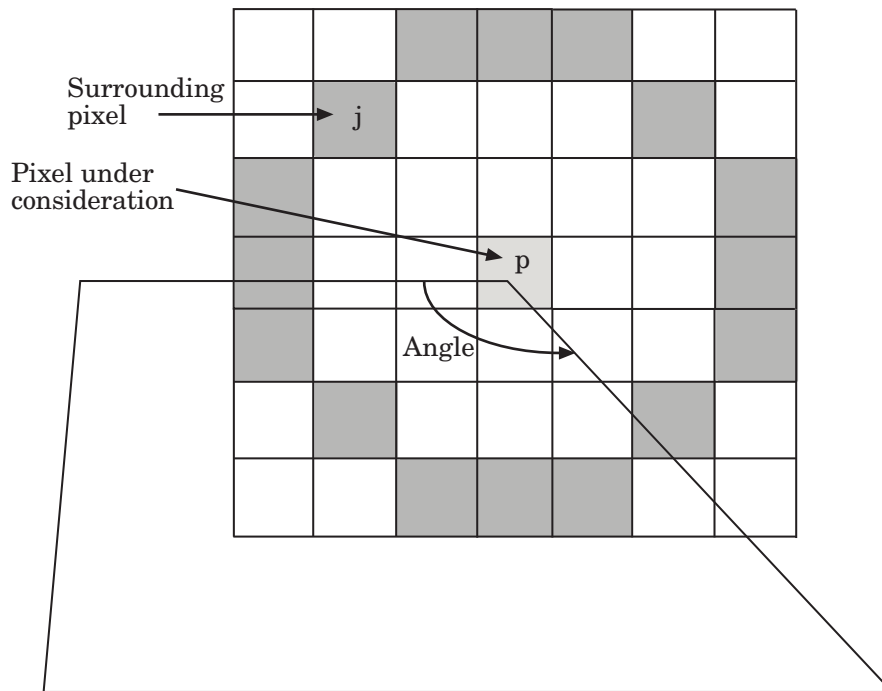
Use the **Coefficients for separable smoothing filter** parameter to define a vector of filter coefficients. The block multiplies this vector of coefficients by its transpose to create a matrix of filter coefficients,  $w$ .

## Local Intensity Comparison

This method determines that a pixel is a possible corner if it has either,  $N$  contiguous valid bright surrounding pixels, or  $N$  contiguous dark surrounding pixels. Specifying the value of  $N$  is discussed later in this section. The next section explains how the block finds these surrounding pixels.

Suppose that  $p$  is the pixel under consideration and  $j$  is one of the pixels surrounding  $p$ . The locations of the other surrounding pixels are denoted by the shaded areas in the following figure.





$I_p$  and  $I_j$  are the intensities of pixels  $p$  and  $j$ , respectively. Pixel  $j$  is a valid bright surrounding pixel if  $I_j - I_p \geq T$ . Similarly, pixel  $j$  is a valid dark surrounding pixel if  $I_p - I_j \geq T$ . In these equations,  $T$  is the value you specified for the **Intensity comparison threshold** parameter.

The block repeats this process to determine whether the block has  $N$  contiguous valid surrounding pixels. The value of  $N$  is related to the value you specify for the **Maximum angle to be considered a corner (in degrees)**, as shown in the following table.

Number of Valid Surrounding Pixels, $N$	Angle (degrees)
15	22.5
14	45
13	67.5
12	90

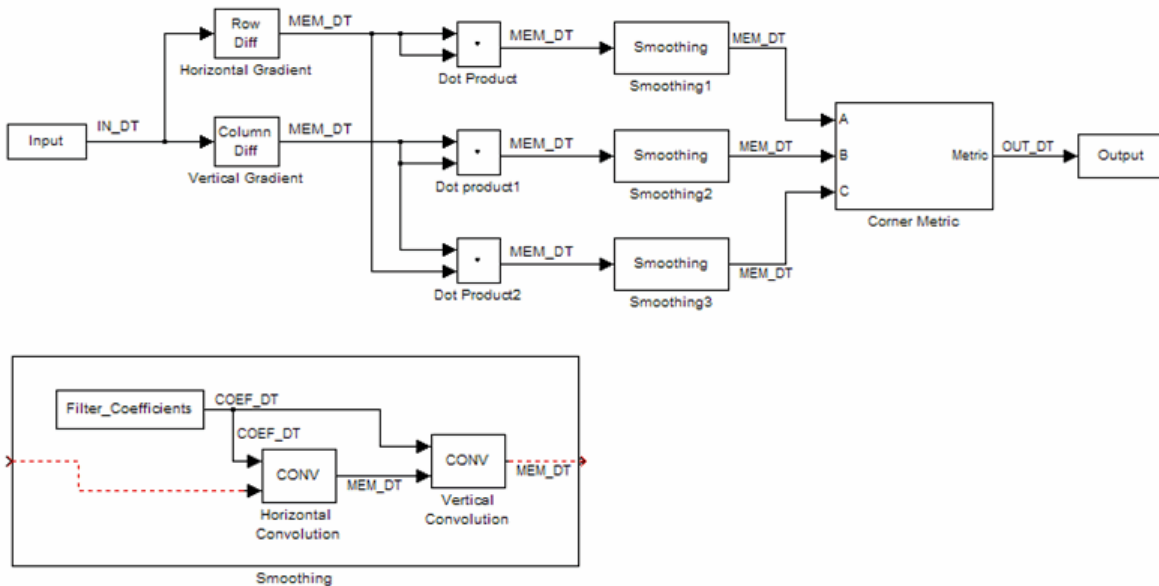
Number of Valid Surrounding Pixels, N	Angle (degrees)
11	112.5
10	135
9	157.5

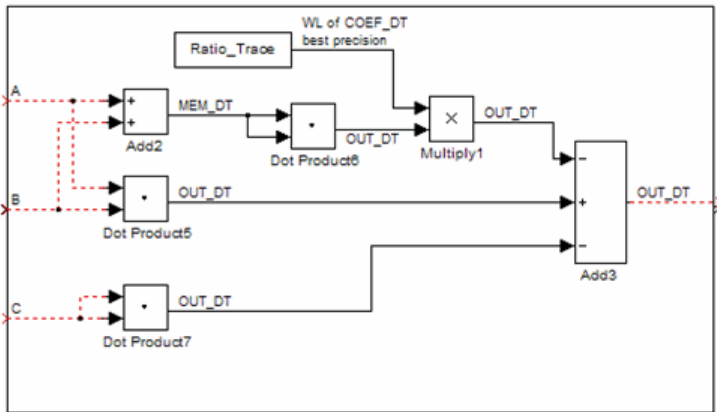
After the block determines that a pixel is a possible corner, it computes its corner metric using the following equation:

$$R = \max \left( \sum_{j:I_j \geq I_p + T} |I_p - I_j| - T, \sum_{j:I_j \leq I_p - T} |I_p - I_j| - T, \right)$$

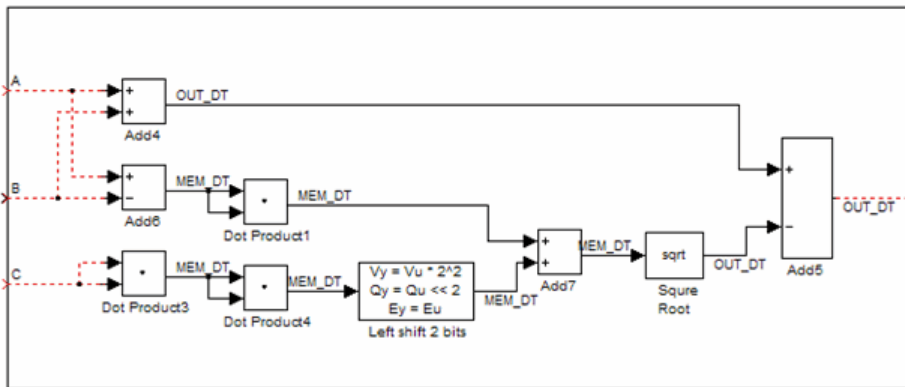
### Fixed-Point Data Types

The following diagram shows the data types used in the Corner Detection block for fixed-point signals. These diagrams apply to the Harris corner detection and minimum eigenvalue methods only.





Corner Metric by Harris Algorithm



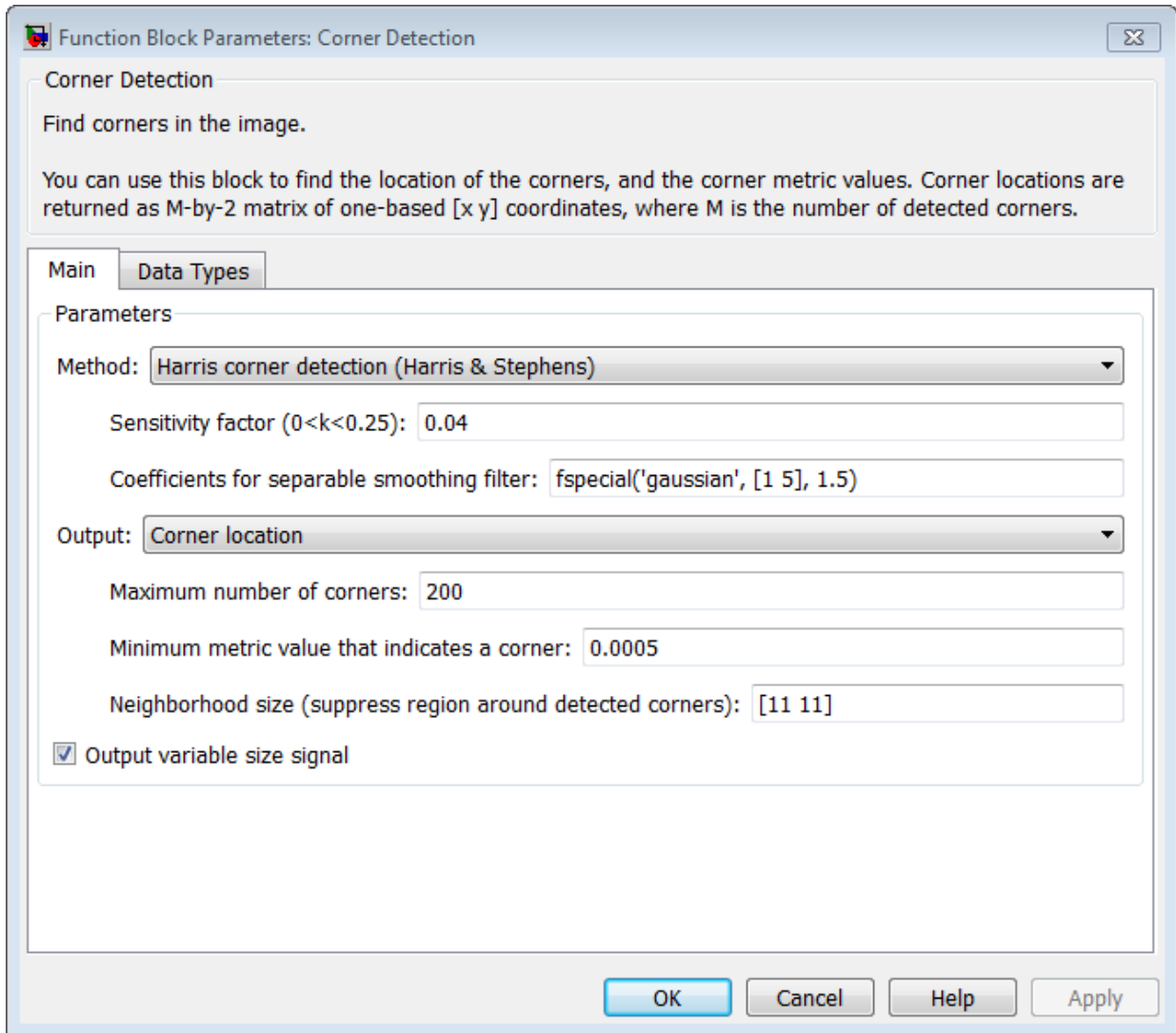
Corner Metric by Minimum Eigenvalue Algorithm

The following table summarizes the variables used in the previous diagrams.

Variable Name	Definition
IN_DT	Input data type
MEM_DT	Memory data type
OUT_DT	Metric output data type
COEF_DT	Coefficients data type

## Dialog Box

The Corner Detection dialog box appears as shown in the following figure.



### Method

Specify the method to use to find the corner values. Your choices are **Harris corner detection** (Harris & Stephens), **Minimum eigenvalue** (Shi & Tomasi), and **Local intensity comparison** (Rosten & Drummond).

#### **Sensitivity factor ( $0 < k < 0.25$ )**

Specify the sensitivity factor,  $k$ . The smaller the value of  $k$  the more likely the algorithm is to detect sharp corners. This parameter is visible if you set the **Method** parameter to **Harris corner detection** (Harris & Stephens). This parameter is tunable.

#### **Coefficients for separable smoothing filter**

Specify a vector of filter coefficients for the smoothing filter. This parameter is visible if you set the **Method** parameter to **Harris corner detection** (Harris & Stephens) or **Minimum eigenvalue** (Shi & Tomasi).

#### **Intensity comparison threshold**

Specify the threshold value used to find valid surrounding pixels. This parameter is visible if you set the **Method** parameter to **Local intensity comparison** (Rosten & Drummond). This parameter is tunable.

#### **Maximum angle to be considered a corner (in degrees)**

Specify the maximum corner angle. This parameter is visible if you set the **Method** parameter to **Local intensity comparison** (Rosten & Drummond). This parameter is tunable for Simulation only.

#### **Output**

Specify the block output. Your choices are **Corner location**, **Corner location and metric matrix**, and **Metric matrix**. The block outputs the corner locations in an  $M$ -by-2 matrix of [x y] coordinates, where  $M$  represents the number of corners. The block outputs the corner metric value in a matrix, the same size as the input image.

When you set the this parameter to **Corner location** or **Corner location and metric matrix**, the **Maximum number of corners**, **Minimum metric value that indicates a corner**, and **Neighborhood size (suppress region around detected corners)** parameters appear on the block.

To determine the final corner values, the block follows this process:

- 1 Find the pixel with the largest corner metric value.
- 2 Verify that the metric value is greater than or equal to the value you specified for the **Minimum metric value that indicates a corner** parameter.

- 3 Suppress the region around the corner value by the size defined in the **Neighborhood size (suppress region around detected corners)** parameter.

The block repeats this process until it finds all the corners in the image or it finds the number of corners you specified in the **Maximum number of corners** parameter.

The corner metric values computed by the **Minimum eigenvalue** and **Local intensity comparison** methods are always non-negative. The corner metric values computed by the **Harris corner detection** method can be negative.

**Maximum number of corners**

Enter the maximum number of corners you want the block to find. This parameter is visible if you set the **Output** parameter to **Corner location** or **Corner location and metric matrix**.

**Minimum metric value that indicates a corner**

Specify the minimum corner metric value. This parameter is visible if you set the **Output** parameter to **Corner location** or **Corner location and metric matrix**. This parameter is tunable.

**Neighborhood size (suppress region around detected corners)**

Specify the size of the neighborhood around the corner metric value over which the block zeros out the values. Enter a two-element vector of positive odd integers, [r c]. Here, r is the number of rows in the neighborhood and c is the number of columns. This parameter is visible if you set the **Output** parameter to **Corner location** or **Corner location and metric matrix**.

The **Data Types** pane of the Corner Detection dialog box appears as shown in the following figure.

Function Block Parameters: Corner Detection

**Corner Detection**  
Find corners in the image.

You can use this block to find the location of the corners, and the corner metric values. Corner locations are returned as M-by-2 matrix of one-based [x y] coordinates, where M is the number of detected corners.

Main | **Data Types**

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

Fixed-point operational parameters

Rounding mode: **Floor**    Overflow mode: **Wrap**

Fixed-point data types

[Open Data Type Diagram](#)

	Data Type	Signed	Word length	Fraction length
Coefficients	<b>Specify word length</b>	Yes	16	16
Product output	<b>Binary point scaling</b>	Yes	32	0
Accumulator	<b>Binary point scaling</b>	Yes	32	0
Memory	<b>Binary point scaling</b>	Yes	32	0
Metric output	<b>Same as accumulator</b>			

Lock data type settings against changes by the fixed-point tools

OK    Cancel    Help    Apply

### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

## Coefficients

Choose how to specify the word length and the fraction length of the coefficients:

- When you select **Same word length as input**, the word length of the coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select **Specify word length**, you can enter the word length of the coefficients, in bits. The block automatically sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the coefficients, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the coefficients. The bias of all signals in the Computer Vision System Toolbox software is 0.

## Product output

As shown in the following figure, the output of the multiplier is placed into the product output data type and scaling.



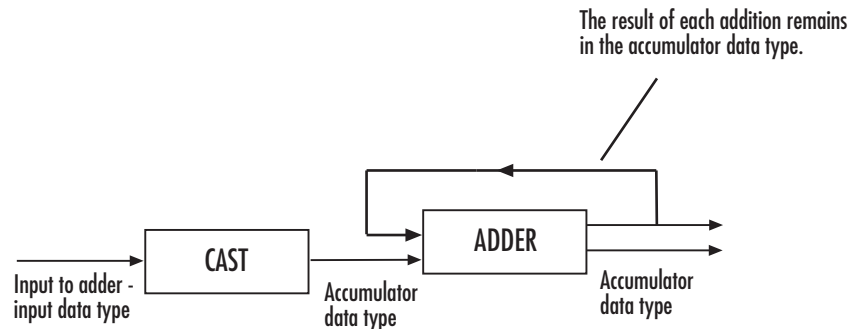
Use this parameter to specify how to designate the product output word and fraction lengths.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox software is 0.

## Accumulator



As shown in the following figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it.



Use this parameter to specify how to designate this accumulator word and fraction lengths:

- When you select **Same as input**, these characteristics match those of the input.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox software is 0.

### Memory

Choose how to specify the memory word length and fraction length:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of 0.

### Metric output

Choose how to specify the metric output word length and fraction length:

- When you select **Same as accumulator**, these characteristics match those of the accumulator.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of 0.

**Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

[1] C. Harris and M. Stephens. “A Combined Corner and Edge Detector.” *Proceedings of the 4th Alvey Vision Conference*. August 1988, pp. 147-151.

[2] J. Shi and C. Tomasi. “Good Features to Track.” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. June 1994, pp. 593–600.

[3] E. Rosten and T. Drummond. “Fusing Points and Lines for High Performance Tracking.” *Proceedings of the IEEE International Conference on Computer Vision* Vol. 2 (October 2005): pp. 1508–1511.

## See Also

Find Local Maxima	Computer Vision System Toolbox software
Estimate Geometric Transformation	Computer Vision System Toolbox software
matchFeatures	Computer Vision System Toolbox software
extractFeatures	Computer Vision System Toolbox software
detectSURFFeatures	Computer Vision System Toolbox software

**Introduced in R2007b**

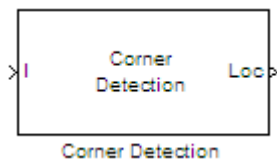
## Corner Detection (To Be Removed)

Calculate corner metric matrix and find corners in images

### Library

Analysis & Enhancement

vipanalysis



### Description

---

**Note:** This Corner Detection block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Corner Detection block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

# Deinterlacing

Remove motion artifacts by deinterlacing input video signal

## Library

Analysis & Enhancement

visionanalysis



## Description

The Deinterlacing block takes the input signal, which is the combination of the top and bottom fields of the interlaced video, and converts it into deinterlaced video using line repetition, linear interpolation, or vertical temporal median filtering.

---

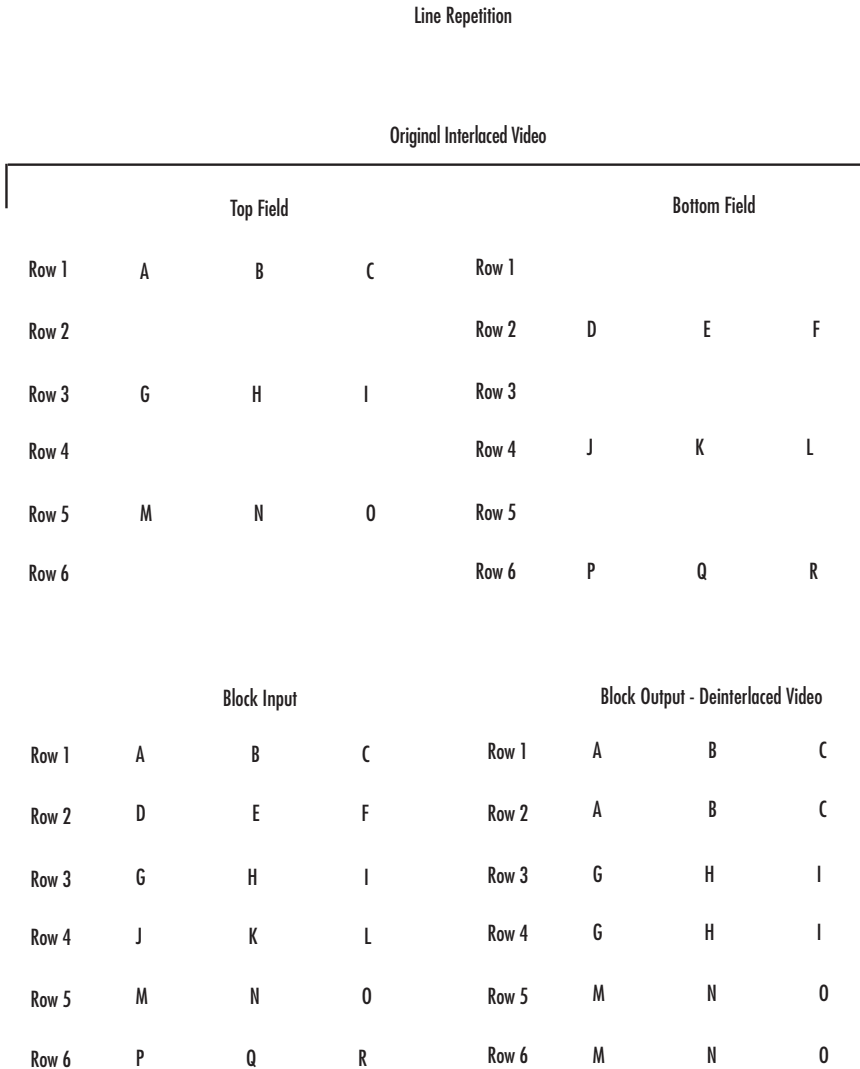
**Note:** This block supports intensity and color images on its ports.

---

Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Combination of top and bottom fields of interlaced video	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Output	Frames of deinterlaced video	Same as Input port	No

Use the **Deinterlacing method** parameter to specify how the block deinterlaces the video.

The following figure illustrates the block's behavior if you select **Line repetition**.



The following figure illustrates the block's behavior if you select **Linear interpolation**.

## Linear Interpolation

## Original Interlaced Video

Top Field			Bottom Field			
Row 1	A	B	C	Row 1		
Row 2				Row 2	D	E
Row 3	G	H	I	Row 3		
Row 4				Row 4	J	K
Row 5	M	N	O	Row 5		
Row 6				Row 6	P	Q

Block Input			Block Output - Deinterlaced Video				
Row 1	A	B	C	Row 1	A	B	C
Row 2	D	E	F	Row 2	$(A+G)/2$	$(B+H)/2$	$(C+I)/2$
Row 3	G	H	I	Row 3	G	H	I
Row 4	J	K	L	Row 4	$(G+M)/2$	$(H+N)/2$	$(I+O)/2$
Row 5	M	N	O	Row 5	M	N	O
Row 6	P	Q	R	Row 6	M	N	O

The following figure illustrates the block's behavior if you select Vertical temporal median filtering.

Vertical Temporal Median Filtering

Original Interlaced Video

Top Field			Bottom Field				
Row 1	A	B	C	Row 1			
Row 2				Row 2	D	E	F
Row 3	G	H	I	Row 3			
Row 4				Row 4	J	K	L
Row 5	M	N	O	Row 5			
Row 6				Row 6	P	Q	R

Block Input

Block Output - Deinterlaced Video

Row 1	A	B	C	Row 1	A	B	C
Row 2	D	E	F	Row 2	median([A,D,G])	median([B,E,H])	median([C,F,I])
Row 3	G	H	I	Row 3	G	H	I
Row 4	J	K	L	Row 4	median([G,J,M])	median([H,K,N])	median([L,O])
Row 5	M	N	O	Row 5	M	N	O
Row 6	P	Q	R	Row 6	M	N	O

## Row-Major Data Format

The MATLAB environment and the Computer Vision System Toolbox software use column-major data organization. However, the Deinterlacing block gives you the option to process data that is stored in row-major format. When you select the **Input image is transposed (data order is row major)** check box, the block assumes that the input buffer contains contiguous data elements from the first row first, then data elements



from the second row second, and so on through the last row. Use this functionality only when you meet all the following criteria:

- You are developing algorithms to run on an embedded target that uses the row-major format.
- You want to limit the additional processing required to take the transpose of signals at the interfaces of the row-major and column-major systems.

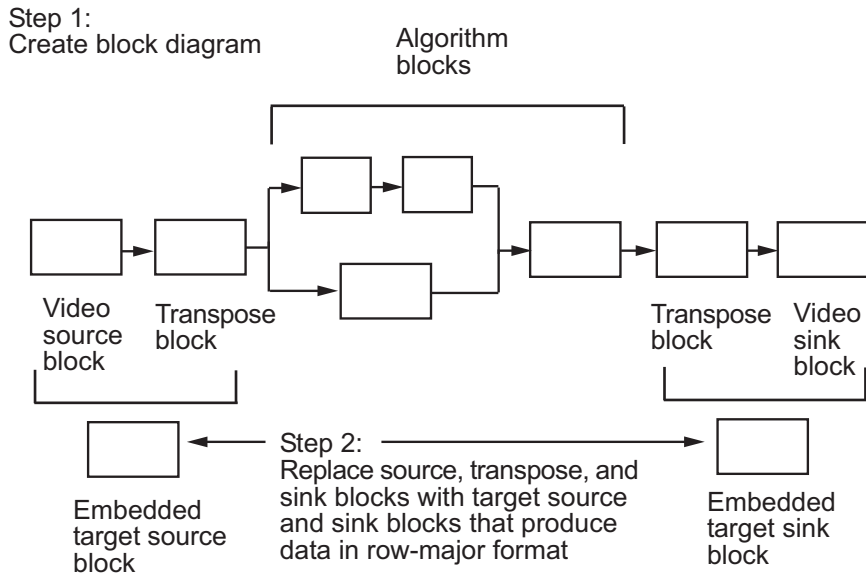
When you use the row-major functionality, you must consider the following issues:

- When you select this check box, the first two signal dimensions of the Deinterlacing block's input are swapped.
- All the Computer Vision System Toolbox blocks can be used to process data that is in the row-major format, but you need to know the image dimensions when you develop your algorithms.

For example, if you use the 2-D FIR Filter block, you need to verify that your filter coefficients are transposed. If you are using the Rotate block, you need to use negative rotation angles, etc.

- Only three blocks have the **Input image is transposed (data order is row major)** check box. They are the Chroma Resampling, Deinterlacing, and Insert Text blocks. You need to select this check box to enable row-major functionality in these blocks. All other blocks must be properly configured to process data in row-major format.

Use the following two-step workflow to develop algorithms in row-major format to run on an embedded target.



See the DM642 EVM Video ADC and DM642 EVM Video DAC reference pages.

## Example

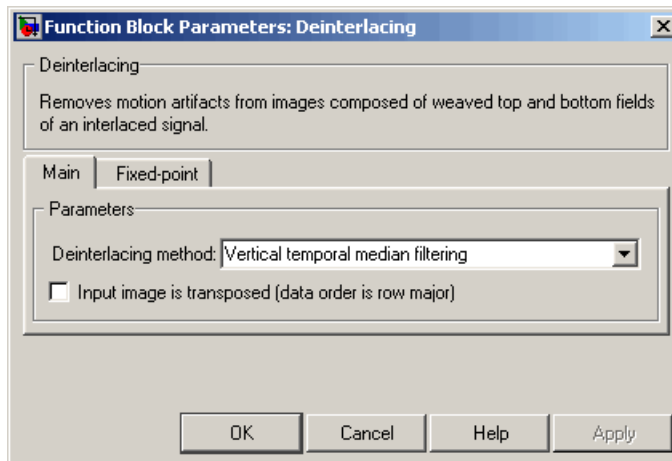
The following example shows you how to use the Deinterlacing block to remove motion artifacts from an image.

- 1 Open the example model by typing

```
ex_deinterlace
```

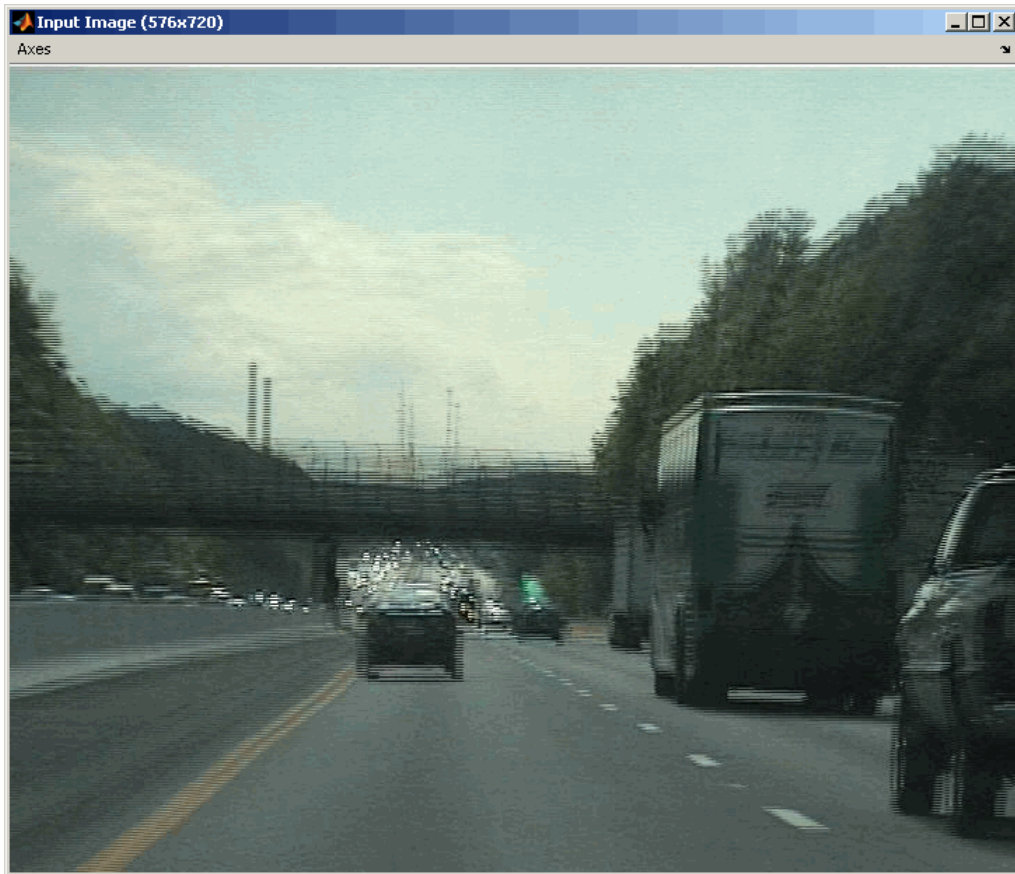
at the MATLAB command prompt.

- 2 Double-click the Deinterlacing block. The model uses this block to remove the motion artifacts from the input image. The **Deinterlacing method** parameter is set to `Vertical temporal median filtering`.

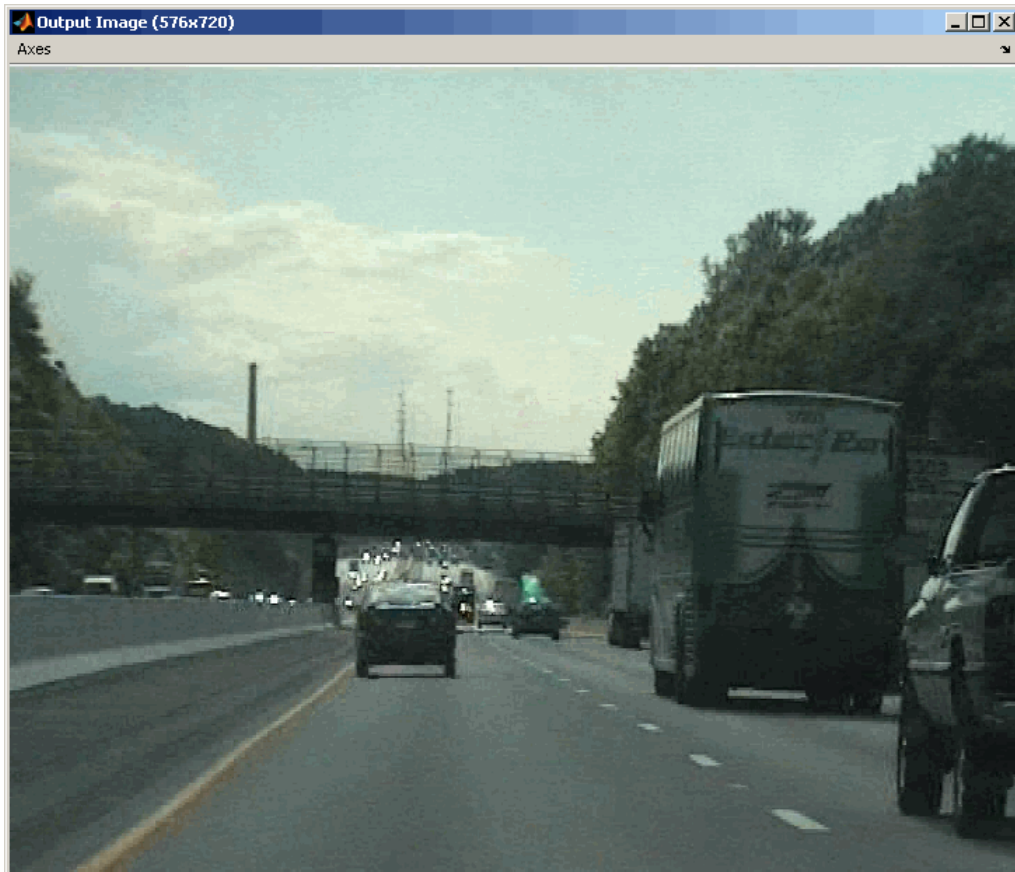


**3** Run the model.

The original image that contains the motion artifacts appears in the Input Image window.

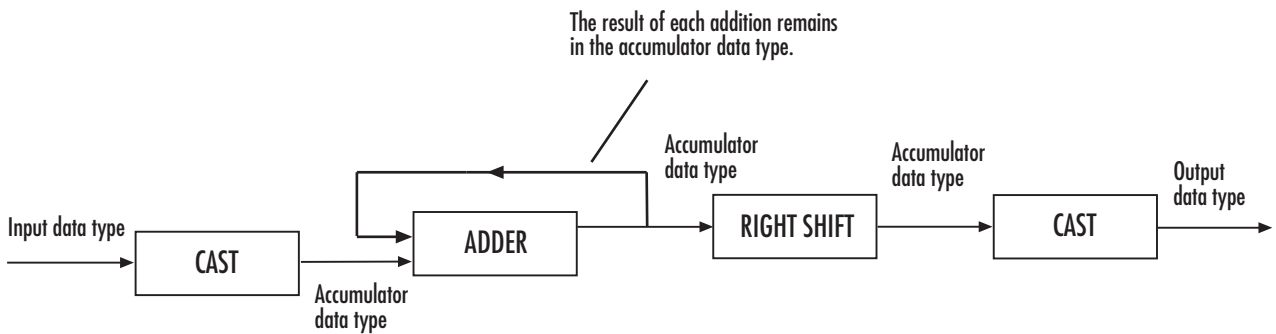


The clearer output image appears in the Output Image window.



## Fixed-Point Data Types

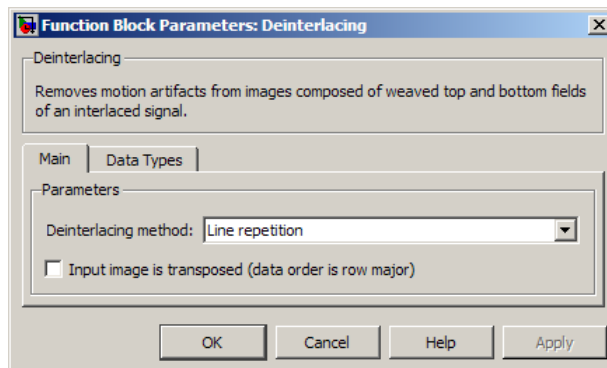
The following diagram shows the data types used in the Deinterlacing block for fixed-point signals.



You can set the product output, accumulator, and output data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Deinterlacing dialog box appears as shown in the following figure.



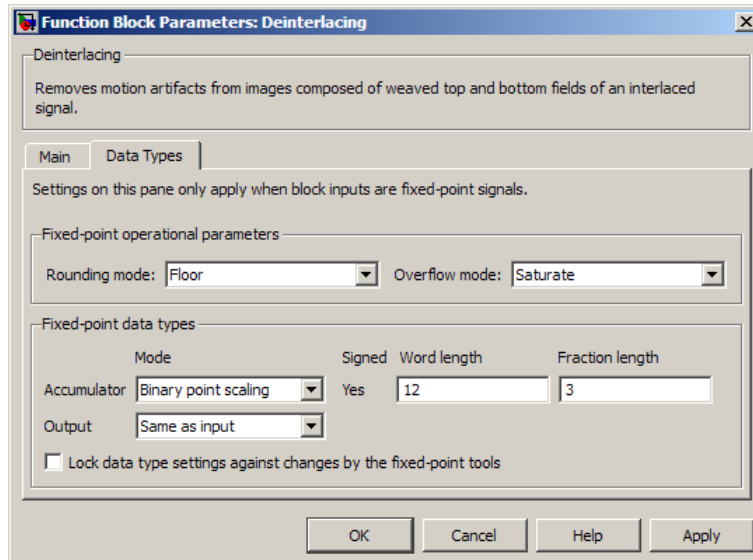
### Deinterlacing method

Specify how the block deinterlaces the video. Your choices are **Line repetition**, **Linear interpolation**, or **Vertical temporal median filtering**.

### Input image is transposed (data order is row major)

When you select this check box, the block assumes that the input buffer contains data elements from the first row first, then data elements from the second row second, and so on through the last row.

The **Data Types** pane of the Deinterlacing dialog box appears as shown in the following figure.



---

**Note:** The parameters on the **Data Types** pane are only available if, for the **Deinterlacing method**, you select **Linear interpolation**.

---

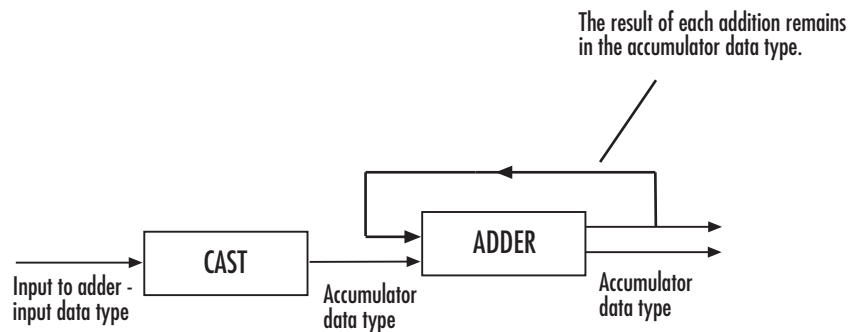
### **Rounding mode**

Select the rounding mode for fixed-point operations.

### **Overflow mode**

Select the overflow mode for fixed-point operations.

### **Accumulator**



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths:

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as input**, these characteristics match those of the input.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Output

Choose how to specify the output word length and fraction length:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of 0.

### Lock data type settings against change by the fixed-point tools



Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

**Introduced before R2006a**

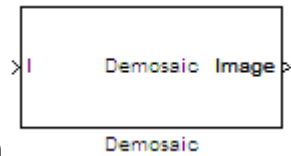
## Demosaic

Demosaic Bayer's format images

## Library

Conversions

visionconversions



## Description

The following figure illustrates a 4-by-4 image in Bayer's format with each pixel labeled R, G, or B.

B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R

The Demosaic block takes in images in Bayer's format and outputs RGB images. The block performs this operation using a gradient-corrected linear interpolation algorithm or a bilinear interpolation algorithm.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Matrix of intensity values <ul style="list-style-type: none"> <li>• If, for the <b>Interpolation algorithm</b> parameter, you select <b>Bilinear</b>, the number of rows and columns must be greater than or equal to 3.</li> <li>• If, for the <b>Interpolation algorithm</b> parameter, you select <b>Gradient-corrected linear</b>, the number of rows and columns must be greater than or equal to 5.</li> </ul>	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
R, G, B	Matrix that represents one plane of the input RGB video stream. Outputs from the R, G, or B ports have the same data type.	Same as I port	No
Image	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes.	Same as I port	No

Use the **Interpolation algorithm** parameter to specify the algorithm the block uses to calculate the missing color information. If you select **Bilinear**, the block spatially averages neighboring pixels to calculate the color information. If you select **Gradient-corrected linear**, the block uses a Weiner approach to minimize the mean-squared error in the interpolation. This method performs well on the edges of objects in the image. For more information, see [1].

Use the **Sensor alignment** parameter to specify the alignment of the input image. Select the sequence of R, G and B pixels that correspond to the 2-by-2 block of pixels in

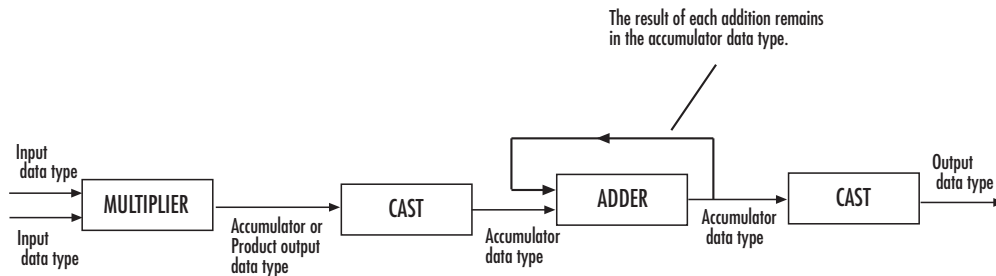
the top-left corner of the image. You specify the sequence in left-to-right, top-to-bottom order. For example, for the image at the beginning of this reference page, you would select BGGR.

Both methods use symmetric padding at the image boundaries. For more information, see the **Image Pad** block reference page.

Use the **Output image signal** parameter to specify how to output a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

## Fixed-Point Data Types

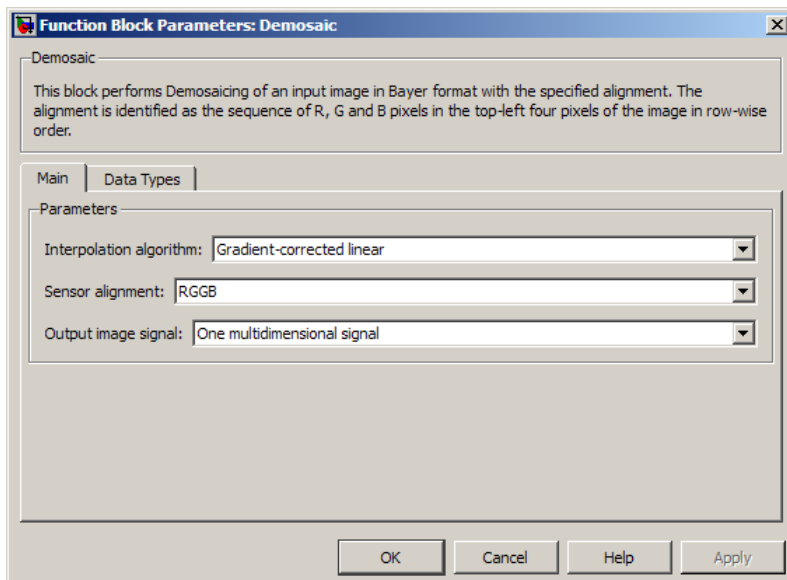
The following diagram shows the data types used in the Demosaic block for fixed-point signals.



You can set the product output and accumulator data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Demosaic dialog box appears as shown in the following figure.



### Interpolation algorithm

Specify the algorithm the block uses to calculate the missing color information. Your choices are **Bilinear** or **Gradient-corrected linear**.

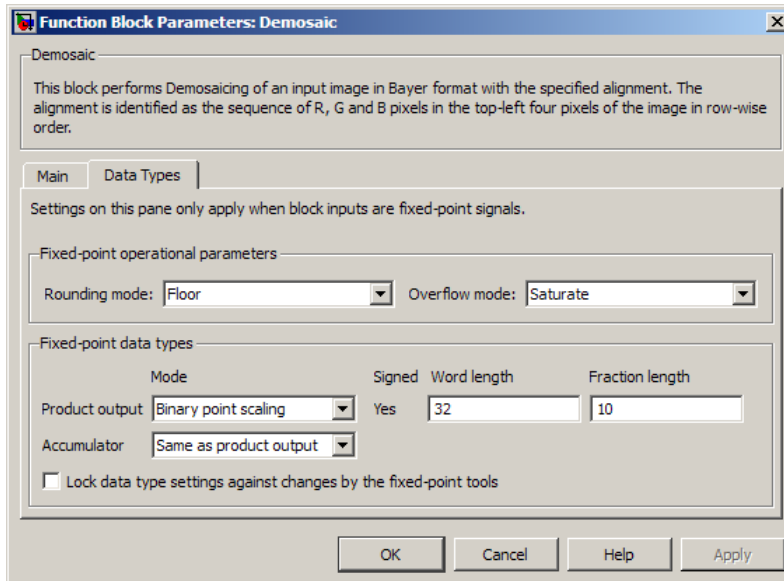
### Sensor alignment

Select the sequence of R, G and B pixels that correspond to the 2-by-2 block of pixels in the top left corner of the image. You specify the sequence in left-to-right, top-to-bottom order.

### Output image signal

Specify how to output a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

The **Data Types** pane of the Demosaic dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Product output



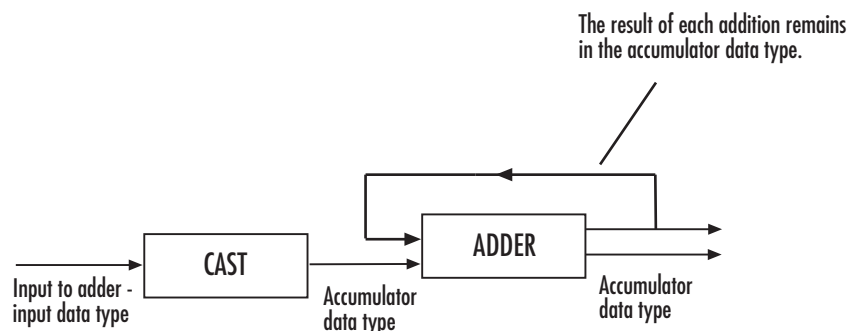
As depicted in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths:

When you select **Same as input**, these characteristics match those of the input to the block.

When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.

When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths:

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as input**, these characteristics match those of the input.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Malvar, Henrique S., Li-wei He, and Ross Cutler. “High-Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images.” *Microsoft Research*, May 2004. [http://research.microsoft.com/pubs/102068/Demosaicing\\_ICASSP04.pdf](http://research.microsoft.com/pubs/102068/Demosaicing_ICASSP04.pdf).
- [2] Gunturk, Bahadir K., John Glotzbach, Yucel Altunbasak, Ronald W. Schafer, and Russel M. Mersereau, “Demosaicking: Color Filter Array Interpolation,” *IEEE Signal Processing Magazine*, Vol. 22, Number 1, January 2005.

**Introduced in R2006b**



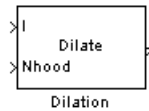
# Dilation

Find local maxima in binary or intensity image

## Library

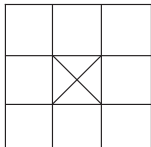
Morphological Operations

visionmorphops

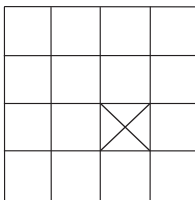


## Description

The Dilate block rotates the neighborhood or structuring element 180 degrees. Then it slides the neighborhood or structuring element over an image, finds the local maxima, and creates the output matrix from these maximum values. If the neighborhood or structuring element has a center element, the block places the maxima there, as illustrated in the following figure.



If the neighborhood or structuring element does not have an exact center, the block has a bias toward the lower-right corner, as a result of the rotation. The block places the maxima there, as illustrated in the following figure.



This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of ones and zeros that represents the neighborhood values	Boolean	No
Output	Vector or matrix of intensity values that represents the dilated image	Same as I port	No

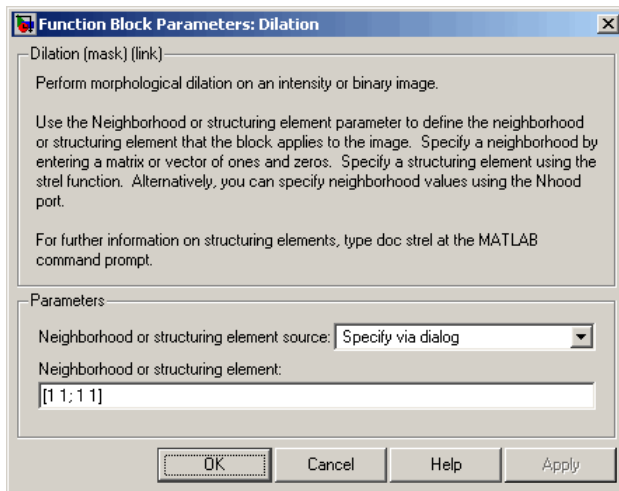
The output signal has the same data type as the input to the I port.

Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the Nhood port appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. You can only specify a structuring element using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the neighborhood or structuring element that the block applies to the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm. If you enter an array of STREL objects, the block applies each object to the entire matrix in turn.

## Dialog Box

The Dilation dialog box appears as shown in the following figure.



### Neighborhood or structuring element source

Specify how to enter your neighborhood or structuring element values. Select **Specify via dialog** to enter the values in the dialog box. Select **Input port** to use the Nhood port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

### Neighborhood or structuring element

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the **strel** function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select **Specify via dialog**.

## References

[1] Soille, Pierre. *Morphological Image Analysis. 2nd ed.* New York: Springer, 2003.

## See Also

Bottom-hat	Computer Vision System Toolbox software
Closing	Computer Vision System Toolbox software

Erosion	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
imdilate	Image Processing Toolbox software
strel	Image Processing Toolbox software

**Introduced before R2006a**

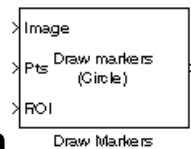
# Draw Markers

Draw markers by embedding predefined shapes on output image

## Library

Text & Graphics

visiontextngfix



## Description

The Draw Markers block can draw multiple circles, x-marks, plus signs, stars, or squares on images by overwriting pixel values. Overwriting the pixel values embeds the shapes.

This block uses Bresenham's circle drawing algorithm to draw circles and Bresenham's line drawing algorithm to draw all other markers.

## Port Description

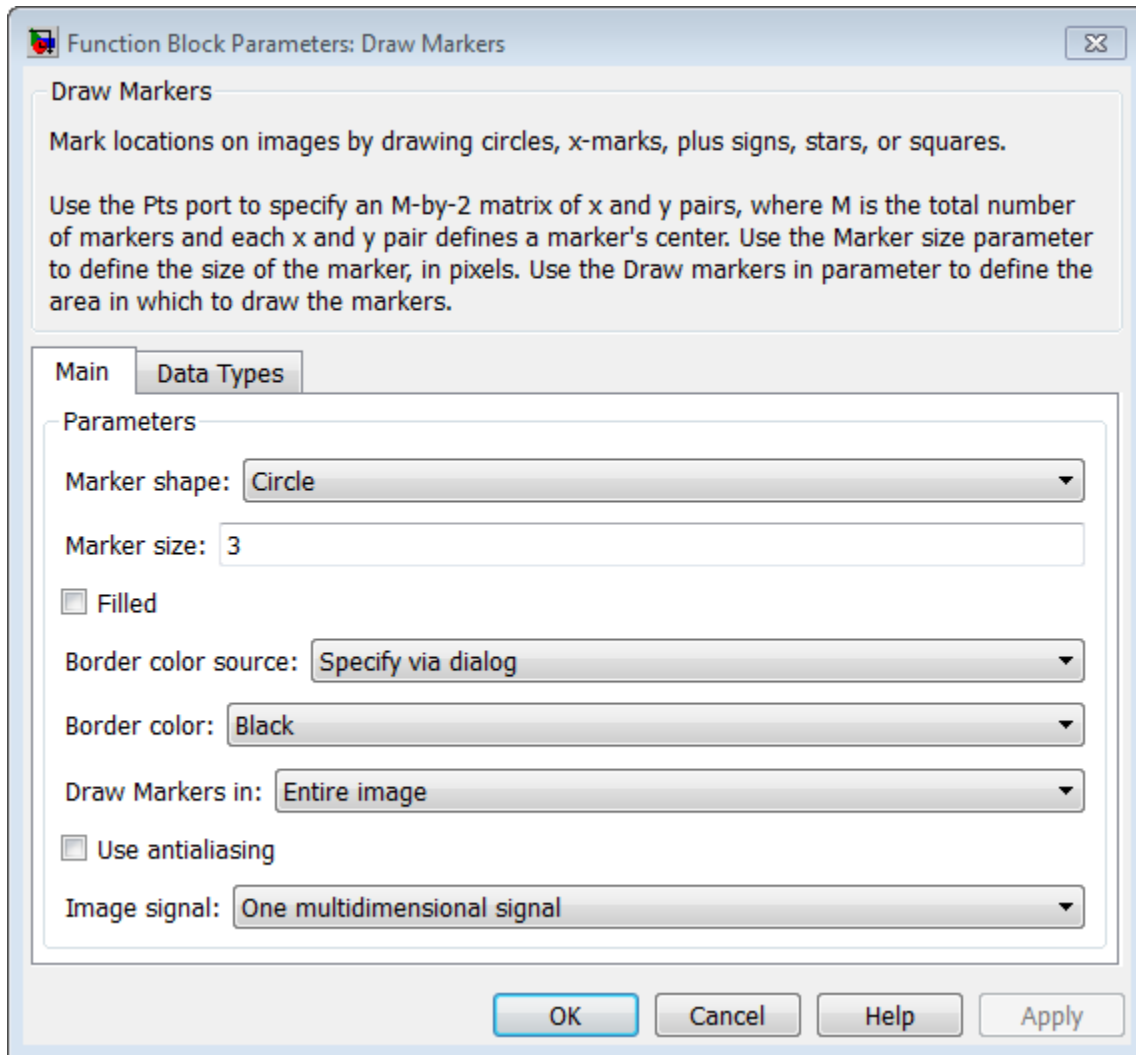
Port	Input/Output	Supported Data Types	Complex Values Supported
Image	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color values where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
R, G, B	Scalar, vector, or matrix that represents one plane of the input RGB video stream. Inputs to the R, G, and B ports must have the same dimensions and data type.	Same as Image port	No
Pts	<p><math>M</math>-by-2 matrix of [x y] coordinates,</p> $\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_M & y_M \end{bmatrix}$ <p>where <math>M</math> is the total number of markers and each [x y] pair defines the center of a marker.</p>	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul> <p>If the input to the Image port is an integer, fixed point, or boolean data type, the input to the Pts port must also be an integer data type.</p>	No
ROI	Four-element vector of integers [x y width height] that define a rectangular area in which to draw the markers. The first two elements represent the one-based [x y] coordinates of the upper-left corner of the area. The second two elements represent the width and height of the area.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Clr	$P$ -element vector or $M$ -by- $P$ matrix where $P$ is the number of color planes.	Same as Image port	No
Output	Scalar, vector, or matrix of pixel values that contain the marker(s)	Same as Image port	No

The output signal is the same size and data type as the inputs to the Image, R, G, and B ports.

## Dialog Box

The Draw Markers dialog box appears as shown in the following figure.



### Marker shape

Specify the type of marker(s) to draw. Your choices are **Circle**, **X-mark**, **Plus**, **Star**, or **Square**.

When you select **Circle**, **X-mark**, or **Star**, and you select the **Use antialiasing** check box, the block performs a smoothing algorithm. The algorithm is similar to the `poly2mask` function to determine which subpixels to draw.

### **Marker size**

Enter a scalar value that represents the size of the marker, in pixels.

Enter a scalar value,  $M$ , that defines a  $(2M+1)$ -by- $(2M+1)$  pixel square into which the marker fits.  $M$  must be greater than or equal to 1.

### **Filled**

Select this check box to fill the marker with an intensity value or a color. This parameter is visible if, for the **Marker shape** parameter, you choose **Circle** or **Square**.

When you select the **Filled** check box, the **Fill color source**, **Fill color** and **Opacity factor (between 0 and 1)** parameters appear in the dialog box.

### **Fill color source**

Specify source for fill color value. You can select **Specify via dialog** or **Input port**. This parameter appears when you select the **Filled** check box. When you select **Input port**, the color input port **clr** appears on the block.

### **Fill color**

If you select **Black**, the marker is black. If you select **White**, the marker is white. If you select **User-specified value**, the **Color value(s)** parameter appears in the dialog box. This parameter is visible if you select the **Filled** check box.

### **Border color source**

Specify source for the border color value to either **Specify via dialog** or **Input port**. Border color options are visible when the fill shapes options are not selected. This parameter is visible if you select the **Filled** check box. When you select **Input port**, the color input port **clr** appears on the block.

### **Border color**

Specify the appearance of the shape's border. If you select **Black**, the border is black. If you select **White**, the border is white. If you select **User-specified value**, the **Color value(s)** parameter appears in the dialog box. This parameter is visible if you clear the **Fill shapes** check box.



### Color value(s)

Specify an intensity or color value for the marker's border or fill. This parameter appears when you set the **Border color** or **Fill color** parameters, to **User-specified value**. Tunable.

The following table describes what to enter for the color value based on the block input and the number of shapes you are drawing.

Block Input	Color Value(s) for Drawing One Marker or Multiple Markers with the Same Color	Color Value(s) for Drawing Multiple Markers with Unique Color
Intensity image	Scalar intensity value	$R$ -element vector where $R$ is the number of markers
Color image	$P$ -element vector where $P$ is the number of color planes	$P$ -by- $R$ matrix where $P$ is the number of color planes and $R$ is the number of markers

For each value in the parameter, enter a number between the minimum and maximum values that can be represented by the data type of the input image. If you enter a value outside this range, the block produces an error message.

### Opacity factor (between 0 and 1)

Specify the opacity of the shading inside the marker, where 0 indicates transparent and 1 indicates opaque. This parameter appears when you select the **Filled** check box. This parameter is tunable.

The following table describes what to enter for the **Opacity factor(s) (between 0 and 1)** parameter based on the block input and the number of markers you are drawing.

Opacity Factor value for Drawing One Marker or Multiple Markers with the Same Color	Opacity Factor value for Drawing Multiple Marker with Unique Color
Scalar intensity value	$R$ -element vector where $R$ is the number of markers

### Draw markers in

Specify the area in which to draw the markers. When you select **Entire image**, you can draw markers in the entire image. When you select **Specify region of**

interest via port, the ROI port appears on the block. Enter a four-element vector, [x y width height], where [x y] are the coordinates of the upper-left corner of the area.

**Use antialiasing**

Perform a smoothing algorithm on the marker. This parameter is visible if, for the **Marker shape** parameter, you select **Circle**, **X-mark**, or **Star**.

**Image signal**

Specify how to input and output a color video signal. When you select **One multidimensional signal**, the block accepts an *M*-by-*N*-by-*P* color video signal, where *P* is the number of color planes, at one port. When you select **Separate color signals**, additional ports appear on the block. Each port accepts one *M*-by-*N* plane of an RGB video stream.

**See Also**

Draw Shapes	Computer Vision System Toolbox software
Insert Text	Computer Vision System Toolbox software

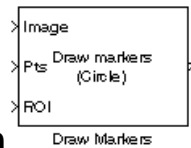
**Introduced before R2006a**

## Draw Markers (To Be Removed)

Draw markers by embedding predefined shapes on output image

### Library

Text & Graphics



### Description

---

**Note:** This Draw Markers block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Draw Markers block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

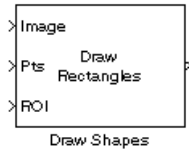
# Draw Shapes

Draw rectangles, lines, polygons, or circles on images

## Library

Text & Graphics

visiontextngfix



## Description

The Draw Shapes block draws multiple rectangles, lines, polygons, or circles on images by overwriting pixel values. As a result, the shapes are embedded on the output image.

This block uses Bresenham's line drawing algorithm to draw lines, polygons, and rectangles. It uses Bresenham's circle drawing algorithm to draw circles.

The output signal is the same size and data type as the inputs to the Image, R, G, and B ports.

You can set the shape fill or border color via the input port or via the input dialog. Use the color input or color parameter to determine the appearance of the rectangle(s), line(s), polygon(s), or circle(s).

## Port Description

Port	Input/Output	Supported Data Types	Complex Values Supported
Image	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color values	<ul style="list-style-type: none"> <li>Double-precision floating point</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
	where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	
R, G, B	Scalar, vector, or matrix that is one plane of the input RGB video stream. Inputs to the R, G, and B ports must have the same dimensions and data type.	Same as Image port	No
Pts	Use integer values to define one-based shape coordinates. If you enter noninteger values, the block rounds them to the nearest integer.	<ul style="list-style-type: none"> <li>• Double-precision floating point (only supported if the input to the I or R, G, and B ports is floating point)</li> <li>• Single-precision floating point (only supported if the input to the I or R, G, and B ports is floating point)</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
ROI	4-element vector of integers [x y width height], that define a rectangular area in which to draw the shapes. The first two elements represent the one-based coordinates of the upper-left corner of the area. The second two elements represent the width and height of the area.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Clr	<p>This port can be used to dynamically specify shape color.</p> <p><i>P</i>-element vector or an <i>M</i>-by-<i>P</i> matrix, where <i>M</i> is the number of shapes, and <i>P</i>, the number of color planes.</p> <p>You can specify a color (RGB), for each shape, or specify one color for all shapes.</p>	Same as Image port	No
Output	Scalar, vector, or matrix of pixel values that contain the shape(s)	Same as Image port	No

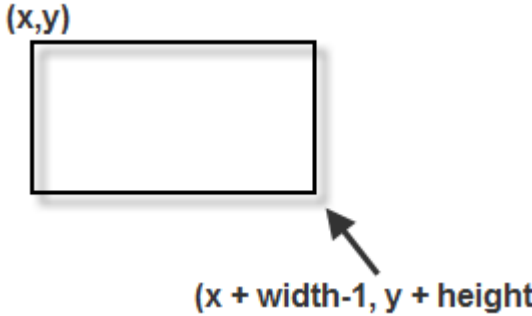
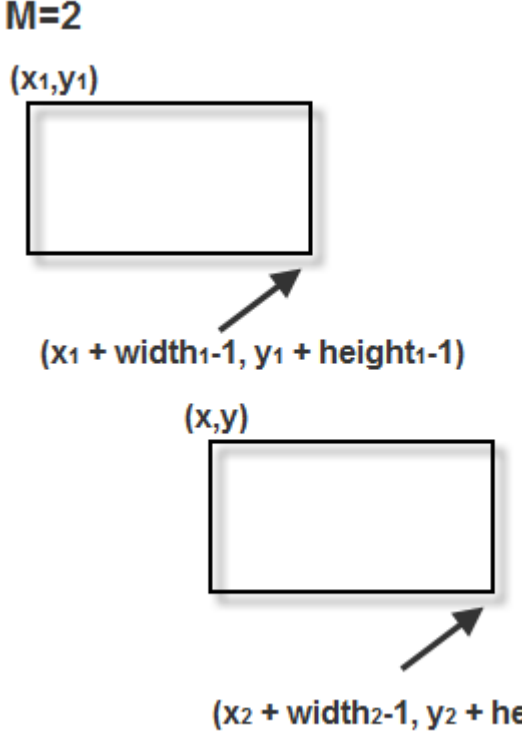
## Drawing Shapes and Lines

Use the **Shape** parameter and **Pts** port to draw the following shapes or lines:

- 
- 
- 
- 

### Drawing Rectangles

The Draw Shapes block lets you draw one or more rectangles. Set the **Shape** parameter to **Rectangles**, and then follow the instructions in the table to specify the input to the **Pts** port to obtain the desired number of rectangles.

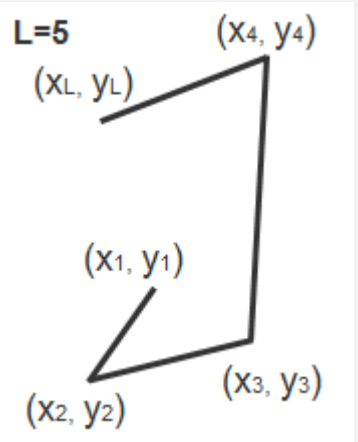
Shape	Input to the Pts Port	Drawn Shape
<p>Single Rectangle</p>	<p>Four-element row vector  <math>[x \ y \ width \ height]</math> where</p> <ul style="list-style-type: none"> <li>• <math>x</math> and <math>y</math> are the one-based coordinates of the upper-left corner of the rectangle.</li> <li>• <math>width</math> and <math>height</math> are the width, in pixels, and height, in pixels, of the rectangle. The values of <math>width</math> and <math>height</math> must be greater than 0.</li> </ul>	 <p><math>(x,y)</math></p> <p><math>(x + width - 1, y + height)</math></p>
<p>M Rectangles</p>	<p><math>M</math>-by-4 matrix</p> $\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$ <p>where each row of the matrix corresponds to a different rectangle and is of the same form as the vector for a single rectangle.</p>	 <p><math>M=2</math></p> <p><math>(x_1,y_1)</math></p> <p><math>(x_1 + width_1 - 1, y_1 + height_1 - 1)</math></p> <p><math>(x,y)</math></p> <p><math>(x_2 + width_2 - 1, y_2 + height_2)</math></p>

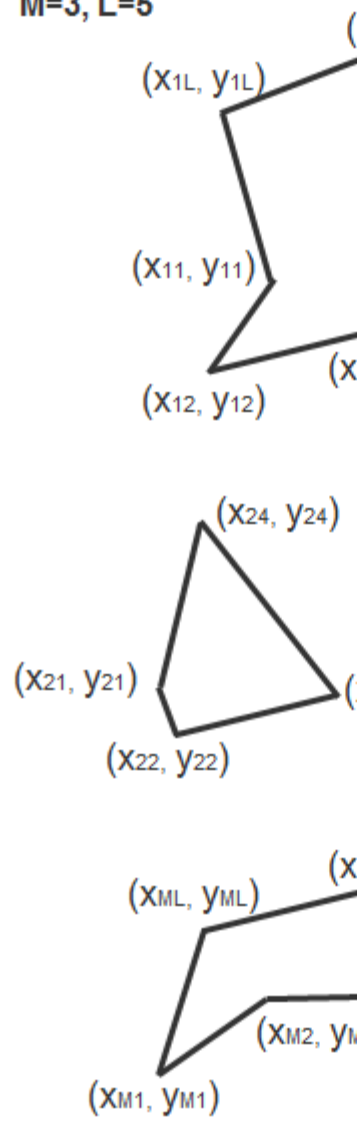
### Drawing Lines and Polylines

The Draw Shapes block lets you draw either a single line, or one or more polylines. You can draw a polyline with a series of connected line segments. Set the **Shape** parameter to **Lines**, and then follow the instructions in the table to specify the input to the Pts port to obtain the desired shape.

Shape	Input to the Pts Port	Drawn Shape
Single Line	<p>Four-element row vector <math>[x_1 \ y_1 \ x_2 \ y_2]</math> where</p> <ul style="list-style-type: none"> <li><math>x_1</math> and <math>y_1</math> are the coordinates of the beginning of the line.</li> <li><math>x_2</math> and <math>y_2</math> are the coordinates of the end of the line.</li> </ul>	
M Lines	<p><math>M</math>-by-4 matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different line and is of the same form as the vector for a single line.</p>	



Shape	Input to the Pts Port	Drawn Shape
<p>Single Polyline with <math>(L-1)</math> Segments</p>	<p>Vector of size <math>2L</math>, where <math>L</math> is the number of vertices, with format, <math>[x_1, y_1, x_2, y_2, \dots, x_L, y_L]</math>.</p> <ul style="list-style-type: none"> <li><math>x_1</math> and <math>y_1</math> are the coordinates of the beginning of the first line segment.</li> <li><math>x_2</math> and <math>y_2</math> are the coordinates of the end of the first line segment and the beginning of the second line segment.</li> <li><math>x_L</math> and <math>y_L</math> are the coordinates of the end of the <math>(L-1)^{\text{th}}</math> line segment.</li> </ul> <p>The polyline always contains <math>(L-1)</math> number of segments because the first and last vertex points do not connect. The block produces an error message when the number of rows is less than two or not a multiple of two.</p>	 <p>The diagram shows a single polyline with <math>L=5</math> segments. The vertices are labeled as <math>(x_1, y_1)</math>, <math>(x_2, y_2)</math>, <math>(x_3, y_3)</math>, <math>(x_4, y_4)</math>, and <math>(x_L, y_L)</math>. The segments connect <math>(x_1, y_1)</math> to <math>(x_2, y_2)</math>, <math>(x_2, y_2)</math> to <math>(x_3, y_3)</math>, <math>(x_3, y_3)</math> to <math>(x_4, y_4)</math>, and <math>(x_4, y_4)</math> to <math>(x_L, y_L)</math>. The first and last vertices are not connected.</p>

Shape	Input to the Pts Port	Drawn Shape
<p><math>M</math> Polylines with <math>(L-1)</math> Segments</p>	<p><math>M</math>-by-<math>2L</math> matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polyline and is of the same form as the vector for a single polyline. When you require one polyline to contain less than <math>(L-1)</math> number of segments, fill the matrix by repeating the coordinates of the last vertex.</p> <p>The block produces an error message if the number of rows is less than two or not a multiple of two.</p>	<p><b>M=3, L=5</b></p> 

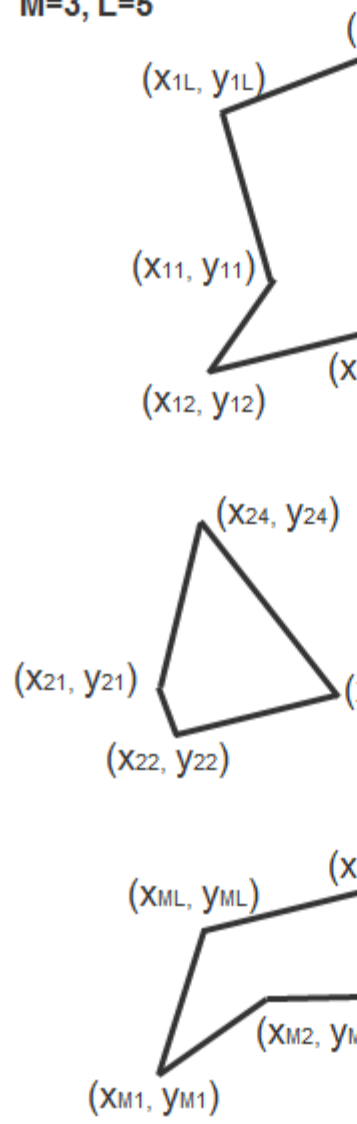
If you select the **Use antialiasing** check box, the block applies an edge smoothing algorithm.

For an example of how to use the Draw Shapes block to draw a line, see “Detect Lines in Images”.

### Drawing Polygons

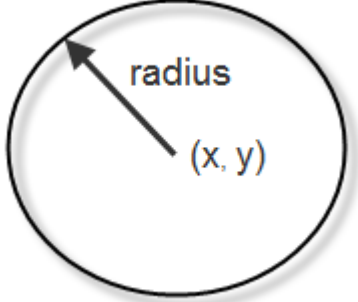
The Draw Shapes block lets you draw one or more polygons. Set the **Shape** parameter to **Polygons**, and then follow the instructions in the table to specify the input to the Pts port to obtain the desired number of polygons.

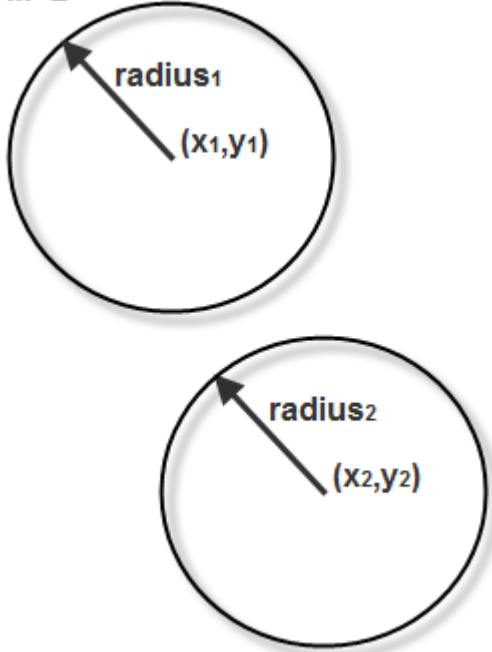
Shape	Input to the Pts Port	Drawn Shape
<p>Single Polygon with <math>L</math> line segments</p>	<p>Row vector of size <math>2L</math>, where <math>L</math> is the number of vertices, with format, <math>[x_1 \ y_1 \ x_2 \ y_2 \ \dots \ x_L \ y_L]</math> where</p> <ul style="list-style-type: none"> <li>• <math>x_1</math> and <math>y_1</math> are the coordinates of the beginning of the first line segment.</li> <li>• <math>x_2</math> and <math>y_2</math> are the coordinates of the end of the first line segment and the beginning of the second line segment.</li> <li>• <math>x_L</math> and <math>y_L</math> are the coordinates of the end of the <math>(L-1)^{\text{th}}</math> line segment and the beginning of the <math>L^{\text{th}}</math> line segment.</li> </ul> <p>The block connects <math>[x_1 \ y_1]</math> to <math>[x_L \ y_L]</math> to complete the polygon. The block produces an error if the number of rows is negative or not a multiple of two.</p>	<p>The diagram shows a closed polygon with five vertices. The vertices are labeled as follows: <math>(x_1, y_1)</math> is the leftmost vertex; <math>(x_2, y_2)</math> is the bottom-left vertex; <math>(x_3, y_3)</math> is the bottom-right vertex; <math>(x_4, y_4)</math> is the top-right vertex; and <math>(x_L, y_L)</math> is the top-left vertex. The text <math>L=5</math> is positioned at the top left of the diagram.</p>

Shape	Input to the Pts Port	Drawn Shape
<p><math>M</math> Polygons with the largest number of line segments in any line being <math>L</math></p>	<p><math>M</math>-by-<math>2L</math> matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polygon and is of the same form as the vector for a single polygon. If some polygons are shorter than others, repeat the ending coordinates to fill the polygon matrix.</p> <p>The block produces an error message if the number of rows is less than two or is not a multiple of two.</p>	<p><b>M=3, L=5</b></p> 

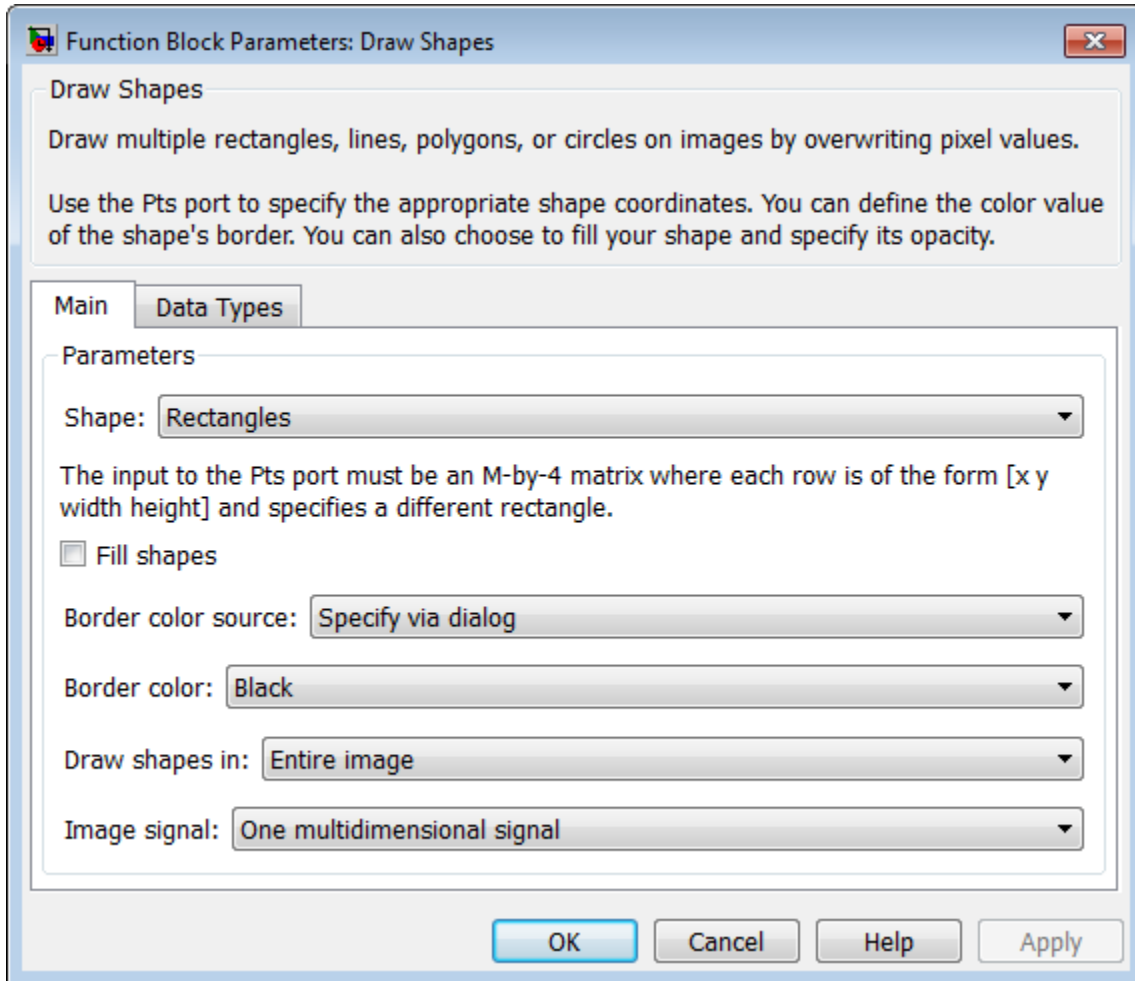
## Drawing Circles

The Draw Shapes block lets you draw one or more circles. Set the **Shape** parameter to **Circles**, and then follow the instructions in the table to specify the input to the Pts port to obtain the desired number of circles.

Shape	Input to the Pts Port	Drawn Shape
Single Circle	<p>Three-element row vector [x y radius] where</p> <ul style="list-style-type: none"><li>• x and y are coordinates for the center of the circle.</li><li>• radius is the radius of the circle, which must be greater than 0.</li></ul>	 A diagram of a circle with a black outline and a light gray drop shadow. The center of the circle is marked with the coordinates (x, y). A line segment with an arrowhead pointing to the circle's edge is labeled "radius".

Shape	Input to the Pts Port	Drawn Shape
<p><math>M</math> Circles</p>	<p><math>M</math>-by-3 matrix</p> $\begin{bmatrix} x_1 & y_1 & radius_1 \\ x_2 & y_2 & radius_2 \\ \vdots & \vdots & \vdots \\ x_M & y_M & radius_M \end{bmatrix}$ <p>where each row of the matrix corresponds to a different circle and is of the same form as the vector for a single circle.</p>	<p><b>M=2</b></p> 

## Dialog Box



### Shape

Specify the type of shape(s) to draw. Your choices are **Rectangles**, **Lines**, **Polygons**, or **Circles**.

The block performs a smoothing algorithm when you select the **Use antialiasing** check box with either **Lines**, **Polygons**, or **Circles**. The block uses an algorithm similar to the `poly2mask` function to determine which subpixels to draw.

### **Fill shapes**

Fill the shape with an intensity value or a color.

When you select this check box, the **Fill color source**, **Fill color** and **Opacity factor (between 0 and 1)** parameters appear in the dialog box.

---

**Note:** If you are generating code and you select the **Fill shapes** check box, the word length of the block input(s) cannot exceed 16 bits.

---

When you do not select the **Fill shapes** check box, the **Border color source**, and **Border color** parameters are available.

### **Fill color source**

Specify source for fill color value to either **Specify via dialog** or **Input port**. This parameter appears when you select the **Fill shapes** check box. When you select **Input port**, the color input port **clr** appears on the block.

### **Fill color**

Specify the fill color for shape. You can specify either **Black**, **White**, or **User-specified value**. When you select **User-specified value**, the **Color value(s)** parameter appears in the dialog box. This parameter is visible if you select the **Fill shapes** check box.

### **Border color source**

Specify source for the border color value to either **Specify via dialog** or **Input port**. Border color options are visible when the fill shapes options are not selected. This appears when you select the **Fill shapes** check box. When you select **Input port**, the color input port **clr** appears on the block.

### **Border color**

Specify the appearance of the shape's border. You can specify either **Black**, **White**, or **User-specified value**. If you select **User-specified value**, the **Color value(s)** parameter appears in the dialog box. This parameter appears when you clear the **Fill shapes** check box.

### **Color value(s)**



Specify an intensity or color value for the shape's border or fill. This parameter applies when you set the **Border color** or **Fill color** parameter to **User-specified value**. This parameter is tunable.

The following table describes what to enter for the color value based on the block input and the number of shapes you are drawing.

<b>Block Input</b>	<b>Color Value(s) for Drawing One Shape or Multiple Shapes with the Same Color</b>	<b>Color Value(s) for Drawing Multiple Shapes with Unique Color</b>
Intensity image	Scalar intensity value	$R$ -element vector where $R$ is the number of shapes
Color image	$P$ -element vector where $P$ is the number of color planes	$R$ -by- $P$ matrix where $P$ is the number of color planes and $R$ is the number of shapes

For each value in the **Color Value(s)** parameter, enter a number between the minimum and maximum values that can be represented by the data type of the input image. If you enter a value outside this range, the block produces an error message.

### Opacity factor (between 0 and 1)

Specify the opacity of the shading inside the shape, where 0 is transparent and 1 is opaque. This parameter is visible if you select the **Fill shapes** check box.

The following table describes what to enter for this parameter based on the block input and the number of shapes you are drawing. This parameter applies when you select the **Filled** check box.

<b>Opacity Factor value for Drawing One Shape or Multiple Shapes with the Same Color</b>	<b>Opacity Factor value for Drawing Multiple Shapes with Unique Color</b>
Scalar intensity value	$R$ -element vector where $R$ is the number of shapes

### Draw shapes in

Specify the type of area in which to draw shapes. You can define one of the following:

- **Entire image**, enables you to draw shapes in the entire image.

- **Specify region of interest via port.** When you select this option, the ROI port appears on the block. Enter a four-element vector of integer values, [ $x$   $y$  width height], where [ $x$   $y$ ] are the coordinates of the upper-left corner of the area.

---

**Note:** If you specify values that are outside the image, the block sets the values to the image boundaries.

---

### Use antialiasing

Perform a smoothing algorithm on the line, polygon, or circle. This parameter is visible if, for the **Shape** parameter, you select **Lines**, **Polygons**, or **Circles**.

### Image signal

Specify how to input and output a color video signal. Select one of the following:

- **One multidimensional signal**, the block accepts an  $M$ -by- $N$ -by- $P$  color video signal, where  $P$  is the number of color planes, at one port.
- **Separate color signals**, additional ports appear on the block. Each port accepts one  $M$ -by- $N$  plane of an RGB video stream.

## See Also

Draw Markers	Computer Vision System Toolbox software
Insert Text	Computer Vision System Toolbox software

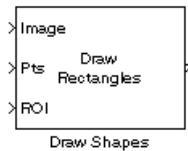
**Introduced before R2006a**

# Draw Shapes (To Be Removed)

Draw rectangles, lines, polygons, or circles on images

## Library

Text & Graphics



## Description

---

**Note:** This Draw Shapes block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Draw Shapes block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

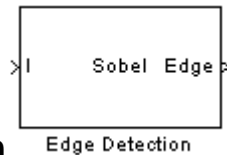
# Edge Detection

Find edges of objects in images using Sobel, Prewitt, Roberts, or Canny method

## Library

Analysis & Enhancement

visionanalysis



## Description

If, for the **Method** parameter, you select **Sobel**, **Prewitt**, or **Roberts**, the Edge Detection block finds the edges in an input image by approximating the gradient magnitude of the image. The block convolves the input matrix with the Sobel, Prewitt, or Roberts kernel. The block outputs two gradient components of the image, which are the result of this convolution operation. Alternatively, the block can perform a thresholding operation on the gradient magnitudes and output a binary image, which is a matrix of Boolean values. If a pixel value is 1, it is an edge.

If, for the **Method** parameter, you select **Canny**, the Edge Detection block finds edges by looking for the local maxima of the gradient of the input image. It calculates the gradient using the derivative of the Gaussian filter. The Canny method uses two thresholds to detect strong and weak edges. It includes the weak edges in the output only if they are connected to strong edges. As a result, the method is more robust to noise, and more likely to detect true weak edges.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>Fixed point (not supported for the Canny method)</li> <li>8-, 16-, 32-bit signed integer (not supported for the Canny method)</li> <li>8-, 16-, 32-bit unsigned integer (not supported for the Canny method)</li> </ul>	
Th	Matrix of intensity values	Same as I port	No
Edge	Matrix that represents a binary image	Boolean	No
Gv	Matrix of gradient values in the vertical direction	Same as I port	No
Gh	Matrix of gradient values in the horizontal direction	Same as I port	No
G45	Matrix of gradient values	Same as I port	No
G135	Matrix of gradient values	Same as I port	No

The output of the Gv, Gh, G45, and G135 ports is the same data type as the input to the I port. The input to the Th port must be the same data type as the input to the I port.

Use the **Method** parameter to specify which algorithm to use to find edges. You can select **Sobel**, **Prewitt**, **Roberts**, or **Canny** to find edges using the Sobel, Prewitt, Roberts, or Canny method.

## Sobel, Prewitt, and Roberts Methods

Use the **Output type** parameter to select the format of the output. If you select **Binary image**, the block outputs a Boolean matrix at the Edge port. The nonzero elements of this matrix correspond to the edge pixels and the zero elements correspond to the background pixels. If you select **Gradient components** and, for the **Method** parameter, you select **Sobel** or **Prewitt**, the block outputs the gradient components that correspond to the horizontal and vertical edge responses at the Gh and Gv ports, respectively. If you select **Gradient components** and, for the **Method** parameter, you select **Roberts**, the block outputs the gradient components that correspond to the 45 and 135 degree edge responses at the G45 and G135 ports, respectively. If you select **Binary**

image and gradient components, the block outputs both the binary image and the gradient components of the image.

Select the **User-defined threshold** check box to define a threshold values or values. If you clear this check box, the block computes the threshold for you.

Use the **Threshold source** parameter to specify how to enter your threshold value. If you select **Specify via dialog**, the **Threshold** parameter appears in the dialog box. Enter a threshold value that is within the range of your input data. If you choose **Input port**, use input port Th to specify a threshold value. This value must have the same data type as the input data. Gradient magnitudes above the threshold value correspond to edges.

The Edge Detection block computes the automatic threshold using the mean of the gradient magnitude squared image. However, you can adjust this threshold using the **Threshold scale factor (used to automatically calculate threshold value)** parameter. The block multiplies the value you enter with the automatic threshold value to determine a new threshold value.

Select the **Edge thinning** check box to reduce the thickness of the edges in your output image. This option requires additional processing time and memory resources.

---

**Note** This block is most efficient in terms of memory usage and processing time when you clear the **Edge thinning** check box and use the **Threshold** parameter to specify a threshold value.

---

## Canny Method

Select the **User-defined threshold** check box to define the low and high threshold values. If you clear this check box, the block computes the threshold values for you.

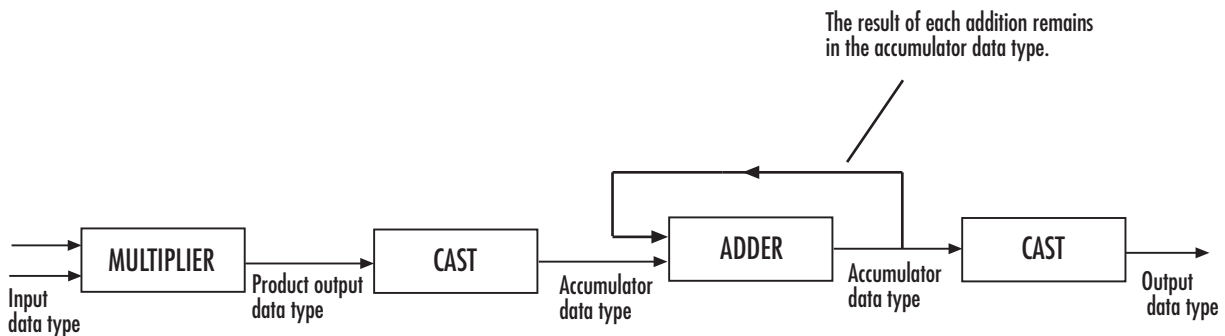
Use the **Threshold source** parameter to specify how to enter your threshold values. If you select **Specify via dialog**, the **Threshold [low high]** parameter appears in the dialog box. Enter the threshold values. If a pixel's magnitude in the gradient image, which is formed by convolving the input image with the derivative of the Gaussian filter, exceeds the high threshold, then the pixel corresponds to a strong edge. Any pixel connected to a strong edge and having a magnitude greater than the low threshold corresponds to a weak edge. If, for the **Threshold source** parameter, you choose **Input port**, use input port Th to specify a two-element vector of threshold values. These values must have the same data type as the input data.

The Edge Detection block computes the automatic threshold values using an approximation of the number of weak and nonedge image pixels. Enter this approximation for the **Approximate percentage of weak edge and nonedge pixels (used to automatically calculate threshold values)** parameter.

Use the **Standard deviation of Gaussian filter** parameter to define the Gaussian filter whose derivative is convolved with the input image.

## Fixed-Point Data Types

The following diagram shows the data types used in the Edge Detection block for fixed-point signals.

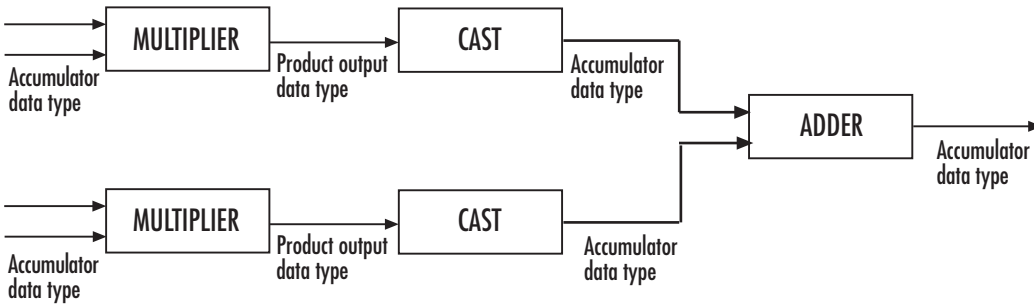


The block squares the threshold and compares it to the sum of the squared gradients to avoid using square roots.

Threshold:



Gradients:

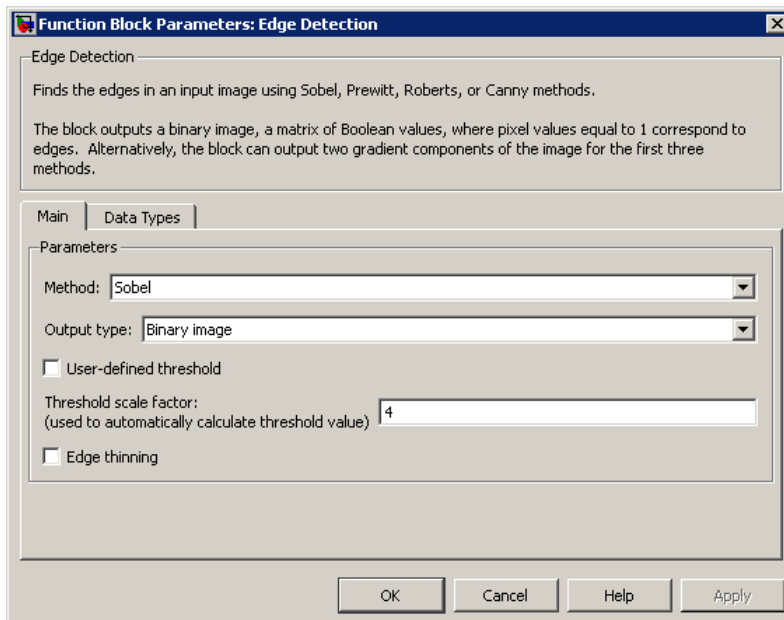


You can set the product output and accumulator data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Edge Detection dialog box appears as shown in the following figure.





### Method

Select the method by which to perform edge detection. Your choices are **Sobel**, **Prewitt**, **Roberts**, or **Canny**.

### Output type

Select the desired form of the output. If you select **Binary image**, the block outputs a matrix that is filled with ones, which correspond to edges, and zeros, which correspond to the background. If you select **Gradient components** and, for the **Method** parameter, you select **Sobel** or **Prewitt**, the block outputs the gradient components that correspond to the horizontal and vertical edge responses. If you select **Gradient components** and, for the **Method** parameter, you select **Roberts**, the block outputs the gradient components that correspond to the 45 and 135 degree edge responses. If you select **Binary image** and **gradient components**, the block outputs both the binary image and the gradient components of the image. This parameter is visible if, for the **Method** parameter, you select **Sobel**, **Prewitt**, or **Roberts**.

### User-defined threshold

If you select this check box, you can enter a desired threshold value. If you clear this check box, the block computes the threshold for you. This parameter is visible

if, for the **Method** parameter, you select Sobel, Prewitt, or Roberts, and, for the **Output type** parameter, you select Binary image or Binary image and gradient components. This parameter is also visible if, for the **Method** parameter, you select Canny.

#### **Threshold source**

If you select Specify via dialog, enter your threshold value in the dialog box. If you choose Input port, use the Th input port to specify a threshold value that is the same data type as the input data. This parameter is visible if you select the **User-defined threshold** check box.

#### **Threshold**

Enter a threshold value that is within the range of your input data. This parameter is visible if, for the **Method** parameter, you select Sobel, Prewitt, or Roberts, you select the **User-defined threshold** check box, and, for **Threshold source** parameter, you select Specify via dialog.

#### **Threshold [low high]**

Enter the low and high threshold values that define the weak and strong edges. This parameter is visible if, for the **Method** parameter, you select Canny. Then you select the **User-defined threshold** check box, and, for **Threshold source** parameter, you select Specify via dialog. Tunable.

#### **Threshold scale factor (used to automatically calculate threshold value)**

Enter a multiplier that is used to adjust the calculation of the automatic threshold. This parameter is visible if, for the **Method** parameter, you select Sobel, Prewitt, or Roberts, and you clear the **User-defined threshold** check box. Tunable.

#### **Edge thinning**

Select this check box if you want the block to perform edge thinning. This option requires additional processing time and memory resources. This parameter is visible if, for the **Method** parameter, you select Sobel, Prewitt, or Roberts, and for the **Output type** parameter, you select Binary image or Binary image and gradient components.

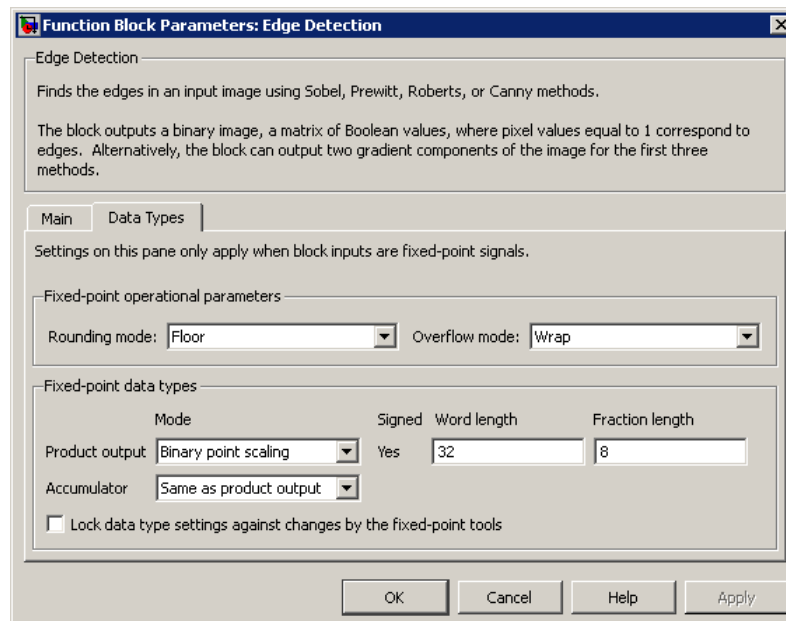
#### **Approximate percentage of weak edge and nonedge pixels (used to automatically calculate threshold values)**

Enter the approximate percentage of weak edge and nonedge image pixels. The block computes the automatic threshold values using this approximation. This parameter is visible if, for the **Method** parameter, you select Canny. Tunable.

#### **Standard deviation of Gaussian filter**

Enter the standard deviation of the Gaussian filter whose derivative is convolved with the input image. This parameter is visible if, for the **Method** parameter, you select **Canny**.

The **Data Types** pane of the Edge Detection dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

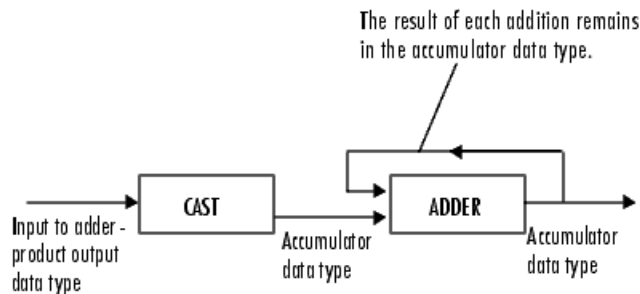
### Product output



Here, the internal coefficients are the Sobel, Prewitt, or Roberts masks. As depicted in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same** as first input, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Gradients

Choose how to specify the word length and fraction length of the outputs of the Gv and Gh ports. This parameter is visible if, for the **Output type** parameter, you choose **Gradient components** or **Binary image and gradient components**:

- When you select **Same as accumulator**, these characteristics match those of the accumulator.
- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Gonzales, Rafael C. and Richard E. Woods. *Digital Image Processing, 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall, 2002.
- [2] Pratt, William K. *Digital Image Processing, 2nd ed.* New York: John Wiley & Sons, 1991.

## See Also

edge	Image Processing Toolbox
------	--------------------------

**Introduced before R2006a**

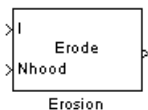
# Erosion

Find local minima in binary or intensity images

## Library

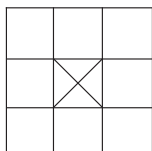
Morphological Operations

visionmorphops

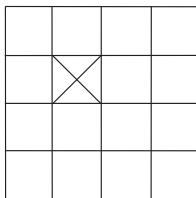


## Description

The Erosion block slides the neighborhood or structuring element over an image, finds the local minima, and creates the output matrix from these minimum values. If the neighborhood or structuring element has a center element, the block places the minima there, as illustrated in the following figure.



If the neighborhood or structuring element does not have an exact center, the block has a bias toward the upper-left corner and places the minima there, as illustrated in the following figure.



This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of 1s and 0s that represents the neighborhood values	Boolean	No
Output	Vector or matrix of intensity values that represents the eroded image	Same as I port	No

The output signal is the same data type as the input to the I port.

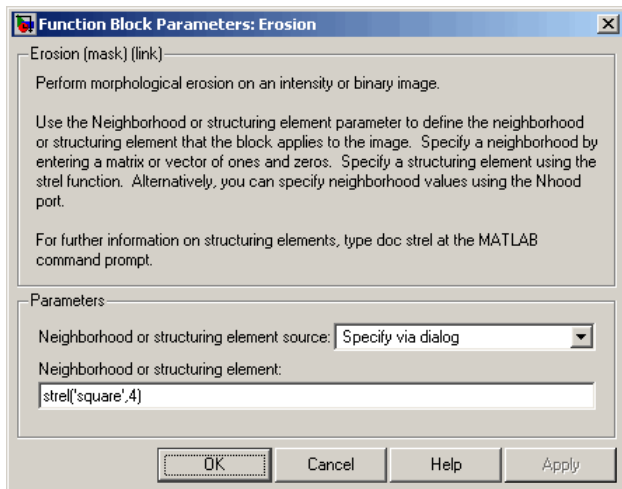
Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the Nhood port appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. You can only specify a structuring element using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the neighborhood or structuring element that the block applies to the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm. If you enter an array of STREL objects, the block applies each object to the entire matrix in turn.

## Dialog Box

The Erosion dialog box appears as shown in the following figure.





### Neighborhood or structuring element source

Specify how to enter your neighborhood or structuring element values. Select **Specify via dialog** to enter the values in the dialog box. Select **Input port** to use the Nhood port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

### Neighborhood or structuring element

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the **strel** function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select **Specify via dialog**.

## References

[1] Soille, Pierre. *Morphological Image Analysis. 2nd ed.* New York: Springer, 2003.

## See Also

Bottom-hat	Computer Vision System Toolbox software
Closing	Computer Vision System Toolbox software

Dilation	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
imerode	Image Processing Toolbox software
strel	Image Processing Toolbox software

**Introduced before R2006a**

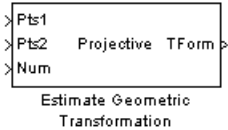
# Estimate Geometric Transformation

Estimate geometric transformation from matching point pairs

## Library

Geometric Transformations

visiongeotforms



## Description

Use the Estimate Geometric Transformation block to find the transformation matrix which maps the greatest number of point pairs between two images. A *point pair* refers to a point in the input image and its related point on the image created using the transformation matrix. You can select to use the Random Sample Consensus (RANSAC) or the Least Median Squares algorithm to exclude outliers and to calculate the transformation matrix. You can also use all input points to calculate the transformation matrix.

Port	Input/Output	Supported Data Types	Complex Values Supported
<b>Pts1/Pts2</b>	<i>M</i> -by-2 Matrix of one-based [x y] point coordinates, where <i>M</i> represents the number of points.	<ul style="list-style-type: none"> <li>• Double</li> <li>• Single</li> <li>• 8, 16, 32-bit signed integer</li> <li>• 8, 16, 32-bit unsigned integer</li> </ul>	No
<b>Num</b>	Scalar value that represents the number of valid points in <b>Pts1</b> and <b>Pts 2</b> .	<ul style="list-style-type: none"> <li>• 8, 16, 32-bit signed integer</li> <li>• 8, 16, 32-bit unsigned integer</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
<b>TForm</b>	3-by-2 or 3-by-3 transformation matrix.	<ul style="list-style-type: none"> <li>• Double</li> <li>• Single</li> </ul>	No
<b>Inlier</b>	$M$ -by-1 vector indicating which points have been used to calculate TForm.	Boolean	No

Ports **Pts1** and **Pts2** are the points on two images that have the same data type. The block outputs the same data type for the transformation matrix

When **Pts1** and **Pts2** are single or double, the output transformation matrix will also have single or double data type. When **Pts1** and **Pts2** images are built-in integers, the option is available to set the transformation matrix data type to either **Single** or **Double**. The **TForm** output provides the transformation matrix. The **Inlier** output port provides the **Inlier** points on which the transformation matrix is based. This output appears when you select the **Output Boolean signal indicating which point pairs are inliers** checkbox.

## RANSAC and Least Median Squares Algorithms

The *RANSAC* algorithm relies on a distance threshold. A pair of points,  $p_i^a$  (image  $a$ , **Pts1**) and  $p_i^b$  (image  $b$ , **Pts 2**) is an inlier only when the distance between  $p_i^b$  and the projection of  $p_i^a$  based on the transformation matrix falls within the specified threshold. The distance metric used in the RANSAC algorithm is as follows:

$$d = \sum_{i=1}^{Num} \min(D(p_i^b, \psi(p_i^a : H)), t)$$

The Least Median Squares algorithm assumes at least 50% of the point pairs can be mapped by a transformation matrix. The algorithm does not need to explicitly specify the

distance threshold. Instead, it uses the median distance between all input point pairs. The distance metric used in the Least Median of Squares algorithm is as follows:

$$d = \text{median}(D(p_1^b, \psi(p_1^a : H)), D(p_2^b, \psi(p_2^a : H)), \dots, D(p_{Num}^b, \psi(p_N^a : H)))$$

For both equations:

$p_i^a$  is a point in image  $a$  (Pts1)

$p_i^b$  is a point in image  $b$  (Pts2)

$\psi(p_i^a : H)$  is the projection of a point on image  $a$  based on transformation matrix  $H$

$D(p_i^b, p_j^b)$  is the distance between two point pairs on image  $b$

$t$  is the threshold

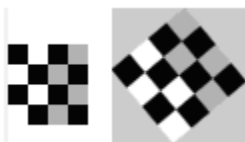
$Num$  is the number of points

The smaller the distance metric, the better the transformation matrix and therefore the more accurate the projection image.

## Transformations

The Estimate Geometric Transformation block supports **Nonreflective similarity**, **affine**, and **projective** transformation types, which are described in this section.

**Nonreflective similarity** transformation supports translation, rotation, and isotropic scaling. It has four degrees of freedom and requires two pairs of points.



The transformation matrix is:

$$H = \begin{bmatrix} h_1 & -h_2 \\ h_2 & h_1 \\ h_3 & h_4 \end{bmatrix}$$

The projection of a point  $[x \ y]$  by  $H$  is:  $[\hat{x} \ \hat{y}] = [x \ y \ 1]H$

**affine** transformation supports nonisotropic scaling in addition to all transformations that the nonreflective similarity transformation supports. It has six degrees of freedom that can be determined from three pairs of noncollinear points.



The transformation matrix is:  $H = \begin{bmatrix} h_1 & h_4 \\ h_2 & h_5 \\ h_3 & h_6 \end{bmatrix}$

The projection of a point  $[x \ y]$  by  $H$  is:  $[\hat{x} \ \hat{y}] = [x \ y \ 1]H$

**Projective** transformation supports tilting in addition to all transformations that the affine transformation supports.



The transformation matrix is :  $h = \begin{bmatrix} h_1 & h_4 & h_7 \\ h_2 & h_5 & h_8 \\ h_3 & h_6 & h_9 \end{bmatrix}$

The projection of a point  $[x \ y]$  by  $H$  is represented by homogeneous coordinates as:

$$[\hat{u} \ \hat{v} \ \hat{w}] = [x \ y \ 1]H$$

## Distance Measurement

For computational simplicity and efficiency, this block uses algebraic distance. The algebraic distance for a pair of points,  $\begin{bmatrix} x^a & y^a \end{bmatrix}^T$  on image  $a$ , and  $\begin{bmatrix} x^b & y^b \end{bmatrix}$  on image  $b$ , according to transformation  $H$ , is defined as follows;

For projective transformation:

$$D(p_i^b, \psi(p_i^a : H)) = ((u^a - w^a x^b)^2 + (v^a - w^a y^b)^2)^{\frac{1}{2}}, \text{ where } \begin{bmatrix} \hat{u}^a & \hat{v}^a & \hat{w}^a \end{bmatrix} = \begin{bmatrix} x^a & y^a & 1 \end{bmatrix} H$$

For Nonreflective similarity or affine transformation:

$$D(p_i^b, \psi(p_i^a : H)) = ((x^a - x^b)^2 + (y^a - y^b)^2)^{\frac{1}{2}},$$

$$\text{where } \begin{bmatrix} \hat{x}^a & \hat{y}^a \end{bmatrix} = \begin{bmatrix} x^a & y^a & 1 \end{bmatrix} H$$

## Algorithm

The block performs a comparison and repeats it  $K$  number of times between successive transformation matrices. If you select the **Find and exclude outliers** option, the RANSAC and Least Median Squares (LMS) algorithms become available. These algorithms calculate and compare a distance metric. The transformation matrix that produces the smaller distance metric becomes the new transformation matrix that the next comparison uses. A final transformation matrix is resolved when either:

- $K$  number of random samplings is performed
- The RANSAC algorithm, when enough number of inlier point pairs can be mapped, (dynamically updating  $K$ )

The Estimate Geometric Transformation algorithm follows these steps:

- 1 A transformation matrix  $H$  is initialized to zeros
- 2 Set **count** = 0 (Randomly sampling).
- 3 While **count** <  $K$ , where  $K$  is total number of random samplings to perform, perform the following;

- a Increment the count;  $\text{count} = \text{count} + 1$ .
  - b Randomly select pair of points from images  $a$  and  $b$ , (2 pairs for Nonreflective similarity, 3 pairs for affine, or 4 pairs for projective).
  - c Calculate a transformation matrix  $H$ , from the selected points.
  - d If  $H$  has a distance metric less than that of  $H$ , then replace  $H$  with  $H$ .  
(Optional for RANSAC algorithm only)
    - i Update  $K$  dynamically.
    - ii Exit out of sampling loop if enough number of point pairs can be mapped by  $H$ .
- 4 Use all point pairs in images  $a$  and  $b$  that can be mapped by  $H$  to calculate a refined transformation matrix  $H$
- 5 Iterative Refinement, (Optional for RANSAC and LMS algorithms)
- a Denote all point pairs that can be mapped by  $H$  as inliers.
  - b Use inlier point pairs to calculate a transformation matrix  $H$ .
  - c If  $H$  has a distance metric less than that of  $H$ , then replace  $H$  with  $H$ , otherwise exit the loop.

## Number of Random Samplings

The number of random samplings can be specified by the user for the RANSAC and Least Median Squares algorithms. You can use an additional option with the RANSAC algorithm, which calculates this number based on an accuracy requirement. The **Desired Confidence** level drives the accuracy.

The calculated number of random samplings,  $K$  used with the RANSAC algorithm, is as follows:

$$K = \frac{\log(1-p)}{\log(1-q^s)}$$

where

- $p$  is the probability of independent point pairs belonging to the largest group that can be mapped by the same transformation. The probability is dynamically calculated based on the number of inliers found versus the total number of points. As the probability increases, the number of samplings,  $K$ , decreases.



- $q$  is the probability of finding the largest group that can be mapped by the same transformation.
- $s$  is equal to the value 2, 3, or 4 for Nonreflective similarity, affine, and projective transformation, respectively.

### **Iterative Refinement of Transformation Matrix**

The transformation matrix calculated from all inliers can be used to calculate a refined transformation matrix. The refined transformation matrix is then used to find a new set of inliers. This procedure can be repeated until the transformation matrix cannot be further improved. This iterative refinement is optional.

## Dialog Box

Function Block Parameters: Estimate Geometric Transformation

Estimate Geometric Transformation

Find the transformation matrix that maps the largest number of points from Pts1 to Pts2.

The Pts1 and Pts2 inputs specify the location of points in two images. Each row in the Pts1 and Pts2 arrays has the format [x y]. The points in the arrays must be ordered to form corresponding location pairs.

When the Find and exclude outliers option is selected, the block randomly selects matching point pairs until it computes a transformation matrix that fits a specified number of inliers, or until a user-specified stopping criterion is reached. It then refines this transformation matrix by using all inliers found in the first step.

Parameters

Transformation type:

Find and exclude outliers

Method:

Algebraic distance threshold for determining inliers:

Determine number of random samplings using:

Number of random samplings:

Stop sampling earlier when a specified percentage of point pairs are determined to be inliers

Perform additional iterative refinement of the transformation matrix

Outputs

Output Boolean signal indicating which point pairs are inliers

When Pts1 and Pts2 are built-in integers, set transformation matrix data type to:

Allow variable-size signal input

OK Cancel Help Apply

**Transformation Type**

Specify transformation type, either **Nonreflective similarity**, **affine**, or **projective** transformation. If you select **projective** transformation, you can also specify a scalar algebraic distance threshold for determining inliers. If you select either **affine** or **projective** transformation, you can specify the distance threshold for determining inliers in pixels. See “Transformations” on page 1-337 for a more detailed discussion. The default value is **projective**.

**Find and exclude outliers**

When selected, the block finds and excludes outliers from the input points and uses only the inlier points to calculate the transformation matrix. When this option is not selected, all input points are used to calculate the transformation matrix.

**Method**

Select either the **RANDOM SAMPLE CONSENSUS (RANSAC)** or the **Least Median of Squares** algorithm to find outliers. See “RANSAC and Least Median Squares Algorithms” on page 1-336 for a more detailed discussion. This parameter appears when you select the **Find and exclude outliers** check box.

**Algebraic distance threshold for determining inliers**

Specify a scalar threshold value for determining inliers. The threshold controls the upper limit used to find the algebraic distance in the RANSAC algorithm. This parameter appears when you set the **Method** parameter to **Random Sample Consensus (RANSAC)** and the **Transformation type** parameter to **projective**. The default value is **1.5**.

**Distance threshold for determining inliers (in pixels)**

Specify the upper limit distance a point can differ from the projection location of its associating point. This parameter appears when you set the **Method** parameter to **Random Sample Consensus (RANSAC)** and you set the value of the **Transformation type** parameter to **Nonreflective similarity** or **affine**. The default value is **1.5**.

**Determine number of random samplings using**

Select **Specified value** to enter a positive integer value for number of random samplings, or select **Desired confidence** to set the number of random samplings as a percentage and a maximum number. This parameter appears when you select **Find and exclude outliers** parameter, and you set the value of the **Method** parameter to **Random Sample Consensus (RANSAC)**.

**Number of random samplings**

Specify the number of random samplings for the algorithm to perform. This parameter appears when you set the value of the **Determine number of random samplings using** parameter to **Specified value**.

**Desired confidence (in %)**

Specify a percent by entering a number between 0 and 100. The **Desired confidence** value represents the probability of the algorithm to find the largest group of points that can be mapped by a transformation matrix. This parameter appears when you set the **Determine number of random samplings using** parameter to **Desired confidence**.

**Maximum number of random samplings**

Specify an integer number for the maximum number of random samplings. This parameter appears when you set the **Method** parameter to **Random Sample Consensus (RANSAC)** and you set the value of the **Determine number of random samplings using** parameter to **Desired confidence**.

**Stop sampling earlier when a specified percentage of point pairs are determined to be inlier**

Specify to stop random sampling when a percentage of input points have been found as inliers. This parameter appears when you set the **Method** parameter to **Random Sample Consensus (RANSAC)**.

**Perform additional iterative refinement of the transformation matrix**

Specify whether to perform refinement on the transformation matrix. This parameter appears when you select **Find and exclude outliers** check box.

**Output Boolean signal indicating which point pairs are inliers**

Select this option to output the inlier point pairs that were used to calculate the transformation matrix. This parameter appears when you select **Find and exclude outliers** check box. The block will not use this parameter with signed or double, data type points.

**When Pts1 and Pts2 are built-in integers, set transformation matrix data type to**

Specify transformation matrix data type as **Single** or **Double** when the input points are built-in integers. The block will not use this parameter with signed or double, data type points.

## Examples

### Calculate transformation matrix from largest group of point pairs

Examples of input data and application of the Estimate Geometric Transformation block appear in the following figures. Figures (a) and (b) show the point pairs. The points are denoted by stars or circles, and the numbers following them show how they are paired. Some point pairs can be mapped by the same transformation matrix. Other point pairs require a different transformation matrix. One matrix exists that maps the largest number of point pairs, the block calculates and returns this matrix. The block finds the point pairs in the largest group and uses them to calculate the transformation matrix. The point pairs connected by the magenta lines are the largest group.

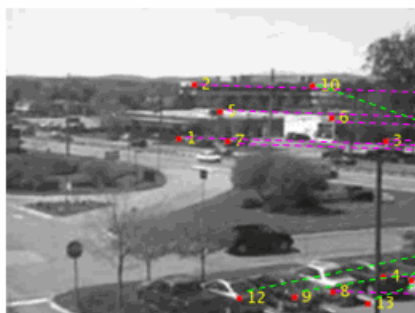
The transformation matrix can then be used to stitch the images as shown in Figure (e).



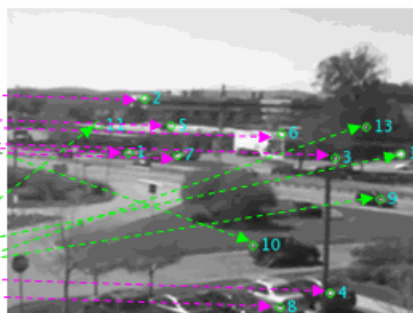
(a)



(b)



(c)



(d)



(e)

## Video Mosaicking

To see an example of the Estimate Geometric Transformation block used in a model with other blocks, see the “Video Mosaicking” example.

## Troubleshooting

The success of estimating the correct geometric transformation depends heavily on the quality of the input point pairs. If you chose the RANSAC or LMS algorithm, the block will randomly select point pairs to compute the transformation matrix and will use the transformation that best fits the input points. There is a chance that all of the randomly selected point pairs may contain outliers despite repeated samplings. In this case, the output transformation matrix, `TForm`, is invalid, indicated by a matrix of zeros.

To improve your results, try the following:

Increase the percentage of inliers in the input points.

Increase the number for random samplings.

For the RANSAC method, increase the desired confidence.

For the LMS method, make sure the input points have 50% or more inliers.

Use features appropriate for the image contents

Be aware that repeated patterns, for example, windows in office building, will cause false matches when you match the features. This increases the number of outliers.

Do not use this function if the images have significant parallax. You can use the `estimateFundamentalMatrix` function instead.

Choose the minimum transformation for your problem.

If a projective transformation produces the error message, “A portion of the input image was transformed to the location at infinity. Only transformation matrices that do not transform any part of the image to infinity are supported.”, it is usually caused by a transformation matrix and an image that would result in an output distortion that does not fit physical reality. If the matrix was an output of the Estimate Geometric Transformation block, then most likely it could not find enough inliers.

## References

R. Hartley and A. Zisserman, “Multiple View Geometry in Computer Vision,” Second edition, Cambridge University Press, 2003

## See Also

cp2tform	Image Processing Toolbox
vipmosaicking	Computer Vision System Toolbox

**Introduced in R2008a**

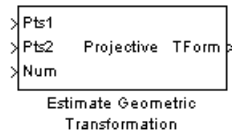


# Estimate Geometric Transformation (To Be Removed)

Estimate geometric transformation from matching point pairs

## Library

Geometric Transformations



## Description

**Note:** This Estimate Geometric Transformation block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated `Estimate Geometric Transformation` block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

**Introduced in R2011b**

# Find Local Maxima

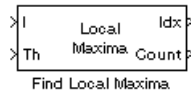
Find local maxima in matrices

## Library

Statistics

visionstatistics

## Description

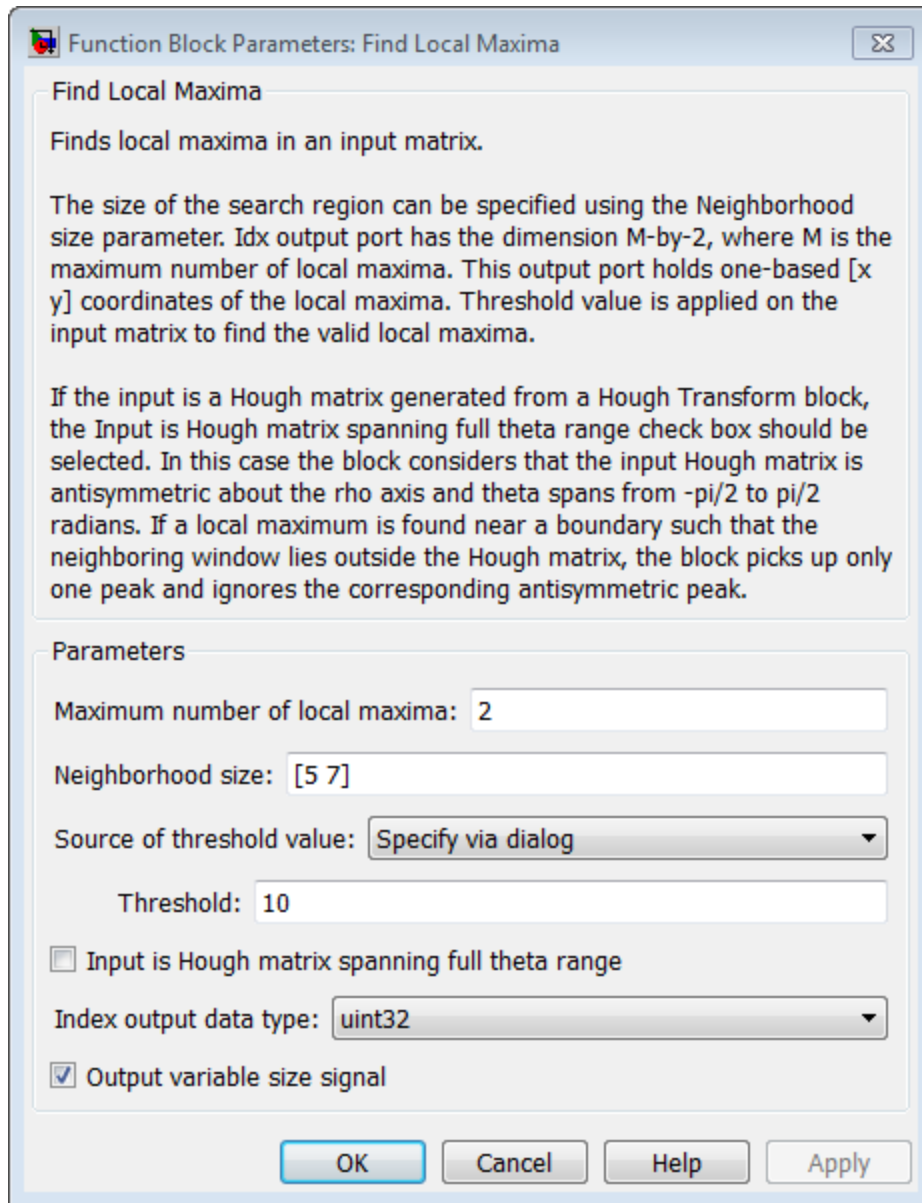


The Find Local Maxima block finds the local maxima within an input matrix. It does so by comparing the maximum value in the matrix to a user-specified threshold. The block considers a value to be a valid local maximum when the maximum value is greater than or equal to the specified threshold. The determination of the local maxima is based on the *neighborhood*, an area around and including the maximum value. After finding the local maxima, the block sets all the matrix values in the neighborhood, including the maximum value, to 0. This step ensures that subsequent searches do not include this maximum. The size of the neighborhood must be appropriate for the data set. That is, the threshold must eliminate enough of the values around the maximum so that false peaks are not discovered. The process repeats until the block either finds all valid maximas or the number of local maximas equal the **Maximum number of local maximas** value. The block outputs one-based [x y] coordinates of the maxima. The data to all input ports must be the same data type.

If the input to this block is a Hough matrix output from the Hough Transform block, select the **Input is Hough matrix spanning full theta range** check box. If you select this check box, the block assumes that the Hough port input is antisymmetric about the rho axis and theta ranges from  $-\pi/2$  to  $\pi/2$  radians. If the block finds a local maxima near the boundary, and the neighborhood lies outside the Hough matrix, then the block detects only one local maximum. It ignores the corresponding antisymmetric maximum.

## Dialog Box

The Find Local Maxima dialog box appears as shown in the following figure.



### Maximum number of local maxima

Specify the maximum number of maxima you want the block to find.

### Neighborhood size

Specify the size of the neighborhood around the maxima over which the block zeros out the values. Enter a two-element vector of positive odd integers,  $[rc]$ . Here,  $r$  represents the number of rows in the neighborhood, and  $c$  represents the number of columns.

### Source of threshold value

Specify how to enter the threshold value. If you select **Input port**, the **Th** port appears on the block. If you select **Specify via dialog**, the **Threshold** parameter appears in the dialog box. Enter a scalar value that represents the value all maxima should meet or exceed.

### Threshold

Enter a scalar value that represents the value all maxima should meet or exceed. This parameter is visible if, for the **Source of threshold value** parameter, you choose **Specify via dialog**.

### Input is Hough matrix spanning full theta range

If you select this check box, the block assumes that the Hough port input is antisymmetric about the rho axis and theta ranges from  $-\pi/2$  to  $\pi/2$  radians.

### Index output data type

Specify the data type of the **Idx** port output. Your choices are **double**, **single**, **uint8**, **uint16**, or **uint32**.

### Output variable size signal

Specify output data type. When you uncheck the **Output variable size signal**, the **Count output data type** parameter appears in the dialog box.

### Count output data types

Specify the data type of the **Count** port output. Your choices are **double**, **single**, **uint8**, **uint16**, or **uint32**. This parameter applies when you clear the **Output variable size signal** check box.

## Examples

See “Detect Lines in Images” in the *Computer Vision System Toolbox User's Guide*.

## Supported Data Types

The block outputs the one-based [x y] coordinates of the maxima at the **Idx** port and the number of valid local maxima found at the **Count** port.

Port	Input/Output	Supported Data Types	Complex Values Supported
I/Hough	Matrix in which you want to find the maxima.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Th	Scalar value that represents the value the maxima should meet or exceed.	Same as I/Hough port	No
Idx	An $M$ -by-2 matrix of one-based [x y] coordinates, where $M$ represents the number of local maximas found.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Count	Scalar value that represents the number of maxima that meet or exceed the threshold value.	Same as Idx port	No

## See Also

Hough Lines	Computer Vision System Toolbox
Corner Detection	Computer Vision System Toolbox
houghpeaks	Image Processing Toolbox
hough	Image Processing Toolbox
vision.HoughLines	Computer Vision System Toolbox

**Introduced before R2006a**

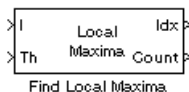
## Find Local Maxima (To Be Removed)

Find local maxima in matrices

### Library

Statistics

### Description



---

**Note:** This Find Local Maxima block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Find Local Maxima block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**



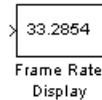
# Frame Rate Display

Calculate average update rate of input signal

## Library

Sinks

visionsinks



## Description

The Frame Rate Display block calculates and displays the average update rate of the input signal. This rate is in relation to the wall clock time. For example, if the block displays 30, the model is updating the input signal 30 times every second. You can use this block to check the video frame rate of your simulation. During code generation, Simulink Coder does not generate code for this block.

---

**Note:** This block supports intensity and color images on its port.

---

Port	Input	Supported Data Types	Complex Values Supported
Input	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No

Use the **Calculate and display rate every** parameter to control how often the block updates the display. When this parameter is greater than 1, the block displays the

average update rate for the specified number of video frames. For example, if you enter 10, the block calculates the amount of time it takes for the model to pass 10 video frames to the block. It divides this time by 10 and displays this average video frame rate on the block.

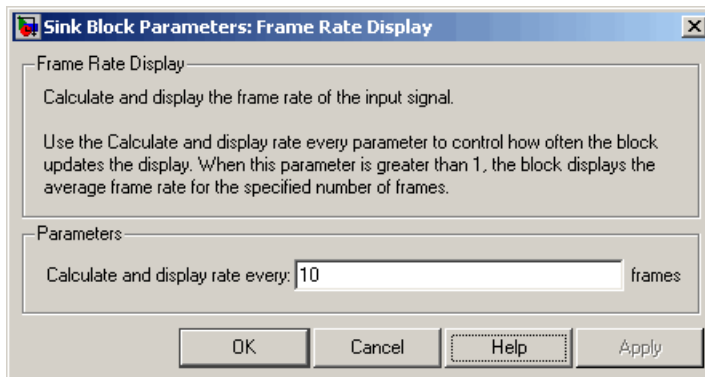
---

**Note:** If you do not connect the Frame Rate Display block to a signal line, the block displays the base (fastest) rate of the Simulink model.

---

## Dialog Box

The Frame Rate Display dialog box appears as shown in the following figure.



### Calculate and display rate every

Use this parameter to control how often the block updates the display.

## See Also

To Multimedia File	Computer Vision System Toolbox software
To Video Display	Computer Vision System Toolbox software
Video To Workspace	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software

**Introduced before R2006a**

## From Multimedia File

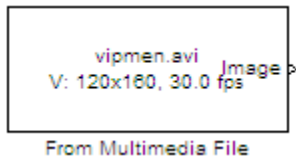
Read video frames and audio samples from compressed multimedia file

### Library

Sources

visionsources

### Description



The **From Multimedia File** block reads audio samples, video frames, or both from a multimedia file. The block imports data from the file into a Simulink model.

---

**Note:** This block supports code generation for the host computer that has file I/O available. You cannot use this block with Simulink Desktop Real-Time™ software because that product does not support file I/O.

---

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The `packNGO` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGO` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Simulink Coder”, “Simulink Shared Library Dependencies”, and “Accelerating Simulink Models” for details.

This block allows you to read WMA/WMV streams to disk or across a network connection. Similarly, the To Multimedia File block allows you to write WMA/WMV streams to disk or across a network connection. If you want to play an MP3/MP4 file in Simulink, but you do not have the codecs, you can re-encode the file as WMA/WMV, which are supported by the Computer Vision System Toolbox.

## Supported Platforms and File Types

The supported file formats available to you depend on the codecs installed on your system.

Platform	Supported File Name Extensions
All Platforms	AVI (.avi)
Windows <sup>®</sup>	<p><b>Image:</b> .jpg, .bmp</p> <p><b>Video:</b> MPEG (.mpeg) MPEG-2 (.mp2) MPEG-1 .mpg MPEG-4, including H.264 encoded video (.mp4, .m4v) Motion JPEG 2000 (.mj2) Windows Media Video (.wmv, .asf, .asx, .asx) and any format supported by Microsoft DirectShow<sup>®</sup> 9.0 or higher.</p> <p><b>Audio:</b> WAVE (.wav) Windows Media Audio File (.wma) Audio Interchange File Format (.aif, .aiff) Compressed Audio Interchange File Format (.aifc), MP3 (.mp3) Sun Audio (.au) Apple (.snd)</p>
Macintosh	<p><b>Video:</b> .avi Motion JPEG 2000 (.mj2) MPEG-4, including H.264 encoded video (.mp4, .m4v) Apple QuickTime Movie (.mov) and any format supported by QuickTime as listed on <a href="http://support.apple.com/kb/HT3775">http://support.apple.com/kb/HT3775</a>.</p>

Platform	Supported File Name Extensions
	<b>Audio:</b> Uncompressed .avi
Linux®	Motion JPEG 2000 (.mj2) Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on <a href="http://gstreamer.freedesktop.org/documentation/plugins.html">http://gstreamer.freedesktop.org/documentation/plugins.html</a> , including Ogg Theora (.ogg).

Windows XP and Windows 7 x64 platform ships with a limited set of 64-bit video and audio codecs. If a compressed multimedia file fails to play, try one of the two alternatives:

- Run the 32-bit version of MATLAB on your Windows XP x64 platform. Windows XP x64 ships with many 32-bit codecs.
- Save the multimedia file to a supported file format listed in the table above.

If you use Windows, use Windows Media player Version 11 or later.

---

**Note:** MJ2 files with bit depth higher than 8-bits is not supported by vision.VideoFileReader. Use VideoReader and VideoWriter for higher bit depths.

---

## Ports

The output ports of the From Multimedia File block change according to the content of the multimedia file. If the file contains only video frames, the **Image**, intensity **I**, or **R, G, B** ports appear on the block. If the file contains only audio samples, the **Audio** port appears on the block. If the file contains both audio and video, you can select the data to emit. The following table describes available ports.

Port	Description
<b>Image</b>	$M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes.
<b>I</b>	$M$ -by- $N$ matrix of intensity values.
<b>R, G, B</b>	Matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports must have same dimensions.
<b>Audio</b>	Vector of audio data.

Port	Description
Y, Cb, Cr	<p>Matrix that represents one frame of the YCbCr video stream. The Y, Cb, Cr ports produce the following outputs:</p> <p>Y: <math>M \times N</math></p> <p style="text-align: center;"><math>\frac{N}{2}</math></p> <p>Cb: <math>M \times \frac{N}{2}</math></p> <p style="text-align: center;"><math>\frac{N}{2}</math></p> <p>Cr: <math>M \times \frac{N}{2}</math></p>

## Sample Rates

The sample rate that the block uses depends on the audio and video sample rate. While the FMMF block operates at a single rate in Simulink, the underlying audio and video streams can produce different rates. In some cases, when the block outputs both audio and video, makes a small adjustment to the video rate.

## Sample Time Calculations Used for Video and Audio Files

$$\text{Sample time} = \frac{\text{ceil}\left(\frac{\text{AudioSampleRate}}{\text{FPS}}\right)}{\text{AudioSampleRate}}$$

When audio sample time,  $\frac{\text{AudioSampleRate}}{\text{FPS}}$  is noninteger, the equation cannot reduce

$$\text{to } \frac{1}{\text{FPS}}.$$

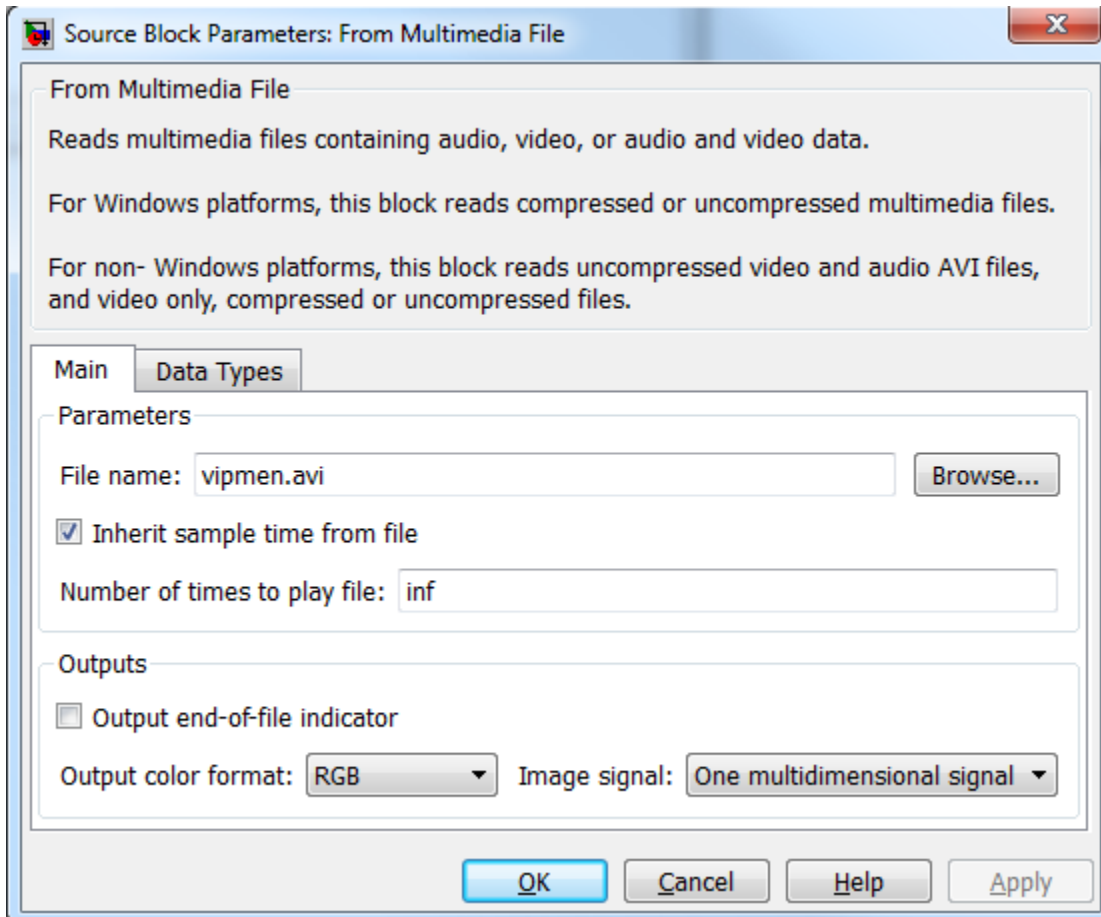
In this case, to prevent synchronization problems, the block drops the corresponding

video frame when the audio stream leads the video stream by more than  $\frac{1}{\text{FPS}}$ .

In summary, the block outputs one video frame at each Simulink time step. To calculate the number of audio samples to output at each time step, the block divides the audio sample rate by the video frame rate (fps). If the audio sample rate does not divide evenly by the number of video frames per second, the block rounds the number of audio samples up to the nearest whole number. If necessary, the block periodically drops a video frame to maintain synchronization for large files.

## Dialog Box

The **Main** pane of the From Multimedia File block dialog appears as follows.



### File name

Specify the name of the multimedia file from which to read. The block determines the type of file (audio and video, audio only, or video only) and provides the associated parameters.

If the location of the file does not appear on your MATLAB path, use the **Browse** button to specify the full path. Otherwise, if the location of this file appears on your



MATLAB path, enter only the file name. On Windows platforms, this parameter supports URLs that point to MMS (Microsoft Media Server) streams.

### **Inherit sample time from file**

Select the **Inherit sample time from file** check box if you want the block sample time to be the same as the multimedia file. If you clear this check box, enter the block sample time in the **Desired sample time** parameter field. The file that the **From Multimedia File** block references, determines the block default sample time. You can also set the sample time for this block manually. If you do not know the intended sample rate of the video, let the block inherit the sample rate from the multimedia file.

### **Desired sample time**

Specify the block sample time. This parameter becomes available if you clear the **Inherit sample time from file** check box.

### **Number of times to play file**

Enter a positive integer or `inf` to represent the number of times to play the file.

### **Output end-of-file indicator**

Use this check box to determine whether the output is the last video frame or audio sample in the multimedia file. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port defaults to 1 when the last video frame or audio sample is output from the block. Otherwise, the output from the EOF port defaults to 0.

### **Multimedia outputs**

Specify **Video and audio**, **Video only**, or **Audio only** output file type. This parameter becomes available only when a video signal has both audio and video.

### **Samples per audio channel**

Specify number of samples per audio channel. This parameter becomes available for files containing audio.

### **Output color format**

Specify whether you want the block to output **RGB**, **Intensity**, or **YCbCr 4:2:2** video frames. This parameter becomes available only for a signal that contains video. If you select **RGB**, use the **Image signal** parameter to specify how to output a color signal.

### **Image signal**

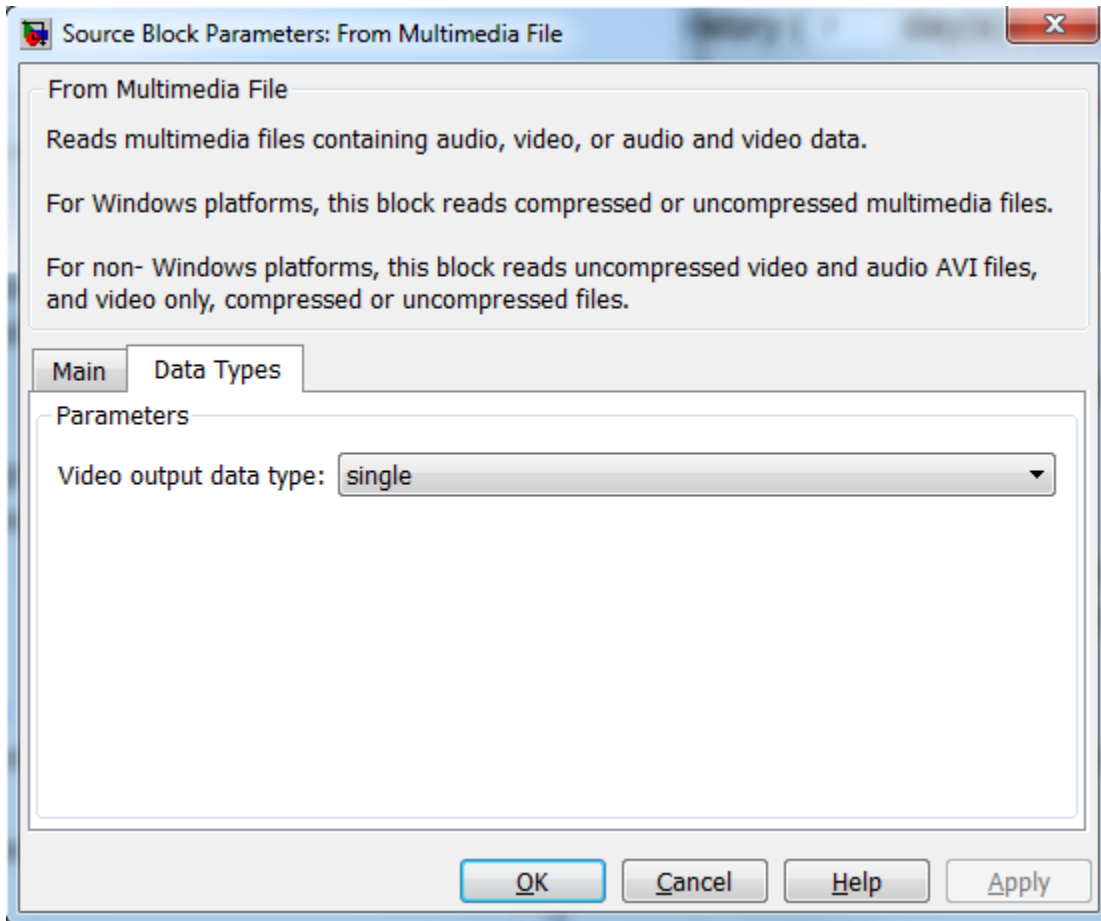
Specify how to output a color video signal. If you select **One multidimensional signal**, the block outputs an  $M$ -by- $N$ -by- $P$  color video signal, where  $P$  is the number

of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one  $M$ -by- $N$  plane of an RGB video stream. This parameter becomes available only if you set the **Image color space** parameter to RGB and the signal contains video.

#### Audio output sampling mode

Select **Sample based** or **Frame based** output. This parameter appears when you specify a file containing audio for the **File name** parameter.

The **Data Types** pane of the **To Multimedia File** block dialog box appears as follows.



**Audio output data type**

Set the data type of the audio samples output at the Audio port. This parameter becomes available only if the multimedia file contains audio. You can choose **double**, **single**, **int16**, or **uint8** types.

**Video output data type**

Set the data type of the video frames output at the **R**, **G**, **B**, or **Image** ports. This parameter becomes available only if the multimedia file contains video. You can choose **double**, **single**, **int8**, **uint8**, **int16**, **uint16**, **int32**, **uint32**, or **Inherit** from file types.

**Supported Data Types**

For source blocks to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. For other data types, the pixel values must be between the minimum and maximum values supported by their data type.

Port	Supported Data Types	Supports Complex Values?
Image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>	No
R, G, B	Same as the Image port	No
Audio	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 16-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul>	No
Y, Cb,Cr	Same as the Image port	No

**See Also**

To Multimedia File      Computer Vision System Toolbox

“Specify Sample Time”      Simulink

**Introduced before R2006a**

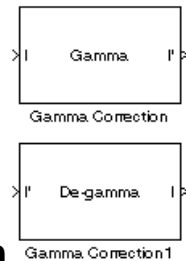
# Gamma Correction

Apply or remove gamma correction from images or video streams

## Library

Conversions

visionconversions



## Description

Use the Gamma Correction block to apply or remove gamma correction from an image or video stream. For input signals normalized between 0 and 1, the block performs gamma correction as defined by the following equations. For integers and fixed-point data types, these equations are generalized by applying scaling and offset values specific to the data type:

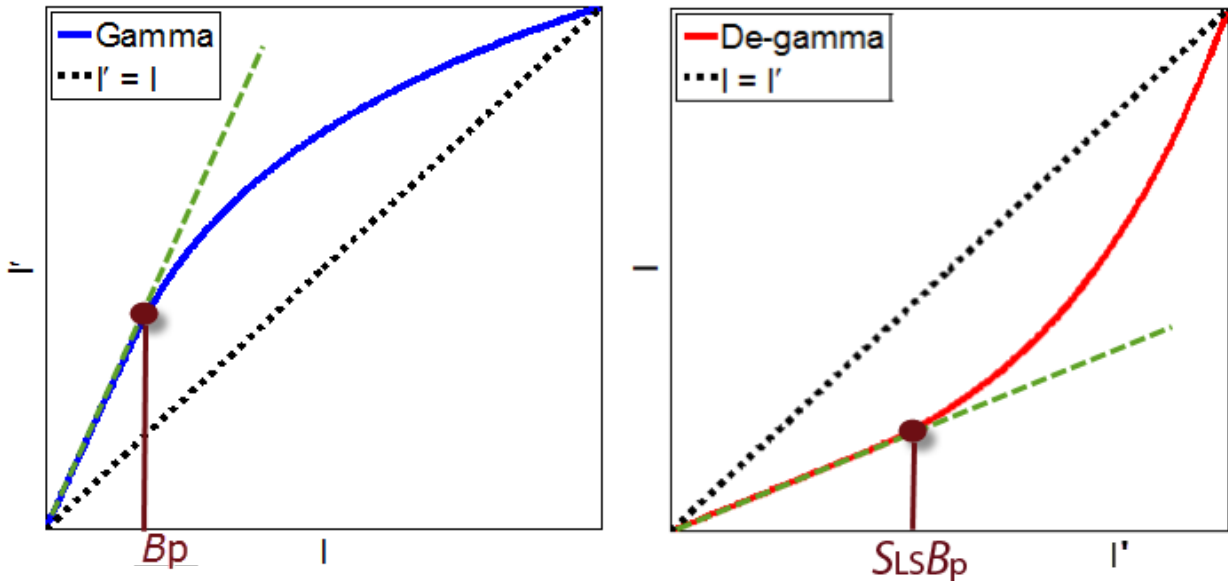
$$S_{LS} = \frac{1}{\frac{\gamma}{B_P^{(\frac{1}{\gamma}-1)}} - \gamma B_P + B_P}$$

$$F_S = \frac{\gamma S_{LS}}{B_P^{(\frac{1}{\gamma}-1)}}$$

$$C_O = F_S B_P^{\frac{1}{\gamma}} - S_{LS} B_P$$

$$I' = \begin{cases} S_{LS}I, & I \leq B_p \\ \frac{1}{F_S I^\gamma - C_O}, & I > B_p \end{cases}$$

$S_{LS}$  is the slope of the straight line segment.  $B_p$  is the break point of the straight line segment, which corresponds to the **Break point** parameter.  $F_S$  is the slope matching factor, which matches the slope of the linear segment to the slope of the power function segment.  $C_O$  is the segment offset, which ensures that the linear segment and the power function segments connect. Some of these parameters are illustrated by the following diagram.



For normalized input signals, the block removes gamma correction, which linearizes the input video stream, as defined by the following equation:

$$I = \begin{cases} \frac{I'}{S_{LS}}, & I' \leq S_{LS}B_p \\ \left( \frac{I' + C_O}{F_S} \right)^\gamma, & I' > S_{LS}B_p \end{cases}$$

Typical gamma values range from 1 to 3. Most monitor gamma values range from 1.8 to 2.2. Check with the manufacturer of your hardware to obtain the exact gamma value. Gamma function parameters for some common standards are shown in the following table:

Standard	Slope	Break Point	Gamma
CIE L*	9.033	0.008856	3
Recommendation ITU-R BT.709-3, Parameter Values for the HDTV Standards for Production and International Programme Exchange	4.5	0.018	$20/9$
sRGB	12.92	0.00304	2.4

---

**Note:** This block supports intensity and color images on its ports.

---

The properties of the input and output ports are summarized in the following table:

Port	Input/Output	Supported Data Types	Complex Values Supported
I	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (up to 16-bit word length)</li> <li>• 8- and 16-bit signed integer</li> <li>• 8- and 16-bit unsigned integer</li> </ul>	No
I'	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	Same as I port	No

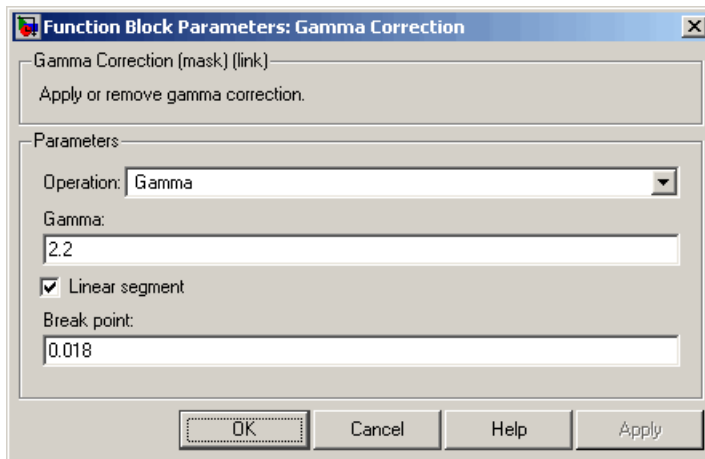
Use the **Operation** parameter to specify the block's operation. If you want to perform gamma correction, select **Gamma**. If you want to linearize the input signal, select **De-gamma**.

If, for the **Operation** parameter, you select **Gamma**, use the **Gamma** parameter to enter the desired gamma value of the output video stream. This value must be greater than or equal to 1. If, for the **Operation** parameter, you select **De-gamma**, use the **Gamma** parameter to enter the gamma value of the input video stream.

Select the **Linear segment** check box if you want the gamma curve to have a linear portion near black. If you select this check box, the **Break point** parameter appears on the dialog box. Enter a scalar value that indicates the *I*-axis value of the end of the linear segment. The break point is shown in the first diagram of this block reference page.

## Dialog Box

The Gamma Correction dialog box appears as shown in the following figure.



### Operation

Specify the block's operation. Your choices are **Gamma** or **De-gamma**.

### Gamma

If, for the **Operation** parameter, you select **Gamma**, enter the desired gamma value of the output video stream. This value must be greater than or equal to 1. If, for the **Operation** parameter, you select **De-gamma**, enter the gamma value of the input video stream.

### Linear segment



Select this check box if you want the gamma curve to have a linear portion near the origin.

**Break point**

Enter a scalar value that indicates the  $I$ -axis value of the end of the linear segment. This parameter is visible if you select the **Linear segment** check box.

## References

- [1] Poynton, Charles. *Digital Video and HDTV Algorithms and Interfaces*. San Francisco, CA: Morgan Kaufman Publishers, 2003.

## See Also

Color Space Conversion	Computer Vision System Toolbox software
imadjust	Image Processing Toolbox software

**Introduced before R2006a**

# Gaussian Pyramid

Perform Gaussian pyramid decomposition

## Library

Transforms

visiontransforms



## Description

The Gaussian Pyramid block computes Gaussian pyramid reduction or expansion to resize an image. The image reduction process involves lowpass filtering and downsampling the image pixels. The image expansion process involves upsampling the image pixels and lowpass filtering. You can also use this block to build a Laplacian pyramid. For more information, see “Examples” on page 1-376.

---

**Note:** This block supports intensity and color images on its ports.

---

Port	Output	Supported Data Types	Complex Values Supported
Input	<p>In <b>Reduce</b> mode, the input can be an M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes.</p> <p>In <b>Expand</b> mode, the input can be a scalar, vector, or</p>	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No

Port	Output	Supported Data Types	Complex Values Supported
	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes.		
Output	<p>In <b>Reduce</b> mode, the output can be a scalar, vector, or matrix that represents one level of a Gaussian pyramid.</p> <p>In <b>Expand</b> mode, the output can be a matrix that represents one level of a Gaussian pyramid.</p>	Same as Input port	No

Use the **Operation** parameter to specify whether to reduce or expand the input image. If you select **Reduce**, the block applies a lowpass filter and then downsamples the input image. If you select **Expand**, the block upsamples and then applies a lowpass filter to the input image.

Use the **Pyramid level** parameter to specify the number of times the block upsamples or downsamples each dimension of the image by a factor of 2. For example, suppose you have a 4-by-4 input image. You set the **Operation** parameter to **Reduce** and the **Pyramid level** to 1. The block filters and downsamples the image and outputs a 2-by-2 pixel output image. If you have an M-by-N input image and you set the **Operation** parameter to **Reduce**, you can calculate the dimensions of the output image using the following equation:

$$\text{ceil}\left(\frac{M}{2}\right)\text{-by-}\text{ceil}\left(\frac{N}{2}\right)$$

You must repeat this calculation for each successive pyramid level. If you have an M-by-N input image and you set the **Operation** parameter to **Expand**, you can calculate the dimensions of the output image using the following equation:

$$\left[(M-1)2^l + 1\right]\text{-by-}\left[(N-1)2^l + 1\right]$$

In the previous equation,  $l$  is the scalar value from 1 to  $\infty$  that you enter for the **Pyramid level** parameter.

Use the **Coefficient source** parameter to specify the coefficients of the lowpass filter. If you select **Default separable filter**  $[1/4 - a/2 \ 1/4 \ a \ 1/4 \ 1/4 - a/2]$ , use the **a** parameter to define the coefficients in the vector of separable filter coefficients. If you select **Specify via dialog**, use the **Coefficient for separable filter** parameter to enter a vector of separable filter coefficients.

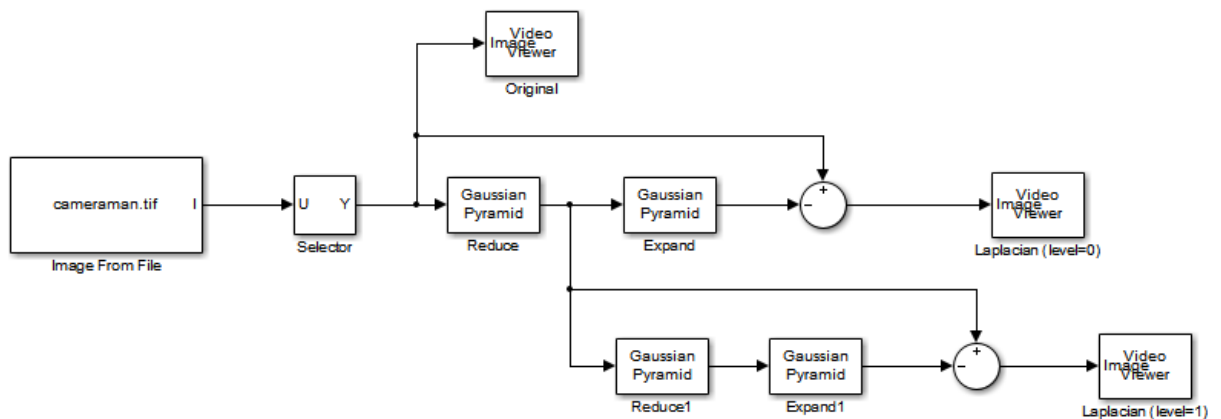
## Examples

The following example model shows how to construct a Laplacian pyramid:

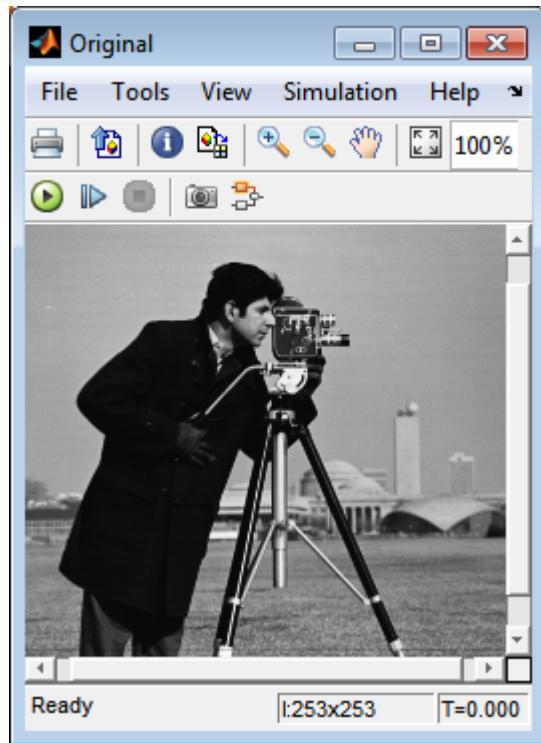
1 Open this model by typing

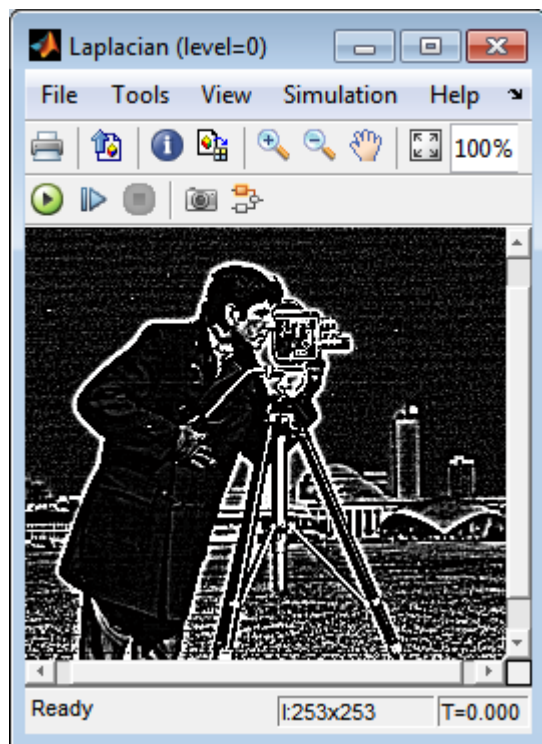
```
ex_laplacian
```

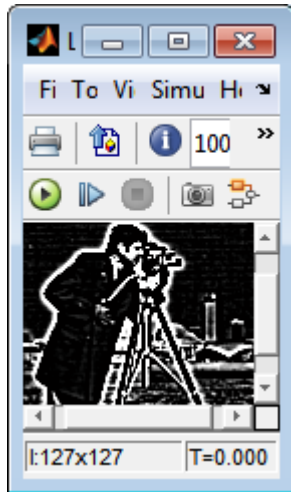
at the MATLAB command prompt.



2 Run the model to see the following results.



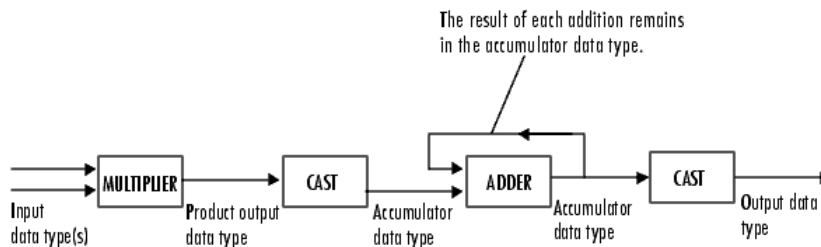




You can construct a Laplacian pyramid if the dimensions of the input image, R-by-C, satisfy  $R = M_R 2^N + 1$  and  $C = M_C 2^N + 1$ , where  $M_R$ ,  $M_C$ , and  $N$  are integers. In this example, you have an input matrix that is 256-by-256. If you set  $M_R$  and  $M_C$  equal to 63 and  $N$  equal to 2, you find that the input image needs to be 253-by-253. So you use a Submatrix block to crop the dimensions of the input image to 253-by-253.

## Fixed-Point Data Types

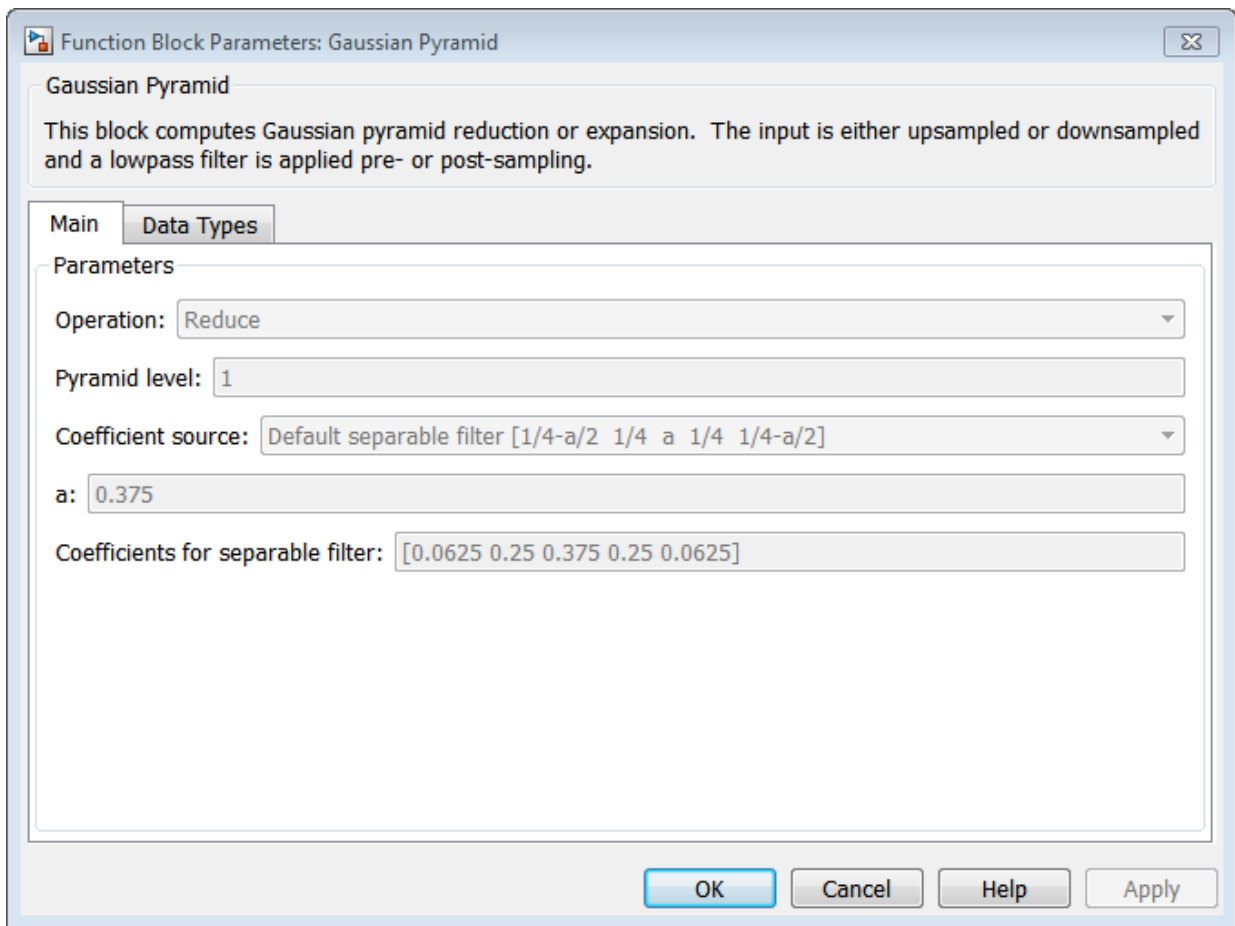
The following diagram shows the data types used in the Gaussian Pyramid block for fixed-point signals:



You can set the coefficients table, product output, accumulator, and output data types in the block mask.

## Dialog Box

The **Main** pane of the Gaussian Pyramid dialog box appears as shown in the following figure.



### Operation



Specify whether you want to reduce or expand the input image.

**Pyramid level**

Specify the number of times the block upsamples or downsamples each dimension of the image by a factor of 2.

**Coefficient source**

Determine how to specify the coefficients of the lowpass filter. Your choices are **Default separable filter**  $[1/4 - a/2 \ 1/4 \ a \ 1/4 \ 1/4 - a/2]$  or **Specify via dialog**.

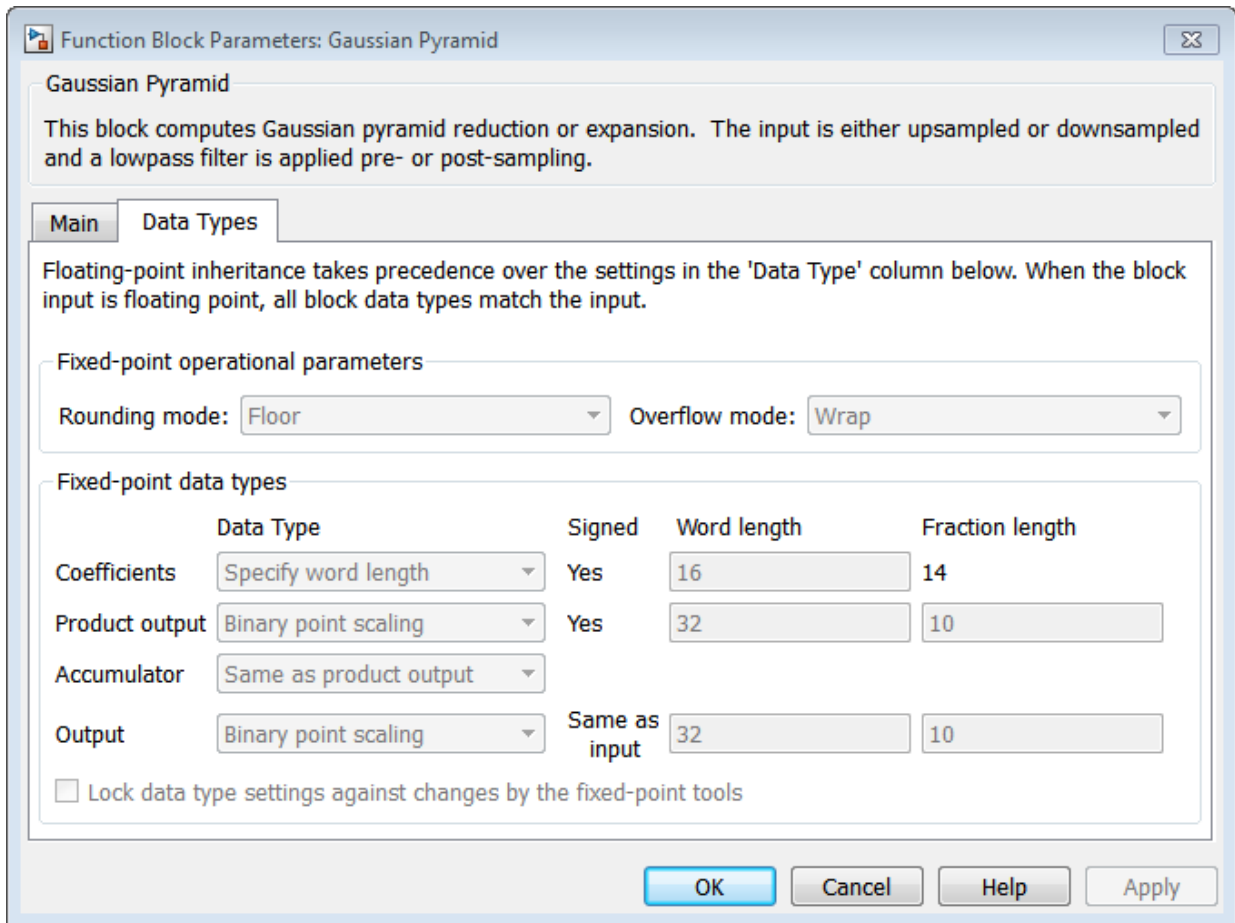
**a**

Enter a scalar value that defines the coefficients in the default separable filter  $[1/4 - a/2 \ 1/4 \ a \ 1/4 \ 1/4 - a/2]$ . This parameter is visible if, for the **Coefficient source** parameter, you select **Default separable filter**  $[1/4 - a/2 \ 1/4 \ a \ 1/4 \ 1/4 - a/2]$ .

**Coefficients for separable filter**

Enter a vector of separable filter coefficients. This parameter is visible if, for the **Coefficient source** parameter, you select **Specify via dialog**.

The **Data Types** pane of the Gaussian Pyramid dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Coefficients

Choose how to specify the word length and the fraction length of the coefficients:

- When you select **Same word length as input**, the word length of the coefficients match that of the input to the block. In this mode, the fraction length

of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select **Specify word length**, you can enter the word length of the coefficients, in bits. The block automatically sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the coefficients, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the coefficients. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

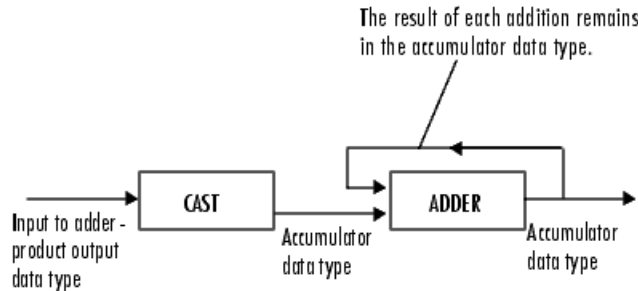
### Product output



As shown in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate the product output word and fraction lengths.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator



As shown in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate the accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## See Also

Resize	Computer Vision System Toolbox software
--------	---

**Introduced before R2006a**

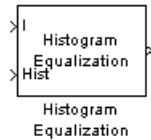
# Histogram Equalization

Enhance contrast of images using histogram equalization

## Library

Analysis & Enhancement

visionanalysis



## Description

The Histogram Equalization block enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image approximately matches a specified histogram.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Hist	Vector of integer values that represents the desired intensity values in each bin	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Output	Matrix of intensity values	Same as I port	No

If the data type of input to the I port is floating point, the input to Hist port must be the same data type. The output signal has the same data type as the input signal.

Use the **Target histogram** parameter to designate the histogram you want the output image to have.

If you select **Uniform**, the block transforms the input image so that the histogram of the output image is approximately flat. Use the **Number of bins** parameter to enter the number of equally spaced bins you want the uniform histogram to have.

If you select **User-defined**, the **Histogram source** and **Histogram** parameters appear on the dialog box. Use the **Histogram source** parameter to select how to specify your histogram. If, for the **Histogram source** parameter, you select **Specify via dialog**, you can use the **Histogram** parameter to enter the desired histogram of the output image. The histogram should be a vector of integer values that represents the desired intensity values in each bin. The block transforms the input image so that the histogram of the output image is approximately the specified histogram.

If, for the **Histogram source** parameter, you select **Input port**, the Hist port appears on the block. Use this port to specify your desired histogram.

---

**Note** The vector input to the Hist port must be normalized such that the sum of the values in all the bins is equal to the number of pixels in the input image. The block does not error if the histogram is not normalized.

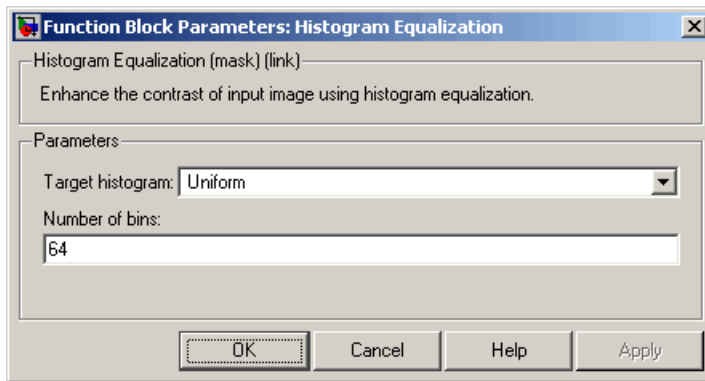
---

## Examples

See “Adjust the Contrast of Intensity Images” and “Adjust the Contrast of Color Images” in the *Computer Vision System Toolbox User's Guide*.

## Dialog Box

The Histogram Equalization dialog box appears as shown in the following figure.



### Target histogram

Designate the histogram you want the output image to have. If you select **Uniform**, the block transforms the input image so that the histogram of the output image is approximately flat. If you select **User-defined**, you can specify the histogram of your output image.

### Number of bins

Enter the number of equally spaced bins you want the uniform histogram to have. This parameter is visible if, for the **Target histogram** parameter, you select **Uniform**.

### Histogram source

Select how to specify your histogram. Your choices are **Specify via dialog** and **Input port**. This parameter is visible if, for the **Target histogram** parameter, you select **User-defined**.

### Histogram

Enter the desired histogram of the output image. This parameter is visible if, for the **Target histogram** parameter, you select **User-defined**.

## See Also

imadjust	Image Processing Toolbox
histeq	Image Processing Toolbox

**Introduced before R2006a**



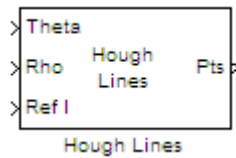
# Hough Lines

Find Cartesian coordinates of lines described by rho and theta pairs

## Library

Transforms

visiontransforms

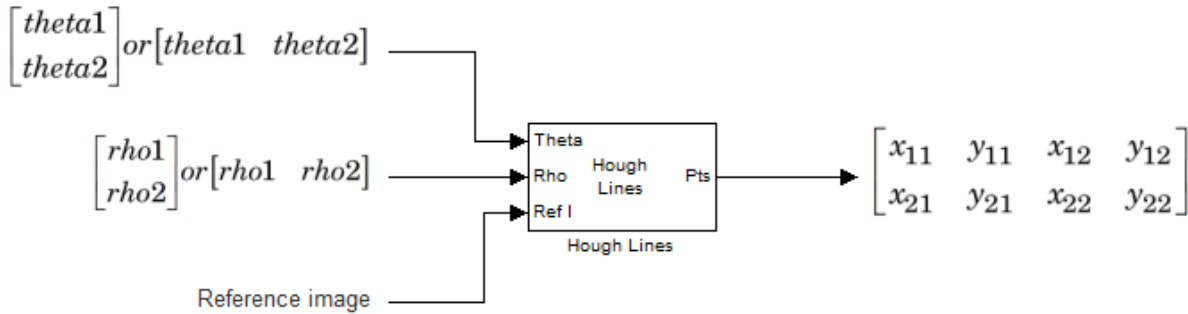


## Description

The Hough Lines block finds the points of intersection between the reference image boundary lines and the line specified by a (rho, theta) pair. The block outputs one-based [x y] coordinates for the points of intersection. The boundary lines indicate the left and right vertical boundaries and the top and bottom horizontal boundaries of the reference image.

If the line specified by the (rho, theta) pair does not intersect two border lines in the reference image, the block outputs the values, [(0,0), (0,0)]. This output intersection value allows the next block in your model to ignore the points. Generally, the Hough Lines block precedes a block that draws a point or shape at the intersection.

The following figure shows the input and output coordinates for the Hough Lines block.

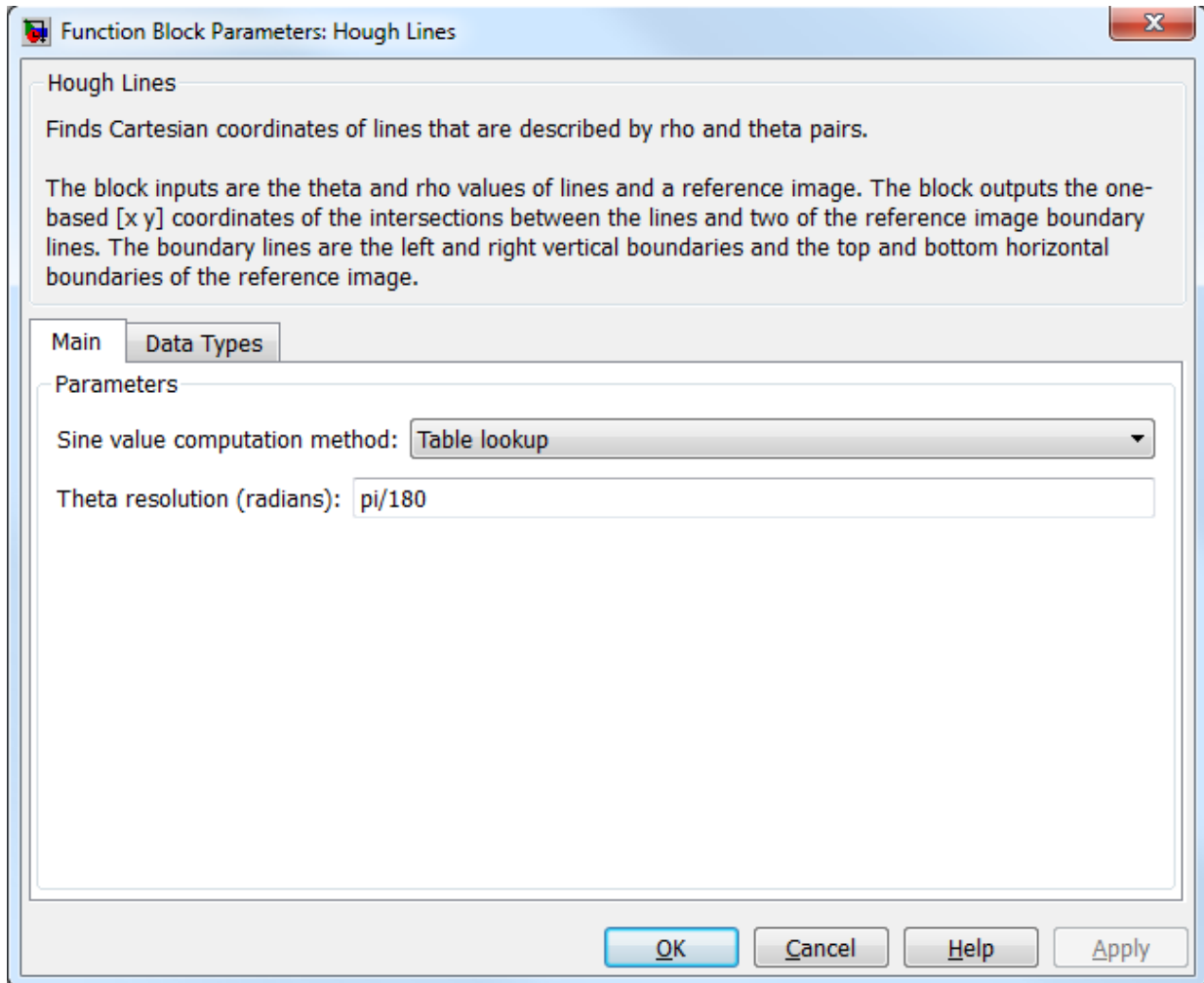


### Port Description

Port	Input/Output	Supported Data Types	Complex Values Supported
Theta	Vector of theta values that represent input lines	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, word length less than or equal to 32)</li> <li>• 8-, 16-, and 32-bit signed integer</li> </ul>	No
Rho	Vector of rho values that represent input lines	Same as Theta port	No
Ref I	Matrix that represents a binary or intensity image or matrix that represents one plane of an RGB image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed-point (signed and unsigned)</li> <li>• Custom data types</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Pts	$M$ -by-4 matrix of intersection values, where $M$ is the number of input lines	<ul style="list-style-type: none"> <li>• 32-bit signed integer</li> </ul>	No

## Dialog Box

The **Main** pane of the Hough Lines dialog box appears as shown in the following figure.



### Sine value computation method

If you select **Trigonometric function**, the block computes sine and cosine values to calculate the intersections of the lines during the simulation. If you select **Table**

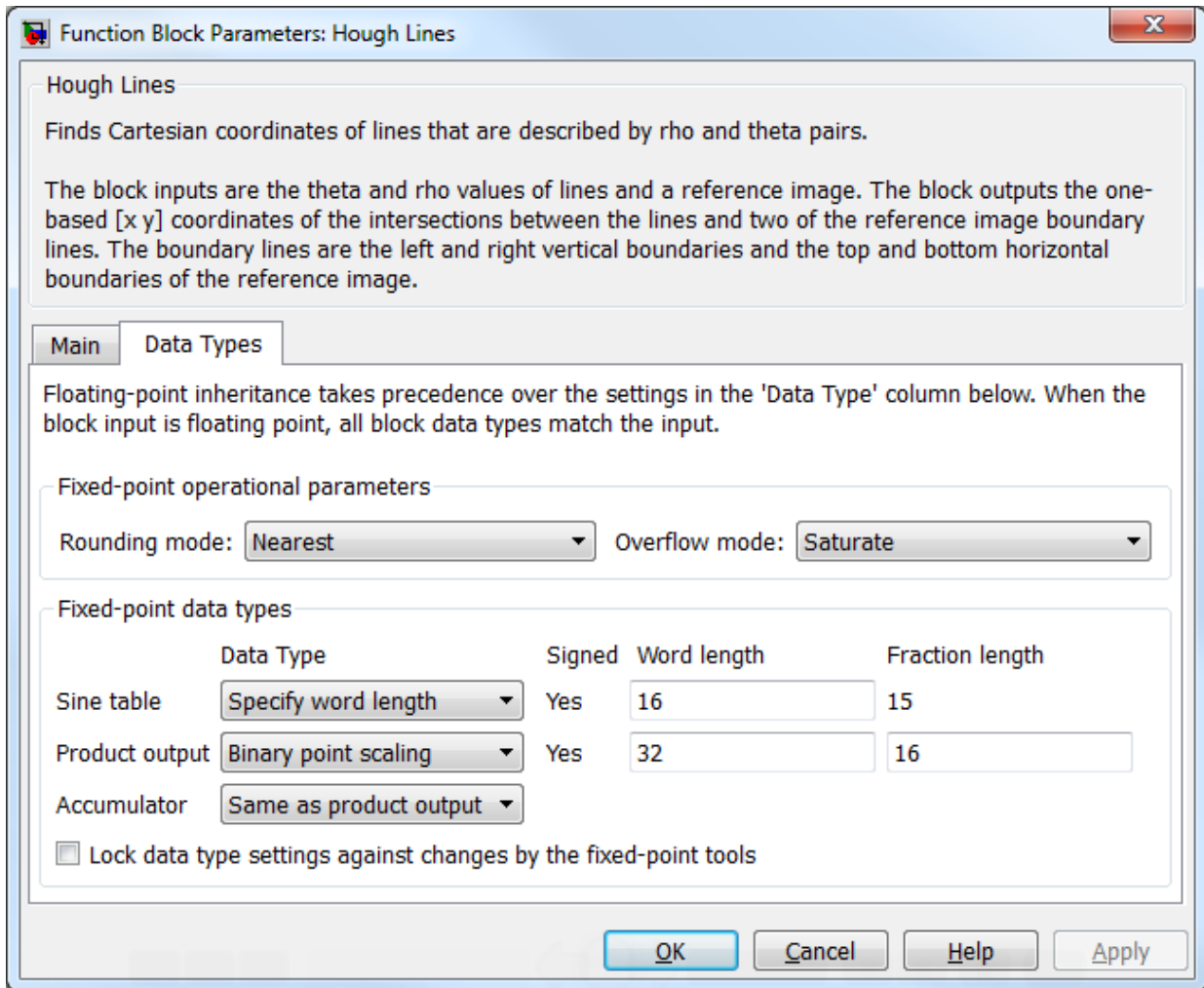
lookup, the block computes and stores the trigonometric values to calculate the intersections of the lines before the simulation starts. In this case, the block requires extra memory.

For floating-point inputs, set the **Sine value computation method** parameter to **Trigonometric function**. For fixed-point inputs, set the parameter to **Table lookup**.

**Theta resolution (radians)**

Use this parameter to specify the spacing of the theta-axis. This parameter appears in the dialog box only if, for the **Sine value computation method** parameter, you select **Table lookup**. parameter appears in the dialog box.

The **Data Types** pane of the Hough Lines dialog box appears as shown in the following figure.

**Rounding mode**

Select the rounding mode for fixed-point operations.

**Overflow mode**

Select the overflow mode for fixed-point operations.

**Sine table**

Choose how to specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one:

When you select **Specify word length**, you can enter the word length of the sine table.

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they saturate and round to **Nearest**.

### **Product output**

Use this parameter to specify how to designate this product output word and fraction lengths:

When you select **Same as first input**, the characteristics match the characteristics of the first input to the block.

When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.

When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. All signals in the Computer Vision System Toolbox blocks have a bias of 0.

See “Multiplication Data Types” for illustrations depicting the use of the product output.

### **Accumulator**

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

When you select **Same as product output** the characteristics match the characteristics of the product output.

When you select **Binary point scaling**, you can enter the **Word length** and the **Fraction length** of the accumulator, in bits.

When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the **Accumulator**. All signals in the Computer Vision System Toolbox software have a bias of 0.

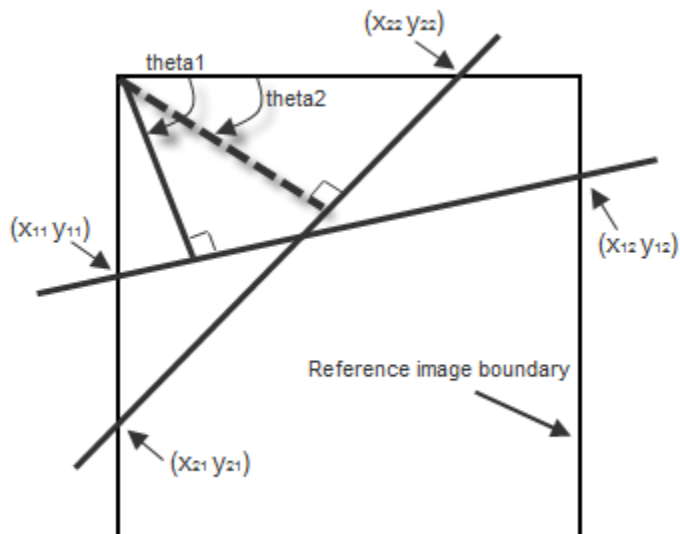
See “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## Examples

The following figure shows Line 1 intersecting the boundaries of the reference image at  $[(x_{11}, y_{11}) (x_{12}, y_{12})]$  and Line 2 intersecting the boundaries at  $[(x_{21}, y_{21}) (x_{22}, y_{22})]$



See “Detect Lines in Images” in the *Computer Vision System Toolbox User Guide*.

## See Also

Find Local Maxima	Computer Vision System Toolbox
Hough Transform	Computer Vision System Toolbox

**Introduced before R2006a**

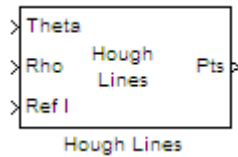


# Hough Lines (To Be Removed)

Find Cartesian coordinates of lines described by rho and theta pairs

## Library

Transforms



## Description

---

**Note:** This Hough Lines block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Hough Lines block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

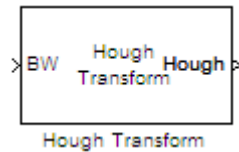
# Hough Transform

Find lines in images

## Library

Transforms

visiontransforms



## Description

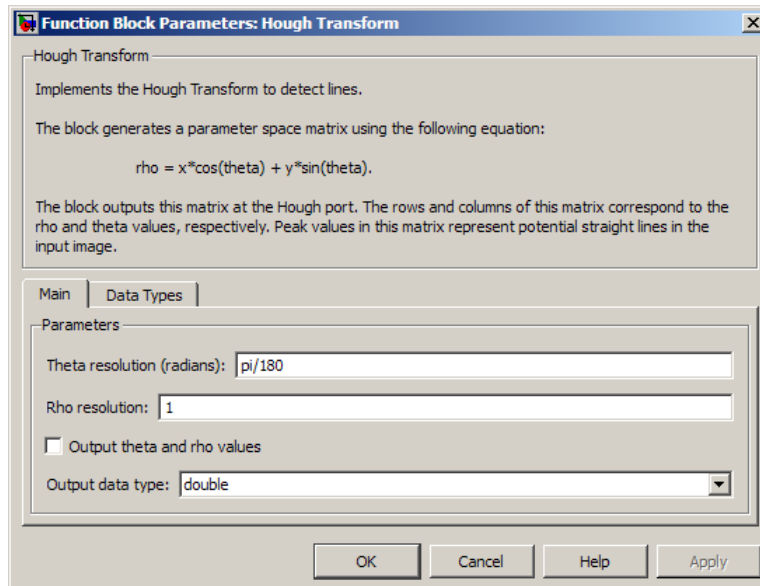
Use the Hough Transform block to find lines in an image. The block outputs the Hough space matrix and, optionally, the *rho*-axis and *theta*-axis vectors. Peak values in the matrix represent potential lines in the input image. Generally, the Hough Transform block precedes the Hough Lines block which uses the output of this block to find lines in an image. You can instead use a custom algorithm to locate peaks in the Hough space matrix in order to identify potential lines.

Port	Input/Output	Supported Data Types	Supported Complex Values
BW	Matrix that represents a binary image	Boolean	No
Hough	Parameter space matrix	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (unsigned, fraction length equal to 0)</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Theta	Vector of theta values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed)</li> </ul>	No

Port	Input/Output	Supported Data Types	Supported Complex Values
		<ul style="list-style-type: none"> <li>8-, 16-, 32-bit signed integer</li> </ul>	
Rho	Vector of rho values	Same as Theta port	No

## Dialog Boxes

The **Main** pane of the Hough Transform dialog box appears as shown in the following figure.



### Theta resolution (radians)

Specify the spacing of the Hough transform bins along the *theta*-axis.

### Rho resolution (pixels)

Specify the spacing of the Hough transform bins along the *rho*-axis.

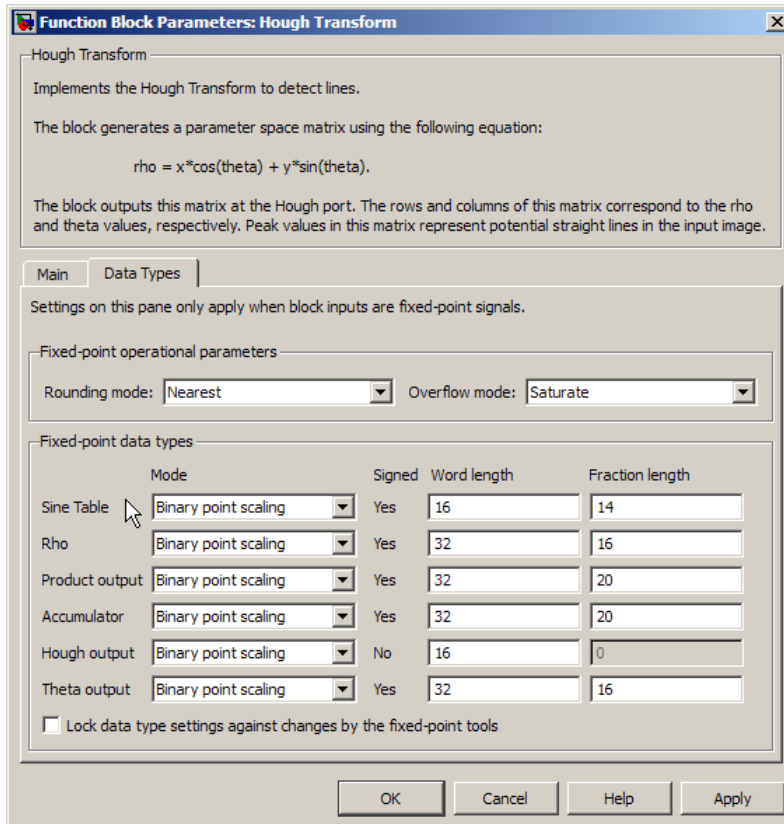
### Output theta and rho values

If you select this check box, the **Theta** and **Rho** ports appear on the block. The block outputs theta and rho-axis vector values at these ports.

## Output data type

Specify the data type of your output signal.

The **Data Types** pane of the Hough Transform block dialog appears as shown in the following figure. The Data Types pane will not show fixed-point parameters when **Output data type** parameter is set to double or single.



## Rounding mode

Select the rounding mode for fixed-point operations.

## Overflow mode

Select the overflow mode for fixed-point operations.

## Sine table

Choose how to specify the word length of the values of the sine table:

- When you select **Binary point scaling**, you can enter the word length of the sine table values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length of the sine table values, in bits.

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they always saturate and round to **Nearest**.

### **Rho**

Choose how to specify the word length and the fraction length of the rho values:

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the rho values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the rho values. All signals in Computer Vision System Toolbox blocks have a bias of 0.

### **Product output**

. Use this parameter to specify how to designate the product output word and fraction lengths:

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. All signals in Computer Vision System Toolbox blocks have a bias of 0.

See “Multiplication Data Types” for illustrations depicting the use of the product output.

### **Accumulator**

Use this parameter to specify how to designate this accumulator word and fraction lengths:

- When you select **Same as product output**, these characteristics match the characteristics of the product output.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.

- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. All signals in Computer Vision System Toolbox blocks have a bias of 0.

See “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block.

### **Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

### **Hough output**

Choose how to specify the word length and fraction length of the Hough output of the block:

- When you select **Binary point scaling**, you can enter the word length of the Hough output, in bits. The fraction length always has a value of 0.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, of the Hough output. The slope always has a value of 0. All signals in Computer Vision System Toolbox blocks have a bias of 0.

### **Theta output**

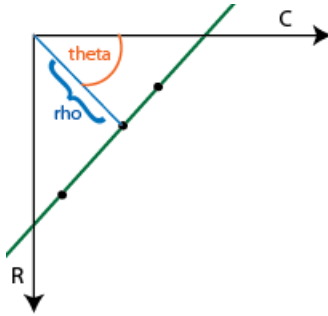
Choose how to specify the word length and fraction length of the theta output of the block:

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the theta output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the theta output. All signals in Computer Vision System Toolbox blocks have a bias of 0.

## **Algorithm**

The Hough Transform block implements the Standard Hough Transform (SHT). The SHT uses the parametric representation of a line:

$$rho = x * \cos(theta) + y * \sin(theta)$$



The variable  $\rho$  indicates the perpendicular distance from the origin to the line.

The variable  $\theta$  indicates the angle of inclination of the normal line from the x-axis.

The range of  $\theta$  is  $-\frac{\pi}{2} \leq \theta < +\frac{\pi}{2}$  with a step-size determined by the **Theta resolution (radians)** parameter. The SHT measures the angle of the line clockwise with respect to the positive x-axis.

The **Hough Transform** block creates an accumulator matrix. The  $(\rho, \theta)$  pair represent the location of a cell in the accumulator matrix. Every valid (logical true) pixel of the input binary image represented by  $(R, C)$  produces a  $\rho$  value for all  $\theta$  values. The block quantizes the  $\rho$  values to the nearest number in the  $\rho$  vector. The  $\rho$  vector depends on the size of the input image and the user-specified  $\rho$  resolution. The block increments a counter (initially set to zero) in those accumulator array cells represented by  $(\rho, \theta)$  pairs found for each pixel. This process validates the point  $(R, C)$  to be on the line defined by  $(\rho, \theta)$ . The block repeats this process for each logical true pixel in the image. The **Hough** block outputs the resulting accumulator matrix.

## Examples

See “Detect Lines in Images” in the *Computer Vision System Toolbox User Guide*.

## See Also

Find Local Maxima	Computer Vision System Toolbox
-------------------	--------------------------------

Hough Lines	Computer Vision System Toolbox
hough	Image Processing Toolbox
houghlines	Image Processing Toolbox
houghpeaks	Image Processing Toolbox

**Introduced before R2006a**



# Image Complement

Compute complement of pixel values in binary or intensity images

## Library

Conversions

visionconversions

## Description



The Image Complement block computes the complement of a binary or intensity image. For binary images, the block replaces pixel values equal to 0 with 1 and pixel values equal to 1 with 0. For an intensity image, the block subtracts each pixel value from the maximum value that can be represented by the input data type and outputs the difference.

For example, suppose the input pixel values are given by  $x(i)$  and the output pixel values are given by  $y(i)$ . If the data type of the input is double or single precision floating-point, the block outputs  $y(i) = 1.0 - x(i)$ . If the input is an 8-bit unsigned integer, the block outputs  $y(i) = 255 - x(i)$ .

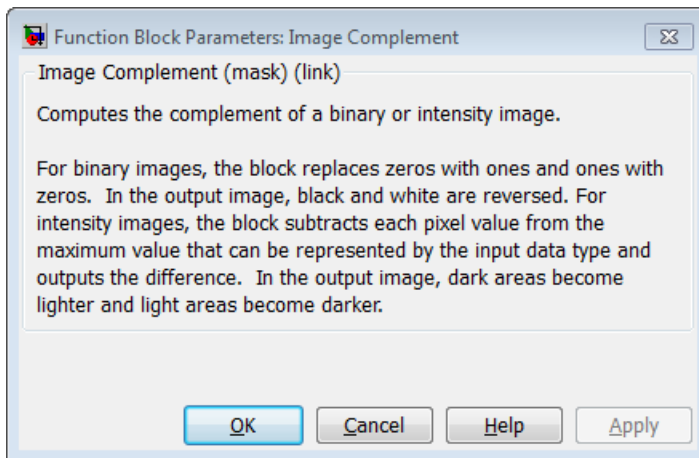
Port	Input/Output	Supported Data Types	Complex Values Supported
Input	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>8-, 16-, 32-bit unsigned integer</li> </ul>	
Output	Complement of a binary or intensity	Same as Input port	No

The dimensions, data type, complexity, and frame status of the input and output signals are the same.

## Dialog Box

The Image Complement dialog box appears as shown in the following figure.



## See Also

Autothreshold	Computer Vision System Toolbox software
Chroma Resampling	Computer Vision System Toolbox software
Color Space Conversion	Computer Vision System Toolbox software
imcomplement	Image Processing Toolbox software

**Introduced before R2006a**

# Image Data Type Conversion

Convert and scale input image to specified output data type

## Library

Conversions

visionconversions



## Description

The Image Data Type Conversion block changes the data type of the input to the user-specified data type and scales the values to the new data type's dynamic range. To convert between data types without scaling, use the [Simulink Data Type Conversion](#) block.

When converting between floating-point data types, the block casts the input into the output data type and clips values outside the range to 0 or 1. When converting to the Boolean data type, the block maps 0 values to 0 and all other values to one. When converting to or between all other data types, the block casts the input into the output data type and scales the data type values into the dynamic range of the output data type. For double- and single-precision floating-point data types, the dynamic range is between 0 and 1. For fixed-point data types, the dynamic range is between the minimum and maximum values that can be represented by the data type.

---

**Note:** This block supports intensity and color images on its ports.

---

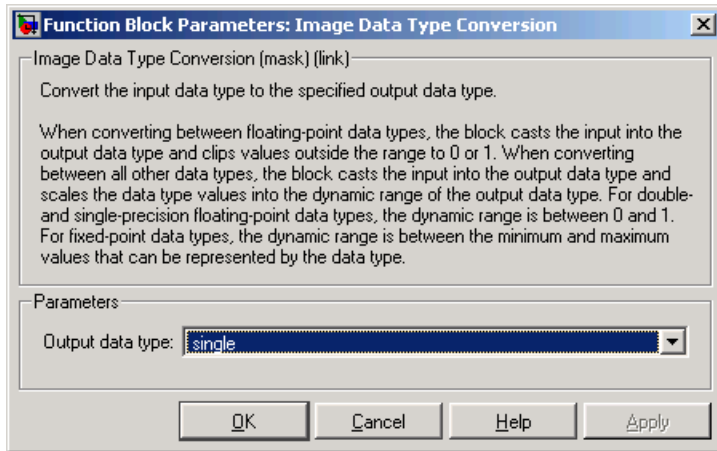
Port	Input/Output	Supported Data Types	Complex Values Supported
Input	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (word length less than or equal to 16)</li> <li>• Boolean</li> <li>• 8-, 16-bit signed integer</li> <li>• 8-, 16-bit unsigned integer</li> </ul>	No
Output	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	Same as Input port	No

The dimensions, complexity, and frame status of the input and output signals are the same.

Use the **Output data type** parameter to specify the data type of your output signal values.

## Dialog Box

The Image Data Type Conversion dialog box appears as shown in the following figure.



## Output data type

Use this parameter to specify the data type of your output signal.

## Signed

Select this check box if you want the output fixed-point data to be signed. This parameter is visible if, for the **Output data type** parameter, you choose **Fixed-point**.

## Word length

Use this parameter to specify the word length of your fixed-point output. This parameter is visible if, for the **Output data type** parameter, you choose **Fixed-point**.

## Fraction length

Use this parameter to specify the fraction length of your fixed-point output. This parameter is visible if, for the **Output data type** parameter, you choose **Fixed-point**.

## See Also

Autothreshold	Computer Vision System Toolbox software
---------------	---

Introduced before R2006a

# Image From File

Import image from image file

## Library

Sources

visionsources



## Description

Use the Image From File block to import an image from a supported image file. For a list of supported file formats, see the `imread` function reference page in the MATLAB documentation. If the image is a  $M$ -by- $N$  array, the block outputs a binary or intensity image, where  $M$  and  $N$  are the number of rows and columns in the image. If the image is a  $M$ -by- $N$ -by- $P$  array, the block outputs a color image, where  $M$  and  $N$  are the number of rows and columns in each color plane,  $P$ .

Port	Output	Supported Data Types	Complex Values Supported
Image	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	Yes
R, G, B	Scalar, vector, or matrix that represents one plane of the	Same as I port	Yes

Port	Output	Supported Data Types	Complex Values Supported
	input RGB video stream. Outputs from the R, G, or B ports have the same dimensions.		

For the Computer Vision System Toolbox blocks to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. If the input pixel values have a different data type than the one you select using the **Output data type** parameter, the block scales the pixel values, adds an offset to the pixel values so that they are within the dynamic range of their new data type, or both.

Use the **File name** parameter to specify the name of the graphics file that contains the image to import into the Simulink modeling and simulation software. If the file is not on the MATLAB path, use the **Browse** button to locate the file. This parameter supports URL paths.

Use the **Sample time** parameter to set the sample period of the output signal.

Use the **Image signal** parameter to specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

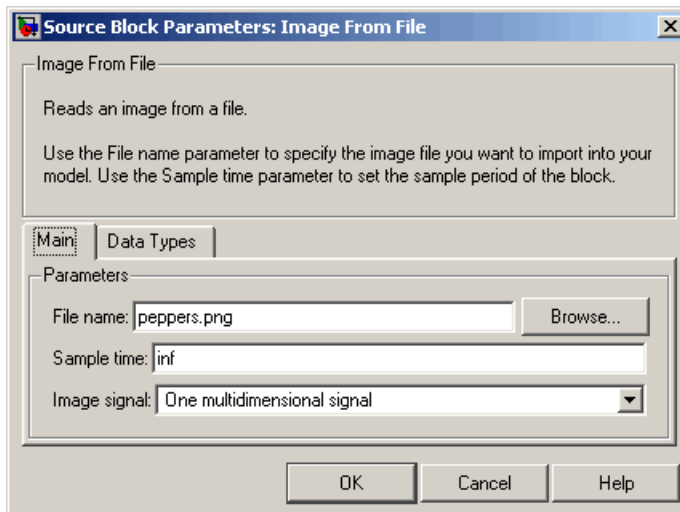
Use the **Output port labels** parameter to label your output ports. Use the spacer character, |, as the delimiter. This parameter is visible if you set the **Image signal** parameter to **Separate color signals**.

On the **Data Types** pane, use the **Output data type** parameter to specify the data type of your output signal.

## Dialog Box

The **Main** pane of the Image From File dialog box appears as shown in the following figure.





### File name

Specify the name of the graphics file that contains the image to import into the Simulink environment.

### Sample time

Enter the sample period of the output signal.

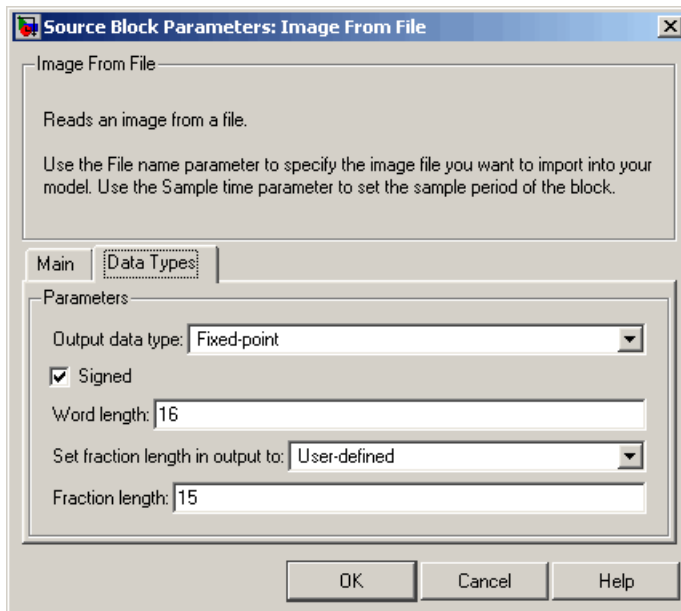
### Image signal

Specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

### Output port labels

Enter the labels for your output ports using the spacer character, |, as the delimiter. This parameter is visible if you set the **Image signal** parameter to **Separate color signals**.

The **Data Types** pane of the Image From File dialog box appears as shown in the following figure.



## Output data type

Specify the data type of your output signal.

## Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible if, from the **Output data type** list, you select **Fixed-point**.

## Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if, from the **Output data type** list, you select **Fixed-point**.

## Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose **Best precision** to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose **User-defined** to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if, from the **Output data type** list, you select **Fixed-point** or when you select **User-defined**.

### Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select **Fixed-point** or **User-defined** for the **Output data type** parameter and **User-defined** for the **Set fraction length in output to** parameter.

### User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from the Fixed-Point Designer™ library. This parameter is only visible when you select **User-defined** for the **Output data type** parameter.

## See Also

From Multimedia File	Computer Vision System Toolbox software
Image From Workspace	Computer Vision System Toolbox software
To Video Display	Video and Image Processing Blockset software
Video From Workspace	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software
<code>im2double</code>	MATLAB software
<code>im2uint8</code>	Image Processing Toolbox software
<code>imread</code>	MATLAB

**Introduced before R2006a**

# Image From Workspace

Import image from MATLAB workspace

## Library

Sources

visionsources



## Description

Use the Image From Workspace block to import an image from the MATLAB workspace. If the image is a M-by-N workspace array, the block outputs a binary or intensity image, where M and N are the number of rows and columns in the image. If the image is a M-by-N-by-P workspace array, the block outputs a color image, where M and N are the number of rows and columns in each color plane, P.

Port	Output	Supported Data Types	Complex Values Supported
Image	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
R, G, B	Scalar, vector, or matrix that represents one plane of the RGB video stream. Outputs	Same as I port	No

Port	Output	Supported Data Types	Complex Values Supported
	from the R, G, or B ports have the same dimensions.		

For the Computer Vision System Toolbox blocks to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. If the input pixel values have a different data type than the one you select using the **Output data type** parameter, the block scales the pixel values, adds an offset to the pixel values so that they are within the dynamic range of their new data type, or both.

Use the **Value** parameter to specify the MATLAB workspace variable that contains the image you want to import into Simulink environment.

Use the **Sample time** parameter to set the sample period of the output signal.

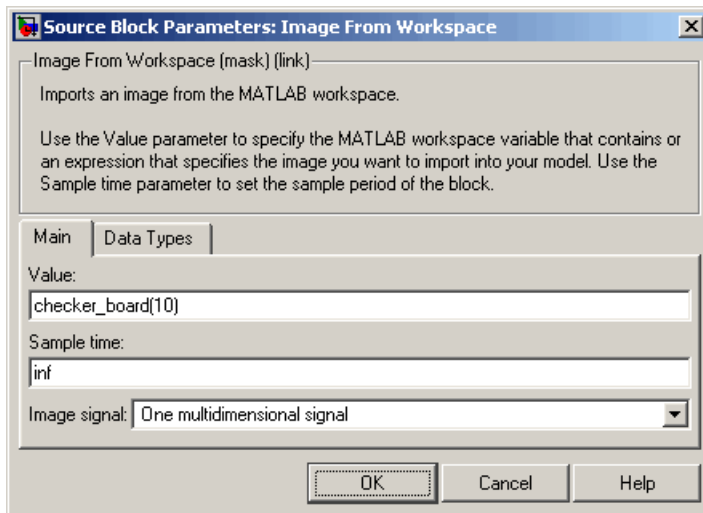
Use the **Image signal** parameter to specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

Use the **Output port labels** parameter to label your output ports. Use the spacer character, |, as the delimiter. This parameter is visible if you set the **Image signal** parameter to **Separate color signals**.

On the **Data Types** pane, use the **Output data type** parameter to specify the data type of your output signal.

## Dialog Box

The **Main** pane of the Image From Workspace dialog box appears as shown in the following figure.



## Value

Specify the MATLAB workspace variable that you want to import into Simulink environment.

## Sample time

Enter the sample period of the output signal.

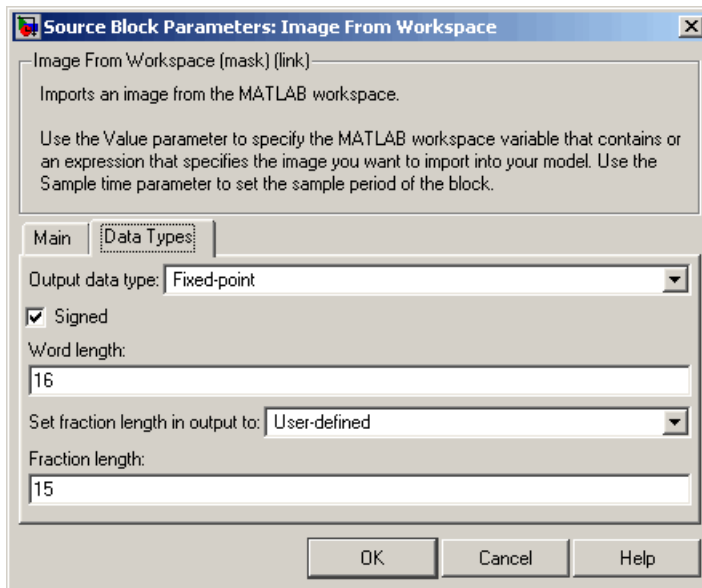
## Image signal

Specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

## Output port labels

Enter the labels for your output ports using the spacer character, |, as the delimiter. This parameter is visible if you set the **Image signal** parameter to **Separate color signals**.

The **Data Types** pane of the Image From Workspace dialog box appears as shown in the following figure.



### Output data type

Specify the data type of your output signal.

### Signed

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned. This parameter is only visible if, from the **Output data type** list, you select **Fixed-point**.

### Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if, from the **Output data type** list, you select **Fixed-point**.

### Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose **Best precision** to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose **User-defined** to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if, from the **Output data type** list, you select **Fixed-point** or when you select **User-defined**.

**Fraction length**

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select **Fixed-point** or **User-defined** for the **Output data type** parameter and **User-defined** for the **Set fraction length in output to** parameter.

**User-defined data type**

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from the Fixed-Point Designer library. This parameter is only visible when you select **User-defined** for the **Output data type** parameter.

**See Also**

From Multimedia File	Computer Vision System Toolbox software
To Video Display	Computer Vision System Toolbox software
Video From Workspace	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software
<code>im2double</code>	Image Processing Toolbox software
<code>im2uint8</code>	Image Processing Toolbox software

**Introduced before R2006a**



# Image Pad

Pad signal along its rows, columns, or both

## Library

Utilities

visionutilities

## Description



The Image Pad block expands the dimensions of a signal by padding its rows, columns, or both. To crop an image, you can use the Simulink **Selector** block, DSP System Toolbox **Submatrix** block, or the Image Processing Toolbox **imcrop** function.

Port	Input/Output	Supported Data Types	Complex Values Supported
Image / I	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal, where $P$ is the number of color planes.	<ul style="list-style-type: none"> <li>• Double-precision floating point.</li> <li>• Single-precision floating point.</li> <li>• Fixed point.</li> <li>• Boolean.</li> <li>• 8-, 16-, 32-bit signed integer.</li> <li>• 8-, 16-, 32-bit unsigned integer.</li> </ul>	Yes
PVal	Scalar value that represents the constant pad value.	Same as I port.	Yes
Output	Padded scalar, vector, or matrix.	Same as I port.	Yes

## Examples

### Pad with a Constant Value

Suppose you want to pad the rows of your input signal with three initial values equal to 0 and your input signal is defined as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Set the Image Pad block parameters as follows:

- **Method** = Constant
- **Pad value source** = Specify via dialog
- **Pad value** = 0
- **Specify** = Output size
- **Add columns to** = Left
- **Output row mode** = User-specified
- **Number of output columns** = 6
- **Add rows to** = No padding

The Image Pad block outputs the following signal:

$$\begin{bmatrix} 0 & 0 & 0 & a_{00} & a_{01} & a_{02} \\ 0 & 0 & 0 & a_{10} & a_{11} & a_{12} \\ 0 & 0 & 0 & a_{20} & a_{21} & a_{22} \end{bmatrix}$$

## Pad by Repeating Border Values

Suppose you want to pad your input signal with its border values, and your input signal is defined as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Set the Image Pad block parameters as follows:

- **Method** = Replicate
- **Specify** = Pad size
- **Add columns to** = Both left and right
- **Number of added columns** = 2
- **Add rows to** = Both top and bottom
- **Number of added rows** = [1 3]

The Image Pad block outputs the following signal:

$$\begin{bmatrix} a_{00} & a_{00} & a_{00} & a_{01} & a_{02} & a_{02} & a_{02} \\ a_{00} & a_{00} & a_{00} & a_{01} & a_{02} & a_{02} & a_{02} \\ a_{10} & a_{10} & a_{10} & a_{11} & a_{12} & a_{12} & a_{12} \\ a_{20} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{22} \\ a_{20} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{22} \\ a_{20} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{22} \\ a_{20} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{22} \end{bmatrix}$$

Input matrix

The border values of the input signal are replicated on the top, bottom, left, and right of the input signal so that the output is a 7-by-7 matrix. The values in the corners of this

output matrix are determined by replicating the border values of the matrices on the top, bottom, left and right side of the original input signal.

### Pad with Mirror Image

Suppose you want to pad your input signal using its mirror image, and your input signal is defined as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Set the Image Pad block parameters as follows:

- **Method** = Symmetric
- **Specify** = Pad size
- **Add columns to** = Both left and right
- **Number of added columns** = [5 6]
- **Add rows to** = Both top and bottom
- **Number of added rows** = 2

The Image Pad block outputs the following signal:

$$\begin{bmatrix} a_{11} & a_{12} & a_{12} & a_{11} & a_{10} & a_{10} & a_{11} & a_{12} & a_{12} & a_{11} & a_{10} & a_{10} & a_{11} & a_{12} \\ a_{01} & a_{02} & a_{02} & a_{01} & a_{00} & a_{00} & a_{01} & a_{02} & a_{02} & a_{01} & a_{00} & a_{00} & a_{01} & a_{02} \\ a_{01} & a_{02} & a_{02} & a_{01} & a_{00} & a_{00} & a_{01} & a_{02} & a_{02} & a_{01} & a_{00} & a_{00} & a_{01} & a_{02} \\ a_{11} & a_{12} & a_{12} & a_{11} & a_{01} & a_{10} & a_{11} & a_{12} & a_{12} & a_{11} & a_{10} & a_{10} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{22} & a_{21} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{21} & a_{20} & a_{20} & a_{21} & a_{22} \\ a_{21} & a_{22} & a_{22} & a_{21} & a_{20} & a_{20} & a_{21} & a_{22} & a_{22} & a_{21} & a_{20} & a_{20} & a_{21} & a_{22} \\ a_{11} & a_{12} & a_{12} & a_{11} & a_{01} & a_{01} & a_{11} & a_{12} & a_{12} & a_{11} & a_{10} & a_{10} & a_{11} & a_{12} \end{bmatrix}$$

Input matrix

The block flips the original input matrix and each matrix it creates about their top, bottom, left, and right sides to populate the 7-by-13 output signal. For example, in the preceding figure, you can see how the block flips the input matrix about its right side to create the matrix directly to its right.

## Pad Using a Circular Repetition of Elements

Suppose you want to pad your input signal using a circular repetition of its values. Your input signal is defined as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Set the Image Pad block parameters as follows:

- **Method** = Circular
- **Specify** = Output size
- **Add columns to** = Both left and right
- **Number of output columns** = 9
- **Add rows to** = Both top and bottom
- **Number of output rows** = 9

The Image Pad block outputs the following signal:

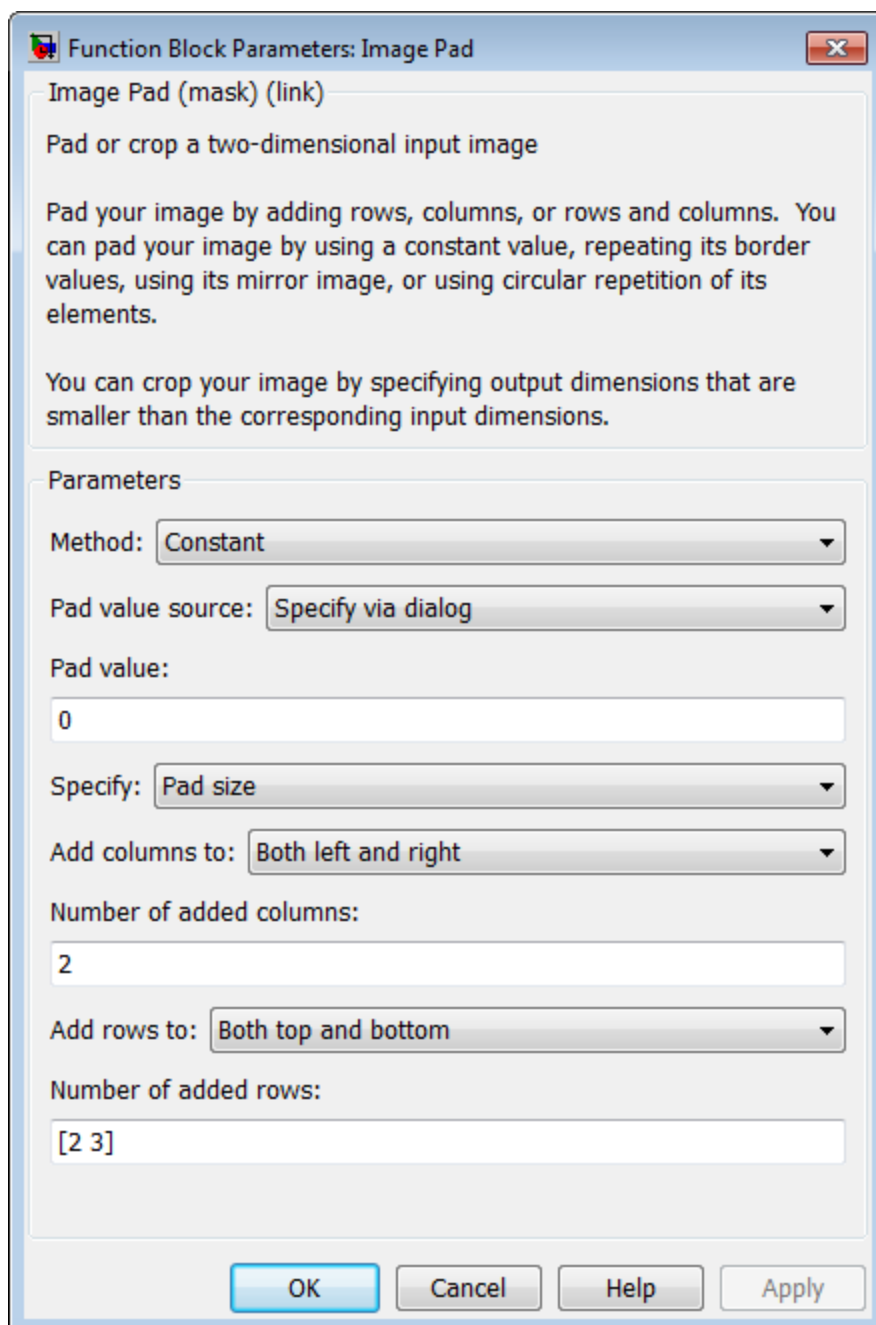
$$\begin{bmatrix}
 a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} \\
 a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} \\
 a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22} \\
 \hline
 a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} \\
 a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} \\
 a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22} \\
 \hline
 a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} & a_{00} & a_{01} & a_{02} \\
 a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} & a_{10} & a_{11} & a_{12} \\
 a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22} & a_{20} & a_{21} & a_{22}
 \end{bmatrix}$$

Input matrix

The block repeats the values of the input signal in a circular pattern to populate the 9-by-9 output matrix.

## Dialog Box

The Image Pad dialog box appears as shown in the following figure.



## Method

Specify how you want the block to pad your signal. The data type of the input signal is the data type of the output signal.

Use the **Method** parameter to specify how you pad the input signal.

- **Constant** — Pad with a constant value
- **Replicate** — Pad by repeating its border values
- **Symmetric** — Pad with its mirror image
- **Circular** — Pad using a circular repetition of its elements

If you set the **Method** parameter to **Constant**, the **Pad value source** parameter appears on the dialog box.

- **Input port** — The PVal port appears on the block. Use this port to specify the constant value with which to pad your signal
- **Specify via dialog** — The **Pad value** parameter appears in the dialog box. Enter the constant value with which to pad your signal.

## Pad value source

If you select **Input port**, the PVal port appears on the block. Use this port to specify the constant value with which to pad your signal. If you select **Specify via dialog**, the **Pad value** parameter becomes available. This parameter is visible if, for the **Method** parameter, you select **Constant**.

## Pad value

Enter the constant value with which to pad your signal. This parameter is visible if, for the **Pad value source** parameter, you select **Specify via dialog**. This parameter is tunable.

## Specify

If you select **Pad size**, you can enter the size of the padding in the horizontal and vertical directions.

If you select **Output size**, you can enter the total number of output columns and rows. This setting enables you to pad the input signal. See the previous section for descriptions of the **Add columns to** and **Add rows to** parameters.

## Add columns to

The **Add columns to** parameter controls the padding at the left, right or both sides of the input signal.



- **Left** — The block adds additional columns on the left side.
- **Right** — The block adds additional columns on the right side.
- **Both left and right** — The block adds additional columns to the left and right side.
- **No padding** — The block does not change the number of columns.

Use the **Add columns to** and **Number of added columns** parameters to specify the size of the padding in the horizontal direction. Enter a scalar value, and the block adds this number of columns to the left, right, or both sides of your input signal. If you set the **Add columns to** parameter to **Both left and right**, you can enter a two element vector. The left element controls the number of columns the block adds to the left side of the signal; the right element controls the number of columns the block adds to the right side of the signal.

### Output row mode

Use the **Output row mode** parameter to describe how to pad the input signal.

- **User-specified** — Use the **Number of output rows** parameter to specify the total number of rows.
- **Next power of two** — The block pads the input signal along the rows until the length of the rows is equal to a power of two. When the length of the input signal's rows is equal to a power of two, the block does not pad the input signal's rows.

### Number of added columns

This parameter controls how many columns are added to the right and/or left side of your input signal. Enter a scalar value, and the block adds this number of columns to the left, right, or both sides of your signal. If, for the **Add columns to** parameter you select **Both left and right**, enter a two-element vector. The left element controls the number of columns the block adds to the left side of the signal and the right element controls how many columns the block adds to the right side of the signal. This parameter is visible if, for the **Specify** parameter, you select **Pad size**.

### Add rows to

The **Add rows to** parameter controls the padding at the top and bottom of the input signal.

- **Top** — The block adds additional rows to the top.
- **Bottom** — The block adds additional rows to the bottom.
- **Both top and bottom** — The block adds additional rows to the top and bottom.
- **No padding** — The block does not change the number of rows.

Use the **Add rows to** and **Number of added rows** parameters to specify the size of the padding in the vertical direction. Enter a scalar value, and the block adds this number of rows to the top, bottom, or both of your input signal. If you set the **Add rows to** parameter to **Both top and bottom**, you can enter a two element vector. The left element controls the number of rows the block adds to the top of the signal; the right element controls the number of rows the block adds to the bottom of the signal.

### **Output column mode**

Describe how to pad the input signal. If you select **User-specified**, the **Row size** parameter appears on the block dialog box. If you select **Next power of two**, the block pads the input signal along the rows until the length of the rows is equal to a power of two. This parameter is visible if, for the **Specify** parameter, you select **Output size**.

Use the **Output column mode** parameter to describe how to pad the input signal.

- **User-specified** — Use the **Number of column rows** parameter to specify the total number of columns.
- **Next power of two** — The block pads the input signal along the columns until the length of the columns is equal to a power of two. When the length of the input signal's columns is equal to a power of two, the block does not pad the input signal's columns.

### **Number of added rows**

This parameter controls how many rows are added to the top, bottom, or both of your input signal. Enter a scalar value and the block adds this number of columns to the top, bottom, or both of your signal. If, for the **Add rows to** parameter you select **Both top and bottom**, enter a two-element vector. The left element controls the number of rows the block adds to the top of the signal and the right element controls how many rows the block adds to the bottom of the signal. This parameter is visible if you set the **Specify** parameter to **Pad size**.

### **Action when truncation occurs**

The following options are available for the **Action when truncation occurs** parameter:

- **None** — Select this option when you do not want to be notified that the input signal is truncated.
- **Warning** — Select this option when you want to receive a warning in the MATLAB Command Window when the input signal is truncated.

- **Error** — Select this option when you want an error dialog box displayed and the simulation terminated when the input signal is truncated.

**See Also**

Selector | Submatrix | imcrop

**Introduced in R2007a**

# Insert Text

Draw text on image or video stream.

## Library

Text & Graphics

visiontextngfix



## Description

The Insert Text block draws formatted text or numbers on an image or video stream. The block uses the FreeType 2.3.5 library, an open-source font engine, to produce stylized text bitmaps. To learn more about the FreeType Project, visit <http://www.freetype.org/>. The Insert Text block does not support character sets other than ASCII.

The Insert Text block lets you draw one or more instances of one or more strings, including:

- A single instance of one text string
- Multiple instances of one text string
- Multiple instances of text, with a different text string at each location

## Port Description

Port	Description	Supported Data Types
Image	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ represents the number of color planes.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point(signed, word length less than or equal to 32.)</li> <li>• Boolean</li> </ul>

Port	Description	Supported Data Types
		<ul style="list-style-type: none"> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>
R, G, B	Matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports have the same dimensions and data type.	Same as Input port
Select	One-based index value that indicates which text string to display.	<ul style="list-style-type: none"> <li>• Double-precision floating point. (This data type is only supported if the input to the I or R, G, and B ports is a floating-point data type.)</li> <li>• Single-precision floating point. (This data type is only supported if the input to the I or R, G, and B ports is a floating-point data type.)</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>

Port	Description	Supported Data Types
Variab	<p>Vector or matrix whose values are used to replace ANSI C printf-style format specifications.</p>	<p>The data types supported by this port depend on the conversion specification you are using in the <b>Text</b> parameter.</p> <p>%d, %i, and %u:</p> <ul style="list-style-type: none"> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul> <p>%c and %s:</p> <ul style="list-style-type: none"> <li>• 8-bit unsigned integer</li> </ul> <p>%f:</p> <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> <p>%o, %x, %X, %e, %E, %g, and %G:</p> <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>
Color	<p>Intensity input — Scalar value used for all strings or a vector of intensity values whose length is equal to the number of strings.</p> <p>Color input — Three-element vector that specifies one color for all strings or a <math>M</math>-by-3 matrix of color values, where <math>M</math> represents the number of strings.</p>	<p>Same as Input port (The input to this port must be the same data type as the input to the Input port.)</p>

Port	Description	Supported Data Types
Location	$M$ -by-2 matrix of one-based [x y] coordinates, where $M$ represents the number of text strings to insert. <b>Location</b> specifies the top-left corner of the text string bounding box.	<ul style="list-style-type: none"> <li>• Double-precision floating point. (This data type is only supported if the input to the I or R, G, and B ports is a floating-point data type.)</li> <li>• Single-precision floating point. (This data type is only supported if the input to the I or R, G, and B ports is a floating-point data type.)</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>
Opacity	Scalar value that is used for all strings or vector of opacity values whose length is equal to the number of strings.	<ul style="list-style-type: none"> <li>• Double-precision floating point. (This data type is only supported if the input to the Input or R, G, and B ports is a double-precision floating-point data type.)</li> <li>• Single-precision floating point. (This data type is only supported if the input to the I or R, G, and B ports is a single-precision floating-point data type.)</li> <li>• <code>ufix8_En7</code> (This data type is only supported if the input to the I or R, G, and B ports is a fixed-point data type.)</li> </ul>

## Row-Major Data Format

MATLAB and the Computer Vision System Toolbox blocks use column-major data organization. However, the Insert Text block gives you the option to process data that is stored in row-major format. When you select the **Input image is transposed (data order is row major)** check box, the block assumes that the input buffer contains contiguous data elements from the first row first, then data elements from the second row second, and so on through the last row. Use this functionality only when you meet all the following criteria:

- You are developing algorithms to run on an embedded target that uses the row-major format.

- You want to limit the additional processing required to take the transpose of signals at the interfaces of the row-major and column-major systems.

When you use the row-major functionality, you must consider the following issues:

- When you select this check box, the first two signal dimensions of the Insert Text block's input are swapped.
- All Computer Vision System Toolbox software blocks can be used to process data that is in the row-major format, but you need to know the image dimensions when you develop your algorithms.

For example, if you use the 2-D FIR Filter block, you need to verify that your filter coefficients are transposed. If you are using the Rotate block, you need to use negative rotation angles, etc.

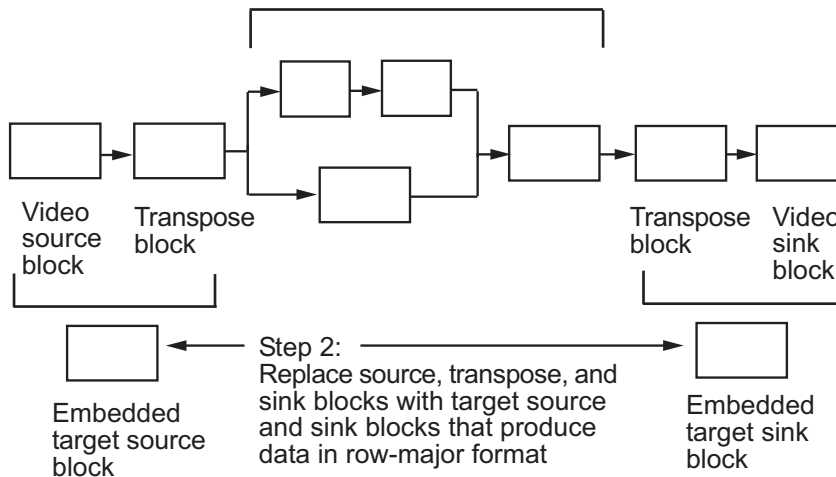
- Only three blocks have the **Input image is transposed (data order is row major)** check box. They are the Chroma Resampling, Deinterlacing, and Insert Text blocks. You need to select this check box to enable row-major functionality in these blocks. All other blocks must be properly configured to process data in row-major format.

Use the following two-step workflow to develop algorithms in row-major format to run on an embedded target.

Step 1:

Create block diagram

Algorithm blocks

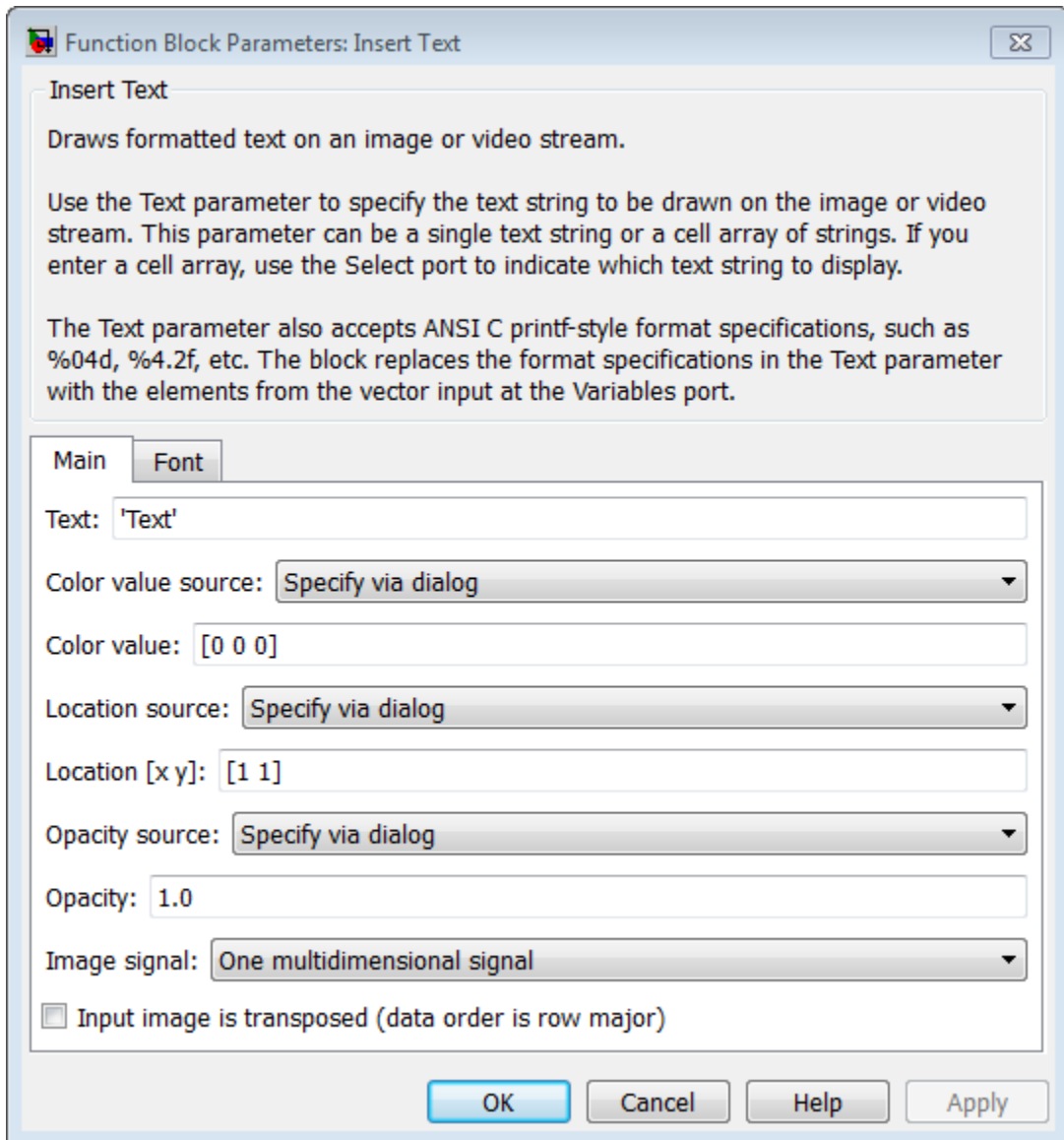




See the DM642 EVM Video ADC and DM642 EVM Video DAC reference pages.

## Dialog Box

The **Main** pane of the Insert Text dialog box appears as shown in the following figure.



## Text

Specify the text string to be drawn on the image or video stream. This parameter can be a single text string, such as 'Figure1', a cell array of strings, such as {'Figure1', 'Figure2'}, or an ANSI C printf-style format specifications, such as %s. To create a **Select** port enter a cell array of strings. To create a **Variables** port, enter ANSI C printf-style format specifications, such as %d, %f, or %s.

When you enter a cell array of strings, the Insert Text block does not display all of the strings simultaneously. Instead, the **Select** port appears on the block to let you indicate which text string to display. The input to this port must be a scalar value, where 1 indicates the first string. If the input is less than 1 or greater than one less than the number of strings in the cell array, no text will be drawn on the image or video frame.

When you enter ANSI C printf-style format specifications, such as %d, %f, or %s, the **Variables** port appears on the block. The block replaces the format specifications in the **Text** parameter with each element of the input vector. Use the %S option to specify a set of text strings for the block to display simultaneously at different locations. For example, using a **Constant** block, enter [uint8('Text1') 0 uint8('Text2')] for the **Constant value** parameter. The following table summarizes the supported conversion specifications.

#### Text Parameter Supported Conversion Specifications

Supported specifications	Support for multiple instances of the same specification	Support for mixed specifications
%d, %i, %u, %c, %f, %o, %x, %X, %e, %E, %g, and %G	Yes	No
%s	No	No

#### Color value source

Select where to specify the text color. Your choices are:

- **Specify via dialog** — the **Color value** parameter appears on the dialog box.
- **Input port** — the Color port appears on the block.

#### Color value

Specify the intensity or color of the text. This parameter is visible if, for the **Color source** parameter, you select **Specify via dialog**. Tunable.

The following table describes how to format the color of the text strings, which depend on the block input and the number of strings you want to insert. Color values for a floating-point data type input image must be between 0 and 1. Color values for an 8-bit unsigned integer data type input image must be between 0 and 255.

### Text String Color Values

Block Input	One Text String	Multiple Text Strings
Intensity image	<b>Color value</b> parameter or the input to the <b>Color</b> port specified as a scalar intensity value	<b>Color value</b> parameter or the input to the <b>Color</b> port specified as a vector of intensity values whose length is equal to the number of strings
Color image	<b>Color value</b> parameter or the input to the <b>Color</b> port specified as an RGB triplet that defines the color of the text	<b>Color value</b> parameter or the input to the <b>Color</b> port specified as an $M$ -by-3 matrix of color values, where $M$ represents the number of strings

### Location source

Indicate where you want to specify the text location. Your choices are:

- **Specify via dialog** — the **Location [x y]** parameter appears on the dialog box.
- **Input port** — the Location port appears on the block.

### Location [x y]

Specify the text location. This parameter is visible if, for the **Location source** parameter, you select **Specify via dialog**. Tunable.

The following table describes how to format the location of the text strings depending on the number of strings you specify to insert. You can specify more than one location regardless of how many text strings you specify, but the only way to get a different text string at each location is to use the %s option for the **Text** parameter to specify a set of text strings. You can enter negative values or values that exceed the dimensions of the input image or video frame, but the text might not be visible.

## Location Parameter Text String Insertion

Parameter	One Instance of One Text String	Multiple Instances of the Same Text String	Multiple Instances of Unique Text Strings
<b>Location [x y]</b> parameter setting or the input to the Location port	Two-element vector of the form [x y] that indicates the top- left corner of the text bounding box.	$M$ -by-2 matrix, where $M$ represents the number of locations at which to display the text string. Each row contains the coordinates of the top- left corner of the text bounding box for the string, e.g., [ $x_1$ $y_1$ ; $x_2$ $y_2$ ]	$M$ -by-2 matrix, where $M$ represents the number of text strings. Each row contains the coordinates of the top- left corner of the text bounding box for the string, e.g., [ $x_1$ $y_1$ ; $x_2$ $y_2$ ].

### Opacity source

Indicate where you want to specify the text's opaqueness. Your choices are:

- **Specify via dialog** — the **Opacity** parameter appears on the dialog box.
- **Input port** — the Opacity port appears on the block.

### Opacity

Specify the opacity of the text. This parameter is visible if, for the **Opacity source** parameter, you select **Specify via dialog**. Tunable.

The following table describes how to format the opacity of the text strings depending on the number of strings you want to insert.

### Text String Opacity Values

Parameter	One Text String	Multiple Text Strings
<b>Opacity</b> parameter setting or the input to the Opacity port	Scalar value between 0 and 1, where 0 is translucent and 1 is opaque	Vector whose length is equal to the number of strings

Use the **Image signal** parameter to specify how to input and output a color video signal:

- **One multidimensional signal** — the block accepts an  $M$ -by- $N$ -by- $P$  color video signal, where  $P$  is the number of color planes, at one port.
- **Separate color signals** — additional ports appear on the block. Each port accepts one  $M$ -by- $N$  plane of an RGB video stream.

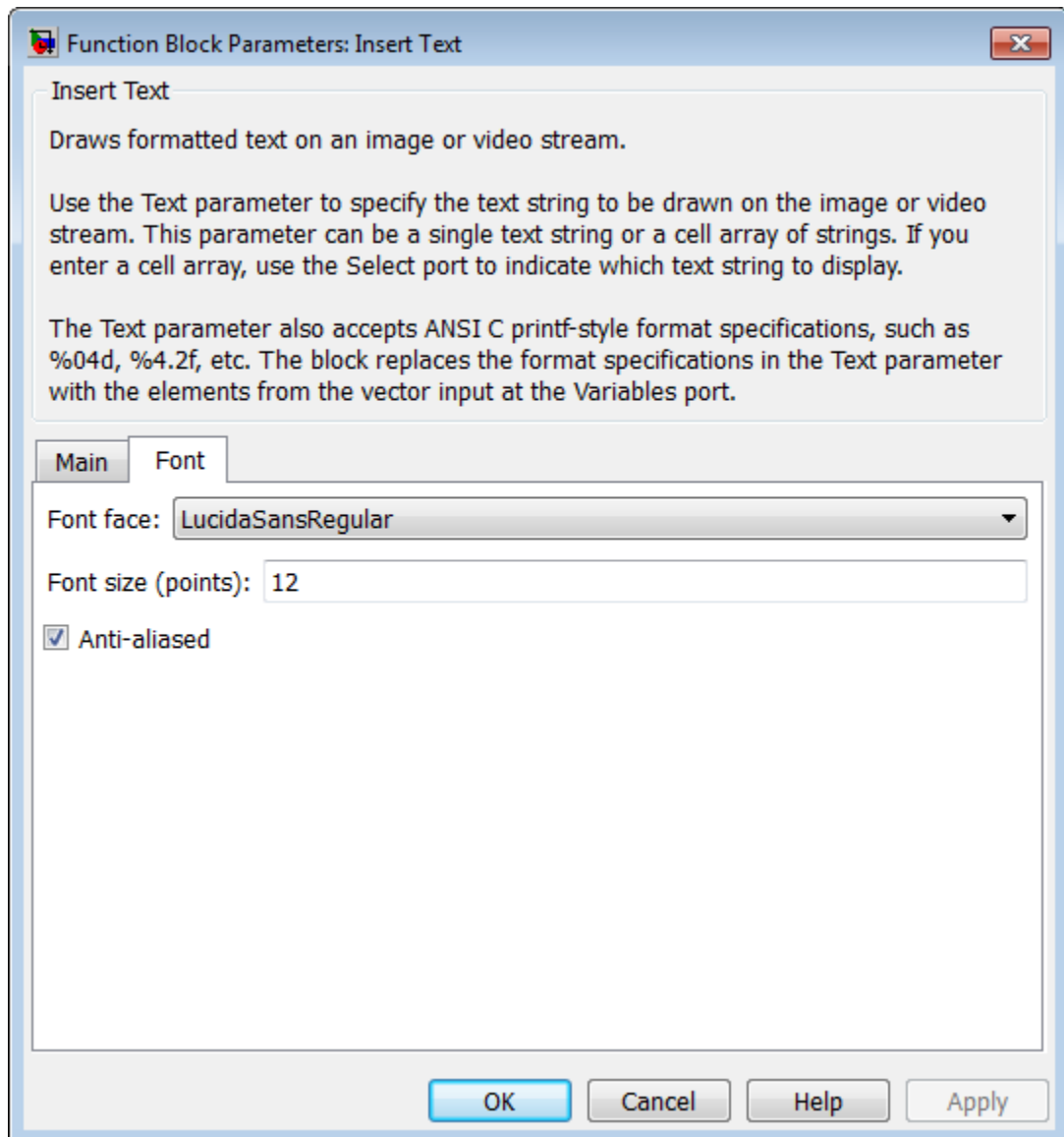
### **Image signal**

Specify how to input and output a color video signal. If you select **One multidimensional signal**, the block accepts an  $M$ -by- $N$ -by- $P$  color video signal, where  $P$  is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port accepts one  $M$ -by- $N$  plane of an RGB video stream.

### **Input image is transposed (data order is row major)**

When you select this check box, the block assumes that the input buffer contains data elements from the first row first, then data elements from the second row second, and so on through the last row.

The **Font** pane of the Insert Text dialog box appears as shown in the following figure.



**Font face**

Specify the font of your text. The block populates this list with the fonts installed on your system. On Windows, the block searches the system registry for font files. On UNIX, the block searches the X Server's font path for font files.

**Font size (points)**

Specify the font size.

**Anti-aliased**

Select this check box if you want the block to smooth the edges of the text. This can be computationally expensive. If you want your model to run faster, clear this check box.

## Examples

- “Annotate Video Files with Frame Numbers”

## See Also

Draw Shapes	Computer Vision System Toolbox
Draw Markers	Computer Vision System Toolbox

**Introduced in R2013a**



# Insert Text (To Be Removed)

Draw text on image or video stream.

## Library

Text & Graphics



## Description

---

**Note:** This Insert Text block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Insert Text block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

# Label

Label connected components in binary images

## Library

Morphological Operations

visionmorphops



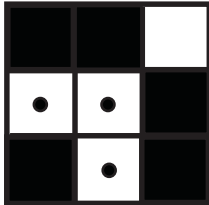
## Description

The Label block labels the objects in a binary image, BW. The background is represented by pixels equal to 0 (black) and objects are represented by pixels equal to 1 (white). At the Label port, the block outputs a label matrix that is the same size as the input matrix. In the label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. At the Count port, the block outputs a scalar value that represents the number of labeled objects.

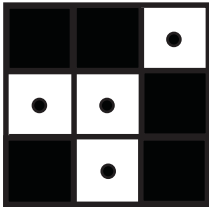
Port	Input/Output	Supported Data Types	Complex Values Supported
BW	Vector or matrix that represents a binary image	Boolean	No
Label	Label matrix	<ul style="list-style-type: none"> <li>8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Count	Scalar that represents the number of labeled objects	Same as Label port	No

Use the **Connectivity** parameter to define which pixels are connected to each other. If you want a pixel to be connected to the other pixels located on the top, bottom, left, and right, select 4. If you want a pixel to be connected to the other pixels on the top, bottom, left, right, and diagonally, select 8.

Consider the following 3-by-3 image. If, for the **Connectivity** parameter, you select **4**, the block considers the white pixels marked by black circles to be connected.



If, for the **Connectivity** parameter, you select **8**, the block considers the white pixels marked by black circles to be connected.



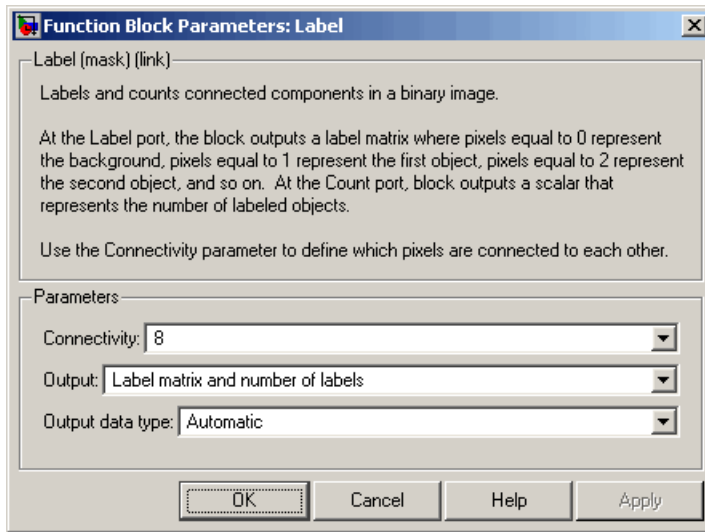
Use the **Output** parameter to determine the block's output. If you select **Label matrix** and **number of labels**, ports **Label** and **Count** appear on the block. The block outputs the label matrix at the **Label** port and the number of labeled objects at the **Count** port. If you select **Label matrix**, the **Label** port appears on the block. If you select **Number of labels**, the **Count** port appears on the block.

Use the **Output data type** parameter to set the data type of the outputs at the **Label** and **Count** ports. If you select **Automatic**, the block calculates the maximum number of objects that can fit inside the image based on the image size and the connectivity you specified. Based on this calculation, it determines the minimum output data type size that guarantees unique region labels and sets the output data type appropriately. If you select **uint32**, **uint16**, or **uint8**, the data type of the output is 32-, 16-, or 8-bit unsigned integers, respectively. If you select **uint16**, or **uint8**, the **If label exceeds data type size, mark remaining regions using** parameter appears in the dialog box. If the number of found objects exceeds the maximum number that can be represented by the output data type, use this parameter to specify the block's behavior. If you select **Maximum value of the output data type**, the remaining regions are labeled with

the maximum value of the output data type. If you select **Zero**, the remaining regions are labeled with zeroes.

## Dialog Box

The Label dialog box appears as shown in the following figure.



### Connectivity

Specify which pixels are connected to each other. If you want a pixel to be connected to the pixels on the top, bottom, left, and right, select **4**. If you want a pixel to be connected to the pixels on the top, bottom, left, right, and diagonally, select **8**.

### Output

Determine the block's output. If you select **Label matrix and number of labels**, the Label and Count ports appear on the block. The block outputs the label matrix at the Label port and the number of labeled objects at the Count port. If you select **Label matrix**, the Label port appears on the block. If you select **Number of labels**, the Count port appears on the block.

### Output data type

Set the data type of the outputs at the Label and Count ports. If you select **Automatic**, the block determines the appropriate data type for the output. If you

select `uint32`, `uint16`, or `uint8`, the data type of the output is 32-, 16-, or 8-bit unsigned integers, respectively.

### If label exceeds data type size, mark remaining regions using

Use this parameter to specify the block's behavior if the number of found objects exceeds the maximum number that can be represented by the output data type. If you select **Maximum value of the output data type**, the remaining regions are labeled with the maximum value of the output data type. If you select **Zero**, the remaining regions are labeled with zeroes. This parameter is visible if, for the **Output data type** parameter, you choose `uint16` or `uint8`.

## See Also

Bottom-hat	Computer Vision System Toolbox software
Closing	Computer Vision System Toolbox software
Dilation	Computer Vision System Toolbox software
Erosion	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
<code>bwlabel</code>	Image Processing Toolbox software
<code>bwlabeln</code>	Image Processing Toolbox software

Introduced before R2006a

# Median Filter

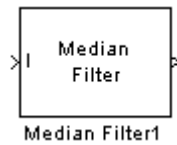
Perform 2-D median filtering

## Library

Filtering and Analysis & Enhancement

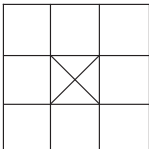
visionanalysis

visionfilter

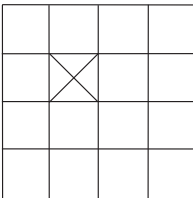


## Description

The Median Filter block replaces the central value of an  $M$ -by- $N$  neighborhood with its median value. If the neighborhood has a center element, the block places the median value there, as illustrated in the following figure.



The block has a bias toward the upper-left corner when the neighborhood does not have an exact center. See the median value placement in the following figure.



The block pads the edge of the input image, which sometimes causes the pixels within  $[M/2\ N/2]$  of the edges to appear distorted. The median value is less sensitive than the

mean to extreme values. As a result, the **Median Filter** block can remove salt-and-pepper noise from an image without significantly reducing the sharpness of the image.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Val	Scalar value that represents the constant pad value	Same as I port	No
Output	Matrix of intensity values	Same as I port	No

If the data type of the input signal is floating point, the output has the same data type. The data types of the signals input to the I and Val ports must be the same.

## Fixed-Point Data Types

The information in this section is applicable only when the dimensions of the neighborhood are even.

For fixed-point inputs, you can specify accumulator and output data types as discussed in “Dialog Box” on page 1-452. Not all these fixed-point parameters apply to all types of fixed-point inputs. The following table shows the output and accumulator data type used for each fixed-point input.

Fixed-Point Input	Output Data Type	Accumulator Data Type
<b>Even M</b>	X	X
<b>Odd M</b>	X	
<b>Odd M and complex</b>	X	X
<b>Even M and complex</b>	X	X

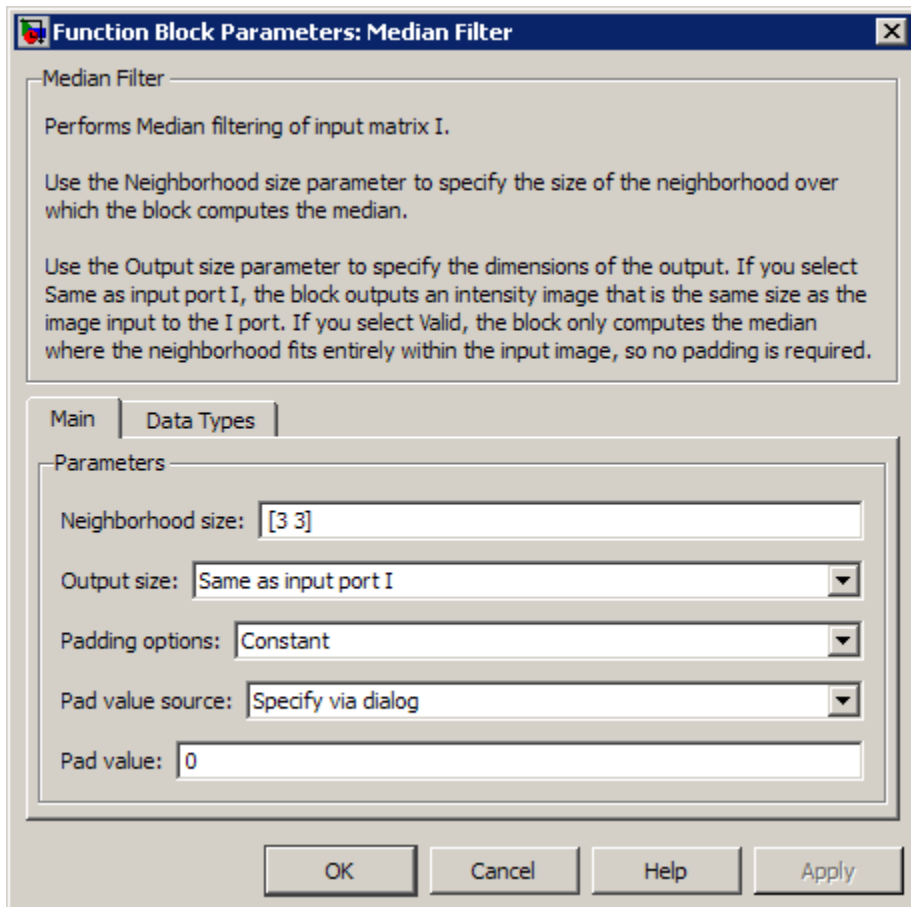
When M is even, fixed-point signals use the accumulator and output data types. The accumulator data type store the result of the sum performed while calculating the

average of the two central rows of the input matrix. The output data type stores the total result of the average.

Complex fixed-point inputs use the accumulator parameters. The calculation for the sum of the squares of the real and imaginary parts of the input occur, before sorting input elements. The accumulator data type stores the result of the sum of the squares.

## Dialog Box

The **Main** pane of the Median Filter dialog box appears as shown in the following figure.





### Neighborhood size

Specify the size of the neighborhood over which the block computes the median.

- Enter a scalar value that represents the number of rows and columns in a square matrix.
- Enter a vector that represents the number of rows and columns in a rectangular matrix.

### Output size

This parameter controls the size of the output matrix.

- If you choose **Same as input port I**, the output has the same dimensions as the input to port I. The **Padding options** parameter appears in the dialog box. Use the **Padding options** parameter to specify how to pad the boundary of your input matrix.
- If you select **Valid**, the block only computes the median where the neighborhood fits entirely within the input image, with no need for padding. The dimensions of the output image are,  $\text{output rows} = \text{input rows} - \text{neighborhood rows} + 1$ ,  
and  
 $\text{output columns} = \text{input columns} - \text{neighborhood columns} + 1$ .

### Padding options

Specify how to pad the boundary of your input matrix.

- Select **Constant** to pad your matrix with a constant value. The **Pad value source** parameter appears in the dialog box
- Select **Replicate** to pad your input matrix by repeating its border values.
- Select **Symmetric** to pad your input matrix with its mirror image.
- Select **Circular** to pad your input matrix using a circular repetition of its elements. This parameter appears if, for the **Output size** parameter, you select **Same as input port I**.

For more information on padding, see the **Image Pad** block reference page.

### Pad value source

Use this parameter to specify how to define your constant boundary value.

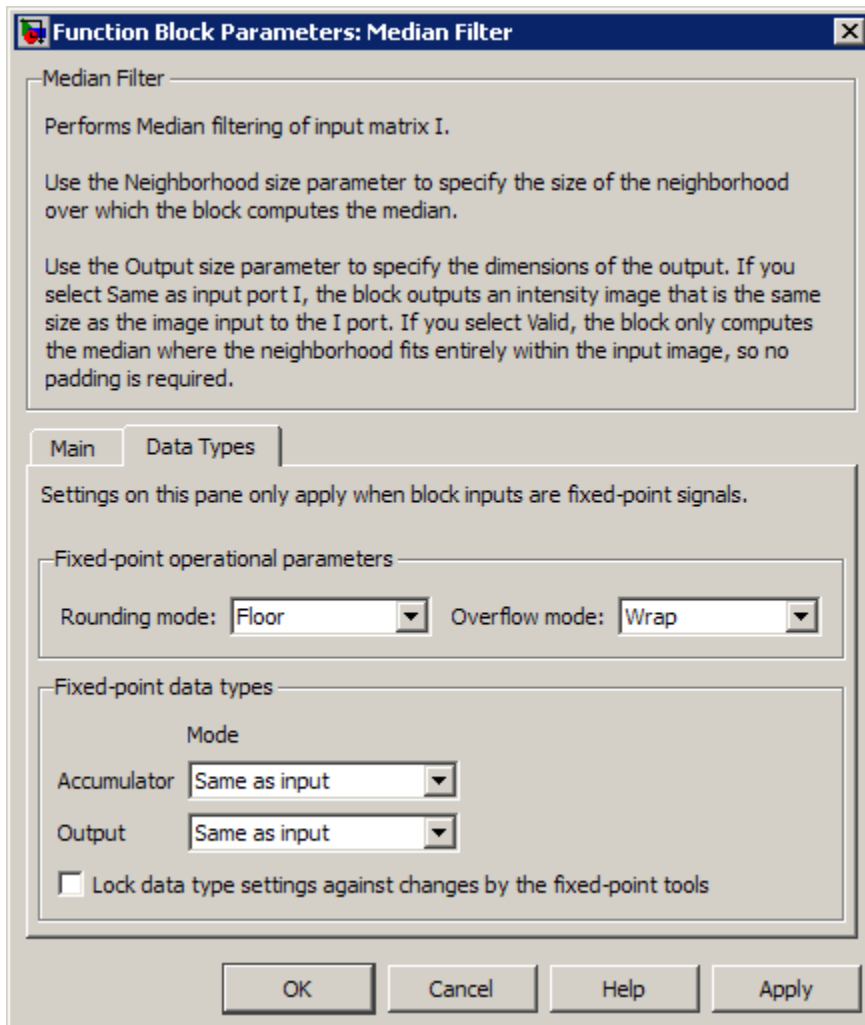
- Select **Specify via dialog** to enter your value in the block parameters dialog box. The **Pad value** parameter appears in the dialog box.

- Select **Input port** to specify your constant value using the **Val** port. This parameter appears if, for the **Padding options** parameter, you select **Constant**.

### **Pad value**

Enter the constant value with which to pad your matrix. This parameter appears if, for the **Pad value source** parameter, you select **Specify via dialog**. Tunable.

The **Data Types** pane of the Median Filter dialog box appears as follows. The parameters on this dialog box becomes visible only when the dimensions of the neighborhood are even.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

---

**Note** Only certain cases require the use of the accumulator and output parameters. Refer to “Fixed-Point Data Types” on page 1-451 for more information.

---

## Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block:

- When you select **Same as input**, these characteristics match the related input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of 0.

## Output

Choose how to specify the output word length and fraction length:

- When you select **Same as input**, these characteristics match the related input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of 0.

## Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Gonzales, Rafael C. and Richard E. Woods. *Digital Image Processing. 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall, 2002.

## See Also

2-D Convolution	Computer Vision System Toolbox
2-D FIR Filter	Computer Vision System Toolbox
medfilt2	Image Processing Toolbox

**Introduced before R2006a**

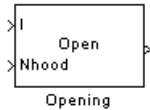
# Opening

Perform morphological opening on binary or intensity images

## Library

Morphological Operations

visionmorphops



## Description

The Opening block performs an erosion operation followed by a dilation operation using a predefined neighborhood or structuring element. This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of ones and zeros that represents the neighborhood values	Boolean	No
Output	Scalar, vector, or matrix of intensity values that represents the opened image	Same as I port	No

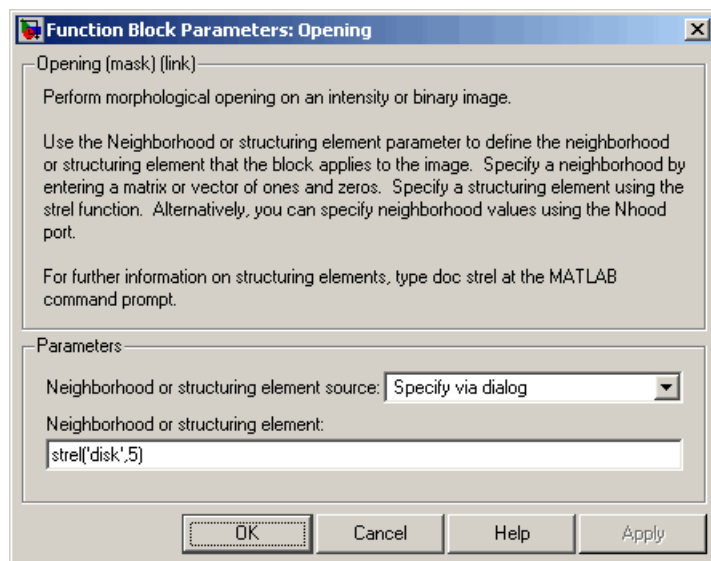
The output signal has the same data type as the input to the I port.

Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the **Nhood** port appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. You can only specify a structuring element using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the region the block moves throughout the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm.

## Dialog Box

The Opening dialog box appears as shown in the following figure.



### Neighborhood or structuring element source

Specify how to enter your neighborhood or structuring element values. Select **Specify via dialog** to enter the values in the dialog box. Select **Input port** to use the `Nhood` port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

### **Neighborhood or structuring element**

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select **Specify via dialog**.

## **References**

[1] Soille, Pierre. *Morphological Image Analysis. 2nd ed.* New York: Springer, 2003.

## **See Also**

Bottom-hat	Computer Vision System Toolbox software
Closing	Computer Vision System Toolbox software
Dilation	Computer Vision System Toolbox software
Erosion	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
Top-hat	Computer Vision System Toolbox software
imopen	Image Processing Toolbox software
strel	Image Processing Toolbox software

**Introduced before R2006a**



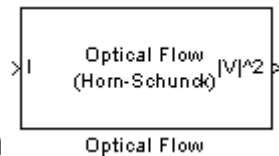
# Optical Flow

Estimate object velocities

## Library

Analysis & Enhancement

visionanalysis



## Description

The Optical Flow block estimates the direction and speed of object motion from one image to another or from one video frame to another using either the Horn-Schunck or the Lucas-Kanade method.

Port	Output	Supported Data Types	Complex Values Supported
I/I1	Scalar, vector, or matrix of intensity values	<ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> <li>Fixed point (supported when the <b>Method</b> parameter is set to <b>Lucas-Kanade</b>)</li> </ul>	No
I2	Scalar, vector, or matrix of intensity values	Same as I port	No
V ^2	Matrix of velocity magnitudes	Same as I port	No
V	Matrix of velocity components in complex form	Same as I port	Yes

To compute the optical flow between two images, you must solve the following optical flow constraint equation:

$$I_x u + I_y v + I_t = 0$$

- $I_x$ ,  $I_y$ , and  $I_t$  are the spatiotemporal image brightness derivatives.
- $u$  is the horizontal optical flow.
- $v$  is the vertical optical flow.

## Horn-Schunck Method

By assuming that the optical flow is smooth over the entire image, the Horn-Schunck method computes an estimate of the velocity field,  $[u \ v]^T$ , that minimizes this equation:

$$E = \iint (I_x u + I_y v + I_t)^2 dx dy + \alpha \iint \left\{ \left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 \right\} dx dy$$

In this equation,  $\frac{\partial u}{\partial x}$  and  $\frac{\partial u}{\partial y}$  are the spatial derivatives of the optical velocity component,  $u$ , and  $\alpha$  scales the global smoothness term. The Horn-Schunck method minimizes the previous equation to obtain the velocity field,  $[u \ v]$ , for each pixel in the image. This method is given by the following equations:

$$u_{x,y}^{k+1} = u_{x,y}^{-k} - \frac{I_x [I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

$$v_{x,y}^{k+1} = v_{x,y}^{-k} - \frac{I_y [I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

In these equations,  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$  is the velocity estimate for the pixel at  $(x,y)$ , and

$\begin{bmatrix} -k & -k \\ u_{x,y} & v_{x,y} \end{bmatrix}$  is the neighborhood average of  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$ . For  $k = 0$ , the initial velocity is 0.

To solve  $u$  and  $v$  using the Horn-Schunck method:

- 1 Compute  $I_x$  and  $I_y$  using the Sobel convolution kernel,  $\begin{bmatrix} -1 & -2 & -1; & 0 & 0 & 0; & 1 & 2 & 1 \end{bmatrix}$ , and its transposed form, for each pixel in the first image.
- 2 Compute  $I_t$  between images 1 and 2 using the  $\begin{bmatrix} -1 & 1 \end{bmatrix}$  kernel.
- 3 Assume the previous velocity to be 0, and compute the average velocity for each pixel using  $\begin{bmatrix} 0 & 1 & 0; & 1 & 0 & 1; & 0 & 1 & 0 \end{bmatrix}$  as a convolution kernel.
- 4 Iteratively solve for  $u$  and  $v$ .

## Lucas-Kanade Method

To solve the optical flow constraint equation for  $u$  and  $v$ , the Lucas-Kanade method divides the original image into smaller sections and assumes a constant velocity in each section. Then, it performs a weighted least-square fit of the optical flow constraint equation to a constant model for  $\begin{bmatrix} u & v \end{bmatrix}^T$  in each section  $\Omega$ . The method achieves this fit by minimizing the following equation:

$$\sum_{x \in \Omega} W^2 [I_x u + I_y v + I_t]^2$$

$W$  is a window function that emphasizes the constraints at the center of each section. The solution to the minimization problem is

$$\begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 I_x I_t \\ \sum W^2 I_y I_t \end{bmatrix}$$

### Lucas-Kanade Difference Filter

When you set the **Temporal gradient filter** to Difference filter  $[-1 \ 1]$ ,  $u$  and  $v$  are solved as follows:

- 1 Compute  $I_x$  and  $I_y$  using the kernel  $[-1 \ 8 \ 0 \ -8 \ 1]/12$  and its transposed form.

If you are working with fixed-point data types, the kernel values are signed fixed-point values with word length equal to 16 and fraction length equal to 15.

- 2 Compute  $I_t$  between images 1 and 2 using the  $[-1 \ 1]$  kernel.
- 3 Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a separable and isotropic 5-by-5 element kernel whose effective 1-D coefficients are  $[1 \ 4 \ 6 \ 4 \ 1]/16$ . If you are working with fixed-point data types, the kernel values are unsigned fixed-point values with word length equal to 8 and fraction length equal to 7.
- 4 Solve the 2-by-2 linear equations for each pixel using the following method:

$$\bullet \text{ If } A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$$

$$\text{Then the eigenvalues of A are } \lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$$

$$\text{In the fixed-point diagrams, } P = \frac{a+c}{2}, Q = \frac{\sqrt{4b^2 + (a-c)^2}}{2}$$

- The eigenvalues are compared to the threshold,  $\tau$ , that corresponds to the value you enter for the threshold for noise reduction. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

A is nonsingular, the system of equations are solved using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

$A$  is singular (noninvertible), the gradient flow is normalized to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.

### Derivative of Gaussian

If you set the temporal gradient filter to **Derivative of Gaussian**,  $u$  and  $v$  are solved using the following steps. You can see the flow chart for this process at the end of this section:

- 1 Compute  $I_x$  and  $I_y$  using the following steps:
  - a Use a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the **Number of frames to buffer for temporal smoothing** parameter.
  - b Use a Gaussian filter and the derivative of a Gaussian filter to smooth the image using spatial filtering. Specify the standard deviation and length of the image smoothing filter using the **Standard deviation for image smoothing filter** parameter.
- 2 Compute  $I_t$  between images 1 and 2 using the following steps:
  - a Use the derivative of a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the **Number of frames to buffer for temporal smoothing** parameter.
  - b Use the filter described in step 1b to perform spatial filtering on the output of the temporal filter.
- 3 Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a gradient smoothing filter. Use the **Standard deviation for gradient smoothing filter** parameter to specify the standard deviation and the number of filter coefficients for the gradient smoothing filter.
- 4 Solve the 2-by-2 linear equations for each pixel using the following method:

- If  $A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$

Then the eigenvalues of A are  $\lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$

- When the block finds the eigenvalues, it compares them to the threshold,  $\tau$ , that corresponds to the value you enter for the **Threshold for noise reduction** parameter. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

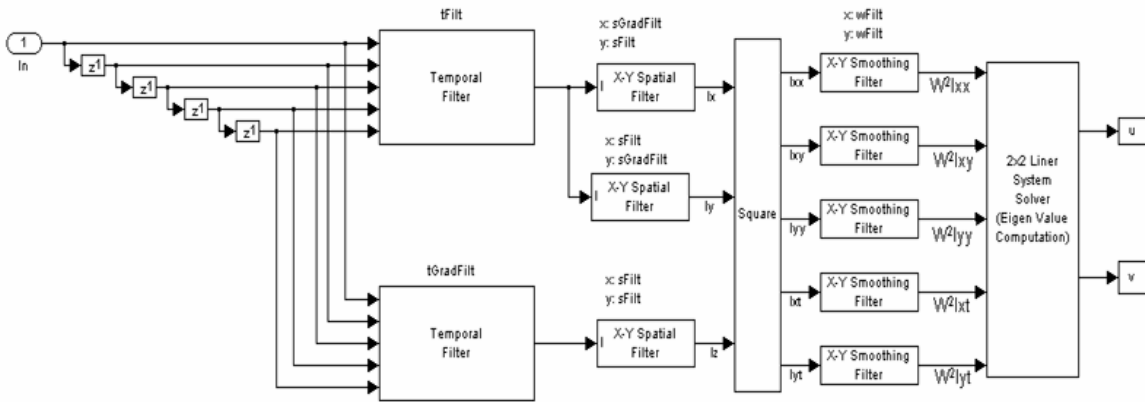
A is nonsingular, so the block solves the system of equations using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

A is singular (noninvertible), so the block normalizes the gradient flow to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.



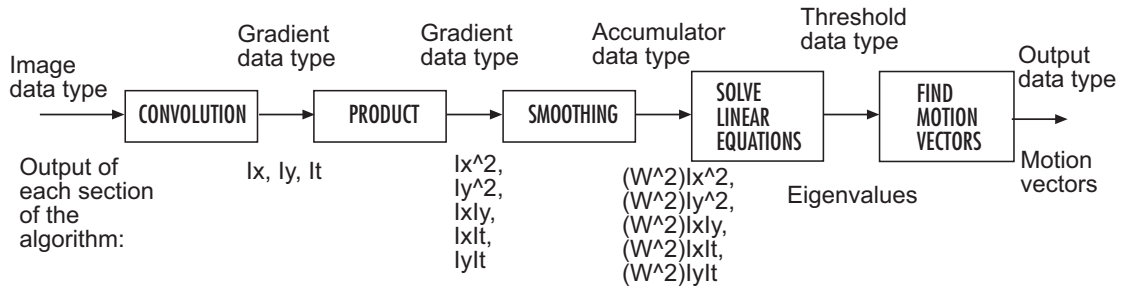
tFilt = Coefficients of Gaussian Filter  
 tGradFilt = Coefficients of the Derivative of a Gaussian Filter

sFilt = Coefficients of Gaussian Filter  
 sGradFilt = Coefficients of the Derivative of a Gaussian Filter

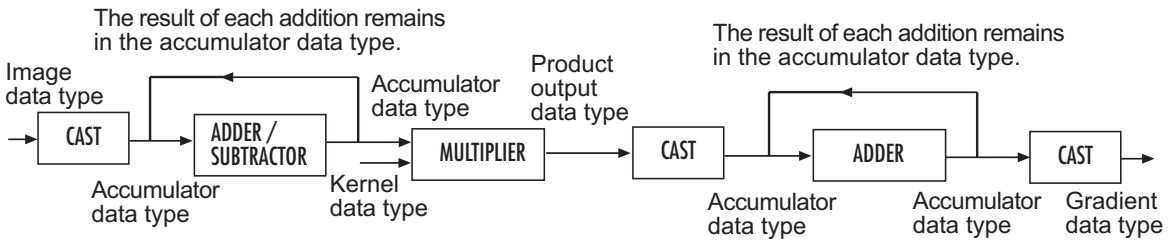
## Fixed-Point Data Type Diagram

The following diagrams shows the data types used in the Optical Flow block for fixed-point signals. The block supports fixed-point data types only when the **Method** parameter is set to Lucas-Kanade.

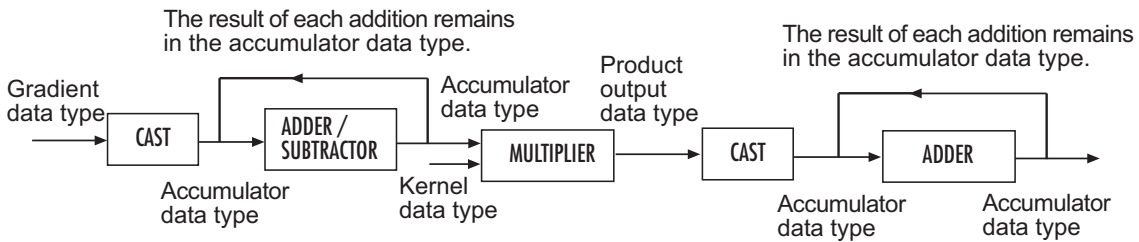
Data type diagram for Optical Flow block's overall algorithm



Data type diagram for convolution algorithm

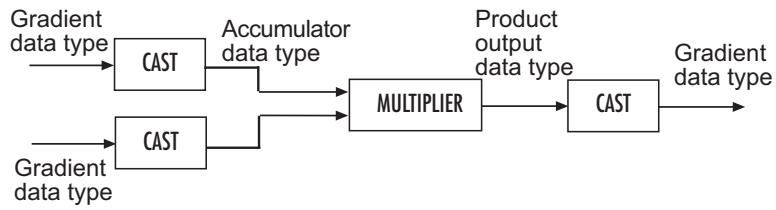


Data type diagram for smoothing algorithm



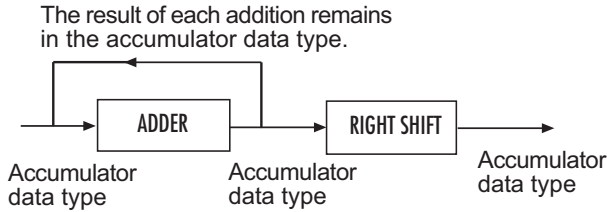


Data type diagram for product algorithm

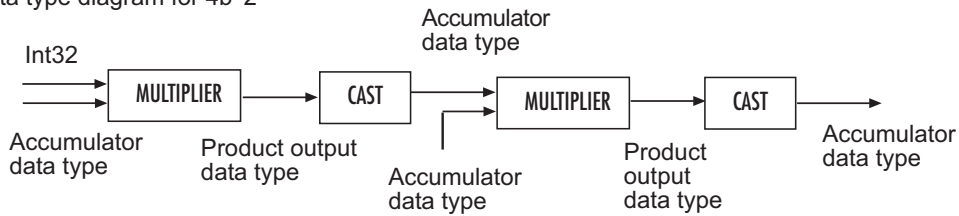


Solving linear equations to compute eigenvalues  
 (see Step 4 in the Lucas-Kanade Method section for the eigenvalue equations)

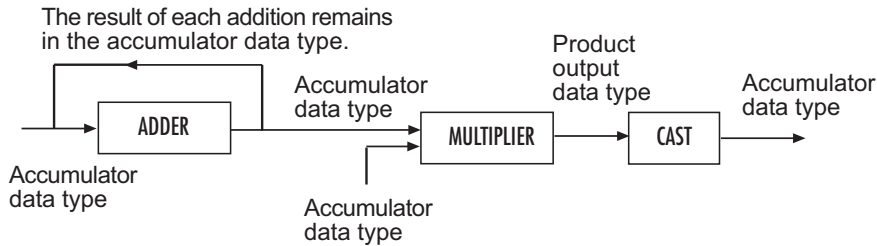
Data type diagram for P



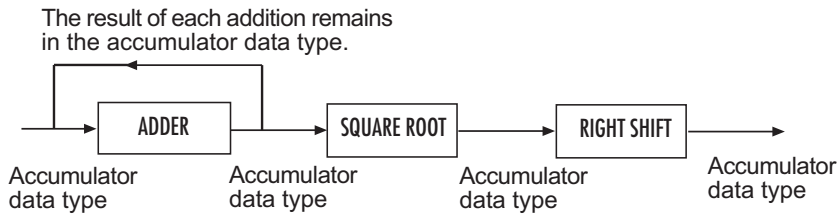
Data type diagram for  $4b^2$



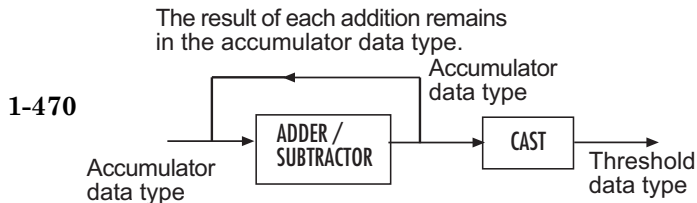
Data type diagram for  $(a-c)^2$



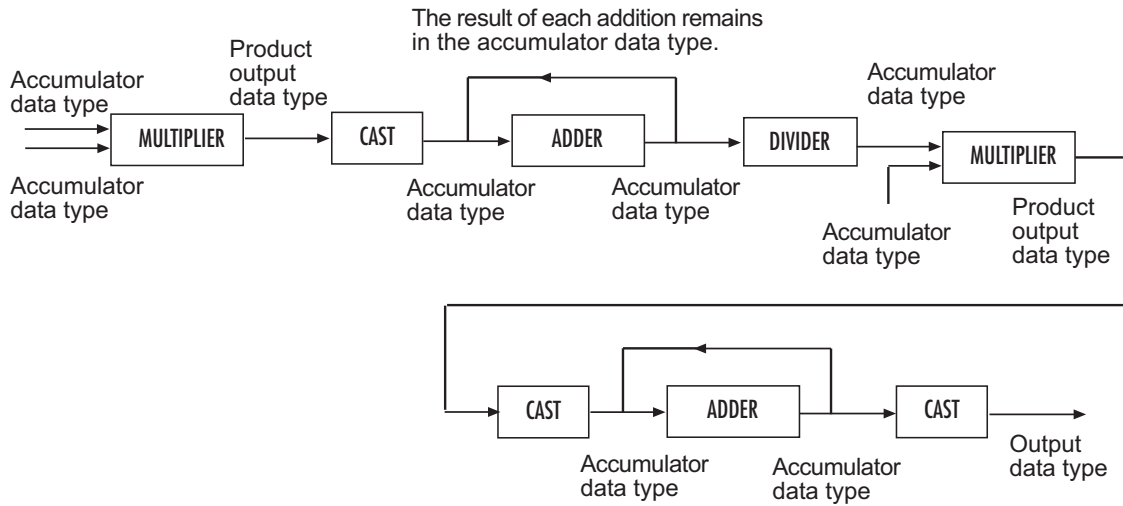
Data type diagram for Q



Data type diagram for eigenvalues



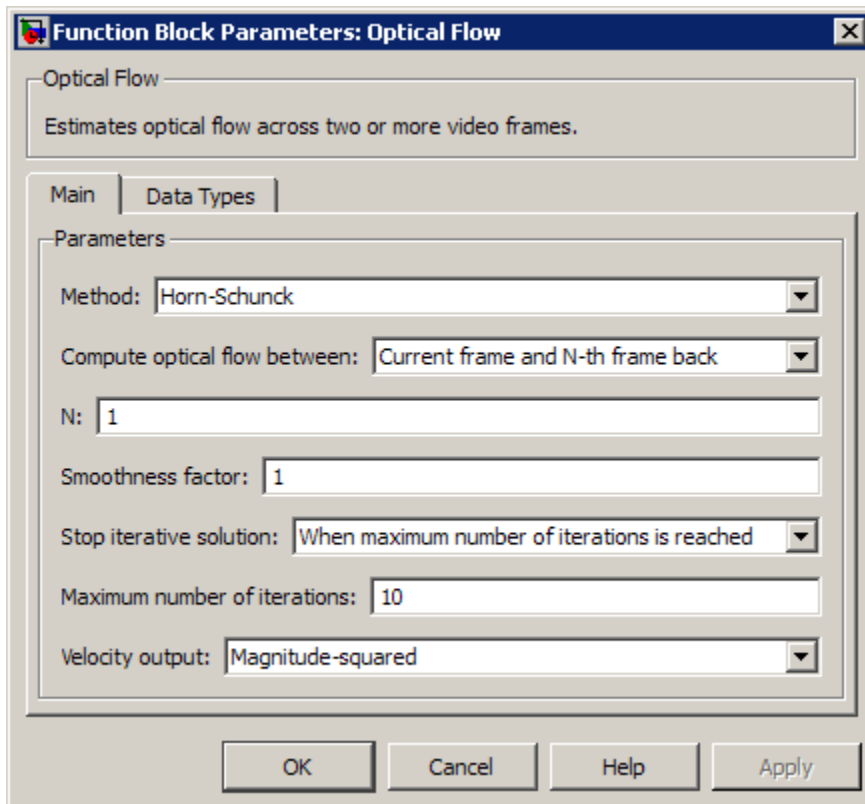
Data type diagram for finding the motion vectors algorithm



You can set the product output, accumulator, gradients, threshold, and output data types in the block mask.

## Dialog Box

The **Main** pane of the Optical Flow dialog box appears as shown in the following figure.



### Method

Select the method the block uses to calculate the optical flow. Your choices are Horn-Schunck or Lucas-Kanade.

### Compute optical flow between

Select **Two images** to compute the optical flow between two images. Select **Current frame and N-th frame back** to compute the optical flow between two video frames that are N frames apart.

This parameter is visible if you set the **Method** parameter to Horn-Schunck or you set the **Method** parameter to Lucas-Kanade and the **Temporal gradient filter** to **Difference filter [-1 1]**.

**N**

Enter a scalar value that represents the number of frames between the reference frame and the current frame. This parameter becomes available if you set the **Compute optical flow between** parameter, you select **Current frame and N-th frame back**.

### Smoothness factor

If the relative motion between the two images or video frames is large, enter a large positive scalar value. If the relative motion is small, enter a small positive scalar value. This parameter becomes available if you set the **Method** parameter to **Horn-Schunck**.

### Stop iterative solution

Use this parameter to control when the block's iterative solution process stops. If you want it to stop when the velocity difference is below a certain threshold value, select **When velocity difference falls below threshold**. If you want it to stop after a certain number of iterations, choose **When maximum number of iterations is reached**. You can also select **Whichever comes first**. This parameter becomes available if you set the **Method** parameter to **Horn-Schunck**.

### Maximum number of iterations

Enter a scalar value that represents the maximum number of iterations you want the block to perform. This parameter is only visible if, for the **Stop iterative solution** parameter, you select **When maximum number of iterations is reached** or **Whichever comes first**. This parameter becomes available if you set the **Method** parameter to **Horn-Schunck**.

### Velocity difference threshold

Enter a scalar threshold value. This parameter is only visible if, for the **Stop iterative solution** parameter, you select **When velocity difference falls below threshold** or **Whichever comes first**. This parameter becomes available if you set the **Method** parameter to **Horn-Schunck**.

### Velocity output

If you select **Magnitude-squared**, the block outputs the optical flow matrix where each element is of the form  $u^2 + v^2$ . If you select **Horizontal and vertical components in complex form**, the block outputs the optical flow matrix where each element is of the form  $u + jv$ .

### Temporal gradient filter

Specify whether the block solves for  $u$  and  $v$  using a difference filter or a derivative of a Gaussian filter. This parameter becomes available if you set the **Method** parameter to Lucas-Kanade.

**Number of frames to buffer for temporal smoothing**

Use this parameter to specify the temporal filter characteristics such as the standard deviation and number of filter coefficients. This parameter becomes available if you set the **Temporal gradient filter** parameter to Derivative of Gaussian.

**Standard deviation for image smoothing filter**

Specify the standard deviation for the image smoothing filter. This parameter becomes available if you set the **Temporal gradient filter** parameter to Derivative of Gaussian.

**Standard deviation for gradient smoothing filter**

Specify the standard deviation for the gradient smoothing filter. This parameter becomes available if you set the **Temporal gradient filter** parameter to Derivative of Gaussian.

**Discard normal flow estimates when constraint equation is ill-conditioned**

Select this check box if you want the block to set the motion vector to zero when the optical flow constraint equation is ill-conditioned. This parameter becomes available if you set the **Temporal gradient filter** parameter to Derivative of Gaussian.

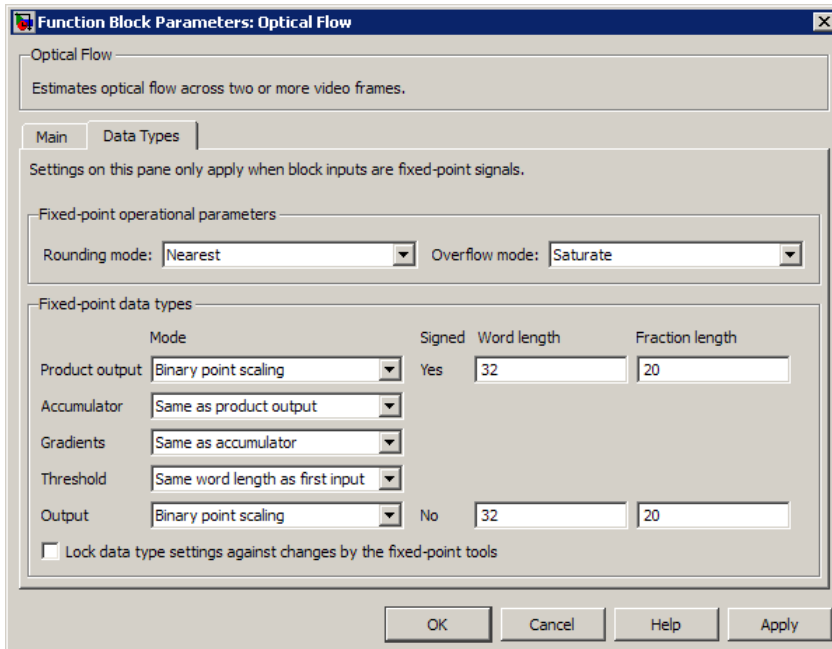
**Output image corresponding to motion vectors (accounts for block delay)**

Select this check box if you want the block to output the image that corresponds to the motion vector being output by the block. This parameter becomes available if you set the **Temporal gradient filter** parameter to Derivative of Gaussian.

**Threshold for noise reduction**

Enter a scalar value that determines the motion threshold between each image or video frame. The higher the number, the less small movements impact the optical flow calculation. This parameter becomes available if you set the **Method** parameter to Lucas-Kanade.

The **Data Types** pane of the Optical Flow dialog box appears as shown in the following figure. The parameters on this dialog box becomes visible only when the Lucas-Kanade method is selected.



### Rounding mode

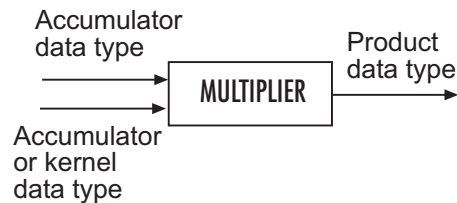
Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Product output

Use this parameter to specify how to designate the product output word and fraction lengths.

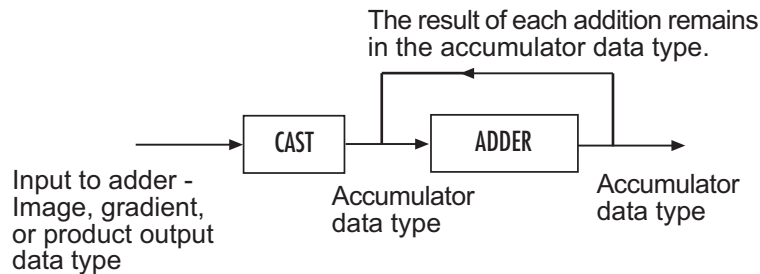


- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output in bits.

- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator

Use this parameter to specify how to designate this accumulator word and fraction lengths.



- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Gradients

Choose how to specify the word length and fraction length of the gradients data type:

- When you select **Same as accumulator**, these characteristics match those of the accumulator.
- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the quotient, in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the quotient. The bias of all signals in the Computer Vision System Toolbox blocks is 0.



## Threshold

Choose how to specify the word length and fraction length of the threshold data type:

- When you select **Same word length as first input**, the threshold word length matches that of the first input.
- When you select **Specify word length**, enter the word length of the threshold data type.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the threshold, in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the threshold. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Output

Choose how to specify the word length and fraction length of the output data type:

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length in bits and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Barron, J.L., D.J. Fleet, S.S. Beauchemin, and T.A. Burkitt. *Performance of optical flow techniques*. CVPR, 1992.

## See Also

Block Matching	Computer Vision System Toolbox software
----------------	---

Gaussian Pyramid	Computer Vision System Toolbox software
------------------	---

**Introduced before R2006a**

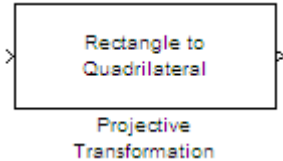
# Projective Transformation (To Be Removed)

Transform quadrilateral into another quadrilateral

## Library

Geometric Transformations

vipgeotforms



## Description

---

**Note:** This block will be removed in a future release. It is recommended that you replace this block with the `Apply Geometric Transformation` and `Estimate Geometric Transformation` block. The replacement will require that you modify your model.

---

Introduced in R2011b

# PSNR

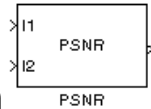
Compute peak signal-to-noise ratio (PSNR) between images

## Library

Statistics

visionstatistics

## Description



The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images. This ratio is often used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.

The *Mean Square Error (MSE)* and the *Peak Signal to Noise Ratio (PSNR)* are the two error metrics used to compare image compression quality. The MSE represents the cumulative squared error between the compressed and the original image, whereas PSNR represents a measure of the peak error. The lower the value of MSE, the lower the error.

To compute the PSNR, the block first calculates the mean-squared error using the following equation:

$$MSE = \frac{\sum_{M,N} [I_1(m,n) - I_2(m,n)]^2}{M * N}$$

In the previous equation,  $M$  and  $N$  are the number of rows and columns in the input images, respectively. Then the block computes the PSNR using the following equation:

$$PSNR = 10 \log_{10} \left( \frac{R^2}{MSE} \right)$$

In the previous equation,  $R$  is the maximum fluctuation in the input image data type. For example, if the input image has a double-precision floating-point data type, then  $R$  is 1. If it has an 8-bit unsigned integer data type,  $R$  is 255, etc.

## Recommendation for Computing PSNR for Color Images

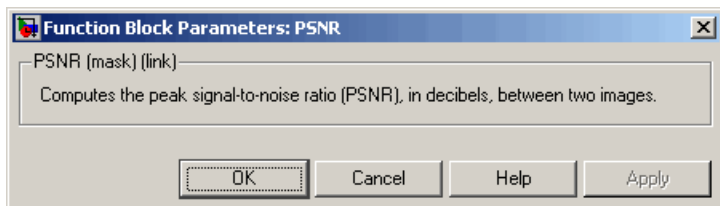
Different approaches exist for computing the PSNR of a color image. Because the human eye is most sensitive to luma information, compute the PSNR for color images by converting the image to a color space that separates the intensity (luma) channel, such as YCbCr. The Y (luma), in YCbCr represents a weighted average of R, G, and B. G is given the most weight, again because the human eye perceives it most easily. With this consideration, compute the PSNR only on the luma channel.

## Ports

Port	Output	Supported Data Types	Complex Values Supported
I1	Scalar, vector, or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
I2	Scalar, vector, or matrix of intensity values	Same as I1 port	No
Output	Scalar value that represents the PSNR	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> </ul> <p>For fixed-point or integer input, the block output is double-precision floating point. Otherwise, the block input and output are the same data type.</p>	No

## Dialog Box

The PSNR dialog box appears as shown in the following figure.



**Introduced before R2006a**

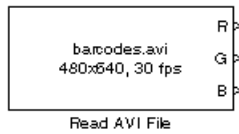
# Read AVI File (To Be Removed)

Read uncompressed video frames from AVI file

## Library

vipobslib

## Description



---

**Note:** The Read AVI File block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox blocks. Use the replacement block `From Multimedia File`.

---

**Introduced in R2011b**

# Read Binary File

Read binary video data from files

## Library

Sources

visionsources

## Description



The Read Binary File block reads video data from a binary file and imports it into a Simulink model.

This block takes user specified parameters that describe the format of the video data. These parameters together with the raw binary file, which stores only raw pixel values, creates the video data signal for a Simulink model. The video data read by this block must be stored in row major format.

---

**Note:** This block supports code generation only for platforms that have file I/O available. You cannot use this block to do code generation with Simulink Desktop Real-Time or Simulink Real-Time™.

---

Port	Output	Supported Data Types	Complex Values Supported
Output	Scalar, vector, or matrix of integer values	<ul style="list-style-type: none"> <li>8-, 16- 32-bit signed integer</li> <li>8-, 16- 32-bit unsigned integer</li> </ul>	No
EOF	Scalar value	Boolean	No



## Four Character Code Video Formats

Four Character Codes (FOURCC) identify video formats. For more information about these codes, see <http://www.fourcc.org>.

Use the **Four character code** parameter to identify the binary file format. Then, use the **Rows** and **Cols** parameters to define the size of the output matrix. These dimensions should match the matrix dimensions of the data inside the file.

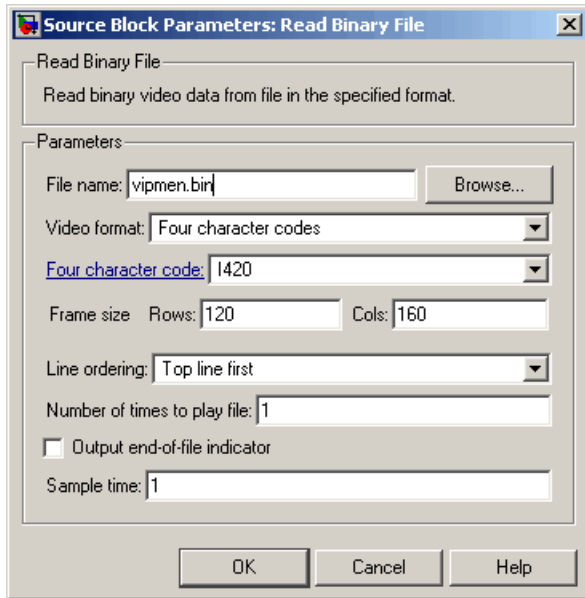
## Custom Video Formats

If your binary file contains data that is not in FOURCC format, you can configure the Read Binary File block to understand a custom format:

- Use the **Bit stream format** parameter to specify whether your data is planar or packed. If your data is packed, use the **Rows** and **Cols** parameters to define the size of the output matrix.
- Use the **Number of output components** parameter to specify the number of components in the binary file. This number corresponds to the number of block output ports.
- Use the **Component**, **Bits**, **Rows**, and **Cols** parameters to specify the component name, bit size, and size of the output matrices, respectively. The block uses the **Component** parameter to label the output ports.
- Use the **Component order in binary file** parameter to specify how the components are arranged within the file.
- Select the **Interlaced video** check box if the binary file contains interlaced video data.
- Select the **Input file has signed data** check box if the binary file contains signed integers.
- Use the **Byte order in binary file** to indicate whether your binary file has little endian or big endian byte ordering.

## Dialog Box

The Read Binary File dialog box appears as shown in the following figure.



### File name

Specify the name of the binary file to read. If the location of this file is on your MATLAB path, enter the filename. If the location of this file is not on your MATLAB path, use the **Browse** button to specify the full path to the file as well as the filename.

### Video format

Specify the format of the binary video data. Your choices are **Four character codes** or **Custom**. See “Four Character Code Video Formats” on page 1-485 or “Custom Video Formats” on page 1-485 for more details.

### Four character code

From the drop-down list, select the binary file format.

### Frame size: Rows, Cols

Define the size of the output matrix. These dimensions should match the matrix dimensions of the data inside the file.

### Line ordering

Specify how the block fills the output matrix. If you select **Top line first**, the block first fills the first row of the output matrix with the contents of the binary file. It then fills the other rows in increasing order. If you select **Bottom line first**,

the block first fills the last row of the output matrix. It then fills the other rows in decreasing order.

### Number of times to play file

Specify the number of times to play the file. The number you enter must be a positive integer or `inf`, to play the file until you stop the simulation.

### Output end-of-file indicator

Specifies the output is the last video frame in the binary file. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last video frame in the binary file is output from the block. Otherwise, the output from the EOF port is 0.

### Sample time

Specify the sample period of the output signal.

**Source Block Parameters: Read Binary File**

Read Binary File  
Read binary video data from file in the specified format.

Parameters

File name:

Video format:

Bit stream format:

Number of output components:

Component	Format	Bits	Rows	Cols
Component 1:	<input text"="" type="text" value="8"/>	<input type="text" value="120"/>	<input type="text" value="160"/>	
Component 2:	<input type="text" value="Cb"/>	<input type="text" value="8"/>	<input type="text" value="60"/>	<input type="text" value="80"/>
Component 3:	<input type="text" value="Cr"/>	<input type="text" value="8"/>	<input type="text" value="60"/>	<input type="text" value="80"/>

Component order in binary file:

Interlaced video

Line ordering:

Input file has signed data

Byte order in binary file:

Number of times to play file:

Output end-of-file indicator

Sample time:

**Bit stream format**

Specify whether your data is planar or packed.

**Frame size: Rows, Cols**

Define the size of the output matrix. This parameter appears when you select a **Bit stream format** parameter of **Packed**.

**Number of output components**

Specify the number of components in the binary file.

**Component, Bits, Rows, Cols**

Specify the component name, bit size, and size of the output matrices, respectively.

**Component order in binary file**

Specify the order in which the components appear in the binary file.

**Interlaced video**

Select this check box if the binary file contains interlaced video data.

**Input file has signed data**

Select this check box if the binary file contains signed integers.

**Byte order in binary file**

Use this parameter to indicate whether your binary file has little endian or big endian byte ordering.

## See Also

From Multimedia File	Computer Vision System Toolbox
Write Binary File	Computer Vision System Toolbox

**Introduced before R2006a**

# Resize

Enlarge or shrink image sizes

## Library

Geometric Transformations

visiongeotforms



## Description

The Resize block enlarges or shrinks an image by resizing the image along one dimension (row or column). Then, it resizes the image along the other dimension (column or row).

This block supports intensity and color images on its ports. When you input a floating point data type signal, the block outputs the same data type.

Shrinking an image can introduce high frequency components into the image and aliasing might occur. If you select the **Perform antialiasing when resize factor is between 0 and 100** check box, the block performs low pass filtering on the input image before shrinking it.

## Port Description

Port	Input/Output	Supported Data Types	Complex Values Supported
Image / Input	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No

Port	Input/Output	Supported Data Types	Complex Values Supported
ROI	Four-element vector [x y width height] that defines the ROI	<ul style="list-style-type: none"> <li>• Double-precision floating point (only supported if the input to the Input port is floating point)</li> <li>• Single-precision floating point (only supported if the input to the Input port is floating point)</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Output	Resized image	Same as Input port	No
Flag	Boolean value that indicates whether the ROI is within the image bounds	Boolean	No

## ROI Processing

To resize a particular region of each image, select the **Enable ROI processing** check box. To enable this option, select the following parameter values.

- **Specify** = Number of output rows and columns
- **Interpolation method** = Nearest neighbor, Bilinear, or Bicubic
- Clear the **Perform antialiasing when resize factor is between 0 and 100** check box.

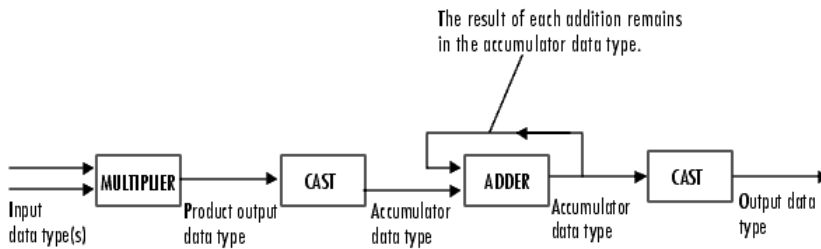
If you select the **Enable ROI processing** check box, the ROI port appears on the block. Use this port to define a region of interest (ROI) in the input matrix, that you want to resize. The input to this port must be a four-element vector, [x y width height]. The first two elements define the upper-left corner of the ROI, and the second two elements define the width and height of the ROI.

If you select the **Enable ROI processing** check box, the **Output flag indicating if any part of ROI is outside image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output.

Flag Port Output	Description
0	ROI is completely inside the input image.
1	ROI is completely or partially outside the input image.

## Fixed-Point Data Types

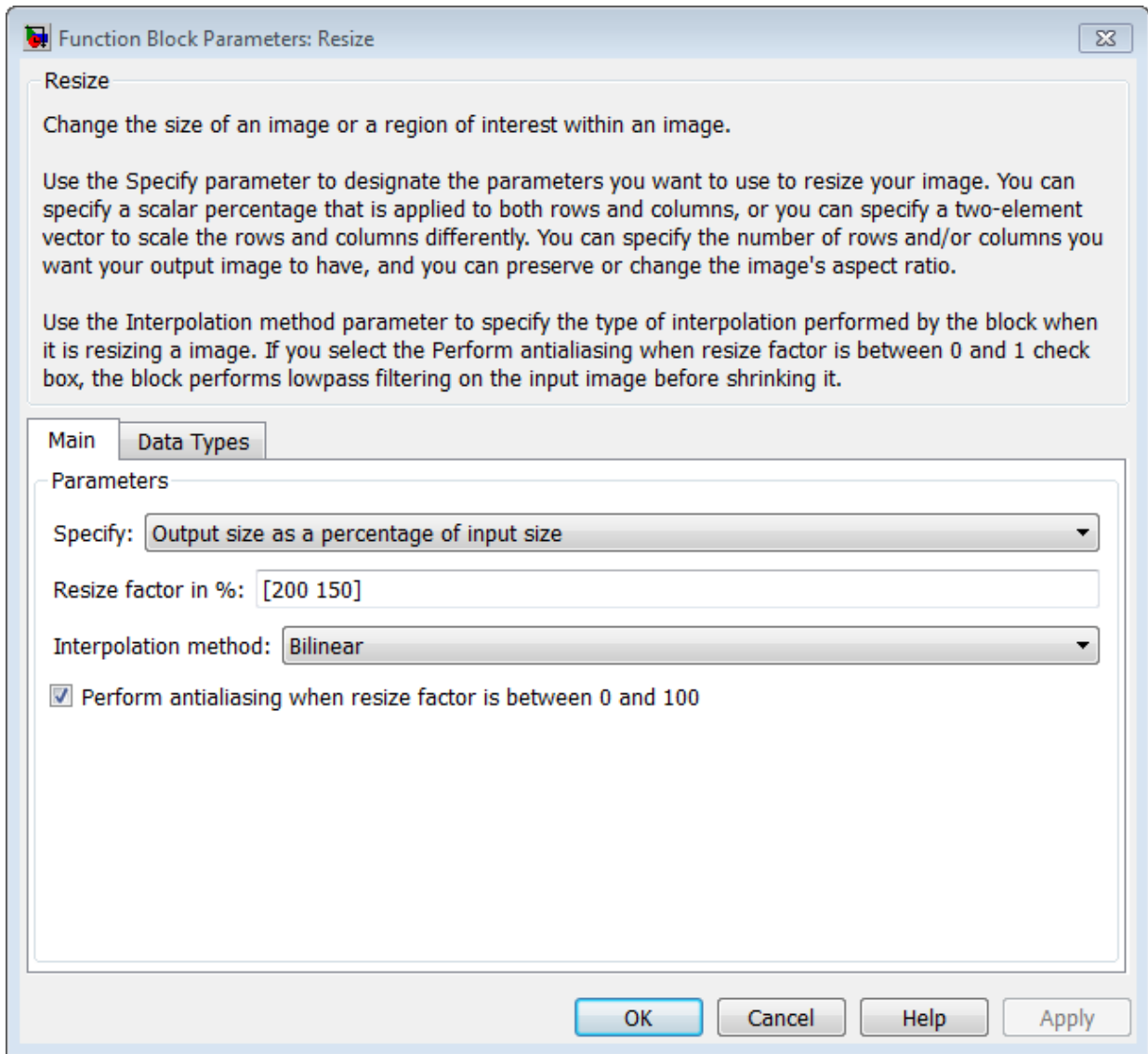
The following diagram shows the data types used in the Resize block for fixed-point signals.



You can set the interpolation weights table, product output, accumulator, and output data types in the block mask.

## Dialog Box

The **Main** pane of the Resize dialog box appears as shown in the following figure:



### Specify

Specify which aspects of the image to resize. Your choices are Output size as a percentage of input size, Number of output columns and preserve



aspect ratio, Number of output rows and preserve aspect ratio, or Number of output rows and columns.

When you select **Output size as a percentage of input size**, the **Resize factor in %** parameter appears in the dialog box. Enter a scalar percentage value that is applied to both rows and columns.

When you select **Number of output columns and preserve aspect ratio**, the **Number of output columns** parameter appears in the dialog box. Enter a scalar value that represents the number of columns you want the output image to have. The block calculates the number of output rows so that the output image has the same aspect ratio as the input image.

When you select **Number of output rows and preserve aspect ratio**, the **Number of output rows** parameter appears in the dialog box. Enter a scalar value that represents the number of rows you want the output image to have. The block calculates the number of output columns so that the output image has the same aspect ratio as the input image.

When you select **Number of output rows and columns**, the **Number of output rows and columns** parameter appears in the dialog box. Enter a two-element vector, where the first element is the number of rows in the output image and the second element is the number of columns. In this case, the aspect ratio of the image can change.

**Resize factor in %**

Enter a scalar percentage value that is applied to both rows and columns or a two-element vector, where the first element is the percentage by which to resize the rows and the second element is the percentage by which to resize the columns. This parameter is visible if, for the **Specify** parameter, you select **Output size as a percentage of input size**.

You must enter a scalar value that is greater than zero. The table below describes the affect of the resize factor value:

Resize factor in %	Resizing of image
0 < resize factor < 100	The block shrinks the image.
resize factor = 100	Image unchanged.

Resize factor in %	Resizing of image
resize factor > 100	The block enlarges the image.

The dimensions of the output matrix depend on the **Resize factor in %** parameter and are given by the following equations:

```
number_output_rows = round(number_input_rows*resize_factor/100);
```

```
number_output_cols = round(number_input_cols*resize_factor/100);
```

### Number of output columns

Enter a scalar value that represents the number of columns you want the output image to have. This parameter is visible if, for the **Specify** parameter, you select **Number of output columns** and **preserve aspect ratio**.

### Number of output rows

Enter a scalar value that represents the number of rows you want the output image to have. This parameter is visible if, for the **Specify** parameter, you select **Number of output rows** and **preserve aspect ratio**.

### Number of output rows and columns

Enter a two-element vector, where the first element is the number of rows in the output image and the second element is the number of columns. This parameter is visible if, for the **Specify** parameter, you select **Number of output rows and columns**.

### Interpolation method

Specify which interpolation method to resize the image.

When you select **Nearest neighbor**, the block uses one nearby pixel to interpolate the pixel value. This option though the most efficient, is the least accurate. When you select **Bilinear**, the block uses four nearby pixels to interpolate the pixel value. When you select **Bicubic** or **Lanczos2**, the block uses 16 nearby pixels to interpolate the pixel value. When you select **Lanczos3**, the block uses 36 surrounding pixels to interpolate the pixel value.

The Resize block performs optimally when you set this parameter to **Nearest neighbor** with one of the following conditions:

- You set the **Resize factor in %** parameter to a multiple of 100.

- Dividing 100 by the **Resize factor in %** parameter value results in an integer value.

**Perform antialiasing when resize factor is between 0 and 100**

If you select this check box, the block performs low-pass filtering on the input image before shrinking it to prevent aliasing.

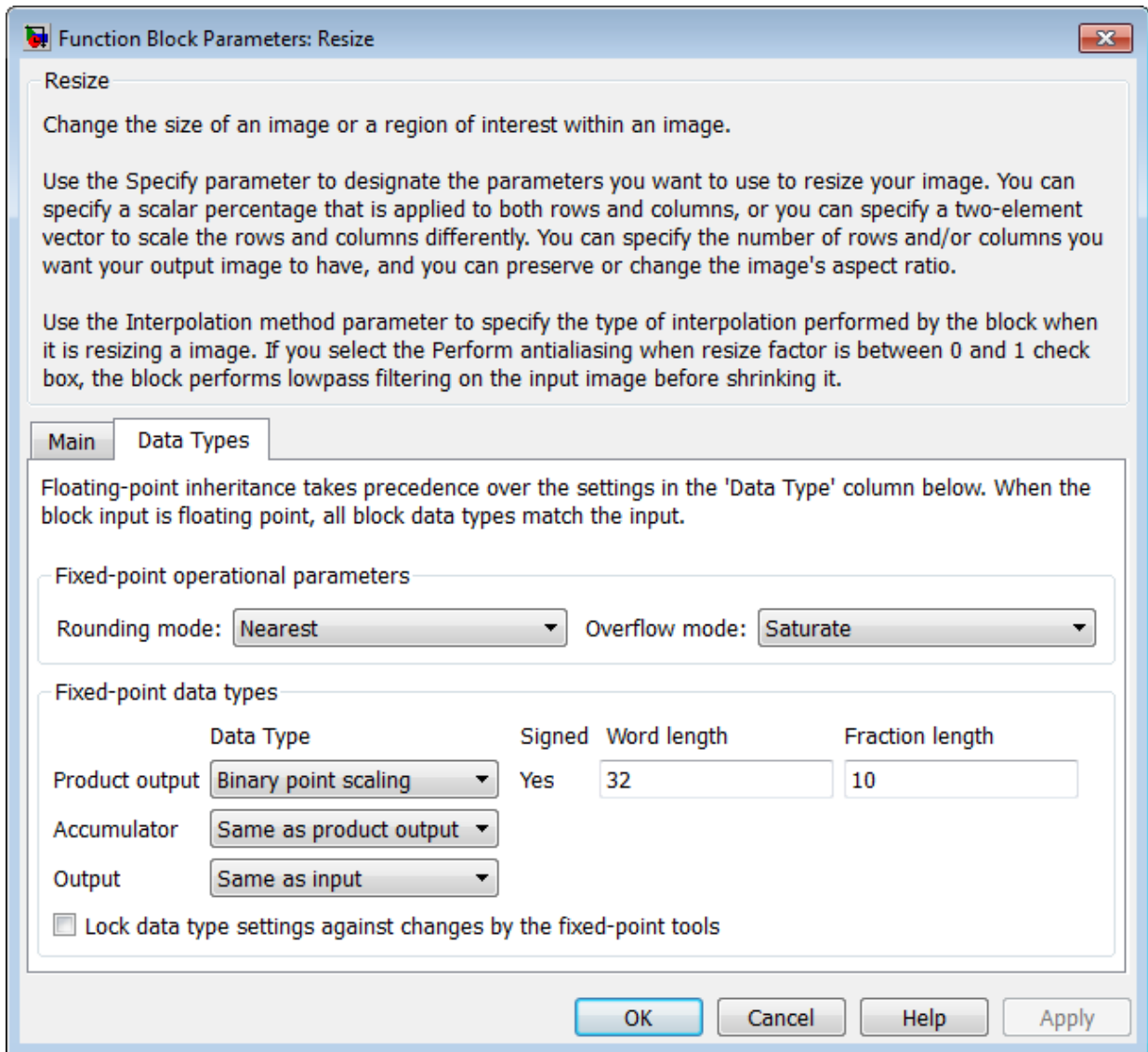
**Enable ROI processing**

Select this check box to resize a particular region of each image. This parameter is available when the **Specify** parameter is set to **Number of output rows and columns**, the **Interpolation method** parameter is set to **Nearest neighbor**, **Bilinear**, or **Bicubic**, and the **Perform antialiasing when resize factor is between 0 and 100** check box is not selected.

**Output flag indicating if any part of ROI is outside image bounds**

If you select this check box, the Flag port appears on the block. The block outputs 1 at this port if the ROI is completely or partially outside the input image. Otherwise, it outputs 0.

The **Data Types** pane of the Resize dialog box appears as shown in the following figure.

**Rounding mode**

Select the rounding mode for fixed-point operations.

**Overflow mode**

Select the overflow mode for fixed-point operations.

### Interpolation weights table

Choose how to specify the word length of the values of the interpolation weights table. The fraction length of the interpolation weights table values is always equal to the word length minus one:

- When you select **Same as input**, the word length of the interpolation weights table values match that of the input to the block.
- When you select **Binary point scaling**, you can enter the word length of the interpolation weights table values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length of the interpolation weights table values, in bits.

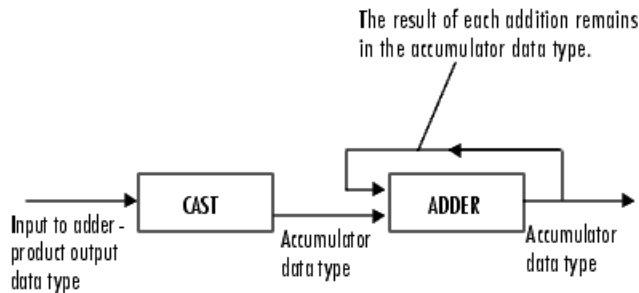
### Product output



As depicted in the preceding diagram, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator



As depicted in the preceding diagram, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Ward, Joseph and David R. Cok. "Resampling Algorithms for Image Resizing and Rotation", *Proc. SPIE Digital Image Processing Applications*, vol. 1075, pp. 260-269, 1989.
- [2] Wolberg, George. *Digital Image Warping*. Washington: IEEE Computer Society Press, 1990.

## See Also

Rotate	Computer Vision System Toolbox software
Shear	Computer Vision System Toolbox software
Translate	Computer Vision System Toolbox software
imresize	Image Processing Toolbox software

**Introduced before R2006a**

## Resize (To Be Removed)

Enlarge or shrink image sizes

### Library

Geometric Transformations

vipgeotforms



### Description

---

**Note:** This Resize block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated `Resize` block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**



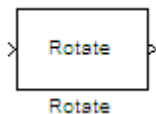
# Rotate

Rotate image by specified angle

## Library

Geometric Transformations

visiongeotforms



## Description

Use the Rotate block to rotate an image by an angle specified in radians.

---

**Note:** This block supports intensity and color images on its ports.

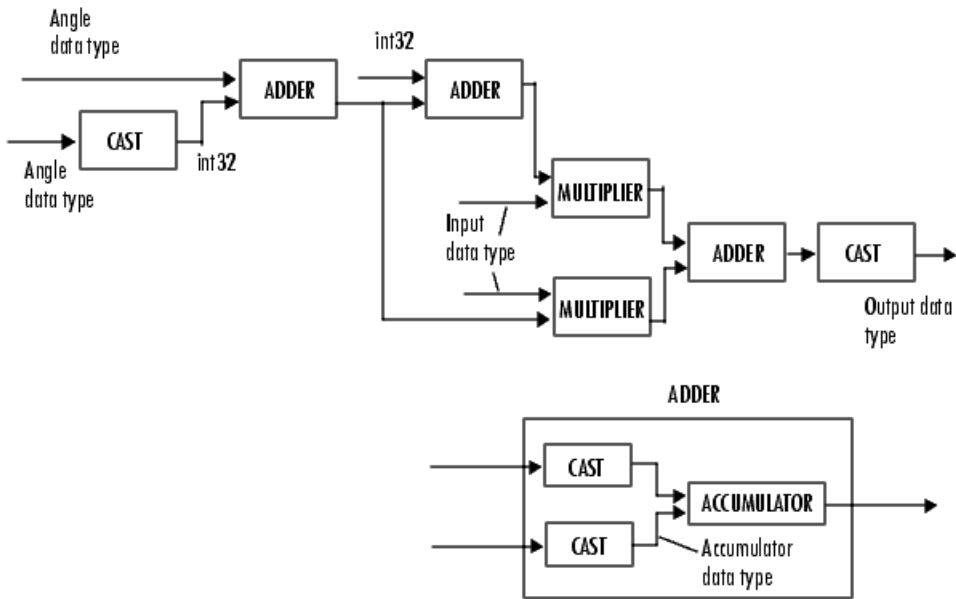
---

Port	Description
<b>Image</b>	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes
<b>Angle</b>	Rotation angle
<b>Output</b>	Rotated matrix

The **Rotate** block uses the 3-pass shear rotation algorithm to compute its values, which is different than the algorithm used by the `imrotate` function in the Image Processing Toolbox.

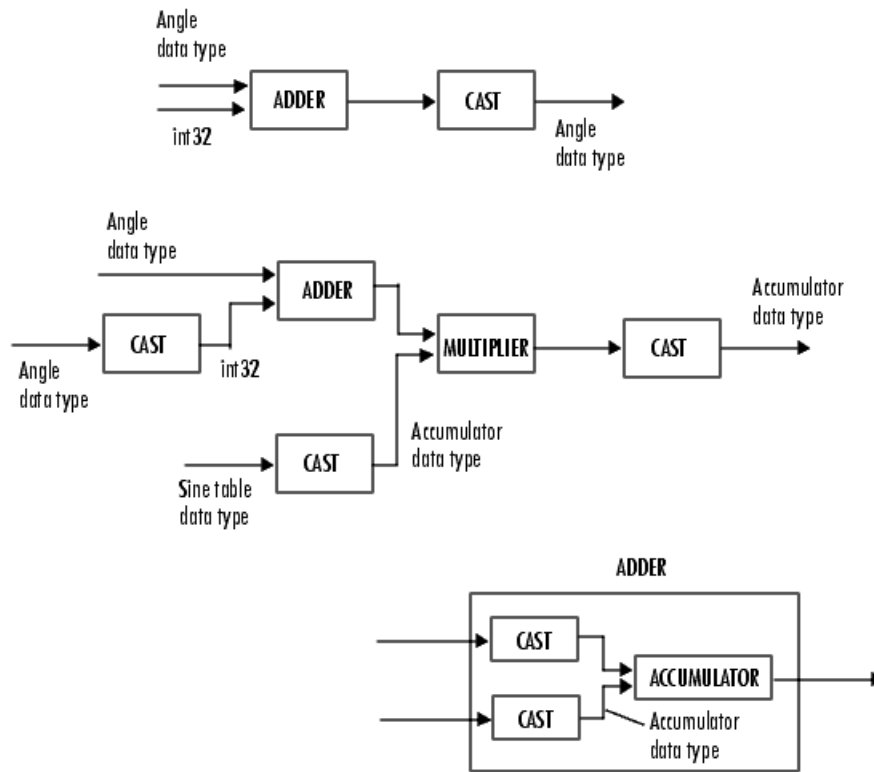
## Fixed-Point Data Types

The following diagram shows the data types used in the **Rotate** block for bilinear interpolation of fixed-point signals.



You can set the angle values, product output, accumulator, and output data types in the block mask.

The Rotate block requires additional data types. The Sine table value has the same word length as the angle data type and a fraction length that is equal to its word length minus one. The following diagram shows how these data types are used inside the block.



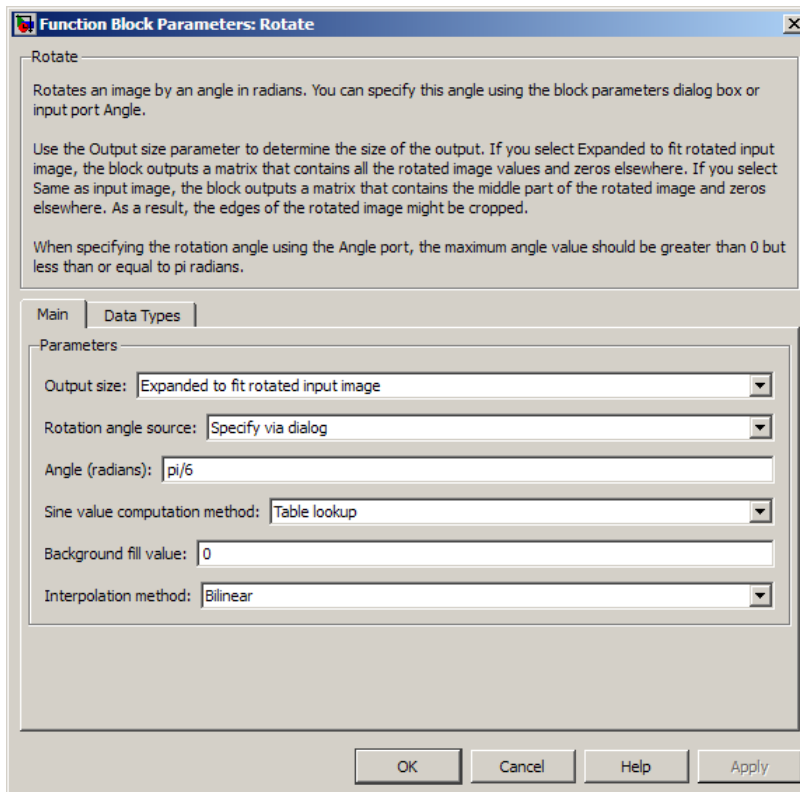

---

**Note** If overflow occurs, the rotated image might appear distorted.

---

## Dialog Box

The **Main** pane of the Rotate dialog box appears as shown in the following figure.



### Output size

Specify the size of the rotated matrix. If you select **Expanded to fit rotated input image**, the block outputs a matrix that contains all the rotated image values. If you select **Same as input image**, the block outputs a matrix that contains the middle part of the rotated image. As a result, the edges of the rotated image might be cropped. Use the **Background fill value** parameter to specify the pixel values outside the image.

### Rotation angle source

Specify how to enter your rotation angle. If you select **Specify via dialog**, the **Angle (radians)** parameter appears in the dialog box.

If you select **Input port**, the Angle port appears on the block. The block uses the input to this port at each time step as your rotation angle. The input to the Angle port must be the same data type as the input to the **I** port.

### Angle (radians)

Enter a real, scalar value for your rotation angle. This parameter is visible if, for the **Rotation angle source** parameter, you select **Specify via dialog**.

When the rotation angle is a multiple of  $\pi/2$ , the block uses a more efficient algorithm. If the angle value you enter for the **Angle (radians)** parameter is within 0.00001 radians of a multiple of  $\pi/2$ , the block rounds the angle value to the multiple of  $\pi/2$  before performing the rotation.

### Maximum angle (enter pi radians to accommodate all positive and negative angles)

Enter the maximum angle by which to rotate the input image. Enter a scalar value, between 0 and  $\pi$  radians. The block determines which angle,  $0 \leq \text{angle} \leq \text{max angle}$ , requires the largest output matrix and sets the dimensions of the output port accordingly.

This parameter is visible if you set the **Output size** parameter, to **Expanded to fit rotated input image**, and the **Rotation angle source** parameter to **Input port**.

### Display rotated image in

Specify how the image is rotated. If you select **Center**, the image is rotated about its center point. If you select **Top-left corner**, the block rotates the image so that two corners of the rotated input image are always in contact with the top and left sides of the output image.

This parameter is visible if, for the **Output size** parameter, you select **Expanded to fit rotated input image**, and, for the **Rotation angle source** parameter, you select **Input port**.

### Sine value computation method

Specify the value computation method. If you select **Trigonometric function**, the block computes sine and cosine values it needs to calculate the rotation of your image during the simulation. If you select **Table lookup**, the block computes and stores the trigonometric values it needs to calculate the rotation of your image before the simulation starts. In this case, the block requires extra memory.

### Background fill value

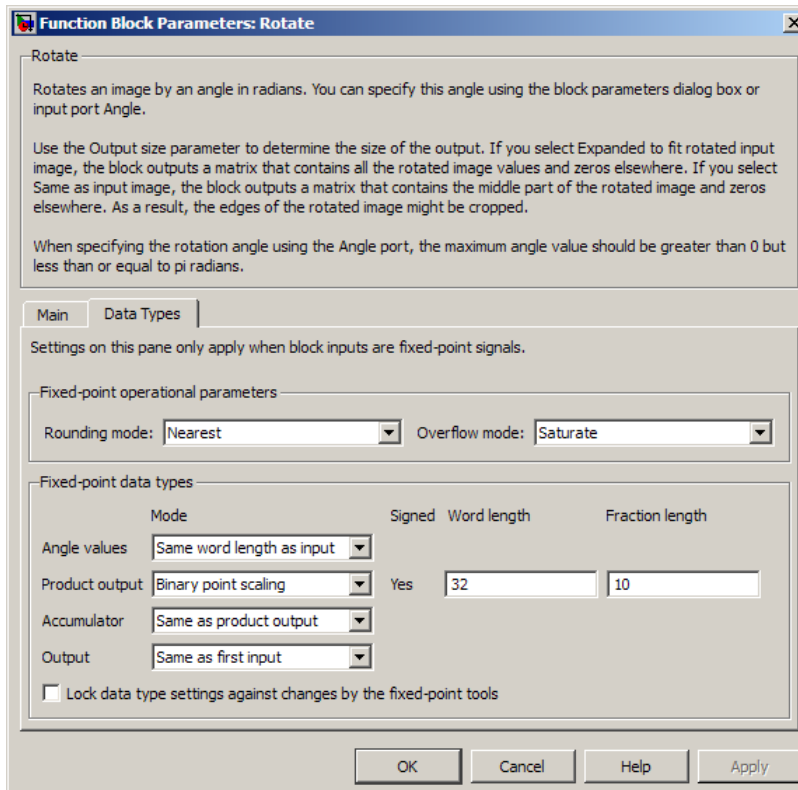
Specify a value for the pixels that are outside the image.

### Interpolation method

Specify which interpolation method the block uses to rotate the image. If you select **Nearest neighbor**, the block uses the value of one nearby pixel for the new pixel value. If you select **Bilinear**, the new pixel value is the weighted average of the four nearest pixel values. If you select **Bicubic**, the new pixel value is the weighted average of the sixteen nearest pixel values.

The number of pixels the block considers affects the complexity of the computation. Therefore, the **Nearest-neighbor** interpolation is the most computationally efficient. However, because the accuracy of the method is proportional to the number of pixels considered, the **Bicubic** method is the most accurate. For more information, see “Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods” in the *Computer Vision System Toolbox User's Guide*.

The **Data Types** pane of the Rotate dialog box appears as shown in the following figure.



### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Angle values

Choose how to specify the word length and the fraction length of the angle values.

- When you select **Same word length as input**, the word length of the angle values match that of the input to the block. In this mode, the fraction length of the angle values is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the angle values.
- When you select **Specify word length**, you can enter the word length of the angle values, in bits. The block automatically sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the angle values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the angle values. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

This parameter is only visible if, for the **Rotation angle source** parameter, you select **Specify via dialog**.

### Product output

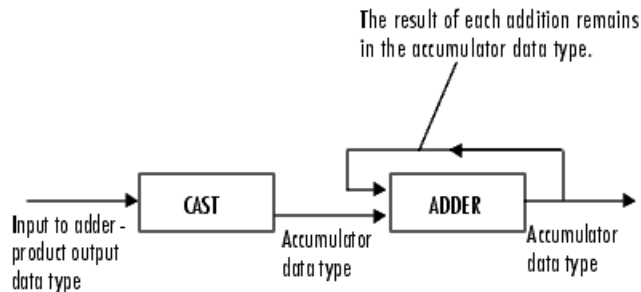


As depicted in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same as first input**, these characteristics match those of the input to the block.

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Accumulator



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.



- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## Supported Data Types

Port	Supported Data Types
Image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>
Angle	Same as Image port
Output	Same as Image port

If the data type of the input signal is floating point, the output signal is the same data type as the input signal.

## References

- [1] Wolberg, George. *Digital Image Warping*. Washington: IEEE Computer Society Press, 1990.

## See Also

Resize	Computer Vision System Toolbox software
--------	---

Translate	Computer Vision System Toolbox software
Shear	Computer Vision System Toolbox software
imrotate	Image Processing Toolbox software

**Introduced before R2006a**

# SAD (To Be Removed)

Perform 2-D sum of absolute differences (SAD)

## Library

vipobslib



## Description

---

**Note:** The SAD block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox software. Use the replacement block **Template Matching**.

---

**Introduced in R2011b**

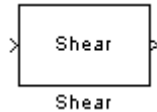
# Shear

Shift rows or columns of image by linearly varying offset

## Library

Geometric Transformations

visiongeotforms



## Description

The Shear block shifts the rows or columns of an image by a gradually increasing distance left or right or up or down.

---

**Note:** This block supports intensity and color images on its ports.

---

Port	Input/Output	Supported Data Types	Complex Values Supported
Image	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
S	Two-element vector that represents the number of pixels by which you want to shift your first and last rows or columns	Same as I port	No
Output	Shifted image	Same as I port	No

If the data type of the input to the I port is floating point, the input to the S port of this block must be the same data type. Also, the block output is the same data type.

Use the **Shear direction** parameter to specify whether you want to shift the rows or columns. If you select **Horizontal**, the first row has an offset equal to the first element of the **Row/column shear values [first last]** vector. The following rows have an offset that linearly increases up to the value you enter for the last element of the **Row/column shear values [first last]** vector. If you select **Vertical**, the first column has an offset equal to the first element of the **Row/column shear values [first last]** vector. The following columns have an offset that linearly increases up to the value you enter for the last element of the **Row/column shear values [first last]** vector.

Use the **Output size after shear** parameter to specify the size of the sheared image. If you select **Full**, the block outputs a matrix that contains the entire sheared image. If you select **Same as input image**, the block outputs a matrix that is the same size as the input image and contains the top-left portion of the sheared image. Use the **Background fill value** parameter to specify the pixel values outside the image.

Use the **Shear values source** parameter to specify how to enter your shear parameters. If you select **Specify via dialog**, the **Row/column shear values [first last]** parameter appears in the dialog box. Use this parameter to enter a two-element vector that represents the number of pixels by which you want to shift your first and last rows or columns. For example, if for the **Shear direction** parameter you select **Horizontal** and, for the **Row/column shear values [first last]** parameter, you enter [50 150], the block moves the top-left corner 50 pixels to the right and the bottom left corner of the input image 150 pixels to the right. If you want to move either corner to the left, enter negative values. If for the **Shear direction** parameter you select **Vertical** and, for the **Row/column shear values [first last]** parameter, you enter [-10 50], the block moves the top-left corner 10 pixels up and the top right corner 50 pixels down. If you want to move either corner down, enter positive values.

Use the **Interpolation method** parameter to specify which interpolation method the block uses to shear the image. If you select **Nearest neighbor**, the block uses the value of the nearest pixel for the new pixel value. If you select **Bilinear**, the new pixel value is the weighted average of the two nearest pixel values. If you select **Bicubic**, the new pixel value is the weighted average of the four nearest pixel values.

The number of pixels the block considers affects the complexity of the computation. Therefore, the nearest-neighbor interpolation is the most computationally efficient. However, because the accuracy of the method is proportional to the number of pixels considered, the bicubic method is the most accurate. For more information, see “Nearest

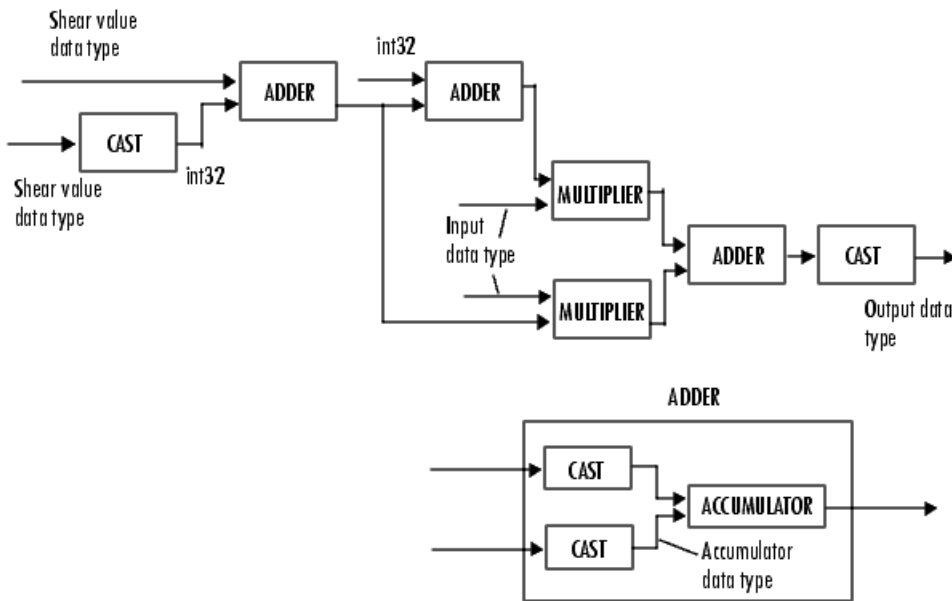
Neighbor, Bilinear, and Bicubic Interpolation Methods” in the *Computer Vision System Toolbox User's Guide*.

If, for the **Shear values source** parameter, you select **Input port**, the S port appears on the block. At each time step, the input to the S port must be a two-element vector that represents the number of pixels by which to shift your first and last rows or columns.

If, for the **Output size after shear** parameter, you select **Full**, and for the **Shear values source** parameter, you select **Input port**, the **Maximum shear value** parameter appears in the dialog box. Use this parameter to enter a real, scalar value that represents the maximum number of pixels by which to shear your image. The block uses this parameter to determine the size of the output matrix. If any input to the S port is greater than the absolute value of the **Maximum shear value** parameter, the block saturates to the maximum value.

## Fixed-Point Data Types

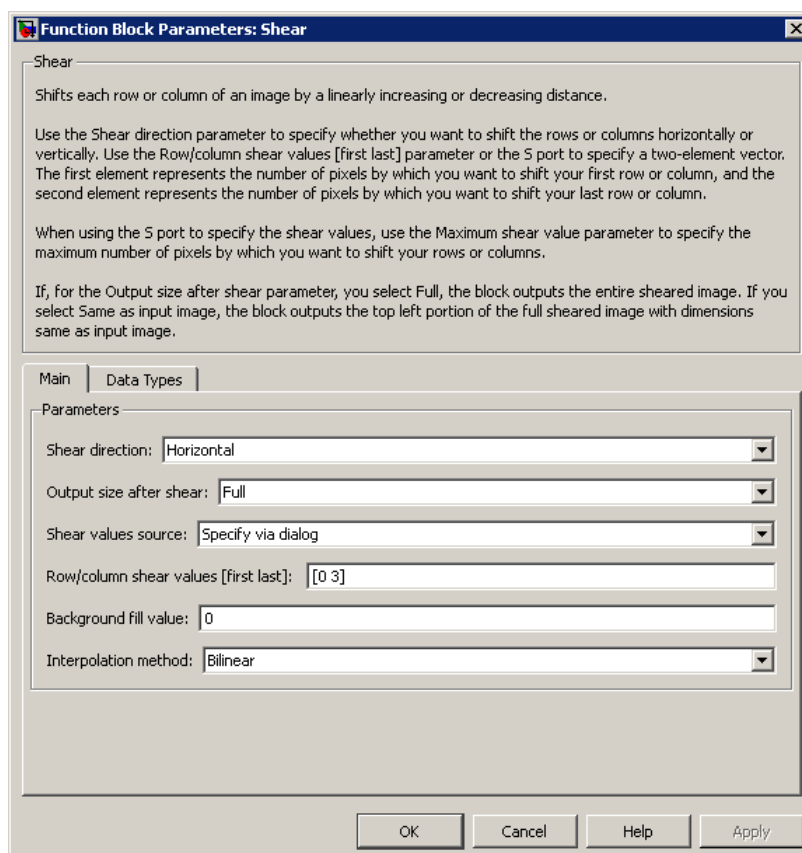
The following diagram shows the data types used in the Shear block for bilinear interpolation of fixed-point signals.



You can set the product output, accumulator, and output data types in the block mask.

## Dialog Box

The **Main** pane of the Shear dialog box appears as shown in the following figure.



### Shear direction

Specify whether you want to shift the rows or columns of the input image. Select **Horizontal** to linearly increase the offset of the rows. Select **Vertical** to steadily increase the offset of the columns.

### **Output size after shear**

Specify the size of the sheared image. If you select **Full**, the block outputs a matrix that contains the sheared image values. If you select **Same as input image**, the block outputs a matrix that is the same size as the input image and contains a portion of the sheared image.

### **Shear values source**

Specify how to enter your shear parameters. If you select **Specify via dialog**, the **Row/column shear values [first last]** parameter appears in the dialog box. If you select **Input port**, port **S** appears on the block. The block uses the input to this port at each time step as your shear value.

### **Row/column shear values [first last]**

Enter a two-element vector that represents the number of pixels by which to shift your first and last rows or columns. This parameter is visible if, for the **Shear values source** parameter, you select **Specify via dialog**.

### **Maximum shear value**

Enter a real, scalar value that represents the maximum number of pixels by which to shear your image. This parameter is visible if, for the **Shear values source** parameter, you select **Input port**.

### **Background fill value**

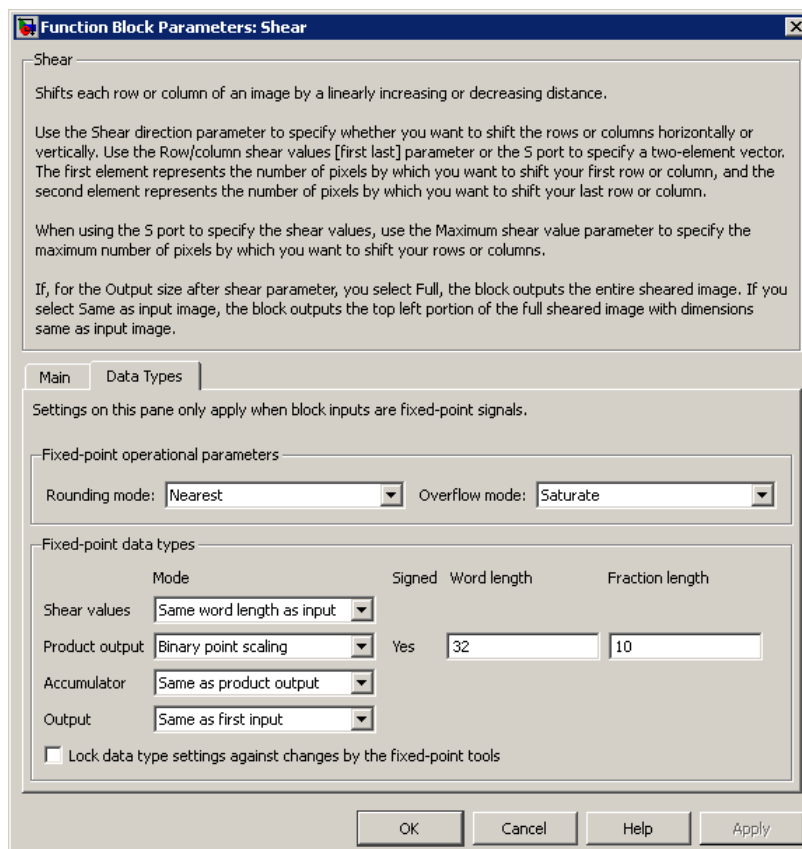
Specify a value for the pixels that are outside the image. This parameter is tunable.

### **Interpolation method**

Specify which interpolation method the block uses to shear the image. If you select **Nearest neighbor**, the block uses the value of one nearby pixel for the new pixel value. If you select **Bilinear**, the new pixel value is the weighted average of the two nearest pixel values. If you select **Bicubic**, the new pixel value is the weighted average of the four nearest pixel values.

The **Data Types** pane of the Shear dialog box appears as shown in the following figure.





## Rounding mode

Select the rounding mode for fixed-point operations.

## Overflow mode

Select the overflow mode for fixed-point operations.

## Shear values

Choose how to specify the word length and the fraction length of the shear values.

- When you select **Same word length as input**, the word length of the shear values match that of the input to the block. In this mode, the fraction length of the shear values is automatically set to the binary-point only scaling that provides

you with the best precision possible given the value and word length of the shear values.

- When you select **Specify word length**, you can enter the word length of the shear values, in bits. The block automatically sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the shear values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the shear values. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

This parameter is visible if, for the **Shear values source** parameter, you select **Specify via dialog**.

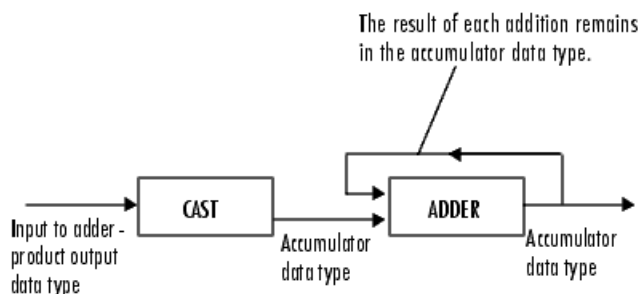
### Product output



As depicted in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same as first input**, these characteristics match those of the first input to the block at the I port.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Accumulator



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the first input to the block at the I port.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the first input to the block at the I port.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

### Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

- [1] Wolberg, George. *Digital Image Warping*. Washington: IEEE Computer Society Press, 1990.

## See Also

Resize	Computer Vision System Toolbox software
Rotate	Computer Vision System Toolbox software
Translate	Computer Vision System Toolbox software

**Introduced before R2006a**

# Template Matching

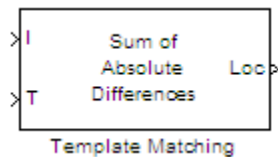
Locate a template in an image

## Library

Analysis & Enhancement

visionanalysis

## Description



The `Template Matching` block finds the best match of a template within an input image. The block computes match metric values by shifting a template over a region of interest or the entire image, and then finds the best match location.

## Port Description

Port	Supported Data Types
<b>I</b> (Input Image)	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, unsigned or both)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>
<b>T</b> (Template)	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, unsigned or both)</li> <li>• Boolean</li> </ul>

Port	Supported Data Types
	<ul style="list-style-type: none"> <li>• 8-bit unsigned integers</li> </ul>
<b>ROI</b> (Region of Interest, [x y width height])	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, unsigned or both)</li> <li>• Boolean</li> <li>• 8-bit unsigned integers</li> </ul>
<b>Metric</b> (Match Metric Values)	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, unsigned or both)</li> <li>• Boolean</li> <li>• 32-bit unsigned integers</li> </ul>
<b>Loc</b> (Best match location [x y])	<ul style="list-style-type: none"> <li>• 32-bit unsigned integers</li> </ul>
<b>NMetric</b> (Metric values in Neighborhood of best match)	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed, unsigned or both)</li> <li>• Boolean</li> <li>• 8-bit unsigned integers</li> </ul>
<b>NValid</b> (Neighborhood valid)	<ul style="list-style-type: none"> <li>• Boolean</li> </ul>
<b>ROIValid</b> (ROI valid)	<ul style="list-style-type: none"> <li>• Boolean</li> </ul>

## Using the Template Matching Block

### Choosing an Output Option

The block outputs either a matrix of match metric values or the zero-based location of the best template match. The block outputs the matrix to the **Metric** port or the location to the **Loc** port. Optionally, the block can output an  $N \times N$  matrix of neighborhood match metric values to the **NMetric** port.

## Input and Output Signal Sizes

The **Template Matching** block does not pad the input data. Therefore, it can only compute values for the match metrics between the input image and the template, where the template is positioned such that it falls entirely on the input image. A set of all such positions of the template is termed as the *valid* region of the input image. The size of the valid region is the difference between the sizes of the input and template images plus one.

$$\text{size}_{\text{valid}} = \text{size}_{\text{input}} - \text{size}_{\text{template}} + 1$$

The output at the **Metric** port for the **Match** metric mode is of the valid image size. The output at the **Loc** port for the **Best match index** mode is a two-element vector of indices relative to the top-left corner of the input image.

The neighborhood metric output at the **NMetric** port is of the size  $N \times N$ , where  $N$  must be an odd number specified in the block mask.

## Defining the Region of Interest (ROI)

To perform template matching on a subregion of the input image, select the **Enable ROI processing** check box. This check box adds the **ROI** input port to the **Template Matching** block. The ROI processing option is only available for the **Best match index** mode.

The ROI port requires a four-element vector that defines a rectangular area. The first two elements represent  $[x \ y]$  coordinates for the upper-left corner of the region. The second two elements represent the width and height of the ROI. The block outputs the best match location index relative to the top left corner of the input image.

## Choosing a Match Metric

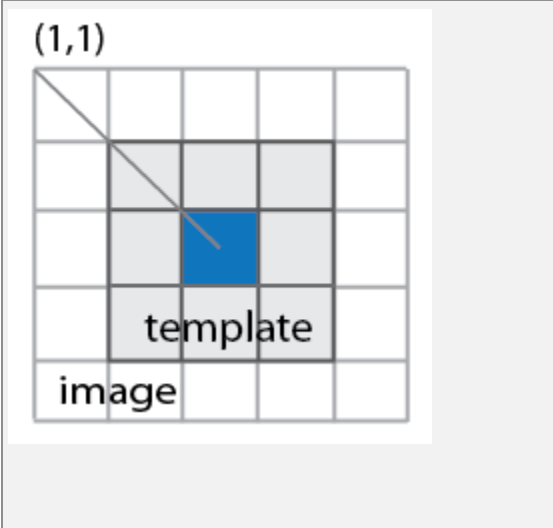
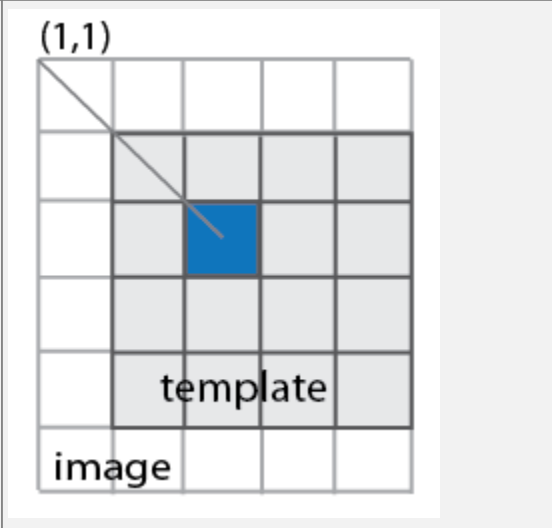
The block computes the match metric at each step of the iteration. Choose the match metric that best suits your application. The block calculates the global optimum for the best metric value. It uses the valid subregion of the input image intersected by the ROI, if provided.

## Returning the Matrix of Match Metric Values

The matrix of the match metrics always implements single-step exhaustive window iteration. Therefore, the block computes the metric values at every pixel.

### Returning the Best Match Location

When in the ROI processing mode, the block treats the image around the ROI as an extension of the ROI subregion. Therefore, it computes the best match locations true to the actual boundaries of the ROI. The block outputs the best match coordinates, relative to the top-left corner of the image. The one-based [x y] coordinates of the location correspond to the center of the template. The following table shows how the block outputs the center coordinates for odd and even templates:

Odd number of pixels in template	Even number of pixels in template
 <p>(1,1)</p> <p>template</p> <p>image</p>	 <p>(1,1)</p> <p>template</p> <p>image</p>

### Returning the Neighborhood Match Metric around the Best Match

When you select **Best match location** to return the matrix of metrics in a neighborhood around the best match, an exhaustive loop computes all the metric values for the  $N$ -by- $N$  neighborhood. This output is particularly useful for performing template matching with subpixel accuracy.

### Choosing a Search Method

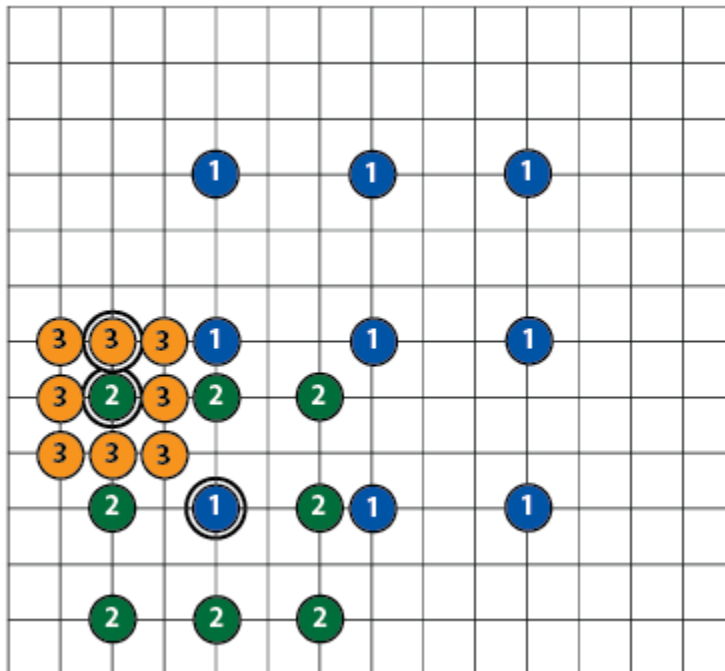
When you select **Best match location** as the output option, you can choose to use either **Exhaustive** or **Three-step** search methods.

The **Exhaustive** search method is computationally intensive because it searches at every pixel location of the image. However, this method provides a more precise result.



The **Three - step** search method is a fast search that uses a neighborhood approach, which does not inspect every pixel. The search starts with a step size equal to or slightly greater than half of the maximum search range and then employs the following steps:

- 1 The block compares nine search points in each step. There is a central point and eight search points located on the search area boundary.
- 2 The block decrements the step size by one, after each step, ending the search with a step size of one pixel.
- 3 At each new step, the block moves the search center to the best matching point resulting from the previous step. The number one blue circles in the figure below represent a search with a starting step size of three. The number two green circles represent the next search, with step size of two, centered around the best match found from the previous search. Finally, the number three orange circles represent the final search, with step size of one, centered around the previous best match.



*Three-Step Search*

## Using the ROIValid and NValid flags for Diagnostics

The **ROIValid** and **NValid** ports represent boolean flags, which track the valid Region of Interest (ROI) and neighborhood. You can use these flags to communicate with the downstream blocks and operations.

### Valid Region of Interest

If you select the **Output flag indicating if ROI is valid** check box, the block adds the **ROIValid** port. If the ROI lies partially outside the valid image, the block only processes the intersection of the ROI and the valid image. The block sets the ROI flag output to this port as follows:

- True, set to 1 indicating the ROI lies completely inside the valid part of the input image.
- False, set to 0 indicating the ROI lies completely or partially outside of the valid part of the input image.

### Valid Neighborhood

The neighborhood matrix of metric values is valid inside of the Region of Interest (ROI). You can use the Boolean flag at the **NValid** port to track the valid neighborhood region. The block sets the neighborhood **NValid** boolean flag output as follows:

- True, set to 1 indicating that the neighborhood containing the best match is completely inside the region of interest.
- False, set to 0 indicating that the neighborhood containing the best match is completely or partially outside of the region of interest.

## Algorithm

The match metrics use a difference equation with general form:

$$d_p(x, y) = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}$$

$l_n^p$  denotes the metric space  $(R^n, d_p)$  for  $R^n$   $n > 1$ .

- **Sum of Absolute Differences (SAD)**

This metric is also known as the *Taxicab* or *Manhattan Distance* metric. It sums the absolute values of the differences between pixels in the original image and the corresponding pixels in the template image. This metric is the  $l^1$  norm of the difference image. The lowest SAD score estimates the best position of template within the search image. The general SAD distance metric becomes:

$$d_1(I_j, T) = \sum_{i=1}^n |I_{i,j} - T_i|$$

- **Sum of Squared Differences (SSD)**

This metric is also known as the *Euclidean Distance* metric. It sums the square of the absolute differences between pixels in the original image and the corresponding pixels in the template image. This metric is the square of the  $l^2$  norm of the difference image. The general SSD distance metric becomes:

$$d_2(I_j, T) = \sum_{i=1}^n |I_{i,j} - T_i|^2$$

- **Maximum Absolute Difference (MaxAD)**

This metric is also known as the *Uniform Distance* metric. It sums the maximum of absolute values of the differences between pixels in the original image and the corresponding pixels in the template image. This distance metric provides the  $l^\infty$  norm of the difference image. The general MaxAD distance metric becomes:

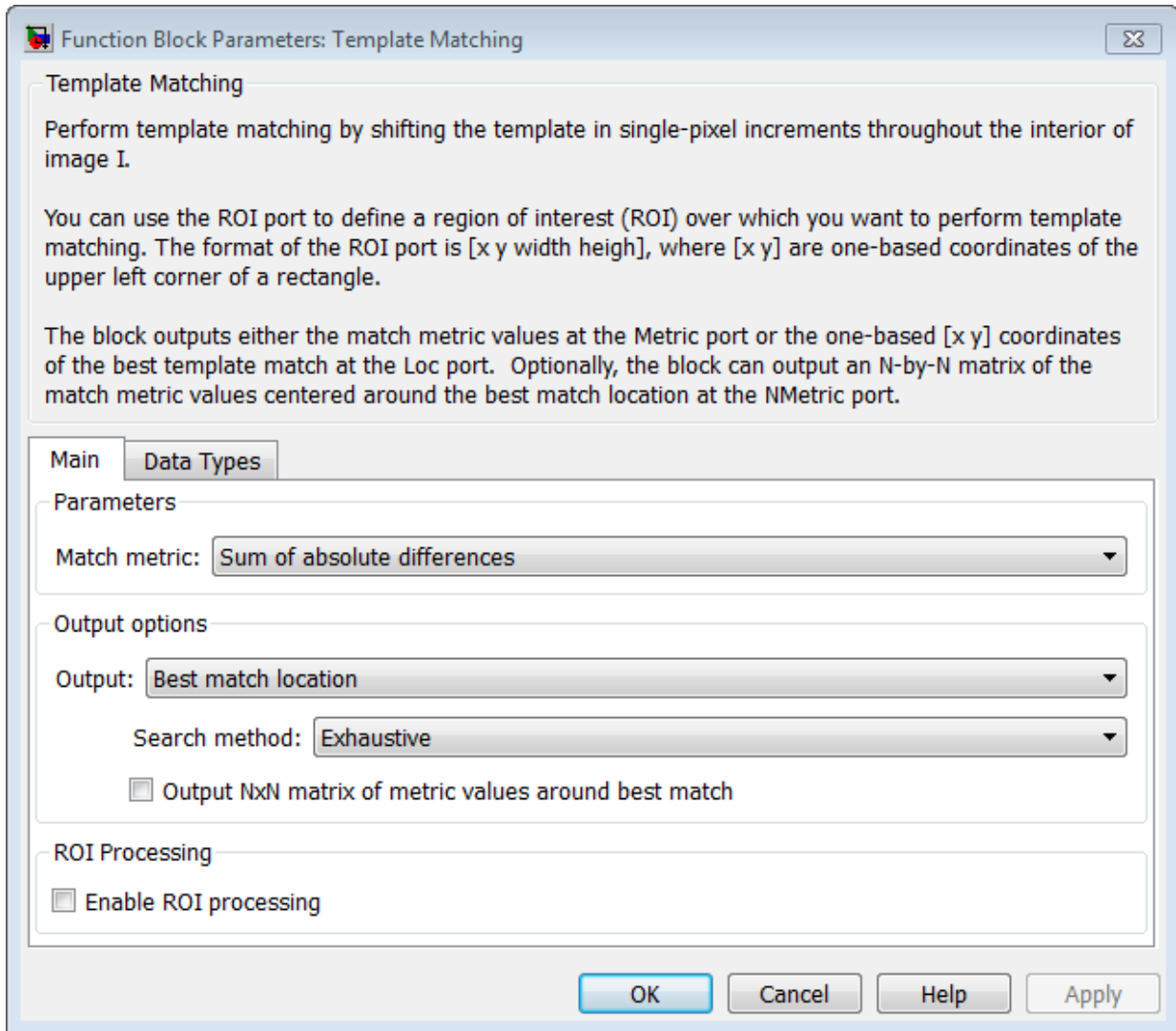
$$d_\infty(I_j, T) = \lim_{x \rightarrow \infty} \sum_{i=1}^n |I_{i,j} - T_i|^x$$

which simplifies to:

$$d_\infty(I_j, T) = \max_i |I_{i,j} - T_i|^p$$

## Main Dialog Box

The **Main** pane of the Template Matching block appears as shown in the following figure.



## Match metric

Select one of three types of match metrics:

- Sum of absolute differences (SAD)
- Sum of squared differences (SSD)
- Maximum absolute difference (MaxAD)

## Output

Select one of two output types:

- **Metric matrix**  
Select this option to output the match metric matrix. This option adds the **Metric** output port to the block.
- **Best match location**  
Select this option to output the [x y] coordinates for the location of the best match. This option adds the **Loc** output port to the block. When you select **Best match location**, the **Search method**, **Output NxN matrix of metric values around best match**, and **Enable ROI processing** parameter options appear.

## Search method

This option appears when you select **Best match location** for the **Output** parameter. Select one of two search methods.

- Exhaustive
- Three-step

## Output NxN matrix of metric values around best match

This option appears when you select **Best match location** for the **Output** parameter. Select the check box to output a matrix of metric values centered around the best match. When you do so, the block adds the **NMetric** and **NValid** output ports.

## N

This option appears when you select the **Output NxN matrix of metric values around best match** check box. Enter an integer number that determines the size of the

$N$ -by- $N$  output matrix centered around the best match location index.  $N$  must be an odd number.

### **Enable ROI processing**

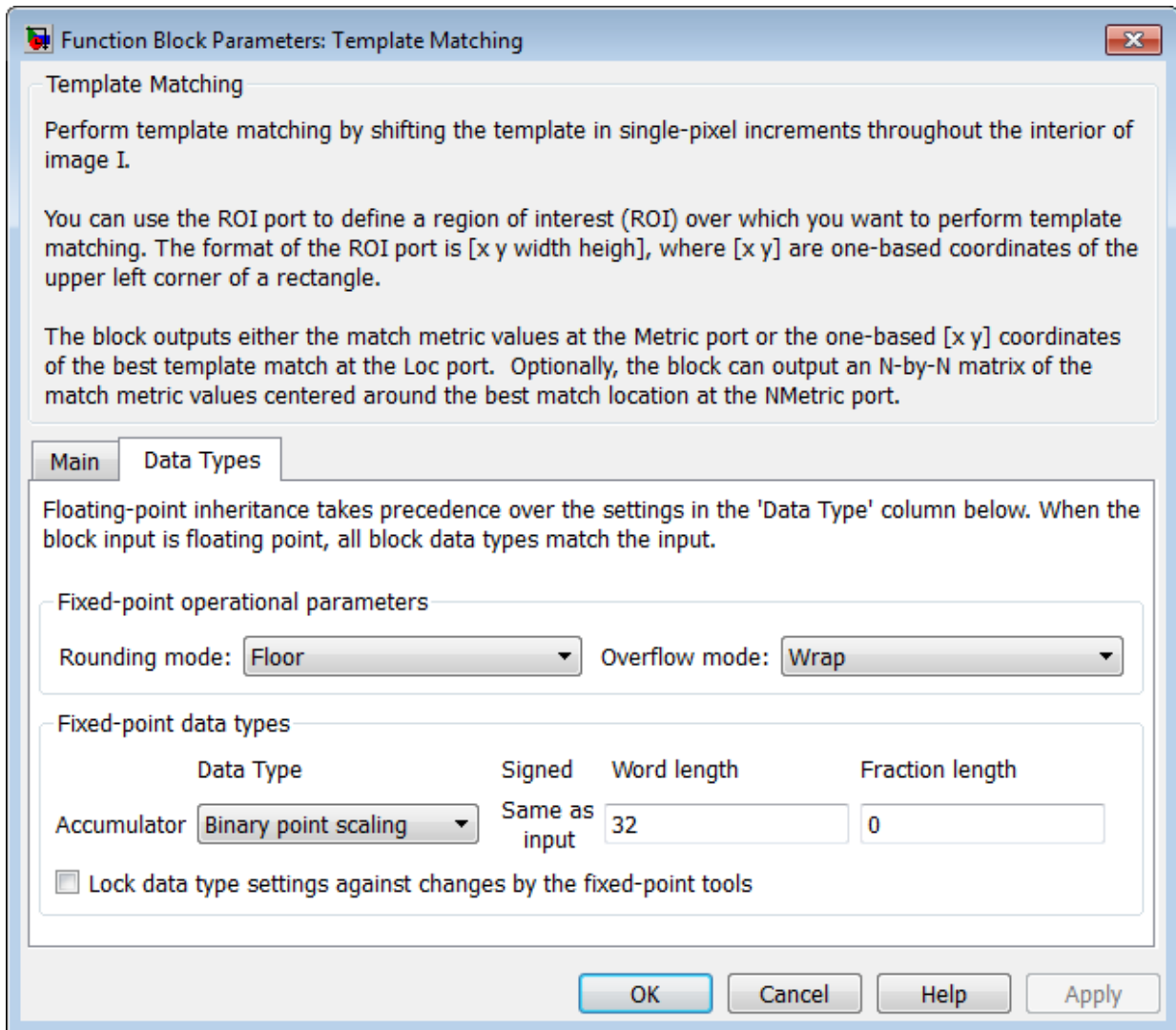
This option appears when you select **Best match location** for the **Output** parameter. Select the check box for the **Template Matching** block to perform region of interest processing. When you do so, the block adds the **ROI** input port, and the **Output flag indicating if ROI is valid** check box appears. The **ROI** input must have the format [x y width height], where [x y] are the coordinates of the upper-left corner of the ROI.

### **Output flag indicating if ROI is valid**

This option appears when you select the **Enable ROI processing** check box. Select the check box for the **Template Matching** block to indicate whether the ROI is within the valid region of the image boundary. When you do so, the block adds the **ROIValid** output port.

## **Data Types Dialog Box**

The **Data Types** pane of the **Template Matching** block dialog box appears as shown in the following figure.



## Rounding mode

Select the “Rounding Modes” for fixed-point operations.

## Overflow mode

Select the Overflow mode for fixed-point operations.

- Wrap
- Saturate

## Product output

- Use this parameter to specify how to designate the product output word and fraction lengths. Refer to “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:
  - When you select **Same as input**, these characteristics match those of the input to the block.
  - When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
  - When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

## Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

- When you select **Same as product output** the characteristics match the characteristics of the product output. See “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

When you select **Binary point scaling**, you can enter the **Word length** and the **Fraction length** of the accumulator, in bits.

When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the **Accumulator**. All signals in the Computer Vision System Toolbox software have a bias of 0.

The block casts inputs to the **Accumulator** to the accumulator data type. It adds each element of the input to the output of the adder, which remains in



the accumulator data type. Use this parameter to specify how to designate this accumulator word and fraction lengths.

## Output

Choose how to specify the **Word length**, **Fraction length** and **Slope** of the **Template Matching** output:

- When you select **Same as first input**, these characteristics match the characteristics of the accumulator.
- When you select **Binary point scaling**, you can enter the **Word length** and **Fraction length** of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the **Word length**, in bits, and the **Slope** of the output. All signals in the Computer Vision System Toolbox software have a bias of 0.

The **Output** parameter on the Data Types pane appears when you select **Metric matrix** or if you select **Best match location** and the **Output NxN matrix of metric values around best match** check box is selected.

## Lock data type settings against change by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## Reference

- [1] Koga T., et. Al. Motion-compensated interframe coding for video conferencing. In National Telecommunications Conference. Nov. 1981, G5.3.1–5, New Orleans, LA.
- [2] Zakai M., "General distance criteria" *IEEE Transaction on Information Theory*, pp. 94–95, January 1964.
- [3] Yu, J., J. Amores, N. Sebe, Q. Tian, "A New Study on Distance Metrics as Similarity Measurement" IEEE International Conference on Multimedia and Expo, 2006 .

## See Also

Block Matching	Image Processing Toolbox
Video Stabilization	Computer Vision System Toolbox

**Introduced in R2009b**

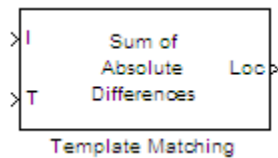
# Template Matching (To Be Removed)

Locate a template in an image

## Library

Analysis & Enhancement

## Description



---

**Note:** This Template Matching block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated `Template Matching` block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**

## To Multimedia File

Write video frames and audio samples to multimedia file

### Library

Sinks

visionsinks

### Description



The **To Multimedia File** block writes video frames, audio samples, or both to a multimedia (.avi, .wav, .wma, .mj2, .mp4, .m4v, or .wmv) file.

You can compress the video frames or audio samples by selecting a compression algorithm. You can connect as many of the input ports as you want. Therefore, you can control the type of video and/or audio the multimedia file receives.

---

**Note** This block supports code generation for platforms that have file I/O available. You cannot use this block with Simulink Desktop Real-Time software, because that product does not support file I/O.

This block performs best on platforms with Version 11 or later of Windows Media® Player software. This block supports only uncompressed RGB24 AVI files on Linux and Mac platforms.

Windows 7 UAC (User Account Control), may require administrative privileges to encode .mwv and .mwa files.

---

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Simulink Coder”, “Simulink Shared Library Dependencies”, and “Accelerating Simulink Models” for details.

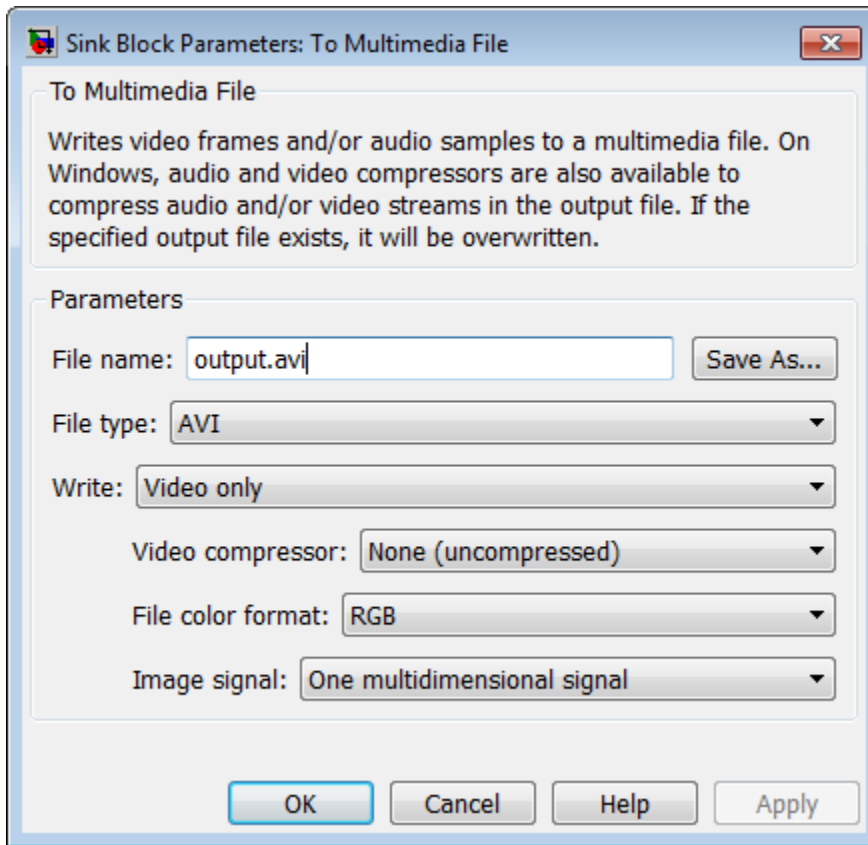
This block allows you to write `.wma`/`.mwv` streams to disk or across a network connection. Similarly, the `From Multimedia File` block allows you to read `.wma`/`.mwv` streams to disk or across a network connection. If you want to play an MP3/MP4 file in Simulink, but you do not have the codecs, you can re-encode the file as `.wma`/`.mwv`, which are supported by the Computer Vision System Toolbox.

## Ports

Port	Description
<b>Image</b>	$M$ -by- $N$ -by-3 matrix RGB, Intensity, or YCbCr 4:2:2 signal.
<b>R, G, B</b>	Matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B port must have the same dimensions and data type.
<b>Audio</b>	Vector of audio data
<b>Y, Cb, Cr</b>	Matrix that represents one frame of the YCbCr video stream. The Y, Cb, and Cr ports use the following dimensions: $Y: M \times N$ $Cb: M \times \frac{N}{2}$ $Cr: M \times \frac{N}{2}$

## Dialog Box

The **Main** pane of the `To Multimedia File` block dialog appears as follows.



**File name**

Specify the name of the multimedia file. The block saves the file in your current folder. To specify a different file or location, click the **Save As...** button.

**File type**

Specify the file type of the multimedia file. You can select `avi`, `wav`, `wma`, or `wmv`.

**Write**

Specify whether the block writes video frames, audio samples, or both to the multimedia file. You can select `Video` and `audio`, `Video only`, or `Audio only`.

**Video Quality**

Quality of the video, specified as an integer scalar in the range [0 100]. This parameter applies only when you set **File Name** to MPEG4 and **Write** to Video only. By default, this parameter is set to 75.

**Compression Factor (>1)**

Specify the compression factor as an integer scalar greater than 1. This parameter is applicable only when the **File type** is set to MJ2000 and **Video compressor** is set to Lossy. By default, this parameter is set to 10.

**Audio compressor**

Select the type of compression algorithm to use to compress the audio data. This compression reduces the size of the multimedia file. Choose **None (uncompressed)** to save uncompressed audio data to the multimedia file.

---

**Note:** The other items available in this parameter list are the audio compression algorithms installed on your system. For information about a specific audio compressor, see the documentation for that compressor.

---

**Audio data type**

Select the audio data type. You can use the **Audio data type** parameter only for uncompressed wave files.

**Video compressor**

Select the type of compression algorithm to use to compress the video data. This compression reduces the size of the multimedia file. Choose **None (uncompressed)** to save uncompressed video data to the multimedia file.

---

**Note:** The other items available in this parameter list are the video compression algorithms installed on your system. For information about a specific video compressor, see the documentation for that compressor.

---

**File color format**

Select the color format of the data stored in the file. You can select either RGB or YCbCr 4:2:2.

**Image signal**

Specify how the block accepts a color video signal. If you select **One multidimensional signal**, the block accepts an *M*-by-*N*-by-*P* color video signal,

where  $P$  is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port accepts one  $M$ -by- $N$  plane of an RGB video stream.

## Supported Data Types

For the block to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. Any other data type requires the pixel values between the minimum and maximum values supported by their data type.

Check the specific codecs you are using for supported audio rates.

Port	Supported Data Types	Supports Complex Values?
Image	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• Boolean</li><li>• 8-, 16- 32-bit signed integers</li><li>• 8-, 16- 32-bit unsigned integers</li></ul>	No
R, G, B	Same as Image port	No
Audio	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• 16-bit signed integers</li><li>• 32-bit signed integers</li><li>• 8-bit unsigned integers</li></ul>	No
Y, Cb, Cr	Same as Image port	No

## See Also

From Multimedia File Computer Vision System Toolbox

**Introduced before R2006a**



# To Video Display

Display video data

## Library

Sinks

visionsinks



## Description

The `To Video Display` block displays video frames. This block is capable of displaying high definition video at high frame rates. It provides a lightweight, high performance, simple display, which accepts RGB and YCbCr formatted images. This block also generates code.

The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra `.dll` files when doing so. The `packNGO` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGO` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Simulink Coder”, “Simulink Shared Library Dependencies”, and “Accelerating Simulink Models” for details.

For the block to display video data properly, double- and single-precision floating-point pixel values must be from 0 to 1. For any other data type, the pixel values must be between the minimum and maximum values supported by their data type.

You can set the display for full screen, normal or, to maintain one-to-one size. When you save the model, the size and position of the display window is saved. Any changes while working with the model should be saved again in order that these preferences are

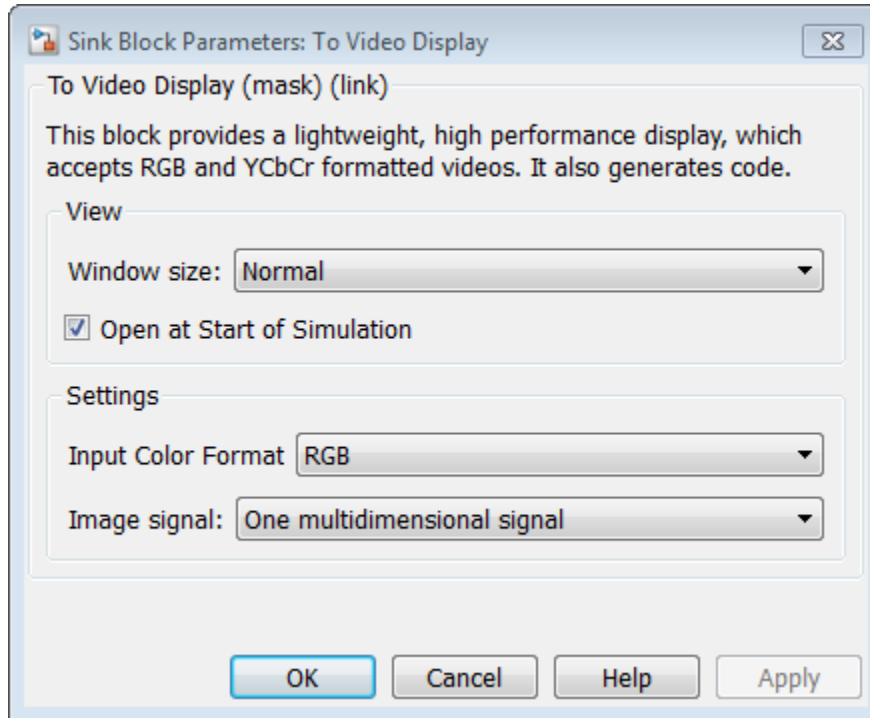
maintained when you run the model. The minimum display width of the window varies depending on your system's font size settings.

### Rapid Accelerator

When you set your model to run in “Accelerator Mode”, and do not select the **Open at Start of Simulation** option, the block will not be included during the run, and therefore the video display will not be visible.

## Dialog Box

The To Video Display dialog box appears as shown in the following figure.



### View Options

#### Window Size

Select **Full-screen** mode to display your video stream in a full screen window. To exit or return to other applications from full-screen, select the display and use the **Esc** key.

Select **Normal** mode to modify display size at simulation start time.

Select **True Size (1:1)** mode to display a one-to-one pixel ratio of input image at simulation start time. The block displays the same information per pixel and does not change size from the input size. You can change the display size after the model starts.

### Open at Start of Simulation

Select **Open at Start of Simulation** from the **View** menu for the display window to appear while running the model. If not selected, you can double click the block to display the window.

## Settings Options

### Input Color Format

Select the color format of the data stored in the input image.

Select **RGB** for the block to accept a matrix that represents one plane of the RGB video stream. Inputs to the **R**, **G**, or **B** ports must have the same dimension and data type.

Select **YCbCr 4:2:2** for the block to accept a matrix that represents one frame of the YCbCr video stream. The **Y** port accepts an  $M$ -by- $N$  matrix. The **Cb** and **Cr** ports accepts an  $M$ -by- $N/2$  matrix.

### Image Signal

Specify how the block accepts a color video signal.

Select **One multidimensional signal**, for the block to accept an  $M$ -by- $N$ -by-3 color video signal at one port.

Select **Separate color signals**, for additional ports to appear on the block. Each port accepts one  $M$ -by- $N$  plane of an RGB video stream.

## Supported Data Types

Port	Supported Data Types
Image	<ul style="list-style-type: none"> <li>Double-precision floating point</li> </ul>

Port	Supported Data Types
	<ul style="list-style-type: none"> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16, and 32-bit signed integer</li> <li>• 8-, 16, and 32-bit unsigned integer</li> </ul>
R, G, B	Same as Image port
YCbCr 4:2:2	Same as Image ports

## See Also

Frame Rate Display	Computer Vision System Toolbox software
From Multimedia File	Computer Vision System Toolbox software
To Multimedia File	Computer Vision System Toolbox software
Video To Workspace	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software

**Introduced before R2006a**

# Top-hat

Perform top-hat filtering on intensity or binary images

## Library

Morphological Operations

visionmorphops

## Description



The Top-hat block performs top-hat filtering on an intensity or binary image using a predefined neighborhood or structuring element. Top-hat filtering is the equivalent of subtracting the result of performing a morphological opening operation on the input image from the input image itself. This block uses flat structuring elements only.

Port	Input/Output	Supported Data Types	Complex Values Supported
I	Vector or matrix of intensity values	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>	No
Nhood	Matrix or vector of 1s and 0s that represents the neighborhood values	Boolean	No
Output	Scalar, vector, or matrix that represents the filtered image	Same as I port	No

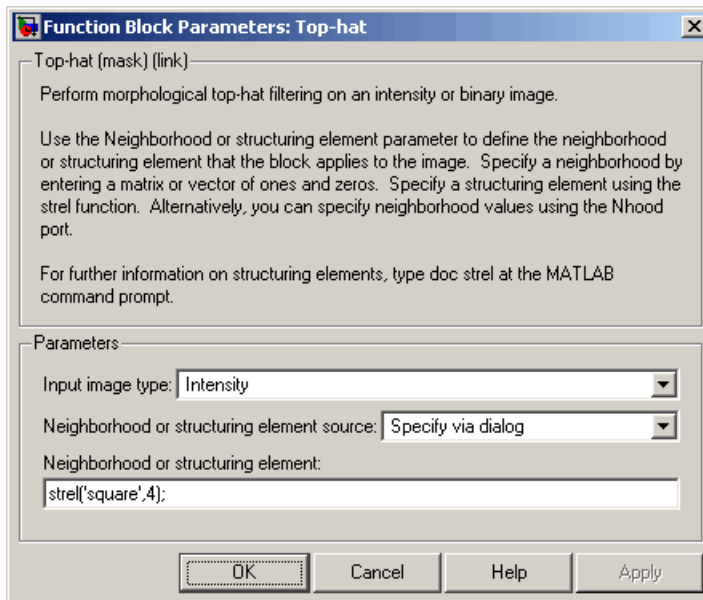
If your input image is a binary image, for the **Input image type** parameter, select **Binary**. If your input image is an intensity image, select **Intensity**.

Use the **Neighborhood or structuring element source** parameter to specify how to enter your neighborhood or structuring element values. If you select **Specify via dialog**, the **Neighborhood or structuring element** parameter appears in the dialog box. If you select **Input port**, the **Nhood port** appears on the block. Use this port to enter your neighborhood values as a matrix or vector of 1s and 0s. Choose your structuring element so that it matches the shapes you want to remove from your image. You can only specify it using the dialog box.

Use the **Neighborhood or structuring element** parameter to define the region the block moves throughout the image. Specify a neighborhood by entering a matrix or vector of 1s and 0s. Specify a structuring element with the `strel` function from the Image Processing Toolbox. If the structuring element is decomposable into smaller elements, the block executes at higher speeds due to the use of a more efficient algorithm.

## Dialog Box

The Top-hat dialog box appears as shown in the following figure.



**Input image type**

If your input image is a binary image, select **Binary**. If your input image is an intensity image, select **Intensity**.

**Neighborhood or structuring element source**

Specify how to enter your neighborhood or structuring element values. Select **Specify via dialog** to enter the values in the dialog box. Select **Input port** to use the Nhood port to specify the neighborhood values. You can only specify a structuring element using the dialog box.

**Neighborhood or structuring element**

If you are specifying a neighborhood, this parameter must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the **strel** function from the Image Processing Toolbox. This parameter is visible if, for the **Neighborhood or structuring element source** parameter, you select **Specify via dialog**.

**See Also**

Bottom-hat	Computer Vision System Toolbox software
Closing	Computer Vision System Toolbox software
Dilation	Computer Vision System Toolbox software
Erosion	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
Opening	Computer Vision System Toolbox software
imtophat	Image Processing Toolbox software
strel	Image Processing Toolbox software

**Introduced before R2006a**

## Trace Boundaries (To Be Removed)

Trace object boundaries in binary images

### Library

Analysis & Enhancement

### Description

---

**Note:** This Trace Boundaries block will be removed in a future release. It uses the zero-based, [row column] coordinate system. It is recommended that you replace this block with the updated Trace Boundary block that uses the one-based, [x y] coordinate system.

Refer to “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” in the R2011b Release Notes for details related to these changes.

---

**Introduced in R2011b**



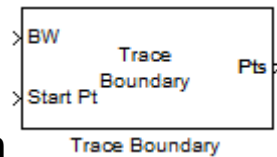
# Trace Boundary

Trace object boundaries in binary images

## Library

Analysis & Enhancement

visionanalysis



## Description

The Trace Boundary block traces object boundaries in binary images, where nonzero pixels represent objects and 0 pixels represent the background.

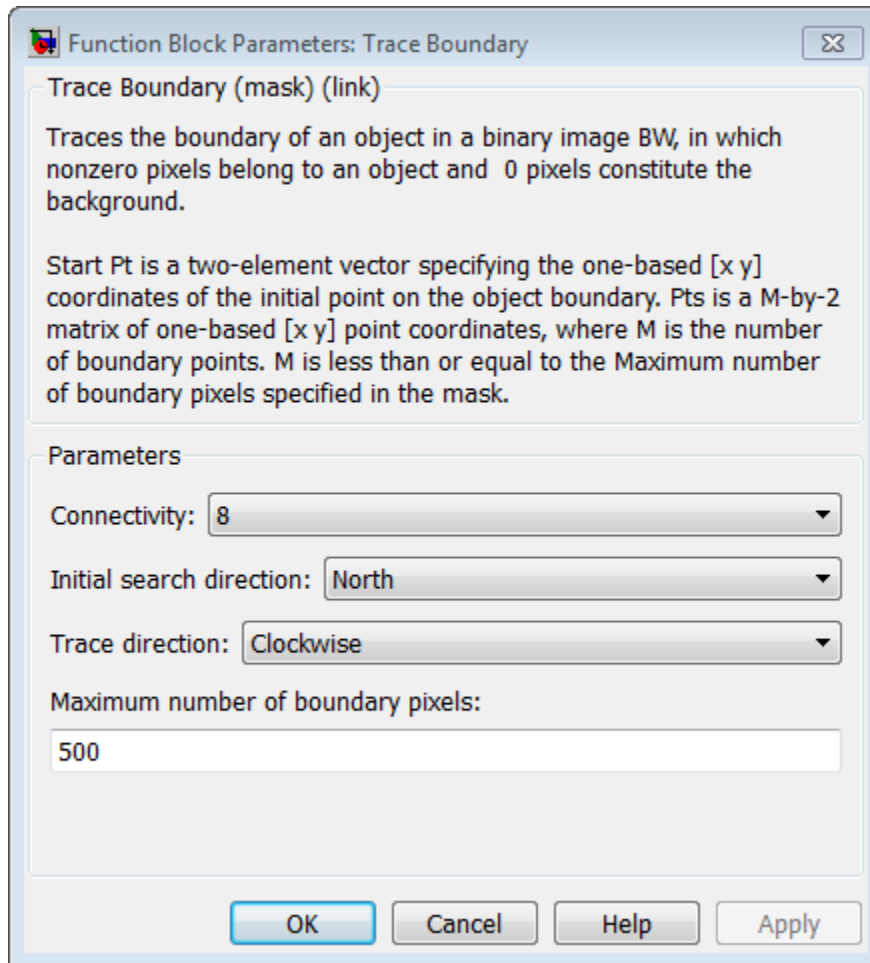
## Port Descriptions

Port	Input/Output	Supported Data Types
BW	Vector or matrix that represents a binary image	Boolean
Start Pt	One-based [x y] coordinates of the boundary starting point.	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>
Pts	$M$ -by-2 matrix of [x y] coordinates of the boundary points, where $M$ represents the number of traced boundary pixels. $M$ must	Same as Start Pts port

Port	Input/Output	Supported Data Types
	<p>be less than or equal to the value specified by the <b>Maximum number of boundary pixels</b> parameter.</p> $\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_m & y_m \end{bmatrix}$	

## Dialog Box

The Trace Boundary dialog box appears as shown in the following figure.



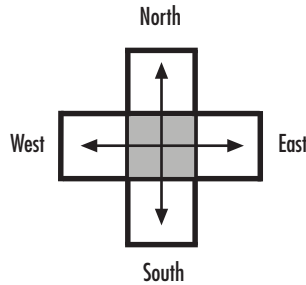
### Connectivity

Specify which pixels are connected to each other. If you want a pixel to be connected to the pixels on the top, bottom, left, and right, select 4. If you want a pixel to be connected to the pixels on the top, bottom, left, right, and diagonally, select 8. For more information about this parameter, see the `Label` block reference page.

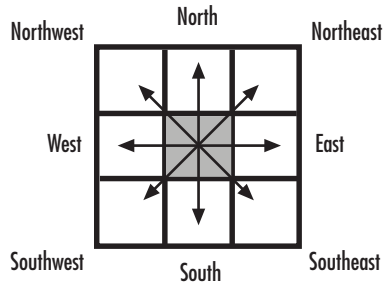
### Initial search direction

Specify the first direction in which to look to find the next boundary pixel that is connected to the starting pixel.

If, for the **Connectivity** parameter, you select **4**, the following figure illustrates the four possible initial search directions:



If, for the **Connectivity** parameter, you select **8**, the following figure illustrates the eight possible initial search directions:



### Trace direction

Specify the direction in which to trace the boundary. Your choices are **Clockwise** or **Counterclockwise**.

### Maximum number of boundary pixels

Specify the maximum number of boundary pixels for each starting point. The block uses this value to preallocate the number of rows of the **Pts** port output matrix so that it can hold all the boundary pixel location values.

Use the **Maximum number of boundary pixels** parameter to specify the maximum number of boundary pixels for the starting point.

## See Also

Edge Detection	Computer Vision System Toolbox software
Label	Computer Vision System Toolbox software
bwboundaries	Image Processing Toolbox software
bwtraceboundary	Image Processing Toolbox software

**Introduced in R2011b**

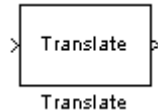
# Translate

Translate image in 2-D plane using displacement vector

## Library

Geometric Transformations

visiongeotforms



## Description

Use the Translate block to move an image in a two-dimensional plane using a displacement vector, a two-element vector that represents the number of pixels by which you want to translate your image. The block outputs the image produced as the result of the translation.

---

**Note:** This block supports intensity and color images on its ports.

---

Port	Input/Output	Supported Data Types	Complex Values Supported
Image / Input	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	No
Offset	Vector of values that represent the number of pixels by which to translate the image	Same as I port	No
Output	Translated image	Same as I port	No

The input to the Offset port must be the same data type as the input to the Image port. The output is the same data type as the input to the Image port.

Use the **Output size after translation** parameter to specify the size of the translated image. If you select **Full**, the block outputs a matrix that contains the entire translated image. If you select **Same as input image**, the block outputs a matrix that is the same size as the input image and contains a portion of the translated image. Use the **Background fill value** parameter to specify the pixel values outside the image.

Use the **Translation values source** parameter to specify how to enter your displacement vector. If you select **Specify via dialog**, the **Offset** parameter appears in the dialog box. Use it to enter your displacement vector, a two-element vector,  $[r \ c]$ , of real, integer values that represent the number of pixels by which you want to translate your image. The  $r$  value represents how many pixels up or down to shift your image. The  $c$  value represents how many pixels left or right to shift your image. The axis origin is the top-left corner of your image. For example, if you enter  $[2.5 \ 3.2]$ , the block moves the image 2.5 pixels downward and 3.2 pixels to the right of its original location. When the displacement vector contains fractional values, the block uses interpolation to compute the output.

Use the **Interpolation method** parameter to specify which interpolation method the block uses to translate the image. If you translate your image in either the horizontal or vertical direction and you select **Nearest neighbor**, the block uses the value of the nearest pixel for the new pixel value. If you translate your image in either the horizontal or vertical direction and you select **Bilinear**, the new pixel value is the weighted average of the four nearest pixel values. If you translate your image in either the horizontal or vertical direction and you select **Bicubic**, the new pixel value is the weighted average of the sixteen nearest pixel values.

The number of pixels the block considers affects the complexity of the computation. Therefore, the nearest-neighbor interpolation is the most computationally efficient. However, because the accuracy of the method is roughly proportional to the number of pixels considered, the bicubic method is the most accurate. For more information, see “Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods” in the *Computer Vision System Toolbox User's Guide*.

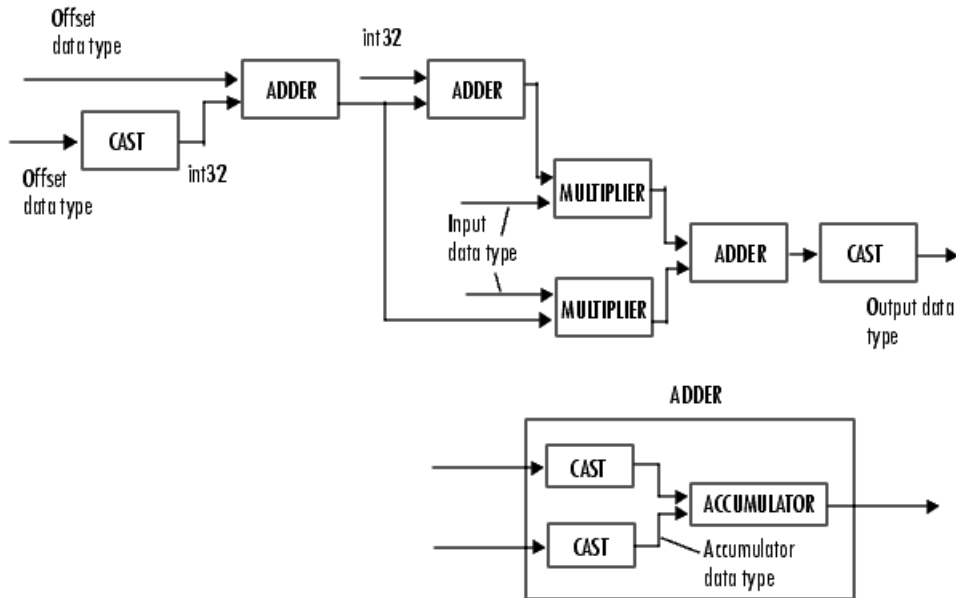
If, for the **Output size after translation** parameter, you select **Full**, and for the **Translation values source** parameter, you select **Input port**, the **Maximum offset** parameter appears in the dialog box. Use the **Maximum offset** parameter to enter a two-element vector of real, scalar values that represent the maximum number of pixels by which you want to translate your image. The block uses this parameter to

determine the size of the output matrix. If the input to the Offset port is greater than the **Maximum offset** parameter values, the block saturates to the maximum values.

If, for the **Translation values source** parameter, you select **Input port**, the Offset port appears on the block. At each time step, the input to the Offset port must be a vector of real, scalar values that represent the number of pixels by which to translate your image.

## Fixed-Point Data Types

The following diagram shows the data types used in the Translate block for bilinear interpolation of fixed-point signals.

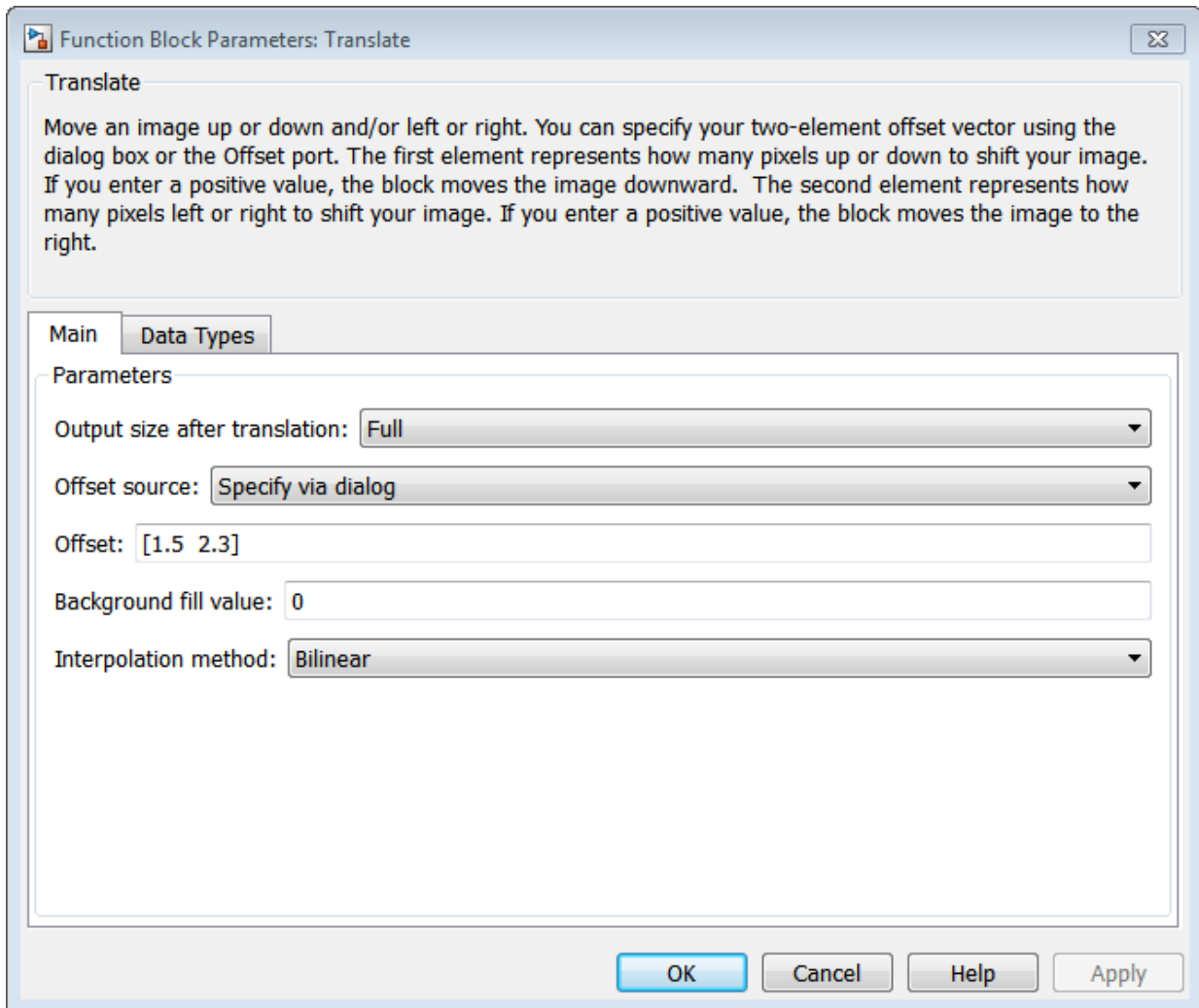


You can set the product output, accumulator, and output data types in the block mask as discussed in the next section.

## Dialog Box

The **Main** pane of the Translate dialog box appears as shown in the following figure.





### Output size after translation

If you select **Full**, the block outputs a matrix that contains the translated image values. If you select **Same as input image**, the block outputs a matrix that is the same size as the input image and contains a portion of the translated image.

### Translation values source

Specify how to enter your translation parameters. If you select **Specify via dialog**, the **Offset** parameter appears in the dialog box. If you select **Input port**, port **O** appears on the block. The block uses the input to this port at each time step as your translation values.

**Offset source**

Enter a vector of real, scalar values that represent the number of pixels by which to translate your image.

**Background fill value**

Specify a value for the pixels that are outside the image.

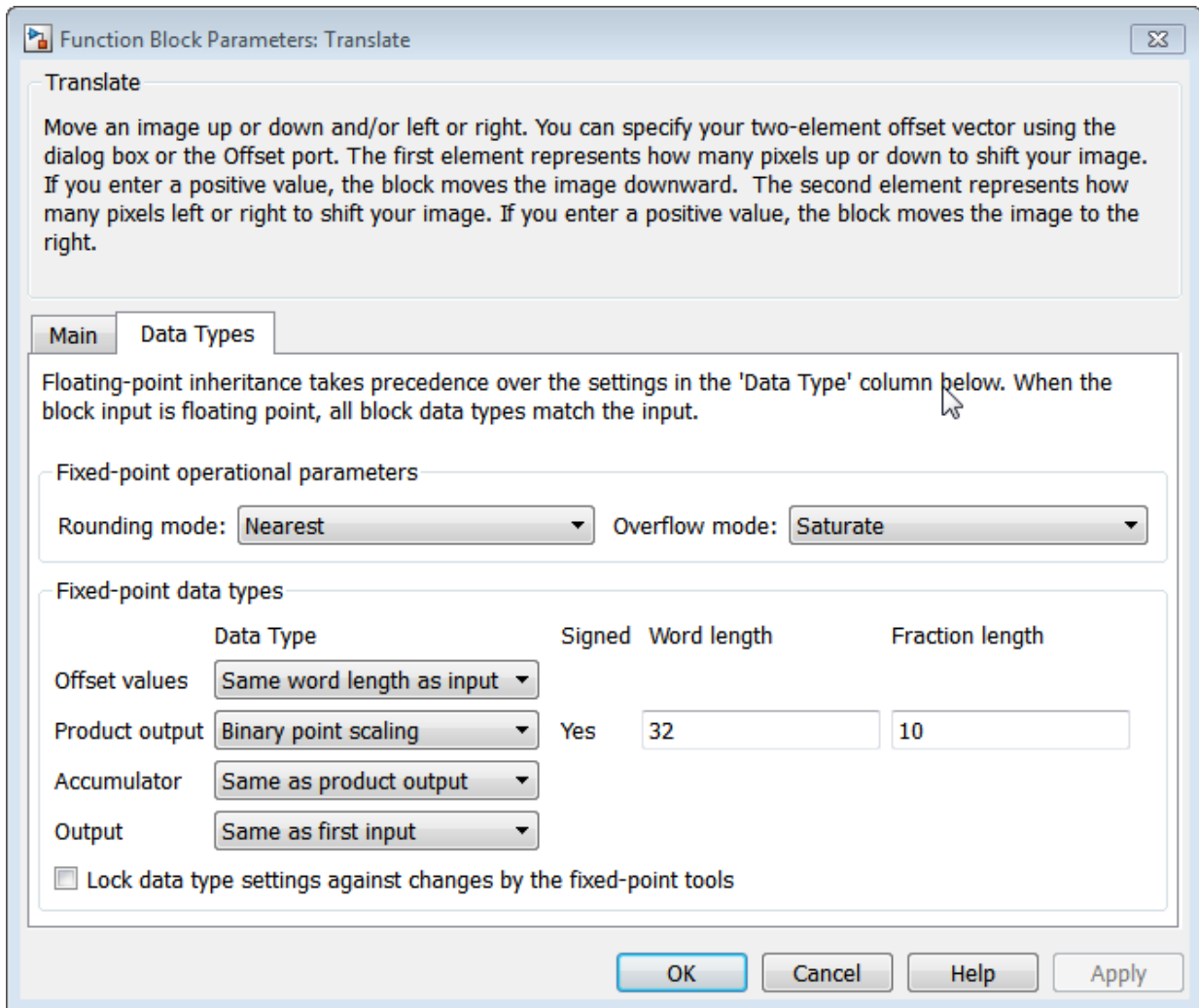
**Interpolation method**

Specify which interpolation method the block uses to translate the image. If you select **Nearest neighbor**, the block uses the value of one nearby pixel for the new pixel value. If you select **Bilinear**, the new pixel value is the weighted average of the four nearest pixel values. If you select **Bicubic**, the new pixel value is the weighted average of the sixteen nearest pixel values.

**Maximum offset**

Enter a vector of real, scalar values that represent the maximum number of pixels by which you want to translate your image. This parameter must have the same data type as the input to the **Offset** port. This parameter is visible if, for the **Output size after translation** parameter, you select **Full** and, for the **Translation values source** parameter, you select **Input port**.

The **Data Types** pane of the Translate dialog box appears as shown in the following figure.



### **Rounding mode**

Select the rounding mode for fixed-point operations.

### **Overflow mode**

Select the overflow mode for fixed-point operations.

### **Offset values**

Choose how to specify the word length and the fraction length of the offset values.

- When you select **Same word length as input**, the word length of the offset values match that of the input to the block. In this mode, the fraction length of the offset values is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the offset values.
- When you select **Specify word length**, you can enter the word length of the offset values, in bits. The block automatically sets the fraction length to give you the best precision.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the offset values, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the offset values. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

This parameter is visible if, for the **Translation values source** parameter, you select **Specify via dialog**.

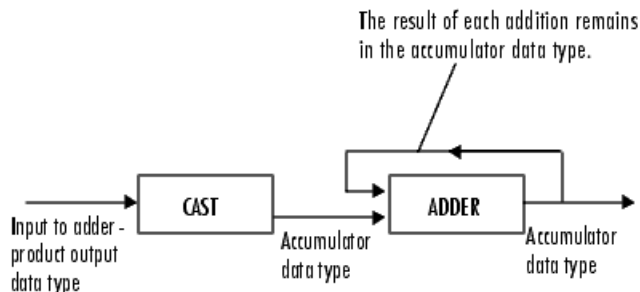
### Product output



As depicted in the previous figure, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how to designate this product output word and fraction lengths.

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Accumulator



As depicted in the previous figure, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how to designate this accumulator word and fraction lengths.

- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

## Output

Choose how to specify the word length and fraction length of the output of the block:

- When you select **Same as first input**, these characteristics match those of the first input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. The bias of all signals in the Computer Vision System Toolbox blocks is 0.

**Lock data type settings against change by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask. For more information, see `fxptdlg`, a reference page on the Fixed-Point Tool in the Simulink documentation.

## References

[1] Wolberg, George. *Digital Image Warping*. Washington: IEEE Computer Society Press, 1990.

## See Also

Resize	Computer Vision System Toolbox software
Rotate	Computer Vision System Toolbox software
Shear	Computer Vision System Toolbox software

**Introduced before R2006a**

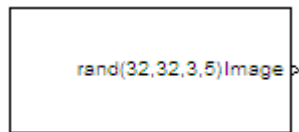
# Video From Workspace

Import video signal from MATLAB workspace

## Library

Sources

visionsources



## Description

The **Video From Workspace** block imports a video signal from the MATLAB workspace. If the video signal is a M-by-N-by-T workspace array, the block outputs an intensity video signal, where M and N are the number of rows and columns in a single video frame, and T is the number of frames in the video signal. If the video signal is a M-by-N-by-C-by-T workspace array, the block outputs a color video signal, where M and N are the number of rows and columns in a single video frame, C is the number of color channels, and T is the number of frames in the video stream. In addition to the video signals previously described, this block supports fi objects.

---

**Note:** If you generate code from a model that contains this block, Simulink Coder takes a long time to compile the code because it puts all of the video data into the .c file. Before you generate code, you should convert your video data to a format supported by the From Multimedia File block or the Read Binary File block.

---

Port	Output	Supported Data Types	Complex Values Supported
Image	M-by-N matrix of intensity values or an M-by-N-by-P color video	<ul style="list-style-type: none"> <li>Double-precision floating point</li> </ul>	No

Port	Output	Supported Data Types	Complex Values Supported
	signal where P is the number of color planes	<ul style="list-style-type: none"> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	
R, G, B	Scalar, vector, or matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports have the same dimensions.	Same as I port	No

For the Computer Vision System Toolbox blocks to display video data properly, double- and single-precision floating-point pixel values must be from 0 to 1. This block does not scale pixel values.

Use the **Signal** parameter to specify the MATLAB workspace variable from which to read. For example, to read an AVI file, use the following syntax:

```
mov = VideoReader('filename.avi')
```

If `filename.avi` has a colormap associated with it, the AVI file must satisfy the following conditions or the block produces an error:

- The colormap must be empty or have 256 values.
- The data must represent an intensity image.
- The data type of the image values must be `uint8`.

Use the **Sample time** parameter to set the sample period of the output signal.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, repeat the final value, or generate 0s until the end of the simulation. The **Form output after final value by** parameter controls this behavior:

- When you specify **Setting To Zero**, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.



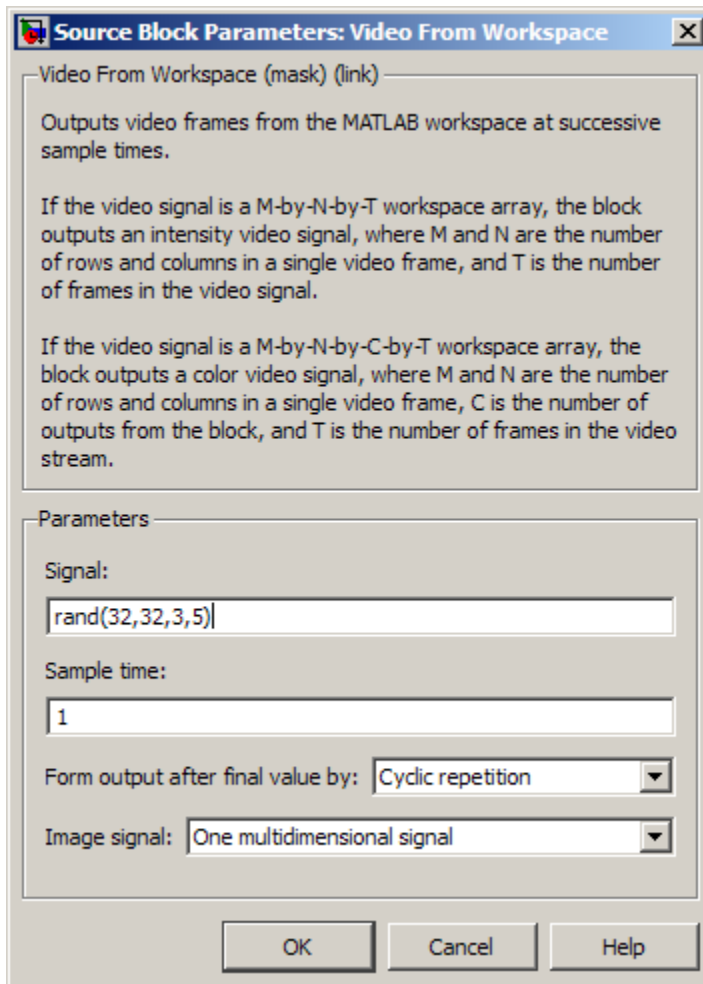
- When you specify **Holding Final Value**, the block repeats the final frame for the duration of the simulation after generating the last frame of the signal.
- When you specify **Cyclic Repetition**, the block repeats the signal from the beginning after it reaches the last frame in the signal.

Use the **Image signal** parameter to specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

Use the **Output port labels** parameter to label your output ports. Use the spacer character, |, as the delimiter. This parameter is available when the **Image signal** parameter is set to **Separate color signals**.

## Dialog Box

The Video From Workspace dialog box appears as shown in the following figure.

**Signal**

Specify the MATLAB workspace variable that contains the video signal, or use the `VideoReader` function to specify an AVI filename.

**Sample time**

Enter the sample period of the output.

**Form output after final value by**

Specify the output of the block after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (**Setting to zero**), repeat the final value (**Holding Final Value**) or repeat the entire signal from the beginning (**Cyclic Repetition**).

### Image signal

Specify how the block outputs a color video signal. If you select **One multidimensional signal**, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream.

### Output port labels

Enter the labels for your output ports using the spacer character, |, as the delimiter. This parameter is available when the **Image signal** parameter is set to **Separate color signals**.

## See Also

From Multimedia File	Computer Vision System Toolbox software
Image From Workspace	Computer Vision System Toolbox software
Read Binary File	Computer Vision System Toolbox software
To Video Display	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software

**Introduced before R2006a**

# Video To Workspace

Export video signal to MATLAB workspace

## Library

Sinks

visionsinks



## Description

The Video To Workspace block exports a video signal to the MATLAB workspace. If the video signal is represented by intensity values, it appears in the workspace as a three-dimensional  $M$ -by- $N$ -by- $T$  array, where  $M$  and  $N$  are the number of rows and columns in a single video frame, and  $T$  is the number of frames in the video signal. If it is a color video signal, it appears in the workspace as a four-dimensional  $M$ -by- $N$ -by- $C$ -by- $T$  array, where  $M$  and  $N$  are the number of rows and columns in a single video frame,  $C$  is the number of inputs to the block, and  $T$  is the number of frames in the video stream. During code generation, Simulink Coder does not generate code for this block.

---

**Note:** This block supports intensity and color images on its ports.

---

Port	Input	Supported Data Types	Complex Values Supported
Image	$M$ -by- $N$ matrix of intensity values or an $M$ -by- $N$ -by- $P$ color video signal where $P$ is the number of color planes	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• Boolean</li> </ul>	No

Port	Input	Supported Data Types	Complex Values Supported
		<ul style="list-style-type: none"> <li>• 8-, 16-, 32-bit signed integer</li> <li>• 8-, 16-, 32-bit unsigned integer</li> </ul>	
R, G, B	Scalar, vector, or matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports have the same dimensions.	Same as I port	No

Use the **Variable name** parameter to specify the MATLAB workspace variable to which to write the video signal.

Use the **Number of inputs** parameter to determine the number of inputs to the block. If the video signal is represented by intensity values, enter 1. If it is a color (R, G, B) video signal, enter 3.

Use the **Limit data points to last** parameter to determine the number of video frames, T, you want to export to the MATLAB workspace.

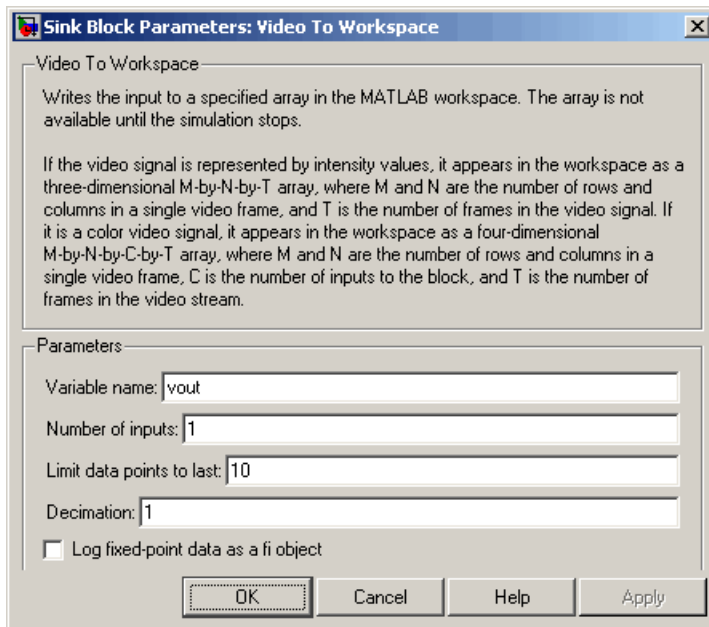
If you want to downsample your video signal, use the **Decimation** parameter to enter your decimation factor.

If your video signal is fixed point and you select the **Log fixed-point data as a fi object** check box, the block creates a fi object in the MATLAB workspace.

Use the **Input port labels** parameter to label your input ports. Use the spacer character, |, as the delimiter. This parameter is available if the **Number of inputs** parameter is greater than 1.

## Dialog Box

The Video To Workspace dialog box appears as shown in the following figure.



### Variable name

Specify the MATLAB workspace variable to which to write the video signal.

### Number of inputs

Enter the number of inputs to the block. If the video signal is black and white, enter 1. If it is a color (R, G, B) video signal, enter 3.

### Limit data points to last

Enter the number of video frames to export to the MATLAB workspace.

### Decimation

Enter your decimation factor.

### Log fixed-point data as a fi object

If your video signal is fixed point and you select this check box, the block creates a fi object in the MATLAB workspace. For more information of fi objects, see the Fixed-Point Designer documentation.

### Input port labels

Enter the labels for your input ports using the spacer character, |, as the delimiter. This parameter is available if the **Number of inputs** parameter is greater than 1.

## See Also

To Multimedia File	Computer Vision System Toolbox software
To Video Display	Computer Vision System Toolbox software
Video Viewer	Computer Vision System Toolbox software

**Introduced before R2006a**

# Video Viewer

Display binary, intensity, or RGB images or video streams

## Library

Sinks

visionsinks



## Description

The Video Viewer block enables you to view a binary, intensity, or RGB image or a video stream. The block provides simulation controls for play, pause, and step while running the model. The block also provides pixel region analysis tools. During code generation, Simulink Coder software does not generate code for this block.

---

**Note:** The To Video Display block supports code generation.

---

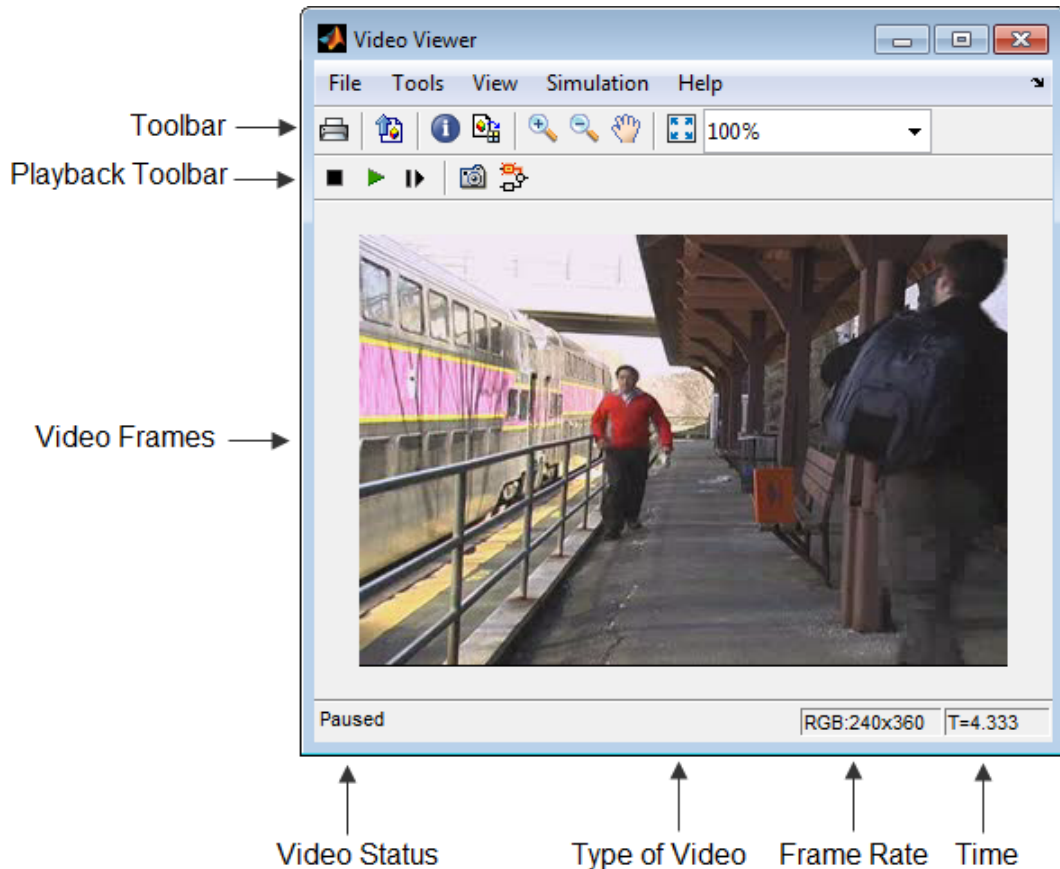
See the following table for descriptions of both input types.

Input	Description
Image	M-by-N matrix of intensity values or an M-by-N-by-P color video signal where P is the number of color planes.
R/G/B	Scalar, vector, or matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B ports must have the same dimensions and data type.

Select **File > Image Signal** to set the input to either **Image** or **RGB**.












## Dialogs



### Toolbar







### Toolbar

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	<b>File &gt; Print</b>	<b>Ctrl+P</b>	Print the current display window. Printing is only available when the display is not changing. You can enable printing by placing the display in snapshot mode, or

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
			<p>by pausing or stopping model simulation, or simulating the model in step-forward mode.</p> <p>To print the current window to a figure rather than sending it to your printer, select <b>File &gt; Print to figure</b>.</p>
	<b>File &gt; Export to Image Tool</b>	<b>Ctrl+E</b>	Send the current video frame to the Image Tool. For more information, see “Explore Images with the Image Viewer App” in the Image Processing Toolbox documentation.
<p><b>Note:</b> The Image Tool can only know that the frame is an intensity image if the colormap of the frame is grayscale (<code>gray(256)</code>). Otherwise, the Image Tool assumes the frame is an indexed image and disables the <b>Adjust Contrast</b> button.</p>			
	<b>Tools &gt; Video Information</b>	<b>V</b>	View information about the video data source.
	<b>Tools &gt; Pixel Region</b>	N/A	Open the Pixel Region tool. For more information about this tool, see the Image Processing Toolbox documentation.
	<b>Tools &gt; Zoom In</b>	N/A	Zoom in on the video display.
	<b>Tools &gt; Zoom Out</b>	N/A	Zoom out of the video display.
	<b>Tools &gt; Pan</b>	N/A	Move the image displayed in the GUI.
	<b>Tools &gt; Maintain Fit to Window</b>	N/A	Scale video to fit GUI size automatically. Toggle the button on or off.
	N/A	N/A	Enlarge or shrink the video frame. This option becomes available if you do not select the <b>Maintain Fit to Window</b> .

### Playback Toolbar

### Playback Toolbar

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	<b>Simulation &gt; Stop</b>	<b>S</b>	Stop the video.
	<b>Simulation &gt; Play</b>	<b>P, Space bar</b>	Play the video.
	<b>Simulation &gt; Pause</b>	<b>P, Space bar</b>	Pause the video. This button appears only when the video is playing.
	<b>Simulation &gt; Step Forward</b>	<b>Right arrow, Page Down</b>	Step forward one frame.
	<b>Simulation &gt; Simulink Snapshot</b>	N/A	Click this button to freeze the display in the viewer window.
File-menu only	<b>Simulation &gt; Drop Frames to Improve Performance</b>	<b>Ctrl+R</b>	Enable the viewer to drop video frames to improve performance.
	<b>View &gt; Highlight Simulink Signal</b>	<b>Ctrl+L</b>	In the model window, highlight the Simulink signal the viewer is displaying.

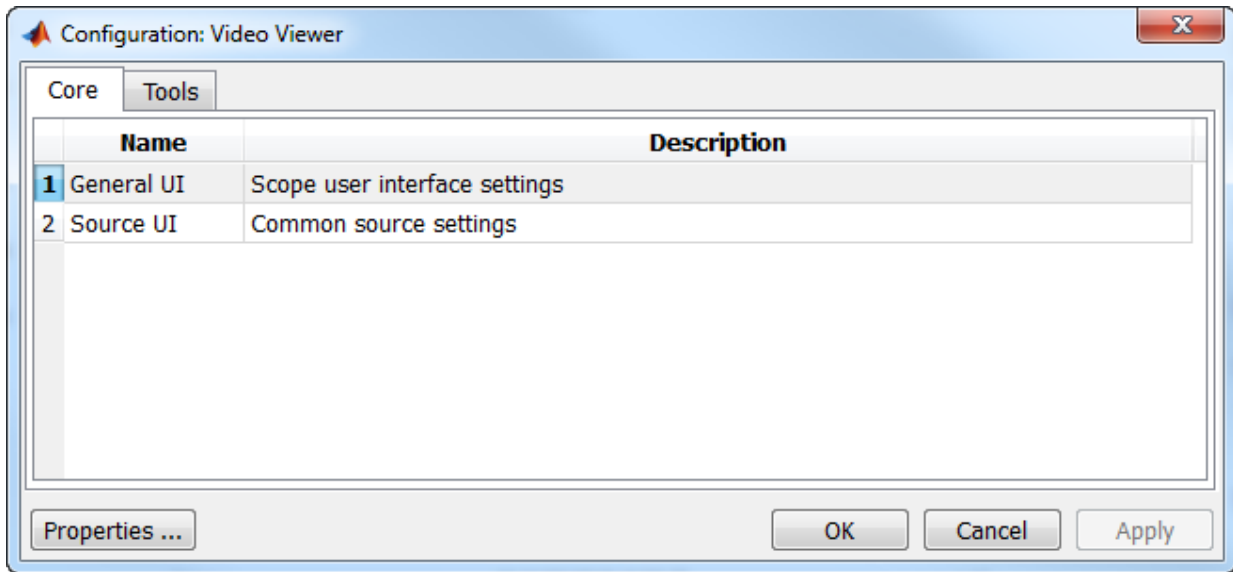
## Setting Viewer Configuration

The Video Viewer Configuration preferences enables you to change the behavior and appearance of the graphic user interface (GUI) as well as the behavior of the playback shortcut keys.

- To open the Configuration dialog box, select **File > Configuration Set > Edit**.
- To save the configuration settings for future use, select **File > Configuration Set > Save as**.

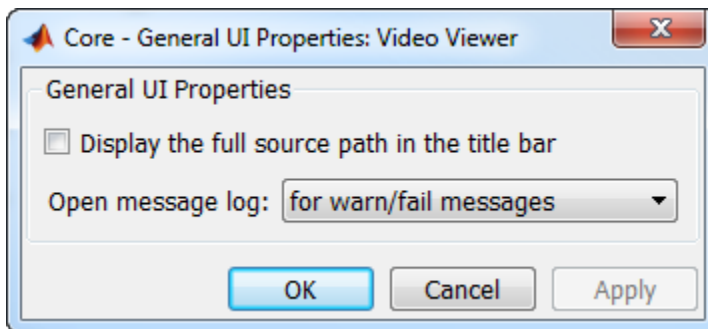
## Core Pane

The Core pane in the Viewer Configuration dialog box controls the GUI's general settings.



### General UI

Click **General UI**, and click the **Options** button to open the General UI Options dialog box.



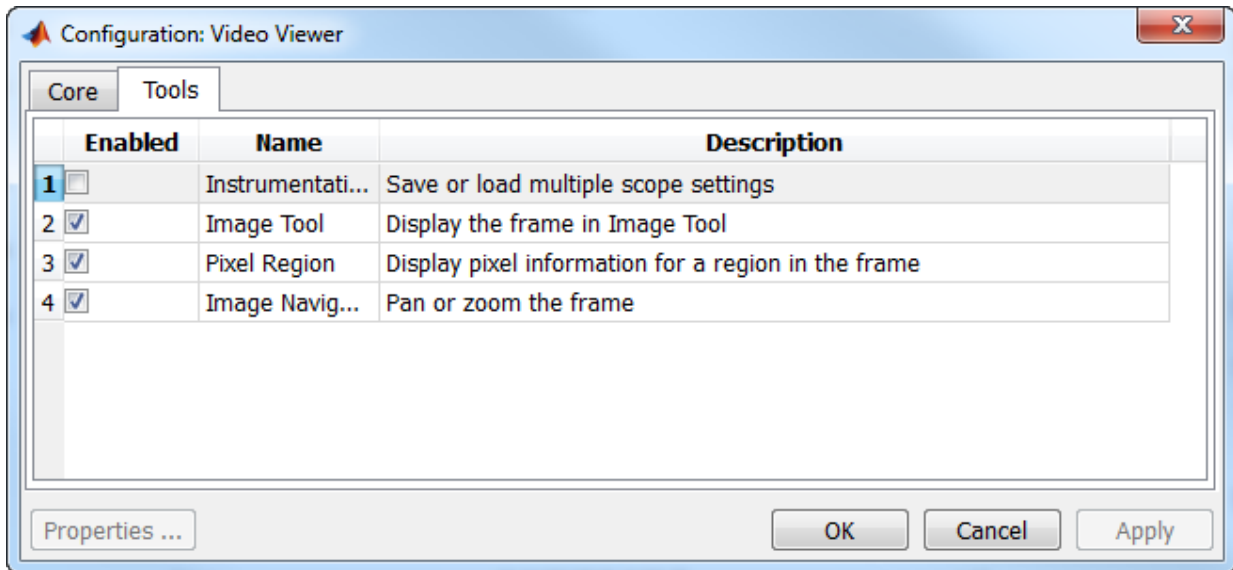
If you select the **Display the full source path in the title bar** check box, the GUI displays the model name and full Simulink path to the video data source in the title bar. Otherwise, it displays a shortened name.

Use the **Open message log:** parameter to control when the Message log window opens. You can use this window to debug issues with video playback. Your choices are for

any new messages, for warn/fail messages, only for fail messages, or manually.

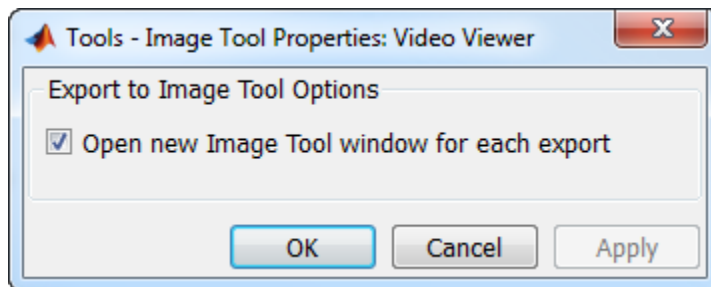
### Tools Pane

The Tools pane in the Viewer Configuration dialog box contains the tools that appear on the Video Viewer GUI. Select the **Enabled** check box next to the tool name to specify which tools to include on the GUI.



### Image Tool

Click **Image Tool**, and then click the **Options** button to open the Image Tool Options dialog box.



Select the **Open new Image Tool window for export** check box if you want to open a new Image Tool for each exported frame.

### Pixel Region

Select the **Pixel Region** check box to display and enable the pixel region GUI button. For more information on working with pixel regions see Getting Information about the Pixels in an Image.


### Image Navigation Tools

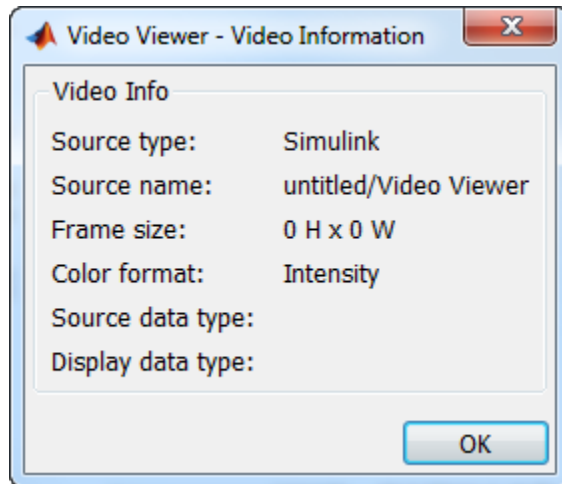
Select the **Image Navigation Tools** check box to enable the pan-and-zoom GUI button.

### Instrumentation Set

Select the **Instrumentation Set** check box to enable the option to load and save viewer settings. The option appears in the **File** menu.

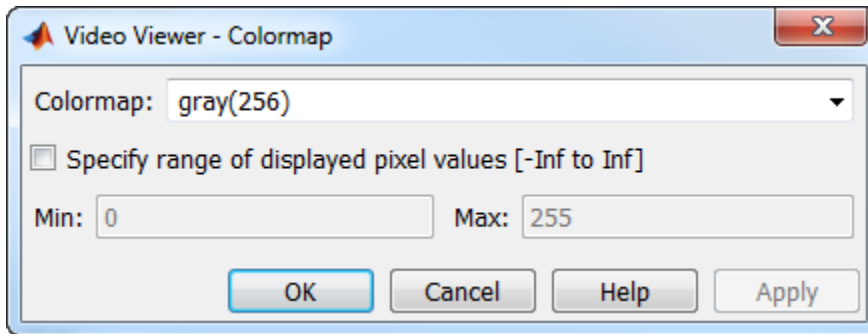
## Video Information

The Video Information dialog box lets you view basic information about the video. To open this dialog box, you can select **Tools > Video Information**, click the information button , or press the **V** key.



## Colormap for Intensity Video

The Colormap dialog box lets you change the colormap of an intensity video. You cannot access the parameters on this dialog box when the GUI displays an RGB video signal. To open this dialog box for an intensity signal, select **Tools > Colormap** or press **C**.



Use the **Colormap** parameter to specify the colormap to apply to the intensity video.

If you know that the pixel values do not use the entire data type range, you can select the **Specify range of displayed pixel values** check box and enter the range for your data. The dialog box automatically displays the range based on the data type of the pixel values.

### Status Bar

A status bar appear along the bottom of the Video Viewer. It displays information pertaining to the video status (running, paused or ready), type of video (Intensity or RGB) and video time.

### Message Log

The Message Log dialog provides a system level record of configurations and extensions used. You can filter what messages to display by **Type** and **Category**, view the records, and display record details.

The **Type** parameter allows you to select either **All**, **Info**, **Warn**, or **Fail** message logs. The **Category** parameter allows you to select either **Configuration** or **Extension** message summaries. The **Configuration** messages indicate when a new configuration file is loaded. The **Extension** messages indicate a component is registered. For example, you might see a **Simulink** message, which indicates the component is registered and available for configuration.

### Saving the Settings of Multiple Video Viewer GUIs

The Video Viewer GUI enables you to save and load the settings of multiple GUI instances. Thus, you only need to configure the Video Viewer GUIs that are associated

with your model once. To save the GUI settings, select **File > Instrumentation Sets > Save Set**. To open the preconfigured GUIs, select **File > Instrumentation Sets > Load Set**.

## Supported Data Types

Port	Supported Data Types
Image	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integer</li> <li>• 8-, 16-, and 32-bit unsigned integer</li> </ul>
R/G/B	Same as Image port

## See Also

From Multimedia File	Computer Vision System Toolbox software
To Multimedia File	Computer Vision System Toolbox software
To Video Display	Computer Vision System Toolbox software
Video To Workspace	Computer Vision System Toolbox software
implay	Image Processing Toolbox

**Introduced before R2006a**



# Warp

Apply projective or affine transformation to an image

## Library

Geometric Transformations

visiongeotforms



## Description

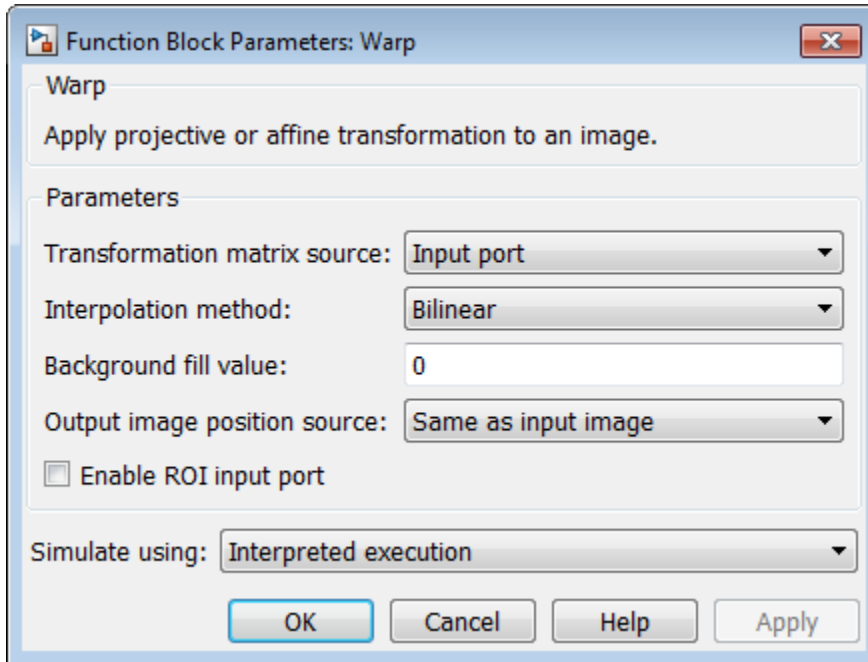
The Warp block applies a “Projective Transformation” on page 1-585 or “Affine Transformation” on page 1-585 transformation to an image. You can transform the entire image or portions of the image using either a polygonal or rectangular region of interest (ROI).

## Input Port Descriptions

Port	Input/Output	Description	Supported Data Types
<b>Image</b>	Input	<p><math>M</math>-by-<math>N</math> grayscale image or <math>M</math>-by-<math>N</math>-by-3 truecolor image.</p> <ul style="list-style-type: none"> <li><math>M</math>— Number of rows in the image.</li> <li><math>N</math>— Number of columns in the image.</li> </ul>	<ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> <li>8- or 16-bit unsigned integers</li> <li>16-bit signed integers</li> <li>logical</li> </ul>
<b>TForm</b>	Input	<p>When you set <b>Transformation matrix source</b> to Input port, the <b>TForm</b> port accepts these inputs:</p>	<ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> </ul>

Port	Input/Output	Description	Supported Data Types
		<ul style="list-style-type: none"> <li>• 3-by-2 matrix (affine transform).</li> <li>• 3-by-3 matrix (projective transform).</li> </ul> <p>When you set <b>Transformation matrix source</b> to <b>Custom</b>, specify the source in the <b>Transformation matrix</b> field.</p>	
<b>ROI</b>	Input	<p>When you enable the <b>ROI</b> input port, you can also enable an <b>Err_roi</b> output port to indicate if any part of an <b>ROI</b> is outside the input image. The <b>ROI</b> input port accepts an ROI rectangle, specified as a 4-element vector: [<i>x y width height</i>].</p>	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, or 32-bit signed integers</li> <li>• 8-, 16-, or 32-bit unsigned integers</li> </ul>
<b>Image</b>	Output	Transformed image.	Same as input
<b>Err_roi</b>	Output	Indicates if any part of an ROI is outside the input image.	Boolean

## Dialog Box



### Transformation matrix source

Input matrix source, specified as either **Input port** or **Custom**. If you select **Custom**, you can enter the transformation matrix parameter in the field that appears with this selection.

### Transformation matrix

Custom transformation matrix, specified as a 3-by-2 or 3-by-3 matrix. This parameters appears when you set **Transformation matrix source** to **Custom**.

## Interpolation method

Interpolation method used to calculate output pixel values, specified as **Nearest neighbor**, **Bilinear**, or **Bicubic**. See “Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods” for an overview of these methods.

## Background fill value

Value of the pixels that are outside of the input image, specified as either a scalar value or a 3-element vector.

## Output image position source

Source of the output image size, specified as either **Same as input image** or **Custom**. If you select **Custom**, you can specify the bounding box in the field that appears with this selection.

## Output image position vector [x y width height]

Position, width, and height of the output image, specified as a 4-element vector: [*x y width height*]. This parameter appears when you set **Output image position source** to **Custom**.

## Enable ROI input port

Select this check box to enable the **ROI** input port. Use this port to specify the rectangular region you want to transform.

## Enable output port indicating if any part of ROI is outside input image

Select this check box to enable the **Err\_roi** output port.

## Algorithms

The size of the transformation matrix dictates the transformation type.

## Projective Transformation

In a projective transformation, the relationship between the input and the output points is defined by:

$$\begin{cases} \hat{x} = \frac{xh_1 + yh_2 + h_3}{xh_7 + yh_8 + h_9} \\ \hat{y} = \frac{xh_4 + yh_5 + h_6}{xh_7 + yh_8 + h_9} \end{cases}$$

where  $h_1, h_2, \dots, h_9$  are transformation coefficients.

You must arrange the transformation coefficients as a 3-by-3 matrix as in:

$$H = \begin{bmatrix} h_1 & h_4 & h_7 \\ h_2 & h_5 & h_8 \\ h_3 & h_6 & h_9 \end{bmatrix}$$

## Affine Transformation

In an affine transformation, The value of the pixel located at  $[\hat{x}, \hat{y}]$  in the input image determines the value of the pixel located at  $[x, y]$  in the output image. The relationship between the input and output point locations is defined by:

$$\begin{cases} \hat{x} = xh_1 + yh_2 + h_3 \\ \hat{y} = xh_4 + yh_5 + h_6 \end{cases}$$

where  $h_1, h_2, \dots, h_6$  are transformation coefficients.

You must arrange the transformation coefficients as a 3-by-2 matrix:

$$H = \begin{bmatrix} h_1 & h_4 \\ h_2 & h_5 \\ h_3 & h_6 \end{bmatrix}$$

## References

[1] Wolberg, George . *Digital Image Warping*, 3rd edition. IEEE Computer Society Press, 1994.

[2] Hartley, Richard, and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd edition. IEEE Computer Society Press, 2003.

## See Also

`imwarp`, Estimate Geometric Transformation, Resize, RotateShearTranslate

## Related Examples

- “Video Mosaicking”

**Introduced in R2015b**

# Write AVI File (To Be Removed)

Write video frames to uncompressed AVI file

## Library

Sinks



## Description

---

**Note:** The Write AVI File block is obsolete. It may be removed in a future version of the Computer Vision System Toolbox blocks. Use the replacement block `To Multimedia File`.

---

**Introduced in R2011b**

# Write Binary File

Write binary video data to files

## Library

Sinks

visionsinks

## Description



The **Write Binary File** block takes video data from a Simulink model and exports it to a binary file.

This block produces a raw binary file with no header information. It has no encoded information providing the data type, frame rate or dimensionality. The video data for this block appears in row major format.

**Note:** This block supports code generation only for platforms that have file I/O available. You cannot use this block to do code generation with Simulink Desktop Real-Time.

Port	Input	Supported Data Types	Complex Values Supported
Input	Matrix that represents the luma (Y') and chroma (Cb and Cr) components of a video stream	<ul style="list-style-type: none"> <li>8-, 16- 32-bit signed integer</li> <li>8-, 16- 32-bit unsigned integer</li> </ul>	No

## Four Character Code Video Formats

Four Character Codes (FOURCC) identify video formats. For more information about these codes, see <http://www.fourcc.org>.



Use the **Four character code** parameter to identify the video format.

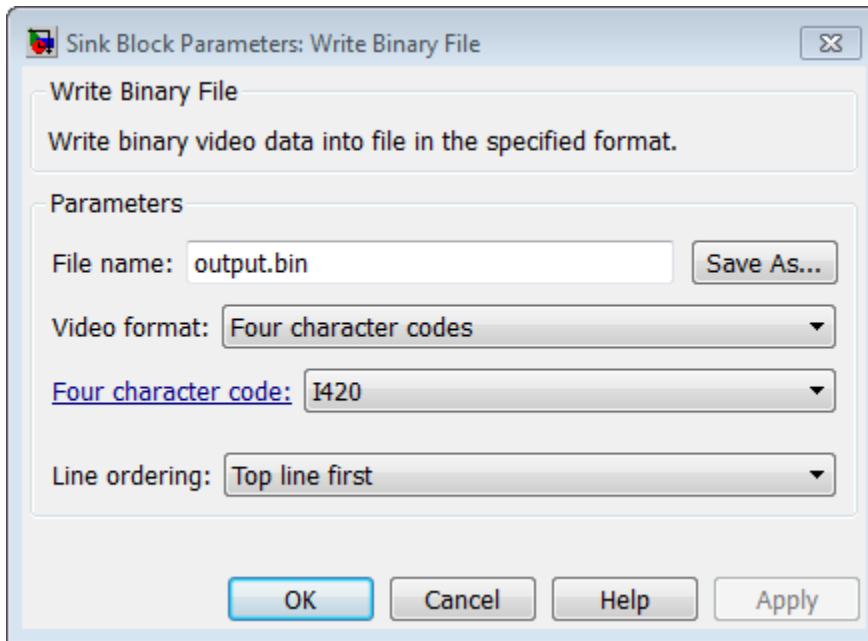
## Custom Video Formats

You can use the Write Binary File block to create a binary file that contains video data in a custom format.

- Use the **Bit stream format** parameter to specify whether you want your data in planar or packed format.
- Use the **Number of input components** parameter to specify the number of components in the video stream. This number corresponds to the number of block input ports.
- Select the **Inherit size of components from input data type** check box if you want each component to have the same number of bits as the input data type. If you clear this check box, you must specify the number of bits for each component.
- Use the **Component** parameters to specify the component names.
- Use the **Component order in binary file** parameter to specify how to arrange the components in the binary file.
- Select the **Interlaced video** check box if the video stream represents interlaced video data.
- Select the **Write signed data to output file** check box if your input data is signed.
- Use the **Byte order in binary file** parameter to specify whether the byte ordering in the output binary file is little endian or big endian.

## Dialog Box

The Write Binary File dialog box appears as shown in the following figure.



**File name**

Specify the name of the binary file. To specify a different file or location, click the **Save As...** button.

**Video format**

Specify the format of the binary video data as either **Four character codes** or **Custom**. See “Four Character Code Video Formats” on page 1-588 or “Custom Video Formats” on page 1-589 for more details.

**Four character code**

From the list, select the binary file format.

**Line ordering**

Specify how the block fills the binary file. If you select **Top line first**, the block first fills the binary file with the first row of the video frame. It then fills the file with the other rows in increasing order. If you select **Bottom line first**, the block first fills the binary file with the last row of the video frame. It then fills the file with the other rows in decreasing order.

**Bit stream format**

Specify whether you want your data in planar or packed format.

**Number of input components**

Specify the number of components in the video stream. This number corresponds to the number of block input ports.

**Inherit size of components from input data type**

Select this check box if you want each component to have the same number of bits as the input data type. If you clear this check box, you must specify the number of bits for each component.

**Component**

Specify the component names.

**Component order in binary file**

Specify how to arrange the components in the binary file.

**Interlaced video**

Select this check box if the video stream represents interlaced video data.

**Write signed data to output file**

Select this check box if your input data is signed.

**Byte order in binary file**

Use this parameter to specify whether the byte ordering in the output binary file is little endian or big endian.

## See Also

Read Binary File	Computer Vision System Toolbox
To Multimedia File	Computer Vision System Toolbox

**Introduced before R2006a**



# Alphabetical List

---

# binaryFeatures class

Object for storing binary feature vectors

## Description

This object provides the ability to pass data between the `extractFeatures` and `matchFeatures` functions. It can also be used to manipulate and plot the data returned by `extractFeatures`.

## Construction

`features= binaryFeatures(featureVectors)` constructs a `binaryFeatures` object from the  $M$ -by- $N$  input matrix, `featureVectors`. This matrix contains  $M$  feature vectors stored in  $N$  uint8 containers.

`features = binaryFeatures(featureVectors,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **featureVectors**

Input feature vectors, specified as an  $M$ -by- $N$  input matrix. This matrix contains  $M$  binary feature vectors stored in  $N$  uint8 containers.

## Properties

### **Features**

Feature vectors, stated as an  $M$ -by- $N$  input matrix. This matrix consists of  $M$  feature vectors stored in  $N$  uint8 containers.

### **NumBits**

Number of bits per feature, which is the number of uint8 feature vector containers times 8.

## NumFeatures

Number of feature vectors contained in the `binaryFeatures` object.

## Examples

### Match Two Sets of Binary Feature Vectors

Input feature vectors.

```
features1 = binaryFeatures(uint8([1 8 7 2; 8 1 7 2]));  
features2 = binaryFeatures(uint8([8 1 7 2; 1 8 7 2]));
```

Match the vectors using the Hamming distance.

```
[indexPairs matchMetric] = matchFeatures(features1, features2)
```

### See Also

`extractFeatures` | `extractHOGFeatures` | `matchFeatures`

**Introduced in R2013a**

# cameraParameters class

Object for storing camera parameters

## Syntax

```
cameraParams = cameraParameters  
cameraParams = cameraParameters(Name,Value)  
cameraParams = cameraParameters(paramStruct)
```

## Description

`cameraParams = cameraParameters` returns an object that contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

`cameraParams = cameraParameters(Name,Value)` configures the camera parameters object properties, specified as one or more `Name,Value` pair arguments. Unspecified properties use default values.

`cameraParams = cameraParameters(paramStruct)` returns a `cameraParameters` object containing the parameters specified by `paramStruct` input. `paramStruct` is returned by the `toStruct` method.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

Use the `toStruct` method to pass a `cameraParameters` object into generated code. See the “Code Generation for Depth Estimation From Stereo Video” example.

## Input Arguments

The object contains intrinsic, extrinsic, lens distortion, and estimation properties.



## Name-Value Pair Arguments

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

Example: 'RadialDistortion',[0 0 0] sets the 'RadialDistortion' to [0 0 0].

### 'IntrinsicMatrix' — Projection matrix

3-by-3 identity matrix (default) | 3-by-3 intrinsic matrix

Projection matrix, specified as the comma-separated pair consisting of 'IntrinsicMatrix' and a 3-by-3 matrix. For the matrix format, the object uses the following format:

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

The coordinates  $[c_x \ c_y]$  represent the optical center (the principal point), in pixels. When the  $x$  and  $y$  axis are exactly perpendicular, the skew parameter,  $s$ , equals 0.

$$f_x = F * s_x$$

$$f_y = F * s_y$$

$F$ , is the focal length in world units, typically expressed in millimeters.

$[s_x, s_y]$  are the number of pixels per world unit in the  $x$  and  $y$  direction respectively.

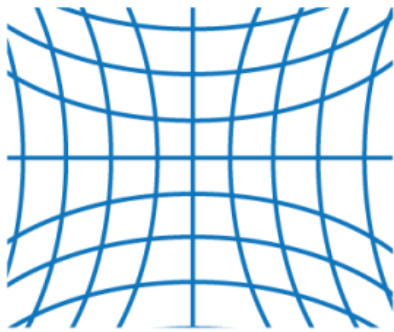
$f_x$  and  $f_y$  are expressed in pixels.

### 'RadialDistortion' — Radial distortion coefficients

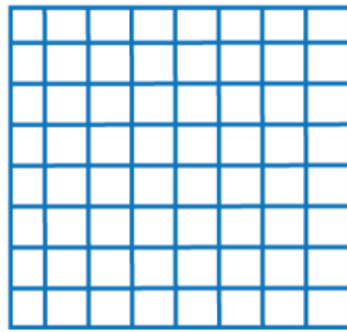
[0 0 0] (default) | 2-element vector | 3-element vector

Radial distortion coefficients, specified as the comma-separated pair consisting of 'RadialDistortion' and either a 2- or 3-element vector. If you specify a 2-element vector, the object sets the third element to 0.

Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.



negative radial distortion  
"pincushion"



no distortion



positive radial distortion  
"barrel"

The camera parameters object calculates the radial distorted location of a point. You can denote the distorted points as  $(x_{\text{distorted}}, y_{\text{distorted}})$ , as follows:

$$x_{\text{distorted}} = x(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$x, y$  = undistorted pixel locations

$k_1, k_2,$  and  $k_3$  = radial distortion coefficients of the lens

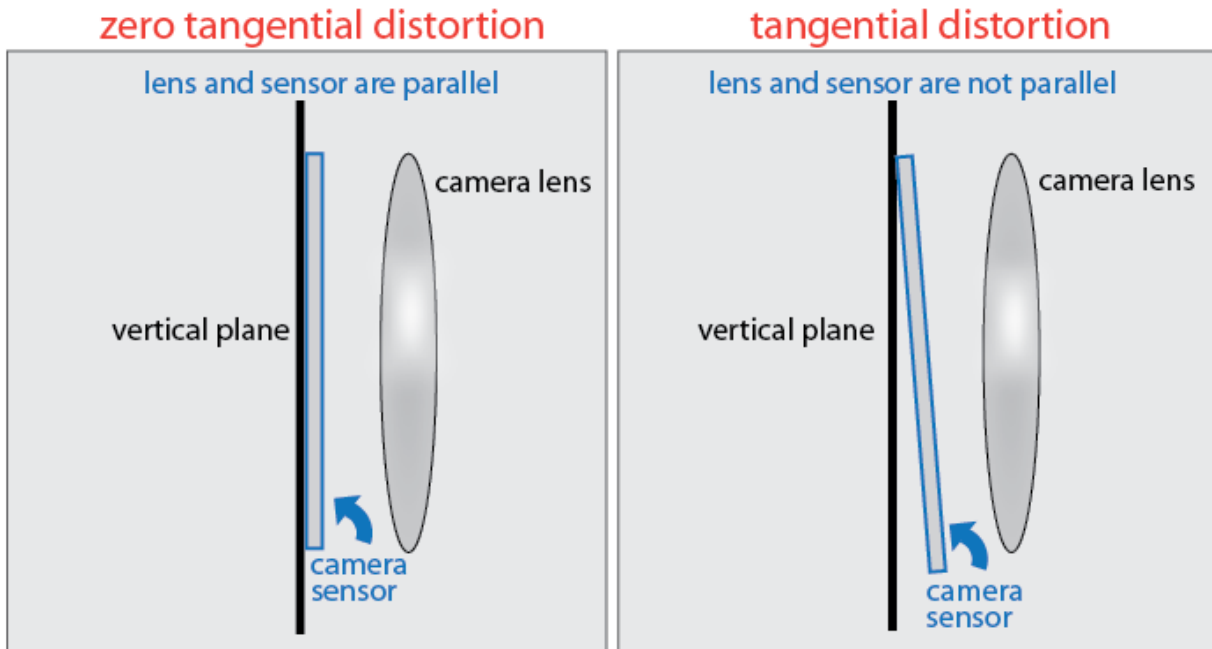
$$r^2 = x^2 + y^2$$

Typically, two coefficients are sufficient. For severe distortion, you can include  $k_3$ . The undistorted pixel locations appear in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

### 'TangentialDistortion' – Tangential distortion coefficients

[0 0]' (default) | 2-element vector

Tangential distortion coefficients, specified as the comma-separated pair consisting of 'TangentialDistortion' and a 2-element vector. Tangential distortion occurs when the lens and the image plane are not parallel.



The camera parameters object calculates the tangential distorted location of a point. You can denote the distorted points as  $(x_{\text{distorted}}, y_{\text{distorted}})$ , as follows:

$$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y]$$

$x, y$  = undistorted pixel locations

$p_1$  and  $p_2$  = tangential distortion coefficients of the lens

$$r^2 = x^2 + y^2$$

The undistorted pixel locations appear in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

### 'RotationVectors' — Camera rotations

$M$ -by-3 matrix | []

Camera rotations, specified as the comma-separated pair consisting of 'RotationVectors' and an  $M$ -by-3 matrix. The matrix contains rotation vectors for  $M$  images, which contain the calibration pattern that estimates the calibration parameters. Each row of the matrix contains a vector that describes the 3-D rotation of the camera relative to the corresponding pattern.

Each vector specifies the 3-D axis about which the camera is rotated. The magnitude of the vector represents the angle of rotation in radians. You can convert any rotation vector to a 3-by-3 rotation matrix using the Rodrigues formula.

You must set the `RotationVectors` and `TranslationVectors` properties together in the constructor to ensure that the number of rotation vectors equals the number of translation vectors. Setting only one property but not the other results in an error.

### 'TranslationVectors' — Camera translations

*M*-by-3 matrix | []

Camera translations, specified as the comma-separated pair consisting of 'RotationVectors' and an *M*-by-3 matrix. This matrix contains translation vectors for *M* images. The vectors contain the calibration pattern that estimates the calibration parameters. Each row of the matrix contains a vector that describes the translation of the camera relative to the corresponding pattern, expressed in world units.

The following equation provides the transformation that relates a world coordinate [*X Y Z*] and the corresponding image point [*x y*]:

$$s \begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \begin{bmatrix} R \\ t \end{bmatrix} K$$

*R* is the 3-D rotation matrix.

*t* is the translation vector.

*K* is the ???.

*s* is a scalar.

This equation does not take distortion into consideration. Distortion is removed by the `undistortImage` function.

You must set the `RotationVectors` and `TranslationVectors` properties together in the constructor to ensure that the number of rotation vectors equals the number of translation vectors. Setting only one property results in an error.

### 'WorldPoints' — World coordinates

*M*-by-2 array | []

World coordinates of key points on calibration pattern, specified as the comma-separated pair consisting of 'WorldPoints' and an *M*-by-2 array. *M* represents the number of key points in the pattern.

**'WorldUnits' — World points units**`'mm'` (default) | string

World points units, specified as the comma-separated pair consisting of `'WorldUnits'` and a string. The string describes the units of measure.

**'EstimateSkew' — Estimate skew flag**`false` (default) | logical scalar

Estimate skew flag, specified as the comma-separated pair consisting of `'EstimateSkew'` and a logical scalar. When you set the logical to `true`, the object estimates the image axes skew. When you set the logical to `false`, the image axes are exactly perpendicular.

**'NumRadialDistortionCoefficients' — Number of radial distortion coefficients**`2` (default) | 3

Number of radial distortion coefficients, specified as the comma-separated pair consisting of `'NumRadialDistortionCoefficients'` and the number `'2'` or `'3'`.

**'EstimateTangentialDistortion' — Estimate tangential distortion flag**`false` (default) | logical scalar

Estimate tangential distortion flag, specified as the comma-separated pair consisting of `'EstimateTangentialDistortion'` and the logical scalar `true` or `false`. When you set the logical to `true`, the object estimates the tangential distortion. When you set the logical to `false`, the tangential distortion is negligible.

**'ReprojectionErrors' — Reprojection errors**`[]` (default) |  $M$ -by-2-by- $P$  array

Reprojection errors, specified as the comma-separated pair of `'ReprojectionErrors'` and an  $M$ -by-2-by- $P$  array of  $[x,y]$  pairs. The  $[x,y]$  pairs represent the translation in  $x$  and  $y$  between the reprojected pattern keypoints and the detected pattern keypoints.

## Properties

**Intrinsic camera parameters:****IntrinsicMatrix — Projection matrix**

3-by-3 identity matrix

Projection matrix, specified as a 3-by-3 identity matrix. The object uses the following format for the matrix format:

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

The coordinates  $[c_x, c_y]$  represent the optical center (the principal point), in pixels. When the  $x$  and  $y$  axis are exactly perpendicular, the skew parameter,  $s$ , equals 0.

$$f_x = F * s_x$$

$$f_y = F * s_y$$

$F$ , is the focal length in world units, typically expressed in millimeters.

$[s_x, s_y]$  are the number of pixels per world unit in the  $x$  and  $y$  direction respectively.

$f_x$  and  $f_y$  are expressed in pixels.

### **PrincipalPoint — Optical center**

2-element vector

Optical center, specified as a 2-element vector  $[cx, cy]$  in pixels. The vector contains the coordinates of the optical center of the camera.

### **FocalLength — Focal length**

2-element vector

Focal length in  $x$  and  $y$ , specified as a 2-element vector  $[fx, fy]$ .

$$fx = F * sx$$

$$fy = F * sy$$

where,  $F$  is the focal length in world units, typically in millimeters, and  $[sx, sy]$  are the number of pixels per world unit in the  $x$  and  $y$  direction respectively. Thus,  $fx$  and  $fy$  are in pixels.

### **Skew — Camera axes skew**

0 (default) | scalar

Camera axes skew, specified as a scalar. If the  $x$  and the  $y$  axes are exactly perpendicular, then set the skew to 0.

### **Camera lens distortion:**

#### **RadialDistortion — Radial distortion coefficients**

[0 0 0] (default) | 2-element vector | 3-element vector

Radial distortion coefficients, specified as either a 2- or 3-element vector. When you specify a 2-element vector, the object sets the third element to 0. Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion. The camera parameters object calculates the radial distorted location of a point. You can denote the distorted points as  $(x_{\text{distorted}}, y_{\text{distorted}})$ , as follows:

$$x_{\text{distorted}} = x(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$x, y$  = undistorted pixel locations

$k_1, k_2,$  and  $k_3$  = radial distortion coefficients of the lens

$$r^2 = x^2 + y^2$$

Typically, two coefficients are sufficient. For severe distortion, you can include  $k_3$ . The undistorted pixel locations appear in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

### **TangentialDistortion** — Tangential distortion coefficients

[0 0] ' (default) | 2-element vector

Tangential distortion coefficients, specified as a 2-element vector. Tangential distortion occurs when the lens and the image plane are not parallel. The camera parameters object calculates the tangential distorted location of a point. You can denote the distorted points as  $(x_{\text{distorted}}, y_{\text{distorted}})$ , as follows:

$$x_{\text{distorted}} = x + [2 * p_1 * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x]$$

$x, y$  = undistorted pixel locations

$p_1$  and  $p_2$  = tangential distortion coefficients of the lens

$$r^2 = x^2 + y^2$$

The undistorted pixel locations appear in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

### **Extrinsic camera parameters:**

#### **RotationMatrices** — 3-D rotation matrix

3-by-3-by- $P$  matrix (read-only)

3-D rotation matrix, specified as a 3-by-3-by- $P$ , with  $P$  number of pattern images. Each 3-by-3 matrix represents the same 3-D rotation as the corresponding vector.

The following equation provides the transformation that relates a world coordinate in the checkerboard's frame  $[X Y Z]$  and the corresponding image point  $[x y]$ :

$$s[x \ y \ 1] = [X \ Y \ Z \ 1] \begin{bmatrix} R \\ t \end{bmatrix} K$$

$R$  is the 3-D rotation matrix.

$t$  is the translation vector.

$K$  is the ???.

$s$  is a scalar.

This equation does not take distortion into consideration. Distortion is removed by the `undistortImage` function.

### **RotationVectors** — 3-D rotation vectors

[ ] (default) |  $M$ -by-3 matrix (read-only)

3-D rotation vectors, specified as a  $M$ -by-3 matrix containing  $M$  rotation vectors. Each vector describes the 3-D rotation of the camera's image plane relative to the corresponding calibration pattern. The vector specifies the 3-D axis about which the camera is rotated, where the magnitude is the rotation angle in radians. The corresponding 3-D rotation matrices are given by the `RotationMatrices` property

### **TranslationVectors** — Camera translations

$M$ -by-3 matrix | [ ]

Camera translations, specified as an  $M$ -by-3 matrix. This matrix contains translation vectors for  $M$  images. The vectors contain the calibration pattern that estimates the calibration parameters. Each row of the matrix contains a vector that describes the translation of the camera relative to the corresponding pattern, expressed in world units.

The following equation provides the transformation that relates a world coordinate in the checkerboard's frame  $[X \ Y \ Z]$  and the corresponding image point  $[x \ y]$ :

$$s[x \ y \ 1] = [X \ Y \ Z \ 1] \begin{bmatrix} R \\ t \end{bmatrix} K$$

$R$  is the 3-D rotation matrix.

$t$  is the translation vector.

$K$  is the ???.

$s$  is a scalar.

This equation does not take distortion into consideration. Distortion is removed by the `undistortImage` function.



You must set the `RotationVectors` and `TranslationVectors` properties in the constructor to ensure that the number of rotation vectors equals the number of translation vectors. Setting only one property but not the other results in an error.

#### **Estimated camera parameter accuracy:**

##### **MeanReprojectionError — Average Euclidean distance**

numeric value (read-only)

Average Euclidean distance between reprojected and detected points, specified as a numeric value in pixels.

##### **ReprojectionErrors — Estimated camera parameters accuracy**

$M$ -by-2-by- $P$  array

Estimated camera parameters accuracy, specified as an  $M$ -by-2-by- $P$  array of  $[x\ y]$  coordinates. The  $[x\ y]$  coordinates represent the translation in  $x$  and  $y$  between the reprojected pattern key points and the detected pattern key points. The values of this property represent the accuracy of the estimated camera parameters.  $P$  is the number of pattern images that estimates camera parameters.  $M$  is the number of keypoints in each image.

##### **ReprojectedPoints — World points reprojected onto calibration images**

$M$ -by-2-by- $P$  array

World points reprojected onto calibration images, specified as an  $M$ -by-2-by- $P$  array of  $[x\ y]$  coordinates.  $P$  is the number of pattern images and  $M$  is the number of keypoints in each image.

#### **Estimate camera parameters settings:**

##### **NumPatterns — Number of calibrated patterns**

integer

Number of calibration patterns that estimates camera extrinsics, specified as an integer. The number of calibration patterns equals the number of translation and rotation vectors.

##### **WorldPoints — World coordinates**

$M$ -by-2 array | []

World coordinates of key points on calibration pattern, specified as an  $M$ -by-2 array.  $M$  represents the number of key points in the pattern.

### **WorldUnits — World points units**

'mm' (default) | string

World points units, specified as a string. The string describes the units of measure.

### **EstimateSkew — Estimate skew flag**

false (default) | logical scalar

Estimate skew flag, specified as a logical scalar. When you set the logical to `true`, the object estimates the image axes skew. When you set the logical to `false`, the image axes are exactly perpendicular.

### **NumRadialDistortionCoefficients — Number of radial distortion coefficients**

2 (default) | 3

Number of radial distortion coefficients, specified as the number '2' or '3'.

### **EstimateTangentialDistortion — Estimate tangential distortion flag**

false (default) | logical scalar

Estimate tangential distortion flag, specified as the logical scalar `true` or `false`. When you set the logical to `true`, the object estimates the tangential distortion. When you set the logical to `false`, the tangential distortion is negligible.

## Methods

pointsToWorld

Determine world coordinates of image points

toStruct

Convert a camera parameters object into a struct

## Output Arguments

### **cameraParams — Camera parameters**

cameraParameters object

Camera parameters, returned as a cameraParameters object. The object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

## Examples

### Remove Distortion From an Image Using The cameraParameters Object

This example shows you how to use the cameraParameters object in a workflow to remove distortion from an image. The example creates a cameraParameters object manually. In practice, use the estimateCameraParameters or the cameraCalibrator app to derive the object.

Create a cameraParameters object manually.

```
IntrinsicMatrix = [715.2699    0         0;
                  0         711.5281    0;
                  565.6995  355.3466    1];
radialDistortion = [-0.3361 0.0921];
cameraParams = cameraParameters('IntrinsicMatrix',IntrinsicMatrix,'RadialDistortion',radialDistortion);
```

Remove distortion from the image.

```
I = imread(fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'calibration', 'fishEye', 'fishEyeImage1.tif'));
J = undistortImage(I,cameraParams);
```

Display the original and undistorted images.

```
figure; imshowpair(imresize(I,0.5), imresize(J,0.5), 'montage');
title('Original Image (left) vs. Corrected Image (right)');
```



- “Measuring Planar Objects with a Calibrated Camera”

- “Code Generation for Depth Estimation From Stereo Video”

### References

- [1] Zhang, Z. “A flexible new technique for camera calibration”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 22, No. 11, pp. 1330–1334, 2000.
- [2] Heikkila, J, and O. Silven. “A Four-step Camera Calibration Procedure with Implicit Image Correction”, *IEEE International Conference on Computer Vision and Pattern Recognition*, 1997.

### See Also

stereoParameters | Camera Calibrator | detectCheckerboardPoints | estimateCameraParameters | generateCheckerboardPoints | showExtrinsics | showReprojectionErrors | undistortImage

### More About

- “Single Camera Calibration App”

**Introduced in R2014a**

# pointsToWorld

**Class:** cameraParameters

Determine world coordinates of image points

## Syntax

```
worldPoints = pointsToWorld(cameraParams,rotationMatrix,  
translationVector,imagePoints)
```

## Description

`worldPoints = pointsToWorld(cameraParams,rotationMatrix,translationVector,imagePoints)` returns world points on the X-Y plane, which correspond to the undistorted image points.

## Input Arguments

### **cameraParams** — Camera parameters

cameraParameters object

Object for storing camera parameters, specified as a cameraParameters returned by the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

### **rotationMatrix** — Camera rotation

3-by-3 matrix

Camera rotation, specified as 3-by-3 matrix. The matrix represents the rotation of the camera in world coordinates.

### **translationVector** — Camera translation

3-element vector

Camera translation, specified as a 3-element vector. The vector represents the translation of the camera in world coordinates.

### **imagePoints** — Undistorted image points

*M*-by-2 matrix

Undistorted image points, specified as an *M*-by-2 matrix containing *M*  $[x, y]$  coordinates of undistorted image points.

`pointsToWorld` does not account for lens distortion. Therefore, the `imagePoints` input must contain image points detected in the undistorted image, or they must be undistorted using the `undistortPoints` function.

## Tips

The `triangulate` function does not account for lens distortion. You can either undistort the images using the `undistortImage` function before detecting the points, or you can undistort the points themselves using the `undistortPoints` function.

## Output Arguments

### **WorldPoints** — World coordinates

*M*-by-2 array

World coordinates, returned as an *M*-by-2 array. *M* represents the number of undistorted points in  $[x, y]$  world coordinates.

## See Also

`cameraParameters` | Camera Calibrator | `estimateCameraParameters` | `extrinsics` | Stereo Camera Calibrator | `undistortImage` | `undistortImage` | `undistortPoints`

## toStruct

**Class:** cameraParameters

Convert a camera parameters object into a struct

## Syntax

```
paramStruct = toStruct(cameraParams)
```

## Description

`paramStruct = toStruct(cameraParams)` returns a struct containing the camera parameters in the `cameraParams` input object. You can use the struct to create an identical `cameraParameters` object. Use the struct for C code generation. You can call `toStruct`, and then pass the resulting structure into the generated code, which re-creates the `cameraParameters` object.

## Input Arguments

**cameraParams** — Camera parameters

cameraParameters object

Camera parameters, specified as a cameraParameters object. The object contains the parameters for the camera.

## Output Arguments

**paramStruct** — Camera parameters

struct

Camera parameters, returned as a struct.

## Related Examples

- “Code Generation for Depth Estimation From Stereo Video”

**Introduced in R2015a**



## pcplayer class

Visualize streaming 3-D point cloud data

### Syntax

```
player = pcplayer(xlimits,ylimits,zlimits)
player = pcplayer(xlimits,ylimits,zlimits,Name,Value)
```

### Description

`player = pcplayer(xlimits,ylimits,zlimits)` returns a player for visualizing 3-D point cloud data streams. The `xlimits`, `ylimits`, and `zlimits` inputs specify the *x*-, *y*-, and *z*- axis limits for the player axes.

Use this player to visualize 3-D point cloud data from devices such as Microsoft® Kinect®.

`player = pcplayer(xlimits,ylimits,zlimits,Name,Value)` returns a player with additional properties specified by one or more `Name, Value` pair arguments.

To improve performance, `pcplayer` automatically downsamples the rendered point cloud during interaction with the figure. The downsampling occurs only for rendering the point cloud and does not affect the saved points.

### Input Arguments

#### **xLimits** — Range of *x*-axis coordinates

1-by-2 vector

Range of *x*-axis coordinates, specified as a 1-by-2 vector in the format  $[min\ max]$ . `pcplayer` does not display data outside these limits.

#### **yLimits** — Range of *y*-axis coordinates

1-by-2 vector

Range of *y*-axis coordinates, specified as a 1-by-2 vector in the format  $[min\ max]$ . `pcplayer` does not display data outside these limits.

### **zLimits** — Range of z-axis coordinates

1-by-2 vector

Range of z-axis coordinates, specified as a 1-by-2 vector in the format [*min max*]. `pcplayer` does not display data outside these limits.

### **Name-Value Pair Arguments**

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

Example: `'VerticalAxisDir', 'Up'`.

### **'MarkerSize'** — Diameter of marker

6 (default) | positive scalar

Diameter of marker, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive scalar. The value specifies the approximate diameter of the point marker. MATLAB graphics defines the unit as points. A marker size larger than six can reduce the rendering performance.

### **'VerticalAxis'** — Vertical axis

'Z' (default) | 'X' | 'Y'

Vertical axis, specified as the comma-separated pair consisting of `'VerticalAxis'` and a string specifying the vertical axis: `'X'`, `'Y'`, or `'Z'`.

### **'VerticalAxisDir'** — Vertical axis direction

'Up' (default) | 'Down'

Vertical axis direction, specified as the comma-separated pair consisting of `'VerticalAxisDir'` and a string specifying the direction of the vertical axis: `'Up'` or `'Down'`.

## **Properties**

### **Axes** — Player axes handle

axes graphics object

Player axes handle, specified as an `axes` graphics object.

## Methods

<code>hide</code>	Hides player figure
<code>isOpen</code>	Visibility of point cloud player figure
<code>show</code>	Shows player figure
<code>view</code>	Display point cloud

## Examples

### View Rotating 3-D Point Cloud

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Define a rotation matrix and 3-D transform.

```
x = pi/180;
R = [ cos(x) sin(x) 0 0
      -sin(x) cos(x) 0 0
        0         0  1 0
        0         0  0 1];
```

```
tform = affine3d(R);
```

Compute `x_y_` limits that ensure that the rotated teapot is not clipped.

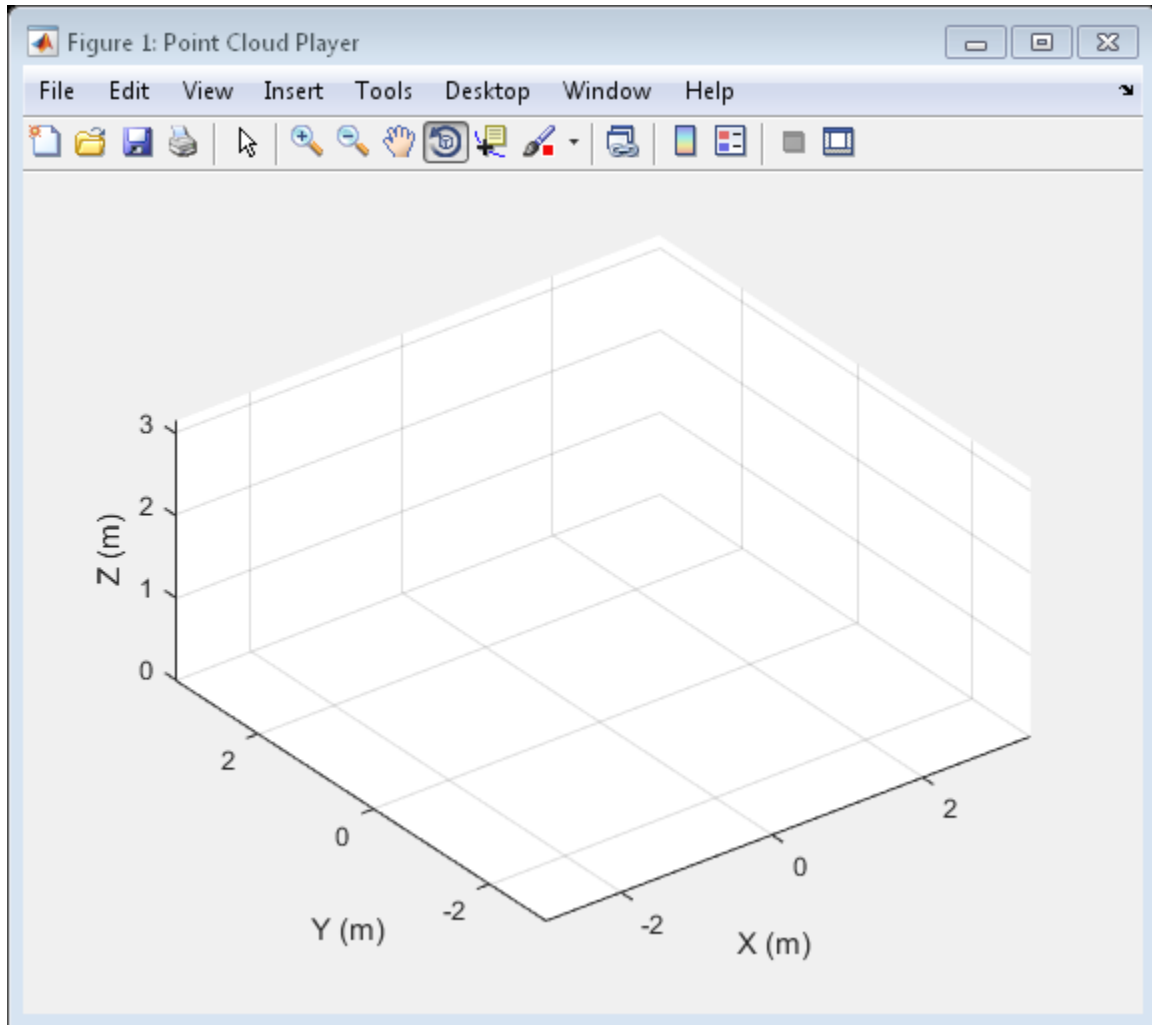
```
lower = min([ptCloud.XLimits ptCloud.YLimits]);
upper = max([ptCloud.XLimits ptCloud.YLimits]);

xlimits = [lower upper];
ylimits = [lower upper];
zlimits = ptCloud.ZLimits;
```

Create the player and customize player axis labels.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

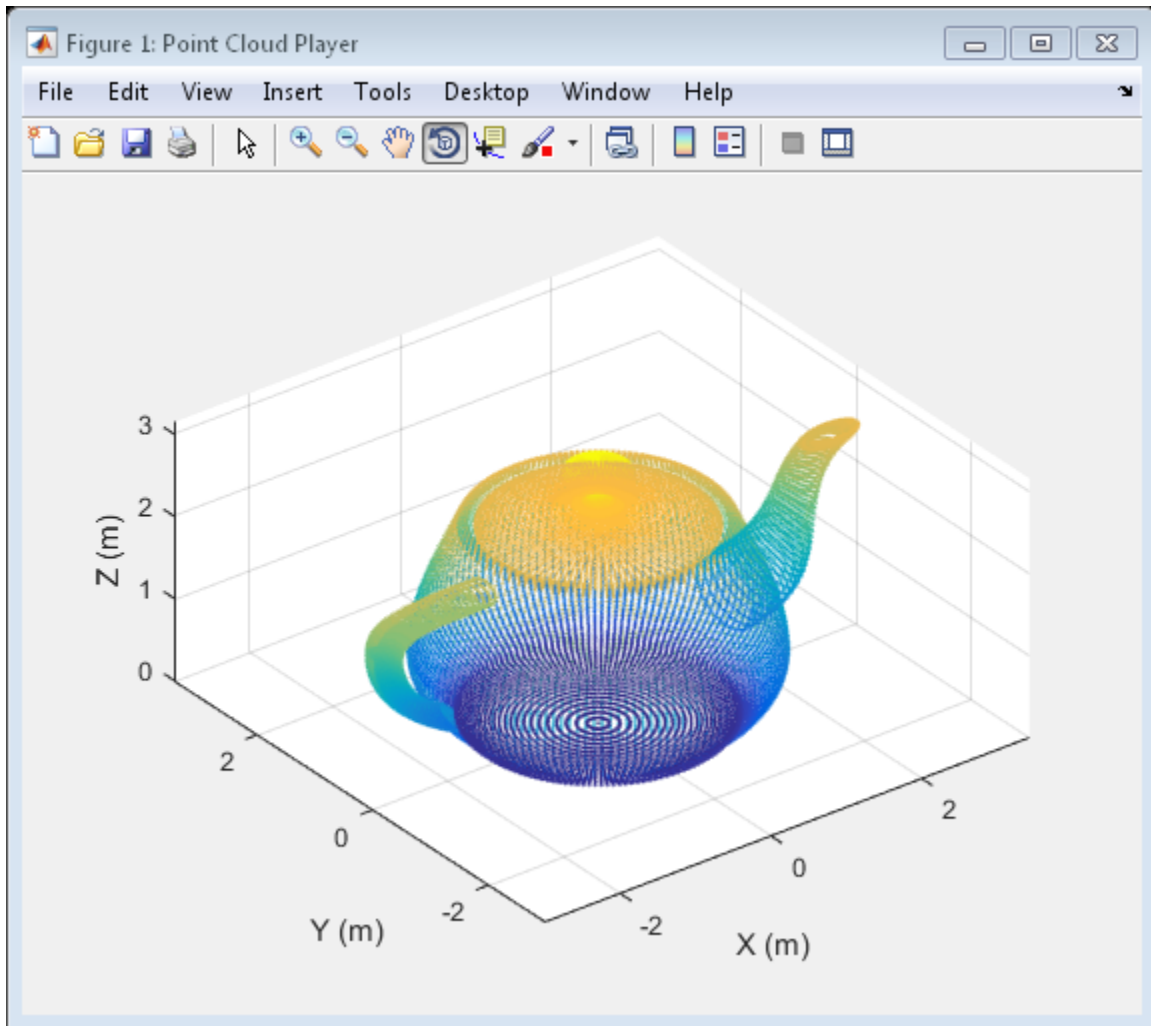
```
xlabel(player.Axes,'X (m)');  
ylabel(player.Axes,'Y (m)');  
zlabel(player.Axes,'Z (m)');
```



Rotate the teapot around the  $z$ -axis.

```
for i = 1:360  
    ptCloud = pctransform(ptCloud,tform);  
end
```

```
view(player,ptCloud);  
end
```



### Terminate a Point Cloud Processing Loop

Create the player and add data.

```
player = pcplayer([0 1],[0 1],[0 1]);
```

Display continuous player figure. Use the `isOpen` method to check if player figure window is open.

```
while isOpen(player)
    ptCloud = pointCloud(rand(1000,3,'single'));
    view(player, ptCloud);
end
```

Terminate while-loop by closing `pcplayer` figure window.

- “3-D Point Cloud Registration and Stitching”

## Algorithms

`pcplayer` supports the `'opengl'` option for the `Renderer` figure property only.

### See Also

`planeModel` | `pointCloud` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcwrite` | `plot3` | `scatter3`

**Introduced in R2015b**

# hide

**Class:** `pcplayer`

Hides player figure

## Syntax

```
hide(player)
```

## Description

`hide(player)` hides the `pcplayer` point cloud figure. To redisplay the player, use `pcplayer.show(player)`.

## Input Arguments

**player** — **Player**

`pcplayer` object

Player for visualizing 3-D point cloud data streams, specified as a `pcplayer` object.

## Examples

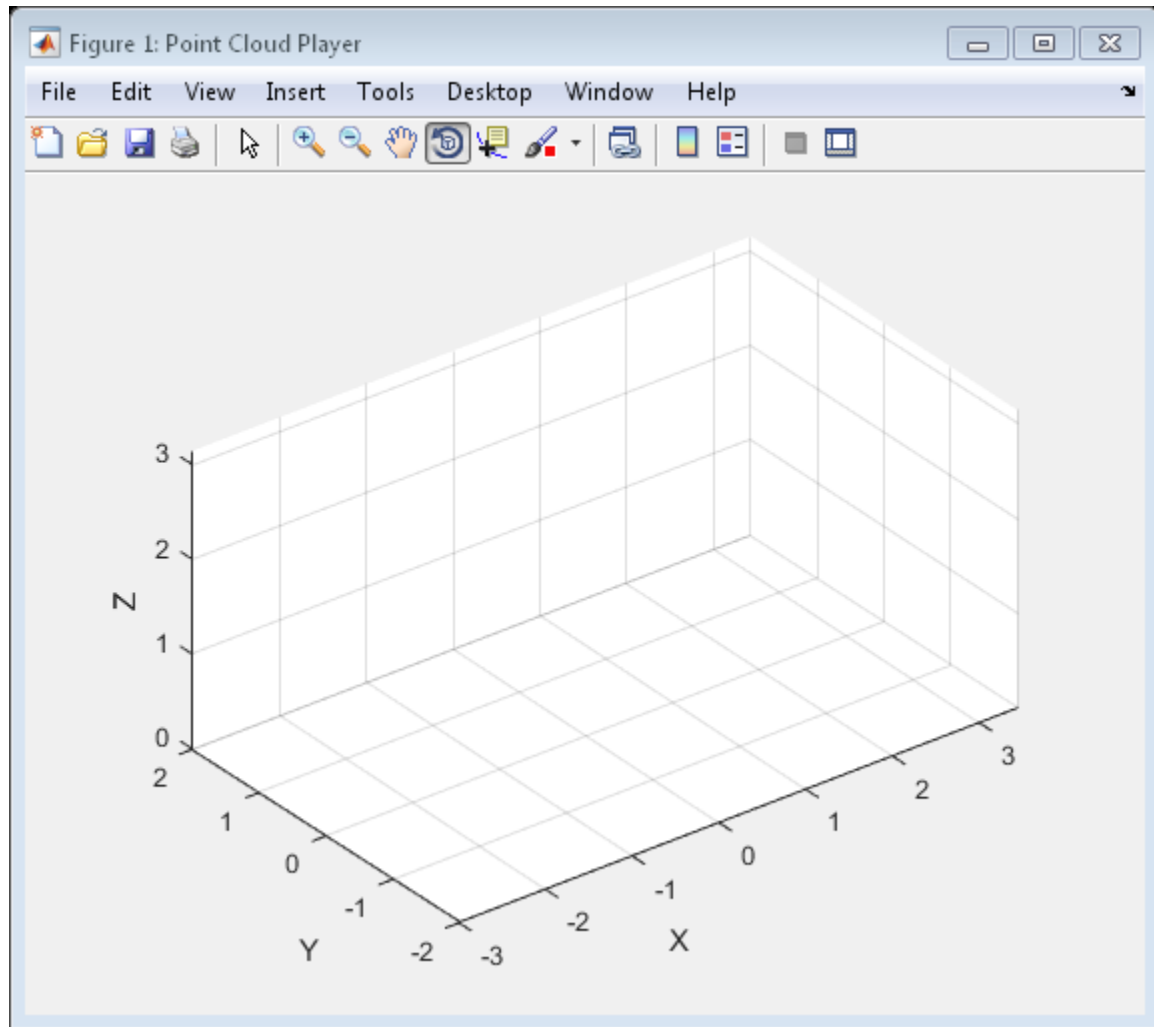
### Hide and Show 3-D Point Cloud Figure

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Create the player and customize player axis labels.

```
player = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits);
```



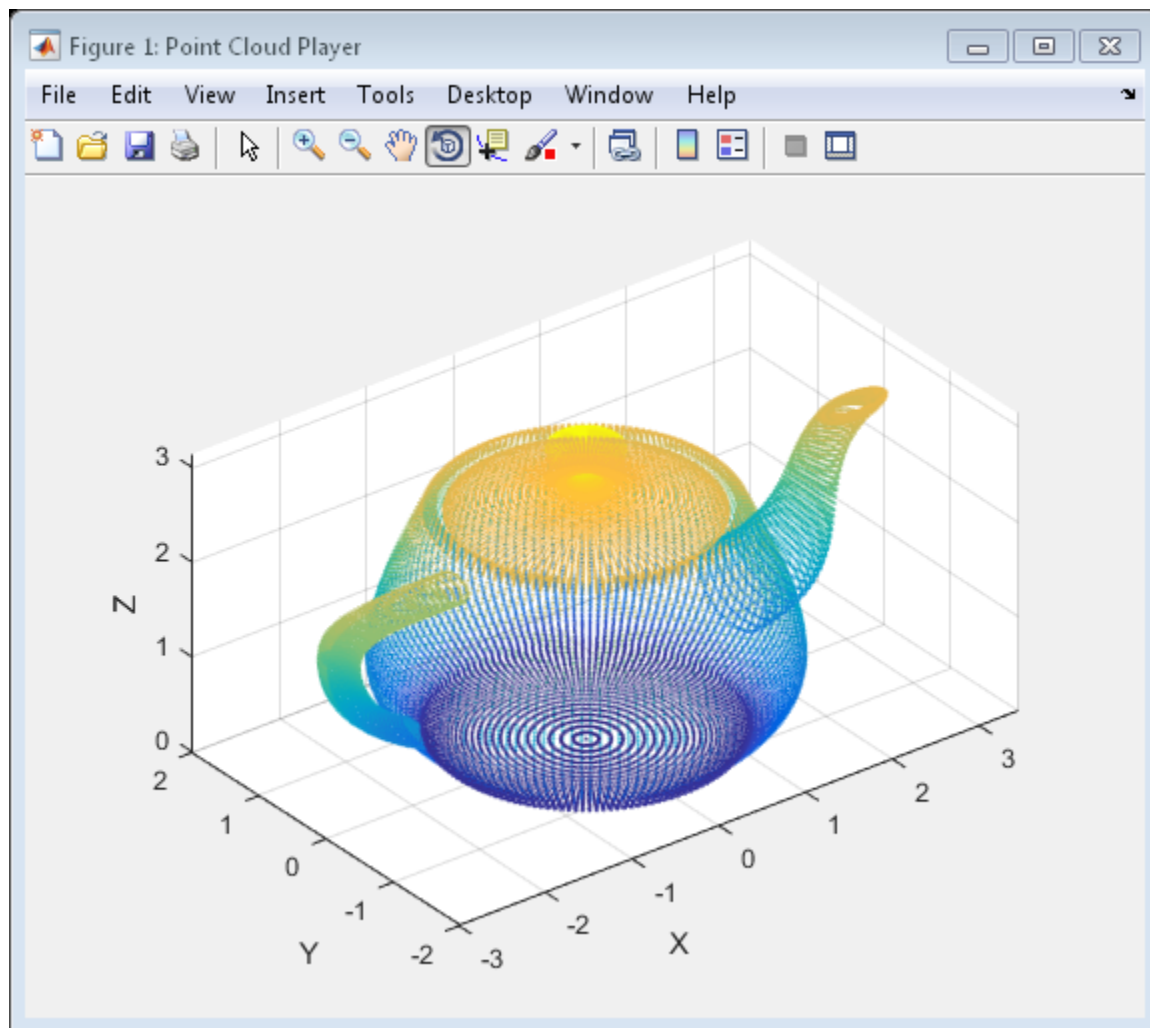
Hide figure.

```
hide(player)
```

Show figure.

```
show(player)  
view(player,ptCloud);
```





Introduced in R2015b

# isOpen

**Class:** `pcplayer`

Visibility of point cloud player figure

## Syntax

```
isOpen(player)
```

## Description

`isOpen(player)` returns `true` or `false` to indicate whether the player is visible.

## Input Arguments

**player** — **Player**  
`pcplayer` object

Player for visualizing 3-D point cloud data streams, specified as a `pcplayer` object

## Examples

### Terminate a Point Cloud Processing Loop

Create the player and add data.

```
player = pcplayer([0 1],[0 1],[0 1]);
```

Display continuous player figure. Use the `isOpen` method to check if player figure window is open.

```
while isOpen(player)
    ptCloud = pointCloud(rand(1000,3,'single'));
    view(player, ptCloud);
end
```

Terminate while-loop by closing poplayer figure window.

**Introduced in R2015b**

# show

**Class:** `pcplayer`

Shows player figure

## Syntax

```
show(player)
```

## Description

`show(player)` makes the point cloud player figure visible again after closing or hiding it. You can use this method at the end of a while loop to show its display code.

## Input Arguments

**player** — **Player**  
`pcplayer` object

Player for visualizing 3-D point cloud data streams, specified as a `pcplayer` object. Use this method to view the figure after you have removed it from display. For example, after you x-out of a figure and you want to view it again. This is particularly useful to use after a while loop that contains display code ends.

## Examples

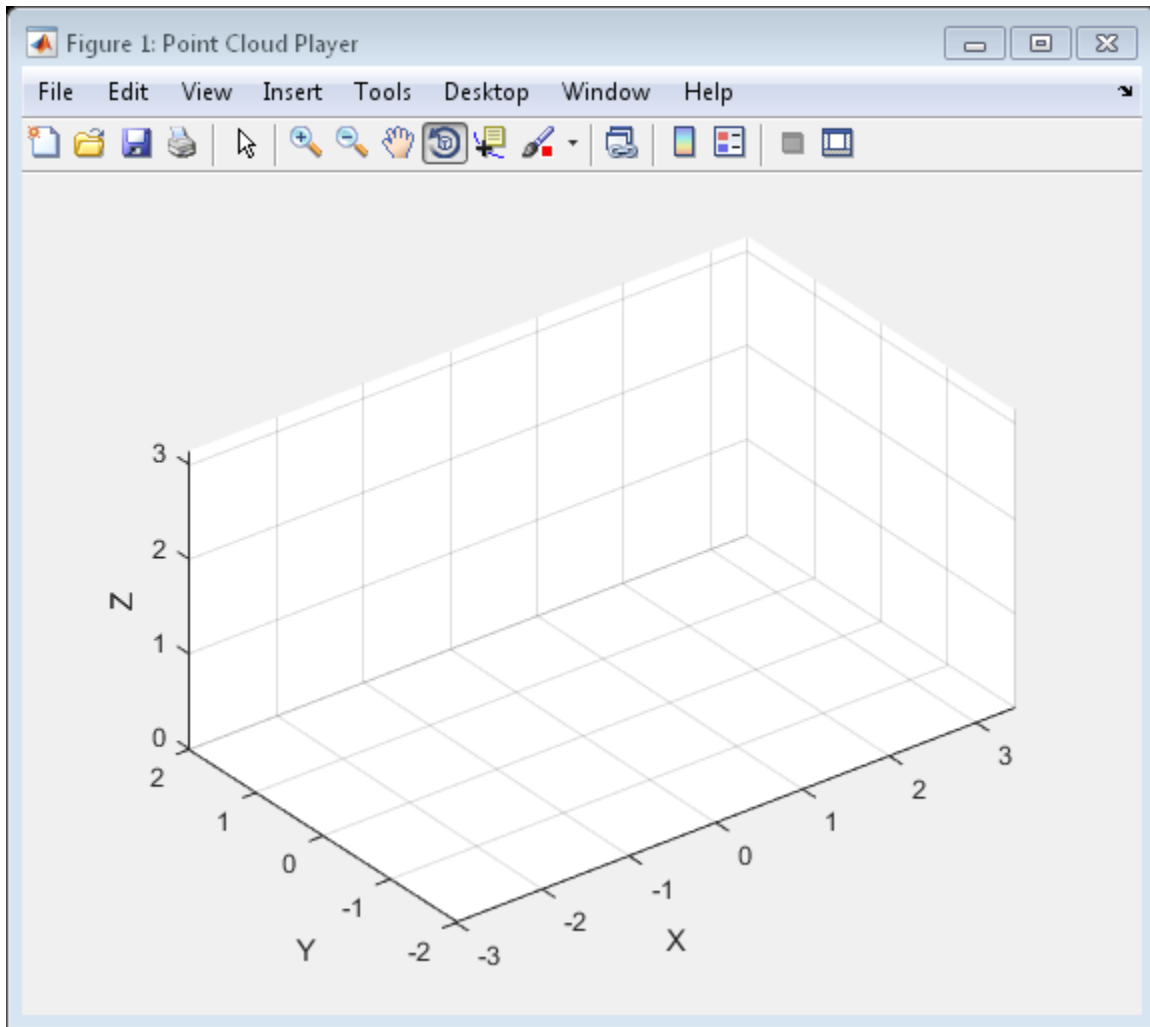
### Hide and Show 3-D Point Cloud Figure

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Create the player and customize player axis labels.

```
player = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits);
```

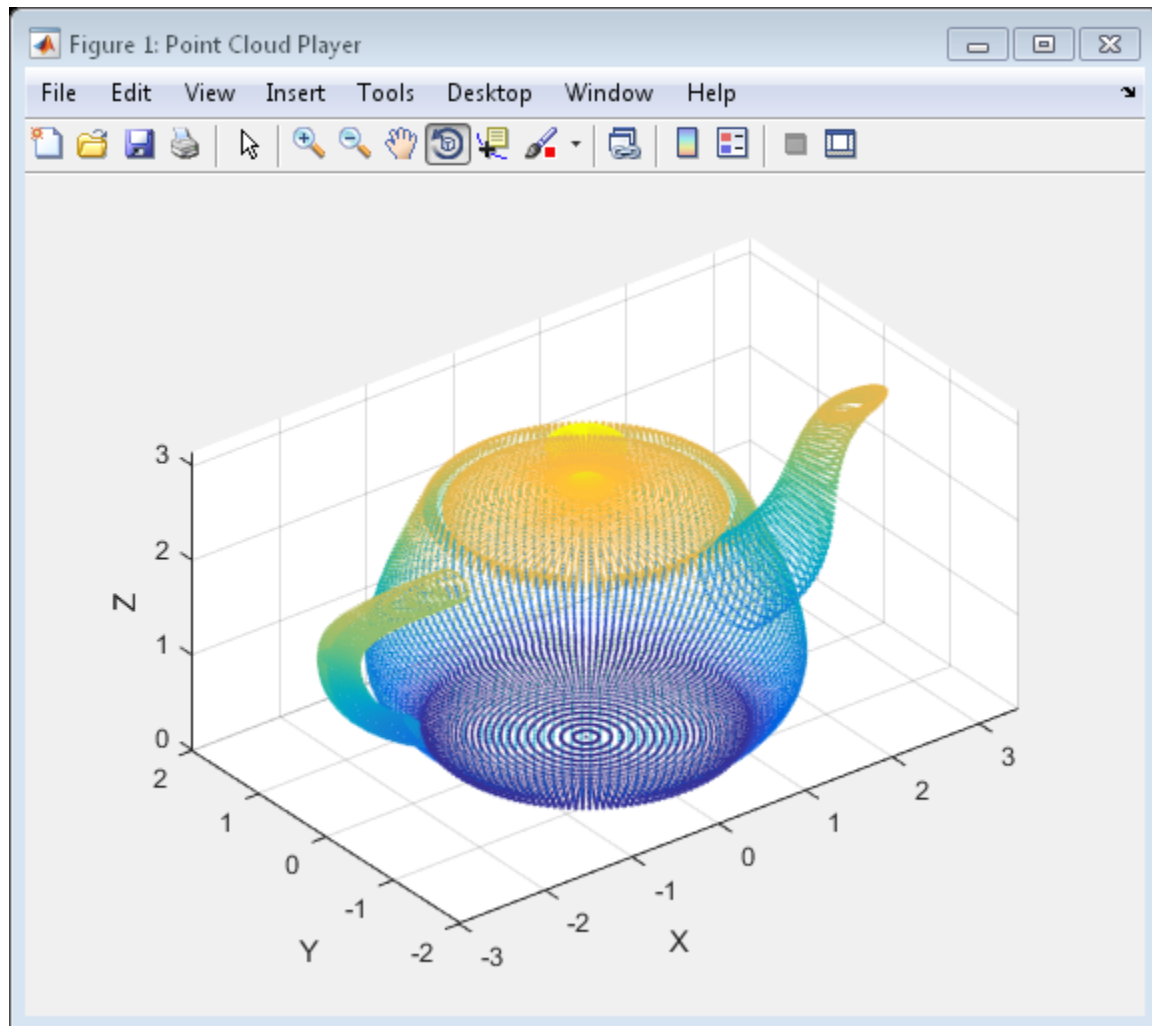


Hide figure.

```
hide(player)
```

Show figure.

```
show(player)  
view(player,ptCloud);
```



Introduced in R2015b

## view

**Class:** pcplayer

Display point cloud

## Syntax

```
view(player,ptCloud)
view(player,xyzPoints)
view(player,xyzPoints,Color)
```

## Description

`view(player,ptCloud)` displays a point cloud in the pcplayer figure window, `player`. The points, locations, and colors are stored in the `ptCloud` object.

`view(player,xyzPoints)` displays the points of a point cloud at the locations specified by the `xyzPoints` matrix. The color of each point is determined by the `z` value.

`view(player,xyzPoints,Color)` displays a point cloud with colors specified by `Color`.

## Input Arguments

### **ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **player** — Player

axes graphics object

Player for visualizing 3-D point cloud data streams, specified as an `axes` graphics object

### **xyzPoints** — Point cloud *x*, *y*, and *z* locations

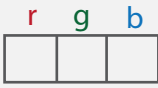
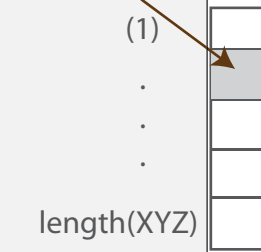
*M*-by-3 numeric matrix | *M*-by-*N*-by-3 numeric matrix

Point cloud  $x$ ,  $y$ , and  $z$  locations, specified as either an  $M$ -by-3 or an  $M$ -by- $N$ -by-3 numeric matrix. The  $M$ -by- $N$ -by-3 numeric matrix is commonly referred to as an *organized point cloud*. The `xyzPoints` numeric matrix contains  $M$  or  $M$ -by- $N$   $[x,y,z]$  points. The  $z$  values in the numeric matrix, which generally correspond to depth or elevation, determine the color of each point.

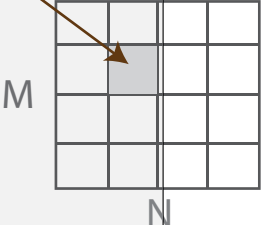
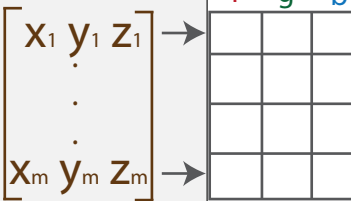
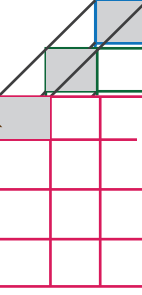
**Color — Point cloud color**

`ColorSpec` (Color Specification) color string | 1-by-3 RGB vector |  $M$ -by-1 vector |  $M$ -by- $N$  matrix |  $M$ -by-3 matrix |  $M$ -by- $N$ -by-3 matrix

Point cloud color of points, specified as a `ColorSpec` (Color Specification) color string, a 1-by-3 RGB vector, an  $M$ -by-1 vector, an  $M$ -by- $N$ ,  $M$ -by-3, or  $M$ -by- $N$ -by-3 matrix. You can specify the same color for all points or a different color for each point. RGB values range between [0, 1] when you set `Color` to `single` or `double` and between [0, 255] when you set `Color` to `uint8`.

Points Input	Color Selection	Valid Values of Color	
xyzPoints	Same color for all points	<code>ColorSpec</code> (Color Specification) color string or a 1-by-3 RGB vector	 <p>1-by-3</p>
	Different color for each point	Vector or $M$ -by- $N$ matrix. The matrix must contain values that are linearly mapped to a color in the current colormap.	<p>prescaled value</p>  <p>length(XYZ)</p> <p>vector</p>



Points Input	Color Selection	Valid Values of Color	
			<p>point(m,n) prescaled value</p>  <p><math>M</math></p> <p><math>N</math></p> <p><math>M</math>-by-<math>N</math></p>
		<p><math>M</math>-by-3 matrix or <math>M</math>-by-<math>N</math>-by-3 matrix containing RGB values for each point.</p>	 <p><math>\begin{bmatrix} x_1 &amp; y_1 &amp; z_1 \\ \vdots &amp; \vdots &amp; \vdots \\ x_m &amp; y_m &amp; z_m \end{bmatrix}</math></p> <p><math>r</math> <math>g</math> <math>b</math></p> <p><math>M</math>-by-3</p>  <p>point(m,n)</p> <p><math>M</math></p> <p><math>N</math></p> <p><math>M</math>-by-<math>N</math>-by-</p>

# Examples

### View Rotating 3-D Point Cloud

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Define a rotation matrix and 3-D transform.

```
x = pi/180;  
R = [ cos(x) sin(x) 0 0  
      -sin(x) cos(x) 0 0  
        0         0   1 0  
        0         0   0 1];
```

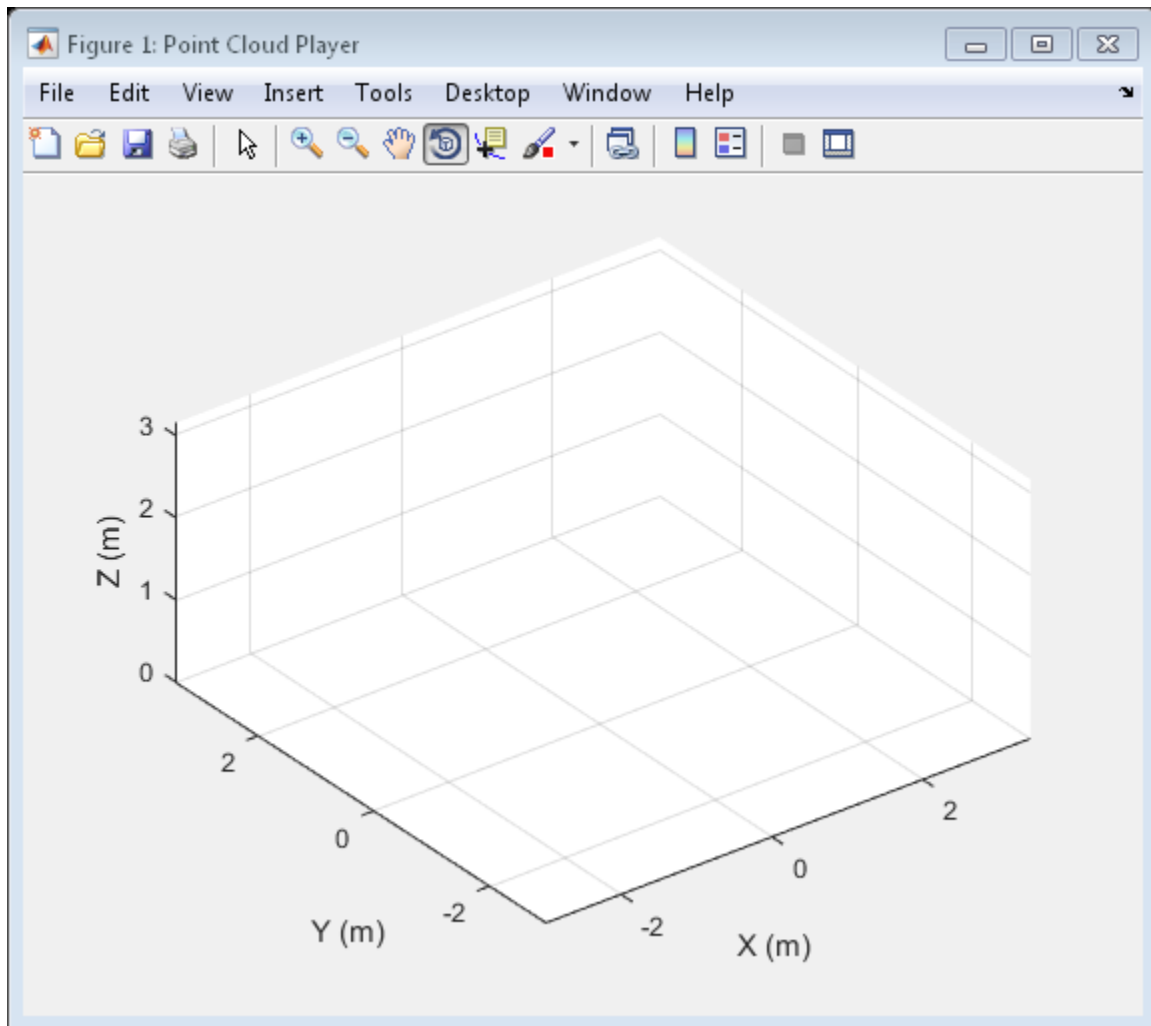
```
tform = affine3d(R);
```

Compute  $x\_y\_$  limits that ensure that the rotated teapot is not clipped.

```
lower = min([ptCloud.XLimits ptCloud.YLimits]);  
upper = max([ptCloud.XLimits ptCloud.YLimits]);  
  
xlimits = [lower upper];  
ylimits = [lower upper];  
zlimits = ptCloud.ZLimits;
```

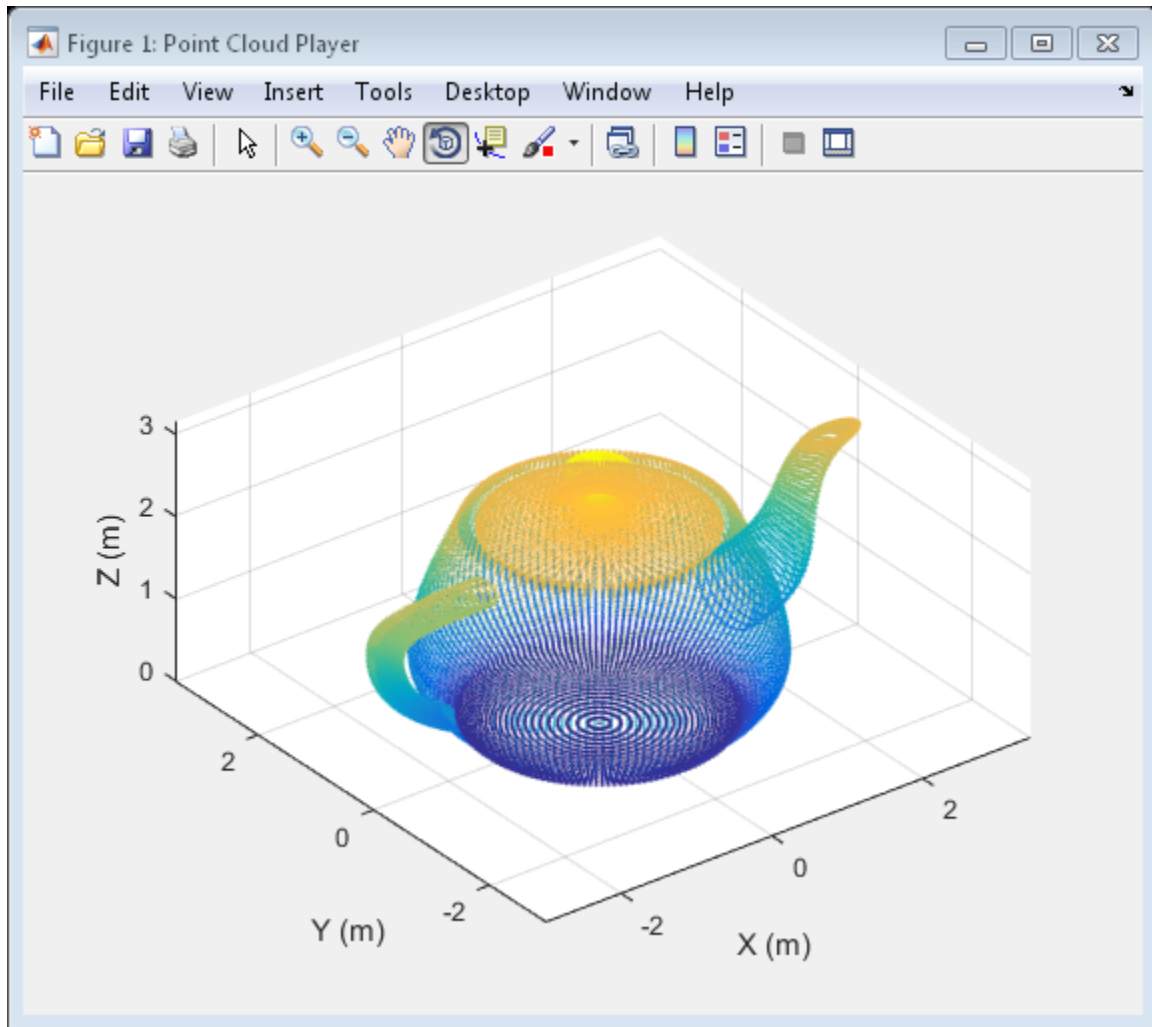
Create the player and customize player axis labels.

```
player = pcplayer(xlimits,ylimits,zlimits);  
  
xlabel(player.Axes,'X (m)');  
ylabel(player.Axes,'Y (m)');  
zlabel(player.Axes,'Z (m)');
```



Rotate the teapot around the z-axis.

```
for i = 1:360
    ptCloud = pctransform(ptCloud,tform);
    view(player,ptCloud);
end
```



Introduced in R2015b

# pointTrack class

Object for storing matching points from multiple views

## Syntax

```
track = pointTrack(viewIDs,points)
```

## Description

`track = pointTrack(viewIDs,points)` returns an object that stores matching 2-D points from multiple views. You can also create this point track object using the `findTracks` method of the `viewSet` object.

## Input Arguments

**viewIDs** — View IDs of camera poses

*M*-element vector

View IDs of camera poses, specified as an *M*-element vector of scalar integers.

**points** — 2-D points that match across multiple camera views

*M*-by-2 matrix

2-D points that match across multiple camera views, specified as an *M*-by-2 matrix of (*x,y*) point coordinates. You can use the `matchFeatures` function to find these points, and then save them using this object.

## Output Arguments

**tracks** — Point track objects

array of `pointTrack` objects

Point track objects, returned as an array of `pointTrack` objects. Each track contains 2-D projections of the same 3-D world point from its corresponding views.

### Examples

#### Create a Point Track Object

Save  $(x, y)$  points and view IDs.

```
points = [10,20;11,21;12,22];  
viewIDs = [1 2 3];
```

Create a `pointTrack` object to save points and IDs.

```
track = pointTrack(viewIDs,points);
```

- “3-D Point Cloud Registration and Stitching”

#### See Also

`viewSet` | `vision.PointTracker` | `bundleAdjustment` | `matchFeatures` | `triangulateMultiview`

#### More About

- “Coordinate Systems”

**Introduced in R2016a**

## viewSet class

Object for managing data for structure-from-motion and visual odometry

### Syntax

```
vSet = viewSet
```

### Description

`vSet = viewSet` returns an empty `viewSet` object that stores views and connections between views. A view includes feature points and an absolute camera pose. A connection between two views includes point correspondences and the relative camera pose between them. Use the `viewSet` methods to find point tracks used by `triangulateMultiview` and `bundleAdjustment` functions.

#### Code Generation Support:

Supports Code Generation: No

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

### Properties

These properties are read-only.

#### **NumViews** — Number of views

integer

Number of views, stored as an integer.

#### **Views** — View attributes

four-column table

View attributes, stored as a four-column table. The table contains columns for `ViewID`, `Points`, `Orientation`, and `Location`. Use the `poses` method to obtain the IDs, orientation, and location for the points.

	1	2	3	4
	ViewId	Points	Orientation	Location
1	1	574x2 single	[]	[]
2	2	503x2 single	[]	[]
3	3	540x2 single	[]	[]
4	4	472x2 single	[]	[]
5	5	421x2 single	[]	[]
6				
7				

**Connections — Pairwise connections between views**

five-column table

Pairwise connections between views, stored as a five-column table. The columns are ViewID1, ViewID2, Matches, RelativeOrientation, and RelativeLocation. The number of entries in the table represent the number of connections. Each index in the Matches column represents a connection between the two views indicated by the view IDs.



The screenshot shows a window titled 'Variables - vSet.Connections' with two tabs: 'vSet' and 'vSet.Connections'. The active tab displays a table with the following data:

	1 ViewId1	2 ViewId2	3 Matches	4 RelativeOrientation	5 RelativeLocation
1	1	2	229x2 uint32	[]	[]
2	2	3	262x2 uint32	[]	[]
3	3	4	203x2 uint32	[]	[]
4	4	5	155x2 uint32	[]	[]
5					
6					
7					

## Output Arguments

### **vSet** – View set object

viewSet object

viewSet object used to store views and connections between the views.

The screenshot shows a window titled 'Variables - vSet' with a tab for 'vSet'. Below the tab, it displays '1x1 viewSet' and a table of properties:

Property	Value
Views	5x4 table
Connections	4x5 table
NumViews	5

### Methods

<code>addView</code>	Add a new view to view set object
<code>updateView</code>	Modify an existing view in a view set object
<code>deleteView</code>	Delete an existing view from view set object
<code>hasView</code>	Check if view exists
<code>addConnection</code>	Add a connection between two views
<code>updateConnection</code>	Modify a connection between two views in a view set object
<code>deleteConnection</code>	Delete a connection between two views from view set object
<code>hasConnection</code>	Check if a connection exists between two views
<code>findTracks</code>	Find matched points across multiple views
<code>poses</code>	Returns camera poses associated to views

### Examples

#### Find Point Tracks Across Sequence of Images

Load images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
images = imageSet(imageDir);
```

Compute features for the first image.

```
I = rgb2gray(read(images, 1));  
pointsPrev = detectSURFFeatures(I);  
[featuresPrev,pointsPrev] = extractFeatures(I,pointsPrev);
```

Create a `viewSet` object.

```
vSet = viewSet;  
vSet = addView(vSet,1,'Points',pointsPrev);
```

Compute features and matches for the rest of the images.

```
for i = 2:images.Count
    I = rgb2gray(read(images,i));
    points = detectSURFFeatures(I);
    [features, points] = extractFeatures(I,points);
    vSet = addView(vSet,i,'Points',points);
    pairsIdx = matchFeatures(featuresPrev,features);
    vSet = addConnection(vSet,i-1,i,'Matches',pairsIdx);
    featuresPrev = features;
end
```

Find point tracks.

```
tracks = findTracks(vSet);
```

- “Structure From Motion From Multiple Views”
- “Structure From Motion From Two Views”
- “Code Generation for Depth Estimation From Stereo Video”

## See Also

`pointTrack` | `bundleAdjustment` | `detectBriskFeatures` | `detectFastFeatures` | `detectHarrisFeatures` | `detectMinEigenFeatures` | `detectMSERFeatures` | `detectSURFFeatures` | `matchFeatures` | `table` | `triangulateMultiview`

## More About

- “Single Camera Calibration App”
- “Structure from Motion”

**Introduced in R2016a**

# addView

**Class:** viewSet

Add a new view to view set object

## Syntax

```
vSet = addView(vSet,viewId)
vSet = addView(vSet,viewId,Name,Value)
```

## Description

`vSet = addView(vSet,viewId)` adds the view specified by `viewID` to the specified `viewSet` object.

`vSet = addView(vSet,viewId,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**vSet — View set object**

`viewSet` object

`viewSet` object.

**viewId — Camera pose view ID**

integer

Camera pose view ID in the `viewSet` object, specified as an integer.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: 'Location', '[0,0,0]'

### 'Points' — Image points

*M*-by-2 matrix | points object

Image points, specified as the comma-separated pair consisting of 'Points' and an *M*-by-2 matrix of [x,y] coordinates or any points object.

### 'Orientation' — Orientation of the second camera relative to the first

3-by-3 matrix

Orientation of the second camera relative to the first, specified as the comma-separated pair consisting of 'Orientation' and a 3-by-3 matrix.

### 'Location' — Location of the second camera relative to the first

three-element vector

Location of the second camera relative to the first, specified as the comma-separated pair consisting of 'Location' and a three-element vector.

## Output Arguments

### vSet — View set object

viewSet object

viewSet object containing the added view specified by viewId.

## Examples

### Add View to View Set Object

Create an empty viewSet object.

```
vSet = viewSet;
```

Detect interest points in the image.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');
```

```
I = imread(fullfile(imageDir, 'image1.jpg'));  
points = detectSURFFeatures(rgb2gray(I));
```

Add the points to the object.

```
vSet = addView(vSet, 1, 'Points', points, 'Orientation', eye(3), 'Location', [0,0,0]);
```

**Introduced in R2016a**

# updateView

**Class:** viewSet

Modify an existing view in a view set object

## Syntax

```
vSet = updateView(vSet,viewId)
vSet = updateView(vSet,viewId,Name,Value)
```

## Description

`vSet = updateView(vSet,viewId)` modifies the view specified by `viewId` in the specified `viewSet` object, `vSet`.

`vSet = updateView(vSet,viewId,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**vSet** — View set object

`viewSet` object

`viewSet` object.

**viewId** — Camera pose view ID

integer

Camera pose view ID in the `viewSet` object, specified as an integer.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: `Location, '[0,0,0]'`

### **'Points' — Image points**

*M*-by-2 matrix | points object

Image points, specified as the comma-separated pair consisting of 'Points' and an *M*-by-2 matrix of  $[x,y]$  coordinates or any points object.

### **'Orientation' — Orientation of the second camera relative to the first camera**

3-by-3 matrix

Orientation of the second camera relative to the first camera, specified as the comma-separated pair consisting of 'Orientation' and a 3-by-3 matrix that represents the  $[x,y,z]$  orientation of the second camera.

### **'Location' — Location of the second camera relative to the first camera**

three-element vector

Location of the second camera relative to the first camera, specified as the comma-separated pair consisting of 'Location' and a three-element vector that represents the  $[x,y,z]$  location of the second camera in the first camera's coordinate system.

## Output Arguments

### **vSet — View set object**

viewSet object

viewSet object containing the modified view specified by `viewId`.

## Examples

### **Update View in View Set Object**

Create an empty `viewSet` object.

```
vSet = viewSet;
```



Detect interest points in the image.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I = imread(fullfile(imageDir,'image1.jpg'));  
points = detectSURFFeatures(rgb2gray(I));
```

Add the points to the object.

```
vSet = addView(vSet,1,'Points',points);
```

Update the view to specify the camera pose.

```
vSet = updateView(vSet, 1,'Orientation',eye(3),'Location',[0,0,0]);
```

**Introduced in R2016a**

# deleteView

**Class:** viewSet

Delete an existing view from view set object

## Syntax

```
vSet = deleteView(vSet,viewId)
```

## Description

`vSet = deleteView(vSet,viewId)` deletes an existing view or a set of views from the specified `viewSet` object,`vSet`.

## Input Arguments

**vSet – View set object**

`viewSet` object

A `viewSet` object.

**viewId – View IDs**

integer scalar | vector

View IDs, specified as an integer scalar for a single view, or as a vector of integers for a set of views.

## Output Arguments

**vSet – View set object**

`viewSet` object

`viewSet` object.

## Examples

### Delete a View from View Set Object

Create an empty viewSet object.

```
vSet = viewSet;
```

Detect interest points in the image.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I = imread(fullfile(imageDir,'image1.jpg'));  
points = detectSURFFeatures(rgb2gray(I));
```

Add a view.

```
vSet = addView(vSet,1,'Points',points);
```

Delete the view.

```
vSet = deleteView(vSet,1);
```

**Introduced in R2016a**

## hasView

**Class:** viewSet

Check if view exists

## Syntax

```
tf = hasView(vSet,viewId)
```

## Description

`tf = hasView(vSet,viewId)` returns 1 if the view specified by `viewID` exists and 0 if it does not exist.

## Input Arguments

**vSet** — View set object

viewSet object

viewSet object.

**viewId1** — View ID

integer

View ID in the viewSet object, specified as an integer.

## Output Arguments

**tf** — Validity of view connection

logical

Validity of view connection, returned as a logical 1 or 0.

## Examples

### Check If View Exists

Create an empty viewSet object.

```
vSet = viewSet;
```

Detect interest points in the image.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I = imread(fullfile(imageDir,'image1.jpg'));  
points = detectSURFFeatures(rgb2gray(I));
```

Add a new view.

```
vSet = addView(vSet, 1, 'Points', points);
```

Confirm that the view with ID 1 exists.

```
tf = hasView(vSet,1);
```

### Introduced in R2016a

# addConnection

**Class:** viewSet

Add a connection between two views

## Syntax

```
vSet = addConnection(vSet,viewId1,viewId2)  
vSet = addConnection(vSet,viewId1,viewId2,Name,Value,)
```

## Description

`vSet = addConnection(vSet,viewId1,viewId2)` adds a connection between two views in the specified viewSet object, `vSet`.

`vSet = addConnection(vSet,viewId1,viewId2,Name,Value,)` uses additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **vSet — View set object**

`viewSet` object

`viewSet` object.

### **viewId1 — View ID 1**

integer

View ID 1 in the `viewSet` object, specified as an integer.

### **viewId2 — View ID 2**

integer

View ID 2 in the `viewSet` object, specified as an integer.

## Name-Value Pair Arguments

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

Example: `'Location', '[0,0,0]'`

### 'Matches' — Indices of matched points between two views

*M*-by-2 matrix

Indices of matched points between two views, specified as the comma-separated pair consisting of 'Matches' and an *M*-by-2 matrix.

### 'Orientation' — Orientation of the second camera relative to the first camera

3-by-3 matrix

Orientation of the second camera relative to the first camera, specified as the comma-separated pair consisting of 'Orientation' and a 3-by-3 matrix that represents the  $[x,y,z]$  orientation of the second camera.

### 'Location' — Location of the second camera relative to the first camera

three-element vector

Location of the second camera relative to the first camera, specified as the comma-separated pair consisting of 'Location' and a three-element vector that represents the  $[x,y,z]$  location of the second camera in the first camera's coordinate system.

## Output Arguments

### **viewSet** — View set object

viewSet object

viewSet object.

## Examples

### Add Connection Between Two Views in View Set Object

Create an empty viewSet object.

```
vSet = viewSet;
```

Read a pair of images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I1 = rgb2gray(imread(fullfile(imageDir,'image1.jpg')));  
I2 = rgb2gray(imread(fullfile(imageDir,'image2.jpg')));
```

Detect interest points in the two images.

```
points1 = detectSURFFeatures(I1);  
points2 = detectSURFFeatures(I2);
```

Add the points to the `viewSet` object.

```
vSet = addView(vSet,1,'Points',points1);  
vSet = addView(vSet,2,'Points',points2);
```

Extract feature descriptors from both images.

```
features1 = extractFeatures(I1,points1);  
features2 = extractFeatures(I2,points2);
```

Match features and store the matches.

```
indexPairs = matchFeatures(features1,features2);
```

Add the connection between the two views.

```
vSet = addConnection(vSet,1,2,'Matches',indexPairs);
```

**Introduced in R2016a**



# updateConnection

**Class:** viewSet

Modify a connection between two views in a view set object

## Syntax

```
vSet = updateConnection(vSet,viewId1,viewId2)
vSet = updateConnection(vSet,viewId1,viewId2,Name,Value)
```

## Description

`vSet = updateConnection(vSet,viewId1,viewId2)` modifies a connection between two views in the specified view set object, `vSet`.

`vSet = updateConnection(vSet,viewId1,viewId2,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. Unspecified properties have default values.

## Input Arguments

### **vSet** — View set object

`viewSet` object

`viewSet` object.

### **viewId1** — View ID 1

integer

View ID 1 in the `viewSet` object, specified as an integer.

### **viewId2** — View ID 2

integer

View ID 2 in the `viewSet` object, specified as an integer.

### Name-Value Pair Arguments

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

Example: `'Location', '[0,0,0]'`

#### **'Matches'** — Indices of matched points between two views

*M*-by-2 matrix

Indices of matched points between two views, specified as the comma-separated pair consisting of `'Matches'` and an *M*-by-2 matrix.

#### **'Orientation'** — Orientation of the second camera relative to the first camera

3-by-3 matrix

Orientation of the second camera relative to the first camera, specified as the comma-separated pair consisting of `'Orientation'` and a 3-by-3 matrix that represents the  $[x,y,z]$  orientation of the second camera.

#### **'Location'** — Location of the second camera relative to the first camera

three-element vector

Location of the second camera relative to the first camera, specified as the comma-separated pair consisting of `'Location'` and a three-element vector that represents the  $[x,y,z]$  location of the second camera in the first camera's coordinate system.

### Output Arguments

#### **viewSet** — View set object

viewSet object

A viewSet object containing the modified connection.

### Examples

#### **Update Connection Between Two Views in View Set Object**

Create an empty viewSet object.

```
vSet = viewSet;
```

Read a pair of images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I1 = rgb2gray(imread(fullfile(imageDir,'image1.jpg')));  
I2 = rgb2gray(imread(fullfile(imageDir,'image2.jpg')));
```

Detect interest points in the two images.

```
points1 = detectSURFFeatures(I1);  
points2 = detectSURFFeatures(I2);
```

Add the points to the viewSet object.

```
vSet = addView(vSet, 1, 'Points', points1);  
vSet = addView(vSet, 2, 'Points', points2);
```

Extract feature descriptors.

```
features1 = extractFeatures(I1, points1);  
features2 = extractFeatures(I2, points2);
```

Match features and store the matches.

```
indexPairs = matchFeatures(features1, features2);  
vSet = addConnection(vSet, 1, 2, 'Matches', indexPairs);
```

Update the connection to store a relative pose between the views.

```
vSet = updateConnection(vSet, 1, 2, 'Orientation', eye(3), 'Location', [1 0 0]);
```

**Introduced in R2016a**

# deleteConnection

**Class:** viewSet

Delete a connection between two views from view set object

## Syntax

```
vSet = deleteConnection(vSet,viewId1,viewId2)
```

## Description

`vSet = deleteConnection(vSet,viewId1,viewId2)` deletes a connection between two views in the specified viewSet object, `vSet`.

## Input Arguments

**vSet — View set object**

viewSet object

viewSet object.

**viewId1 — View ID 1**

integer

View ID 1 in the viewSet object, specified as an integer.

**viewId2 — View ID 2**

integer

View ID 2 in the viewSet object, specified as an integer.

## Output Arguments

**vSet — View set object**

viewSet object

viewSet object.

## Examples

### Delete a Connection Between Two Views In View Set Object

Create an empty viewSet object.

```
vSet = viewSet;
```

Read a pair of images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
I1 = rgb2gray(imread(fullfile(imageDir,'image1.jpg')));  
I2 = rgb2gray(imread(fullfile(imageDir,'image2.jpg')));
```

Detect interest points in the two images.

```
points1 = detectSURFFeatures(I1);  
points2 = detectSURFFeatures(I2);
```

Add the points to the viewSet object.

```
vSet = addView(vSet,1,'Points',points1);  
vSet = addView(vSet,2,'Points',points2);
```

Extract feature descriptors.

```
features1 = extractFeatures(I1,points1);  
features2 = extractFeatures(I2,points2);
```

Match features and store the matches.

```
indexPairs = matchFeatures(features1, features2);  
vSet = addConnection(vSet,1,2,'Matches',indexPairs);
```

Delete the connection between the views.

```
vSet = deleteConnection(vSet,1,2);
```

**Introduced in R2016a**

## hasConnection

**Class:** viewSet

Check if a connection exists between two views

### Syntax

```
tf = hasConnection(vSet,viewId1,viewId2)
```

### Description

`tf = hasConnection(vSet,viewId1,viewId2)` returns true if both views exist and have a connection.

### Input Arguments

**vSet — View set object**

viewSet object

viewSet object.

**viewId1 — View ID 1**

integer

View ID 1 in the viewSet object, specified as an integer.

**viewId2 — View ID 2**

integer

View ID 2 in the viewSet object, specified as an integer.

### Output Arguments

**tf — Validity of view connection**

logical

Validity of view connection, returned as a logical 1 or 0.

## Examples

### Check Whether a Connection Exists Between Two Views

Create an empty `viewSet` object.

```
vSet = viewSet;
```

Add a pair of views.

```
vSet = addView(vSet,1);  
vSet = addView(vSet,2);
```

Add a connection.

```
vSet = addConnection(vSet,1,2);
```

Confirm that the connection exists.

```
tf = hasConnection(vSet,1,2);
```

**Introduced in R2016a**

# findTracks

**Class:** viewSet

Find matched points across multiple views

## Syntax

```
tracks = findTracks(vSet)
tracks = findTracks(vSet,viewIds)
```

## Description

`tracks = findTracks(vSet)` finds point tracks across multiple views.

`tracks = findTracks(vSet,viewIds)` finds point tracks across a subset of views.

## Input Arguments

**vSet** — View set object

viewSet object

viewSet object.

**viewIds** — Subset of views

vector of integers

Subset of views in the viewSet object, specified as a vector of integers.

## Output Arguments

**tracks** — Point track objects

array of pointTrack objects

Point track objects, returned as an array of pointTrack objects. Each track contains 2-D projections of the same 3-D world point.



## Examples

### Find Point Tracks Across Sequence of Images

Load images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','structureFromMotion');  
images = imageSet(imageDir);
```

Compute features for the first image.

```
I = rgb2gray(read(images, 1));  
pointsPrev = detectSURFFeatures(I);  
[featuresPrev,pointsPrev] = extractFeatures(I,pointsPrev);
```

Create a viewSet object.

```
vSet = viewSet;  
vSet = addView(vSet,1,'Points',pointsPrev);
```

Compute features and matches for the rest of the images.

```
for i = 2:images.Count  
    I = rgb2gray(read(images,i));  
    points = detectSURFFeatures(I);  
    [features, points] = extractFeatures(I,points);  
    vSet = addView(vSet,i,'Points',points);  
    pairsIdx = matchFeatures(featuresPrev,features);  
    vSet = addConnection(vSet,i-1,i,'Matches',pairsIdx);  
    featuresPrev = features;  
end
```

Find point tracks.

```
tracks = findTracks(vSet);
```

**Introduced in R2016a**

### poses

**Class:** viewSet

Returns camera poses associated to views

### Syntax

```
cameraPoses = poses(vSet)
cameraPoses = poses(vSet,viewIds)
```

### Description

`cameraPoses = poses(vSet)` returns the camera poses that correspond to the views contained in the input `viewSet`, `object`, `vSet`.

`cameraPoses = poses(vSet,viewIds)` returns the camera poses that correspond to a subset of views specified by the vector `viewIds`.

### Input Arguments

**vSet — View set object**

viewSet object

viewSet object.

**viewId — View IDs**

integer scalar | vector

View IDs, specified as an integer scalar for a single view, or as a vector of integers for a set of views.

### Output Arguments

**cameraPoses — Camera pose information**

three-column table

Camera pose information, returned as a three-column table. The table contains columns for `ViewId`, `Orientation`, and `Location`. The view IDs correspond to the IDs in the `viewSet` object. The orientations are specified as 3-by-3 rotation matrices and locations are specified as three-element vectors. You can pass the `cameraPoses` table to the `triangulateMultiview` and the `bundleAdjustment` functions.

## Examples

### Retrieve Camera Poses from View Set Object

Create an empty `viewSet` object.

```
vSet = viewSet;
```

Add views to the object.

```
vSet = addView(vSet,1,'Orientation',eye(3),'Location',[0,0,0]);  
vSet = addView(vSet,2,'Orientation',eye(3),'Location',[1,0,0]);
```

Retrieve the absolute camera poses.

```
camPoses = poses(vSet);
```

### See Also

`triangulateMultiview` | `bundleAdjustment`

**Introduced in R2016a**

# bagOfFeatures class

Bag of visual words object

## Syntax

```
bag = bagOfFeatures(imgSet)
bag = bagOfFeatures(imgSet, 'CustomExtractor', extractorFcn)
bag = bagOfFeatures(imgSet, Name, Value)
```

## Description

`bag = bagOfFeatures(imgSet)` returns a bag of features object. The `bag` output object is generated using samples from the `imgSet` input.

`bag = bagOfFeatures(imgSet, 'CustomExtractor', extractorFcn)` returns a bag of features that uses a custom feature extractor function to extract features from the output bag to learn its visual vocabulary. `extractorFcn` is a function handle to a custom feature extraction function.

`bag = bagOfFeatures(imgSet, Name, Value)` returns a `bagOfFeatures` object with additional properties specified by one or more `Name, Value` pair arguments.

This object supports parallel computing using multiple MATLAB workers. Enable parallel computing from the “Computer Vision System Toolbox Preferences” dialog box. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision System Toolbox**.

## Input Arguments

### **imgSet** — Image set

array of `imageSet` objects

Image set, specified as an array of `imageSet` objects. The `bagOfFeatures` extracts an equal number of strongest features from each image set contained in the `imgSet` input array.

*number of strongest features = min(number of features found in each set) x*

**StrongestFraction**

The object obtains the **StrongestFraction** value from the **'StrongestFeatures'** property.

### **extractorFcn** — Custom feature extractor function

function handle

Custom feature extractor function, specified the comma-separated pair consisting of **'CustomExtractor'** and a function handle. This custom function extracts features from the output **bagOfFeatures** object to learn the visual vocabulary of the object.

The function, **extractorFcn**, must be specified as a function handle for a file:

```
extractorFcn = @exampleBagOfFeaturesExtractor;
bag = bagOfFeatures(imgSets, 'CustomExtractor', extractorFcn)
where exampleBagOfFeaturesExtractor is a MATLAB function. For example:
```

```
function [features, featureMetrics] = exampleBagOfFeaturesExtractor(img)
...

```

The function must be on the path or in the current working directory.

For more details on the custom extractor function and its input and output requirements, see “Create a Custom Feature Extractor”.

You can open an example function file, and use it as a template by typing the following command at the MATLAB command-line:

```
edit('exampleBagOfFeaturesExtractor.m')
```

### **Name-Value Pair Arguments**

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

Example: **'VocabularySize'**, 500 sets the size of the visual words to include in the bag to 500.

### **'VocabularySize'** — Number of visual words

500 (default) | integer scalar

Number of visual words to include in the `bagOfFeatures` object, specified as the comma-separated pair consisting of `'VocabularySize'` and an integer scalar in the range  $[2, \text{inf}]$ . The `VocabularySize` value corresponds to  $K$  in the K-means clustering algorithm used to quantize features into the visual vocabulary.

**'StrongestFeatures' — Fraction of strongest features**

0.8 (default) |  $[0, 1]$

Fraction of strongest features, specified as the comma-separated pair consisting of `'StrongestFeatures'` and a value in the range  $[0, 1]$ . The value represents the fraction of strongest features to use from each image set contained in `imgSet` input.

**'Verbose' — Enable progress display to screen**

true (default) | false

Enable progress display to screen, specified as the comma-separated pair consisting of `'Verbose'` and the logical true or false.

**'PointSelection' — Selection method for picking point locations**

'Grid' (default) | 'Detector'

Selection method for picking point locations for SURF feature extraction, specified as the comma-separated pair consisting of `'PointSelection'` and the string `'Grid'` or `'Detector'`. There are two stages for feature extraction. First, you select a method for picking the point locations, (SURF `'Detector'` or `'Grid'`), with the `PointSelection` property. The second stage extracts the features. The feature extraction uses a SURF extractor for both point selection methods.

When you set `PointSelection` to `'Detector'`, the feature points are selected using a speeded up robust feature (SURF) detector. Otherwise, the points are picked on a predefined grid with spacing defined by `'GridStep'`. This property applies only when you are not specifying a custom extractor with the `CustomExtractor` property.

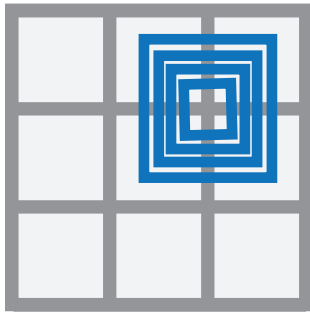
**'GridStep' — Grid step size**

$[8 \ 8]$  (default) | 1-by-2  $[x \ y]$  vector

Grid step size in pixels, specified as the comma-separated pair consisting of `'GridStep'` and an 1-by-2  $[x \ y]$  vector. This property applies only when you set `PointSelection` to `'Grid'` and you are not specifying a custom extractor with the `CustomExtractor` property. The steps in the  $x$  and  $y$  directions define the spacing of a uniform grid. Intersections of the grid lines define locations for feature extraction.

**'BlockWidth' — Patch size to extract upright SURF descriptor**[32 64 96 128] (default) | 1-by- $N$  vector

Patch size to extract upright SURF descriptor, specified as the comma-separated pair consisting of 'BlockWidth' and a 1-by- $N$  vector of  $N$  block widths. This property applies only when you are not specifying a custom extractor with the `CustomExtractor` property. Each element of the vector corresponds to the size of a square block from which the function extracts upright SURF descriptors. Use multiple square sizes to extract multiscale features. All the square specified are used for each extraction points on the grid. This property only applies when you set `PointSelection` to 'Grid'. The block width corresponds to the scale of the feature. The minimum `BlockWidth` is 32 pixels.

**'Upright' — Orientation of SURF feature vector**

true (default) | logical scalar

Orientation of SURF feature vector, specified as the comma-separated pair consisting of 'Upright' and a logical scalar. This property applies only when you are not specifying a custom extractor with the `CustomExtractor` property. Set this property to `true` when you do not need to estimate the orientation of the SURF feature vectors. Set it to `false` when you need the image descriptors to capture rotation information.

## Properties

**CustomExtractor — Custom extraction function**

function handle

Custom feature extractor function, specified as a handle to a function. The custom feature extractor function extracts features used to learn the visual vocabulary for

`bagOfFeatures`. You must specify `'CustomExtractor'` and the function handle, `extractorFcn`, to a custom feature extraction function.

The function, `extractorFcn`, must be specified as a function handle for a file:

```
extractorFcn = @exampleBagOfFeaturesExtractor;  
bag = bagOfFeatures(imgSets, 'CustomExtractor', extractorFcn)  
where exampleBagOfFeaturesExtractor is a MATLAB function such as:  
  
function [features, featureMetrics] = exampleBagOfFeaturesExtractor(img)  
...  
The function must be on the path or in the current working directory.
```

For more details on the custom extractor function and its input and output requirements, see “Create a Custom Feature Extractor”. You can open an example function file, and use it as a template by typing the following command at the MATLAB command-line:

```
edit('exampleBagOfFeaturesExtractor.m')
```

### **VocabularySize** — Number of visual words

500 (default) | integer scalar

Number of visual words to include in the `bagOfFeatures` object, specified as the comma-separated pair consisting of `'VocabularySize'` and an integer scalar in the range `[2, inf)`. The `VocabularySize` value corresponds to  $K$  in the “k-Means Clustering” algorithm used to quantize features into the visual vocabulary.

### **StrongestFeatures** — Fraction of strongest features

0.8 (default) | `[0, 1]`

Fraction of strongest features, specified as the comma-separated pair consisting of `'StrongestFeatures'` and a value in the range `[0, 1]`. The value represents the strongest of strongest features to use from each image set contained in `imgSet` input.

### **PointSelection** — Point selection method

`'Grid'` (default) | `'Detector'`

Selection method for picking point locations for SURF feature extraction, specified as the comma-separated pair consisting of `'PointSelection'` and the string `'Grid'` or `'Detector'`. There are two stages for feature extraction. First, you select a method for picking the point locations, (SURF `'Detector'` or `'Grid'`), with the `PointSelection`



property. The second stage extracts the features. The feature extraction uses a SURF extractor for both point selection methods.

When you set `PointSelection` to `'Detector'`, the feature points are selected using a speeded up robust feature (SURF) detector. Otherwise, the points are picked on a predefined grid with spacing defined by `'GridStep'`. This property applies only when you are not specifying a custom extractor with the `CustomExtractor` property.

### **GridStep — Grid step size**

[ 8 8 ] (default) | 1-by-2 [x y] vector

Grid step size in pixels, specified as the comma-separated pair consisting of `'GridStep'` and an 1-by-2 [x y] vector. This property applies only when you set `PointSelection` to `'Grid'`. The steps in the *x* and *y* directions define the spacing of a uniform grid. Intersections of the grid lines define locations for feature extraction.

### **BlockWidth — Patch size to extract SURF descriptor**

[ 32 64 96 128 ] (default) | 1-by-*N* vector

Patch size to extract SURF descriptor, specified as the comma-separated pair consisting of `'BlockWidth'` and a 1-by-*N* vector of *N* block widths. Each element of the vector corresponds to the size of a square block from which the function extracts SURF descriptors. Use multiple square sizes to extract multiscale features. All the square specified are used for each extraction points on the grid. This property only applies when you set `PointSelection` to `'Grid'`. The block width corresponds to the scale of the feature. The minimum `BlockWidth` is 32 pixels.

### **Upright — Orientation of SURF feature vector**

true (default) | logical scalar

Orientation of SURF feature vector, specified as the comma-separated pair consisting of `'Upright'` and a logical scalar. This property applies only when you are not specifying a custom extractor with the `CustomExtractor` property. Set this property to `true` when you do not need to estimate the orientation of the SURF feature vectors. Set it to `false` when you need the image descriptors to capture rotation information.

## **Methods**

encode

Create histogram of visual word occurrences

# Examples

## Create a Bag of Visual Words

Load two image sets.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');  
imgSets = imageSet(setDir, 'recursive');
```

Pick the first two images from each image set to create training sets.

```
trainingSets = partition(imgSets, 2);
```

Create the bag of features. This process can take a few minutes.

```
bag = bagOfFeatures(trainingSets, 'Verbose', false);
```

Compute histogram of visual word occurrences for one of the images. Store the histogram as feature vector.

```
img = read(imgSets(1), 1);  
featureVector = encode(bag, img);
```

## Create a Bag of Features with a Custom Feature Extractor

Load an image set.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');  
imgSets = imageSet(setDir, 'recursive');
```

Specify a custom feature extractor.

```
extractor = @exampleBagOfFeaturesExtractor;  
bag = bagOfFeatures(imgSets, 'CustomExtractor', extractor, 'Verbose', false)
```

bag =

bagOfFeatures with properties:

```
CustomExtractor: @exampleBagOfFeaturesExtractor  
VocabularySize: 500  
StrongestFeatures: 0.8000
```

- “Image Category Classification Using Bag of Features”

## References

Csurka, G., C. R. Dance, L. Fan, J. Willamowski, and C. Bray. *Visual Categorization with Bags of Keypoints*. Workshop on Statistical Learning in Computer Vision. ECCV 1 (1–22), 1–2.

## See Also

`imageSet` | `imageCategoryClassifier` | `trainImageCategoryClassifier`

## More About

- “Image Classification with Bag of Visual Words”
- “Create a Custom Feature Extractor”

**Introduced in R2014b**

# encode

**Class:** `bagOfFeatures`

Create histogram of visual word occurrences

## Syntax

```
featureVector = encode(bag,I)
[featureVector,words] = encode(bag,I)

featureVector= encode(bag,imgSet)
[featureVector,words] = encode(bag,imgSet)

[ ___ ] = encode( ___ ,Name,Value)
```

## Description

`featureVector = encode(bag,I)` returns a feature vector that represents a histogram of visual word occurrences contained in the input image, `I`. The input `bag` contains the `bagOfFeatures` object.

`[featureVector,words] = encode(bag,I)` optionally returns the visual words as a `visualWords` object. The `visualWords` object stores the visual words that occur in `I` and stores the locations of those words.

`featureVector= encode(bag,imgSet)` returns a feature vector that represents a histogram of visual word occurrences contained in `imgSet`. The input `bag` contains the `bagOfFeatures` object.

`[featureVector,words] = encode(bag,imgSet)` optionally returns an array of `visualWords` objects for each image in `imgSet`. The `visualWords` object stores the visual words that occur in `I` and stores the locations of those words.

`[ ___ ] = encode( ___ ,Name,Value)` returns a feature vector with optional input properties specified by one or more `Name,Value` pair arguments.

This method supports parallel computing using multiple MATLAB workers. Enable parallel computing from the “Computer Vision System Toolbox Preferences” dialog

box. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision System Toolbox**.

## Input Arguments

### **bag** — Bag of features

bagOfFeatures object

Bag of features, specified as a bagOfFeatures object.

### **I** — Input image

grayscale image | truecolor image

Input image, I, specified as a grayscale or truecolor image.

### **imgSet** — Image set

imageSet object | array of imageSet objects

Image set, specified as either an imageSet object or an array of imageSet objects.

## 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**.

### **'Normalization'** — Type of normalization

'L2' (default) | 'none'

Type of normalization applied to the feature vector, specified as the string 'L2' or 'none'.

### **'SparseOutput'** — Output sparsity

false (default) | true

Output sparsity, specified as **true** or **false**. Set this property to **true** to return the visual word histograms in a sparse matrix. Setting this property to **true** reduces

memory consumption for large visual vocabularies where the visual word histograms contain many zero elements.

### 'Verbose' — Enable progress display to screen

true (default) | false

Enable progress display to screen, specified as the comma-separated pair consisting of 'Verbose' and the logical true or false.

## Output Arguments

### featureVector — Histogram of visual word occurrences

1-by-bag.VocabularySize |  $M$ -by-bag.VocabularySize

Histogram of visual word occurrences, specified as either a 1-by-bag.VocabularySize or an  $M$ -by-bag.VocabularySize vector.

- When you specify a single input image set, `encode` returns `featureVector` as a 1-by-bag.VocabularySize vector.
- When you specify an array of image sets for the input, `encode` returns `featureVector` as an  $M$ -by-bag.VocabularySize vector. The number  $M$ , calculated as `sum([imgSet.Count])`, represents the total number of images in the input image set, `imgSet`.

### words — Visual words object

visualWords object

Visual words object, returned as a visual words object or an array of visual words objects. The `visualWords` object stores the visual words that occur in the images and stores the locations of those words.

## Examples

### Encode Image from Image Set into Feature Vector

Load two image sets.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');  
imgSets = imageSet(setDir, 'recursive');
```

**Pick the first two images from each set.**

```
trainingSets = partition(imgSets, 2);
```

**Create bag of features.**

```
bag = bagOfFeatures(trainingSets);
```

```
Creating Bag-Of-Features from 2 image sets.
```

```
-----
```

```
* Image set 1: books.
```

```
* Image set 2: cups.
```

```
* Selecting feature point locations using the Grid method.
```

```
* Extracting SURF features from the selected feature point locations.
```

```
** The GridStep is [8 8] and the BlockWidth is [32 64 96 128].
```

```
* Extracting features from 2 images in image set 1...done. Extracted 38400 features.
```

```
* Extracting features from 2 images in image set 2...done. Extracted 38400 features.
```

```
* Keeping 80 percent of the strongest features from each image set.
```

```
* Using K-Means clustering to create a 500 word visual vocabulary.
```

```
* Number of features          : 61440
```

```
* Number of clusters (K)     : 500
```

```
* Initializing cluster centers...100.00%.
```

```
* Clustering...completed 20/100 iterations (~0.22 seconds/iteration)...converged in 20
```

```
* Finished creating Bag-Of-Features
```

**Encode one of the images into a feature vector.**

```
img = read(trainingSets(1), 1);
```

```
featureVector = encode(bag, img);
```

# imageCategoryClassifier class

Predict image category

## Syntax

```
imageCategoryClassifier
```

## Description

`imageCategoryClassifier` is returned by the `trainImageCategoryClassifier` function. It contains a linear support vector machine (SVM) classifier trained to recognize an image category.

You must have a Statistics and Machine Learning Toolbox™ license to use this classifier.

This classifier supports parallel computing using multiple MATLAB workers. Enable parallel computing using the “Computer Vision System Toolbox Preferences” dialog. To open the Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Select **Computer Vision System Toolbox**.

## Properties

### **Labels** — Category labels

cell array

Category labels, specified as a cell array.

### **NumCategories** — Number of trained categories

integer

Number of trained categories, stored as an integer value.

## Methods

`predict`

Predict image category



evaluate

Evaluate image classifier on collection of image sets

## Examples

### Train, Evaluate, and Apply Image Category Classifier

Load two image categories.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');
imgSets = imageSet(setDir, 'recursive');
```

Separate the two sets into training and test data. Pick 30% of images from each set for the training data and the remainder 70% for the test data.

```
[trainingSets, testSets] = partition(imgSets, 0.3, 'randomize');
```

Create bag of visual words.

```
bag = bagOfFeatures(trainingSets, 'Verbose', false);
```

Train a classifier with the training sets.

```
categoryClassifier = trainImageCategoryClassifier(trainingSets, bag);
```

Training an image category classifier for 2 categories.

```
-----
* Category 1: books
* Category 2: cups

* Encoding features for category 1...done.
* Encoding features for category 2...done.

* Finished training the category classifier. Use evaluate to test the classifier on a t
```

Evaluate the classifier using test images. Display the confusion matrix.

```
confMatrix = evaluate(categoryClassifier, testSets)
```

Evaluating image category classifier for 2 categories.

```
* Category 1: books
* Category 2: cups

* Evaluating 4 images from category 1...done.
* Evaluating 4 images from category 2...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:
```

	PREDICTED	
KNOWN	books	cups
books	0.75	0.25
cups	0.25	0.75

```
* Average Accuracy is 0.75.
```

```
confMatrix =
```

```
    0.7500    0.2500
    0.2500    0.7500
```

Find the average accuracy of the classification.

```
mean(diag(confMatrix))
```

```
ans =
```

```
    0.7500
```

Apply the newly trained classifier to categorize new images.

```
img = imread(fullfile(setDir, 'cups', 'bigMug.jpg'));
[labelIdx, score] = predict(categoryClassifier, img);
```

Display the classification label.

```
categoryClassifier.Labels(labelIdx)
```

```
ans =  
    'cups'
```

- “Image Category Classification Using Bag of Features”

## References

Csurka, G., C. R. Dance, L. Fan, J. Willamowski, and C. Bray *Visual Categorization with Bag of Keypoints*, Workshop on Statistical Learning in Computer Vision, ECCV 1 (1-22), 1-2.

## See Also

`imageSet` | `bagOfFeatures` | `fitcecoc` | `trainImageCategoryClassifier`

## More About

- “Image Classification with Bag of Visual Words”

**Introduced in R2014b**

# predict

**Class:** imageCategoryClassifier

Predict image category

## Syntax

```
[labelIdx,score] = predict(categoryClassifier,I)
[labelIdx,score] = predict(categoryClassifier,imgSet)
[labelIdx,score] = predict(__,Name,Value)
```

## Description

`[labelIdx,score] = predict(categoryClassifier,I)` returns the predicted label index and score for the input image.

This supports parallel computing using multiple MATLAB workers. Enable parallel computing using the “Computer Vision System Toolbox Preferences” dialog. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Select **Computer Vision System Toolbox**.

`[labelIdx,score] = predict(categoryClassifier,imgSet)` returns the predicted label index and score for the images specified in `imgSet`.

`[labelIdx,score] = predict(__,Name,Value)` returns the predicted label index and score with optional input properties specified by one or more `Name,Value` pair arguments. You can use any of the input arguments from previous syntaxes.

## Input Arguments

### **I** – Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input image, specified as either an *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-D grayscale image.

**categoryClassifier** — Image category classifier

imageCategoryClassifier object

Image category classifier, specified as an imageCategoryClassifier object.

**imgSet** — Image set

imageSet object

Image set, specified as an imageSet 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**.

Example: 'Verbose', true sets the value of 'Verbose' to the logical true.

**'Verbose'** — Enable progress display to screen

true (default) | false

Enable progress display to screen, specified as the comma-separated pair consisting of 'Verbose' and the logical true or false.

## Output Arguments

**labelIdx** — Predicted label index $M$ -by-1 vector | scalar

Predicted label index, returned as either an  $M$ -by-1 vector for  $M$  images or a scalar value for a single image. The **labelIdx** output value corresponds to the index of an image set used to train the bag of features. The prediction index corresponds to the class with the lowest average binary loss of the ECOC SVM classifier.

**score** — Prediction score1-by- $N$  vector |  $M$ -by- $N$  matrix

Prediction score, specified as a 1-by- $N$  vector or an  $M$ -by- $N$  matrix.  $N$  represents the number of classes.  $M$  represents the number of images in the imageSet input object,

`imgSet`. The score provides a negated average binary loss per class. Each class is a support vector machine (SVM) multiclass classifier that uses the error-correcting output codes (ECOC) approach.

## Examples

### Predict Category for Image

**Load two image category sets.**

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');
imgSets = imageSet(setDir, 'recursive');
```

**Separate the two sets into training and test data. Pick 30% of images from each set for the training data and the remainder 70% for the test data.**

```
[trainingSets, testSets] = partition(imgSets, 0.3, 'randomize');
```

**Create a bag of visual words.**

```
bag = bagOfFeatures(trainingSets);
```

Creating Bag-Of-Features from 2 image sets.

```
-----
* Image set 1: books.
* Image set 2: cups.

* Selecting feature point locations using the Grid method.
* Extracting SURF features from the selected feature point locations.
** The GridStep is [8 8] and the BlockWidth is [32 64 96 128].

* Extracting features from 2 images in image set 1...done. Extracted 38400 features.
* Extracting features from 2 images in image set 2...done. Extracted 38400 features.

* Keeping 80 percent of the strongest features from each image set.

* Using K-Means clustering to create a 500 word visual vocabulary.
* Number of features           : 61440
* Number of clusters (K)      : 500

* Initializing cluster centers...100.00%.
* Clustering...completed 28/100 iterations (~0.23 seconds/iteration)...converged in 28
```

```
* Finished creating Bag-Of-Features
```

### **Train a classifier.**

```
categoryClassifier = trainImageCategoryClassifier(trainingSets, bag);
```

```
Training an image category classifier for 2 categories.
```

```
-----  
* Category 1: books
```

```
* Category 2: cups
```

```
* Encoding features for category 1...done.
```

```
* Encoding features for category 2...done.
```

```
* Finished training the category classifier. Use evaluate to test the classifier on a t
```

### **Predict the category label for an image in one of the test sets.**

```
[labelIdx, score] = predict(categoryClassifier, testSets(1).read(1))  
categoryClassifier.Labels(labelIdx)
```

```
labelIdx =
```

```
1
```

```
score =
```

```
-0.3459 -0.6541
```

```
ans =
```

```
'books'
```

# evaluate

**Class:** `imageCategoryClassifier`

Evaluate image classifier on collection of image sets

## Syntax

```
confMat = evaluate(classifier,imgSet)
[confMat,knownLabelIdx,predictedLabelIdx,score] = evaluate(
classifier,imgSet)
[ ___ ] = evaluate( ___ ,Name,Value)
```

## Description

`confMat = evaluate(classifier,imgSet)` returns a normalized confusion matrix, `confMat`, that evaluates the input image set using the input classifier.

`[confMat,knownLabelIdx,predictedLabelIdx,score] = evaluate(classifier,imgSet)` additionally returns the corresponding label indexes and score.

`[ ___ ] = evaluate( ___ ,Name,Value)` uses additional input properties specified by one or more `Name,Value` pair arguments. You can use any of the arguments from previous syntaxes.

## Input Arguments

### **I** — Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input image, specified as either an *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-D grayscale image.

### **classifier** — Image category classifier

`imageCategoryClassifier` object

Image category classifier, specified as an `imageCategoryClassifier` object.



**imgSet** — Image set*imageSet* object

Image set, specified as an *imageSet* 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*.

Example: 'Verbose', true sets the value of 'Verbose' to the logical true.

**'Verbose'** — Enable progress display to screen*true* (default) | *false*

Enable progress display to screen, specified as the comma-separated pair consisting of 'Verbose' and the logical *true* or *false*.

## Output Arguments

**knownLabelIdx** — Label index for image set*M*-by-1 vector | scalar

Label index for image set, returned as an *M*-by-1 vector for *M* images. The *knownLabelIdx* output value corresponds to the index of an image set used to train the bag of features.

**predictedLabelIdx** — Predicted label index*M*-by-1 vector

Predicted label index, returned as an *M*-by-1 vector for *M* images. The *predictedLabelIdx* output value corresponds to the index of an image set used to train the bag of features. The predicted index corresponds to the class with the largest value in the score output.

**score** — Prediction score*M*-by-*N* matrix

Prediction score, specified as an  $M$ -by- $N$  matrix.  $N$  represents the number of classes.  $M$  represents the number of images in the `imageSet` input object, `imgSet`. The score provides a negated average binary loss per class. Each class is a support vector machine (SVM) multiclass classifier that uses the error-correcting output codes (ECOC) approach.

## Examples

### Train, Evaluate, and Apply Image Category Classifier

Load two image categories.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');  
imgSets = imageSet(setDir, 'recursive');
```

Separate the two sets into training and test data. Pick 30% of images from each set for the training data and the remainder 70% for the test data.

```
[trainingSets, testSets] = partition(imgSets, 0.3, 'randomize');
```

Create bag of visual words.

```
bag = bagOfFeatures(trainingSets, 'Verbose', false);
```

Train a classifier with the training sets.

```
categoryClassifier = trainImageCategoryClassifier(trainingSets, bag);
```

Training an image category classifier for 2 categories.

```
-----  
* Category 1: books  
* Category 2: cups  
  
* Encoding features for category 1...done.  
* Encoding features for category 2...done.  
  
* Finished training the category classifier. Use evaluate to test the classifier on a t
```

Evaluate the classifier using test images. Display the confusion matrix.

```
confMatrix = evaluate(categoryClassifier, testSets)
```

Evaluating image category classifier for 2 categories.

```

-----
* Category 1: books
* Category 2: cups

* Evaluating 4 images from category 1...done.
* Evaluating 4 images from category 2...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:

```

KNOWN	PREDICTED	
	books	cups
books	0.75	0.25
cups	0.25	0.75

```

* Average Accuracy is 0.75.

```

```

confMatrix =

```

```

    0.7500    0.2500
    0.2500    0.7500

```

Find the average accuracy of the classification.

```

mean(diag(confMatrix))

```

```

ans =

```

```

    0.7500

```

Apply the newly trained classifier to categorize new images.

```

img = imread(fullfile(setDir, 'cups', 'bigMug.jpg'));
[labelIdx, score] = predict(categoryClassifier, img);

```

Display the classification label.

```
categoryClassifier.Labels(labelIdx)
```

```
ans =
```

```
 'cups'
```

## intrinsicsEstimationErrors class

Object for storing standard errors of estimated camera intrinsics and distortion coefficients

### Syntax

```
intrinsicsEstimationErrors
```

### Description

`intrinsicsEstimationErrors` contains the standard errors of estimated camera intrinsics and distortion coefficients. You can access the intrinsics and distortion standard errors using the object properties.

### Properties

#### **SkewError**

Standard error of camera axes skew estimate

#### **FocalLengthError**

Standard error of focal length estimate

#### **PrincipalPointError**

Standard error of principal point estimate

#### **RadialDistortionError**

Standard error of radial distortion estimate

#### **TangentialDistortionError**

Standard error of tangential distortion estimate

### **See Also**

[cameraCalibrationErrors](#) | [stereoCalibrationErrors](#) | [intrinsicEstimationErrors](#) | [extrinsicEstimationErrors](#) | [Camera Calibrator](#) | [Stereo Camera Calibrator](#)

### **More About**

- “Single Camera Calibration App”

**Introduced in R2013b**

# extrinsicsEstimationErrors class

Object for storing standard errors of estimated camera extrinsics

## Syntax

```
extrinsicsEstimationErrors
```

## Description

`extrinsicsEstimationErrors` contains the standard errors of estimated camera extrinsics. You can access the extrinsics standard errors using the object properties.

## Properties

### **RotationVectorsError**

Standard error of camera rotations estimate

### **TranslationVectorsError**

Standard error of camera translations estimate

## See Also

[cameraCalibrationErrors](#) | [stereoCalibrationErrors](#) | [intrinsicsEstimationErrors](#) | [Camera Calibrator](#) | [Stereo Camera Calibrator](#)

## More About

- “Single Camera Calibration App”

**Introduced in R2013b**

## BRISKPoints class

Object for storing BRISK interest points

### Description

This object provides the ability to pass data between the `detectBRISKFeatures` and `extractFeatures` functions. You can also use it to manipulate and plot the data returned by these functions. You can use the object to fill the points interactively in situations where you might want to mix a non-BRISK interest point detector with a BRISK descriptor.

### Tips

Although `BRISKPoints` can hold many points, it is a scalar object. Therefore, `NUMEL(BRISKPoints)` always returns 1. This value can differ from `LENGTH(BRISKPoints)`, which returns the true number of points held by the object.

### Construction

`points = BRISKPoints(Location)` constructs a `BRISKPoints` object from an  $M$ -by-2 array of `[x y]` point coordinates, `Location`.

`points = BRISKPoints(Location,Name,Value)` constructs a `BRISKPoints` object with optional input properties specified by one or more `Name,Value` pair arguments. You can specify each additional property as a scalar or a vector whose length matches the number of coordinates in `Location`.

Code Generation Support
Compile-time constant inputs: No restriction.
Supports MATLAB Function block: No
To index locations with this object, use the syntax: <code>points.Location(idx,:)</code> , for <code>points</code> object. See <code>visionRecoverformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.



**Code Generation Support**

“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### Location

Point locations, specified as an  $M$ -by-2 array of [x y] point coordinates.

## Properties

### Scale

Scale at which the feature is detected, specified as a value greater than or equal to 1.6.

**Default:** 12.0

### Metric

Strength of detected feature, specified as a numeric value. The BRISK algorithm uses a determinant of an approximated Hessian.

**Default:** 0.0

### Orientation

Orientation of the detected feature, specified as an angle, in radians. The angle is measured counterclockwise from the X-axis with the origin specified by the **Location** property. Do not set this property manually. Use the call to **extractFeatures** to fill in this value. The **extractFeatures** function modifies the default value of 0.0. Using BRISK interest points to extract a non-BRISK descriptor, (e.g. SURF, FREAK, MSER, etc.), can alter **Orientation** values. The **Orientation** is mainly useful for visualization purposes.

**Default:** 0.0

### Count

Number of points held by the BRISK object, specified as a numeric value.

Default: 0

## Methods

<code>isempty</code>	Returns true for empty object
<code>length</code>	Number of stored points
<code>plot</code>	Plot BRISK points
<code>selectStrongest</code>	Return points with strongest metrics
<code>selectUniform</code>	Return a uniformly distributed subset of feature points
<code>size</code>	Size of the BRISKPoints object

## Examples

### Detect BRISK Features in an Image

Read an image and detect the BRISK interest points.

```
I = imread('cameraman.tif');  
points = detectBRISKFeatures(I);
```

Select and plot the 10 strongest interest points.

```
strongest = points.selectStrongest(10);  
imshow(I); hold on;  
plot(strongest);
```



**Display the [x y] coordinates.**

```
strongest.Location
```

```
ans =
```

```
136.4033    55.5000  
155.6964    83.7577  
197.0000   233.0000  
117.3680    92.3680  
147.0000   162.0000  
104.0000   229.0000  
129.7972    68.2028  
154.8790    77.0000  
118.0269   174.0269  
131.0000    91.1675
```

## References

- [1] Leutenegger, S., M. Chli, and R. Siegwart. *BRISK: Binary Robust Invariant Scalable Keypoints*, Proceedings of the IEEE International Conference on Computer Vision (ICCV) 2011.

## See Also

MSERRegions | SURFPoints | cornerPoints | detectBRISKFeatures | detectFASTFeatures | detectHarrisFeatures | detectMinEigenFeatures | detectMSERFeatures | detectSURFFeatures | extractFeatures | matchFeatures

**Introduced in R2014a**

# isempty

**Class:** BRISKPoints

Returns true for empty object

## Syntax

`isempty(BRISKPointsObj)`

## Description

`isempty(BRISKPointsObj)` returns a true value, if the `BRISKPointsObj` object is empty.

## **length**

**Class:** BRISKPoints

Number of stored points

## **Syntax**

`length(BRISKPointsObj)`

## **Description**

`length(BRISKPointsObj)` returns the number of stored points in the BRISKPointsObj object.

# plot

**Class:** BRISKPoints

Plot BRISK points

## Syntax

```
briskPoints/plot  
briskPoints/plot(Axes_HANDLE,...)  
briskPoints/plot(Axes_HANDLE,Name,Value)
```

## Description

`briskPoints/plot` plots BRISK points in the current axis.

`briskPoints/plot(Axes_HANDLE,...)` plot points using axes with the handle `AXES_HANDLE`.

`briskPoints/plot(Axes_HANDLE,Name,Value)` Additional control for the `plot` method requires specification of parameters and corresponding values. An additional option is specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **AXES\_HANDLE**

Handle for plot method to use for display. You can set the handle using `gca`.

## 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`.

### **showScale**

Display proportional circle around feature. Set this value to `true` or `false`. When you set this value to `true`, the object draws a circle proportional to the scale of the detected feature, with the feature point located at its center. When you set this value to `false`, the object turns the display of the circle off.

The algorithm represents the scale of the feature with a circle of  $6 * \text{Scale}$  radius. The BRISK algorithm uses this equivalent size of circular area to compute the orientation of the feature.

**Default:** `true`

### **showOrientation**

Display a line corresponding to feature point orientation. Set this value to `true` or `false`. When you set this value to `true`, the object draws a line corresponding to the point's orientation. The object draws the line from the feature point location to the edge of the circle, indicating the scale.

**Default:** `false`

## **Examples**

```
% Extract BRISK features
I = imread('cameraman.tif');
points = detectBRISKFeatures(I);

% Display
imshow(I); hold on;
plot(points);
```



# selectStrongest

**Class:** BRISKPoints

Return points with strongest metrics

## Syntax

```
strongestPoints = BRISKPoints.selectStrongest(N)
```

## Description

`strongestPoints = BRISKPoints.selectStrongest(N)` returns N number of points with strongest metrics.

## Examples

```
% Create object holding 50 points
points = BRISKPoints(ones(50,2), 'Metric', 1:50);

% Keep 2 strongest features
points = points.selectStrongest(2)
```

# selectUniform

**Class:** BRISKPoints

Return a uniformly distributed subset of feature points

## Syntax

```
pointsOut = selectUniform(pointsIn,N,imageSize)
```

## Description

`pointsOut = selectUniform(pointsIn,N,imageSize)` returns N uniformly distributed points with the strongest metrics.

## Input Arguments

### **pointsIn** — BRISK points

BRISKPoints object

SURF points, returned as a BRISKPoints object. This object contains information about BRISK features detected in a grayscale image. You can use the `detectBRISKFeatures` to obtain BRISK points.

### **N** — Number of points

integer

Number of points, specified as an integer.

### **imageSize** — Size of image

2-element vector | 3-element vector

Size of image, specified as a 2- or 3-element vector.

## Examples

### Select A Uniformly Distributed Subset of Features From an Image

Load an image.

```
im = imread('yellowstone_left.png');
```

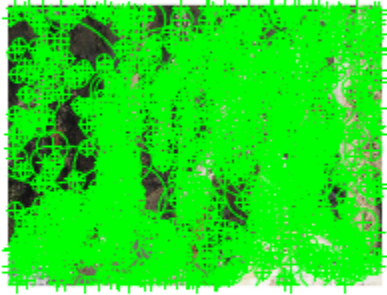
Detect many corners by reducing the quality threshold.

```
points1 = detectBRISKFeatures(rgb2gray(im), 'MinQuality', 0.05);
```

Plot image with detected corners.

```
subplot(1,2,1);  
imshow(im);  
hold on  
plot(points1);  
hold off  
title('Original points');
```

**Original points**



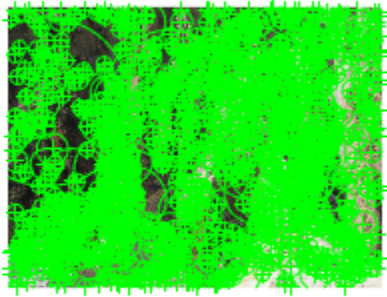
Select a uniformly distributed subset of points.

```
numPoints = 100;  
points2 = selectUniform(points1,numPoints,size(im));
```

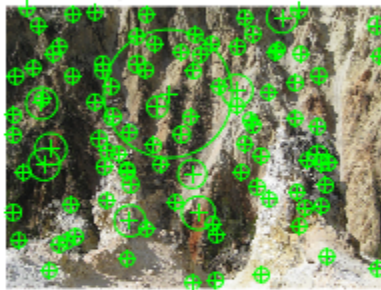
Plot images showing original and subset of points.

```
subplot(1, 2, 2);  
imshow(im);  
hold on  
plot(points2);  
hold off  
title('Uniformly distributed points');
```

**Original points**



**Uniformly distributed points**



### size

**Class:** BRISKPoints

Size of the BRISKPoints object

### Syntax

`size(BRISKPointsObj)`

### Description

`size(BRISKPointsObj)` returns the size of the BRISKPointsObj object.

- `sz = size(v)` returns the vector `[length(v), 1]`
- `sz = size(v, 1)` returns the length of  $v$
- `sz = size(v, N)`, for  $N \geq 2$ , returns 1
- `[M, N] = size(v)` returns `length(v)` for  $M$  and 1 for  $N$

# imageSet class

Define collection of images

## Syntax

```
imgSet = imageSet(imageLocation)
imgSetVector = imageSet(imgFolder, 'recursive')
```

## Construction

`imgSet = imageSet(imageLocation)` returns an object for storing an image data set or a collection of image data sets. You can use this object to manage your image data. The object contains image descriptions, locations of images, and the number of images in your collection.

`imgSetVector = imageSet(imgFolder, 'recursive')` returns a vector of image sets found through a recursive search starting from `imgFolder`. The `imgSetVector` output is a 1-by-*NumFolders* vector, where *NumFolders* is the number of folders that contain at least one image.

## Input Arguments

### **imageLocation** — Image file location

string | cell array

Image file location, specified as a string or a cell array. The string must specify the folder name that contains the images. The cell array must contain image locations.

### **Image file location**

`{'imagePath1','imagePath2', ..., 'imagePathX'}`, where each *imagePath* represents the path to an each image.

### **imgFolder** — Start recursive image search folder

string

Start recursive image search folder, specified as a string. The function searches the folder structure recursively, starting from `imgFolder`.

# Properties

### **Description — Information about the image set**

string

Information about the image set, specified as a string. When you create an image set by recursively searching folders or by specifying a single folder location, the **Description** property is set to the folder name. When you specify individual image files, the **Description** property is not set. You can set the property manually.

Data Types: char

### **Count — Number of images in image set**

integer

Number of images in image set.

Data Types: double | single

### **ImageLocation — Image locations**

cell array of strings

Image locations, given as a cell array of strings.

Data Types: cell

# Methods

partition	Divide image set into subsets
read	Read image at specified index
select	Select subset of images from image set

# Examples

### **Create an Image Set From a Folder of Images**

Read the folder of images.

```
imgFolder = fullfile(toolboxdir('vision'),'visiondata','stopSignImages');
```



```
imgSet = imageSet(imgFolder);
```

**Display the first image in the image set collection.**

```
imshow(read(imgSet,1));
```



**Create an Array of Image Sets from Multiple Folders**

**Identify the path to the image sets.**

```
imgFolder = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'imageSets');
```

**Recursively scan the entire image set folder.**

```
imgSets = imageSet(imgFolder, 'recursive')
```

```
imgSets =
```

1x2 imageSet array with properties:

```
Description  
ImageLocation  
Count
```

**Display the names of the scanned folders.**

```
{imgSets.Description}
```

```
ans =
```

```
    'books'    'cups'
```

**Display 2nd image from the 'cups' folder.**

```
imshow(read(imgSets(2),2));
```



As an alternative to the method below, you can pick the files manually using `imgetfile`:  
`imgFiles = imgetfile('MultiSelect',true);`

**Specify individual images**  
**Create an image set by specifying Individual Images**

```
imgFiles = { fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'stopSignImages', 'ima  
             fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'stopSignImages', 'ima
```

### Create image set.

```
imgSet = imageSet(imgFiles);
```

- “Image Category Classification Using Bag of Features”

### See Also

[imageSet](#) | [imageCategoryClassifier](#) | [bagOfFeatures](#) | [imgetfile](#) | [trainImageCategoryClassifier](#)

**Introduced in R2014b**

# partition

**Class:** imageSet

Divide image set into subsets

## Syntax

```
[set1,set2,...,setN] = partition(imgSet,groupSizes)
[set1,set2,...,setN] = partition(imgSet,groupPercentages)
[set1,set2,...,setN] = partition( ____,method)
```

## Description

[set1,set2,...,setN] = `partition(imgSet,groupSizes)` partitions the input image set, `imgSet`, into the collection of subsets specified in `groupSizes`.

[set1,set2,...,setN] = `partition(imgSet,groupPercentages)` returns the partitioned image sets in terms of percentages.

[set1,set2,...,setN] = `partition( ____,method)` additionally specifies a method, 'sequential' or 'randomized'.

## Input Arguments

**imgSet** — Image set

array of `imageSet` objects

Image set, specified as an array of `imageSet` objects.

**groupSizes** — Group size

scalar

Group size of images, specified as a scalar. The number of output arguments must be between 1 and `length(groupSizes) + 1`.

### Example

If you set `groupSizes` to `[20 60]`, the method returns 20 images in `set1`, 60 images in `set2`, and the remainder of images in `set3`.

### **groupPercentages** — Group size percentage

scalar

Group size of images by percentage.

### Example

If you set `groupPercentages` to `[0.1 0.5]`, the method returns 10% of images in `set1`, 50% in `set2`, and the remainder in `set3`.

### **method** — Image selection method

'sequential' (default) | 'randomized'

Image selection method, specified as either `method` or `'randomized'`. When you set `method` to `'randomized'` the images are randomly selected to form the new sets. When you set `method` to `'sequential'` the images are selected sequentially.

## Examples

### Partition Image Set

#### Create an image set.

```
imgFolder = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'stopSignImages');  
imgSet = imageSet(imgFolder);
```

#### Divide the set into two groups: one with five images and the other with the remainder of the images from `imgSet`.

```
[setA1, setA2] = partition(imgSet,5);
```

#### Randomly partition the set into three groups: one with 20% of the images, the second group with 30%, and the third group with 50%.

```
[setB1, setB2, setB3] = partition(imgSet, [0.2, 0.3], 'randomized');
```

# read

**Class:** imageSet

Read image at specified index

## Syntax

```
image = read(imgSet,idx)
```

## Description

`image = read(imgSet,idx)` returns an image from the `imgSet` image set, located at the index `idx`.

## Input Arguments

**imgSet** — Image set

array of `imageSet` objects

Image set, specified as an array of `imageSet` objects.

**idx** — Image location index

scalar

Image location index, specified as a scalar value.

## Examples

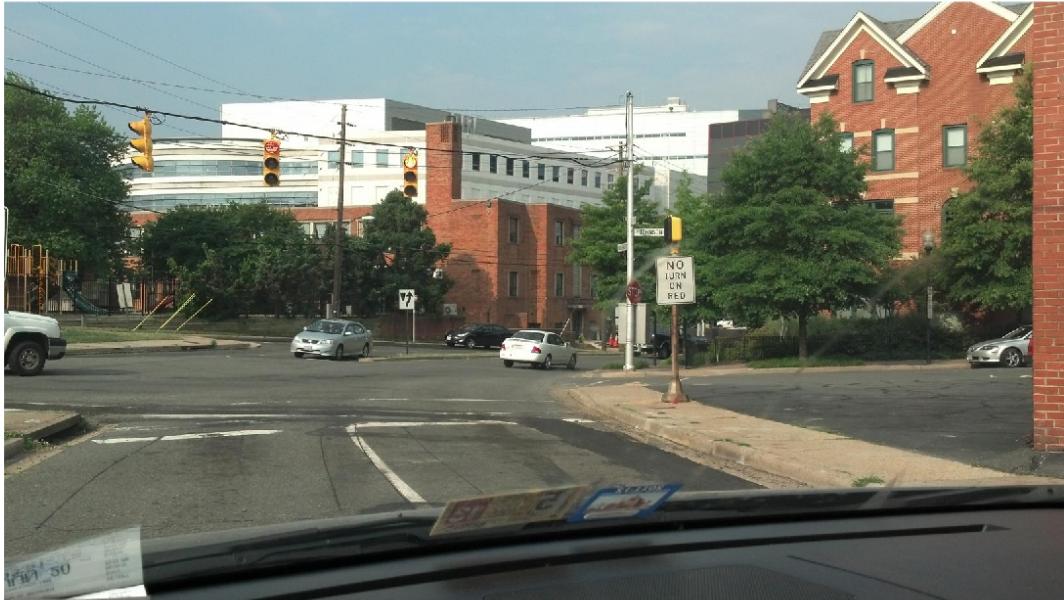
### Display Image from an Image Set

Create an image set.

```
imgFolder = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'stopSignImages');  
imgSet = imageSet(imgFolder);
```

Display the fourth image from the set.

```
imshow(read(imgSet, 4));
```





# select

**Class:** `imageSet`

Select subset of images from image set

## Syntax

```
imgSetOut = select(imgSet,idx)
```

## Description

`imgSetOut = select(imgSet,idx)` returns a new image set, `imgSetOut`, using the selection of images specified by the index `idx`.

## Input Arguments

**imgSet** — Image set

array of `imageSet` objects

Image set, specified as an array of `imageSet` objects.

**idx** — Image location index

scalar | vector of linear indices | vector of logical indices

Image location index, specified as a scalar, vector of linear indices, or a vector of logical indices. The function uses the `idx` index to select the subset of images.

## Examples

**Select Images Specified by an Index**

**Read images from a folder.**

```
imgFolder = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'stopSignImages');  
imgSet = imageSet(imgFolder);
```

**Select images 2 and 4 from the image set.**

```
imgSetOut = select(imgSet, [2, 4]);
```

**Select every other image from the image set.**

```
imgSetOut2 = select(imgSet, 1:2:imgSet.Count);
```

# invertedImageIndex class

Search index that maps visual words to images

## Syntax

```
imageIndex = invertedImageIndex(bag)
imageIndex = invertedImageIndex(bag, 'SaveFeatureLocations', tf)
imageIndex = invertedImageIndex( ____, Name, Value)
```

## Construction

`imageIndex = invertedImageIndex(bag)` returns a search index object that you can use with the `retrieveImages` function to search for an image. The object stores the visual word-to-image mapping based on the input `bag`, a `bagOfFeatures` object.

`imageIndex = invertedImageIndex(bag, 'SaveFeatureLocations', tf)` optionally specifies whether or not to save the feature location data in `imageIndex`.

`imageIndex = invertedImageIndex( ____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments, using any of the preceding syntaxes.

## Input Arguments

### **bag** — Bag of visual words

`bagOfFeatures` object

Bag of visual words, specified as a `bagOfFeatures` object.

### **SaveFeatureLocations** — Save feature locations

`true` (default) | `false`

Save feature locations, specified as a logical scalar. When you set this property to `true`, the image feature locations are saved in the `imageIndex` output object. Use location

data to verify the spatial or geometric image search results. If you do not require feature locations, set this property to `false` to reduce memory consumption.

## Properties

### **ImageLocation** — Indexed image locations

cell array

Indexed image locations, stored as a cell array.

### **ImageWords** — Visual words

1-by- $M$  vector of `visualWords` objects

Visual words, stored as a 1-by- $M$  vector of `visualWords` objects for each indexed image. The `visualWords` object contains the `WordIndex`, `Location`, `VocabularySize`, and `Count` properties for each indexed image.

### **WordFrequency** — Word occurrence

$M$ -by-1 vector

Word occurrence, specified as an  $M$ -by-1 vector. The vector contains the percentage of images in which each visual word occurs. These percentages are analogous to document frequency in text retrieval applications. The `WordFrequency` property contains the percentage of images in which each visual word occurs. It is often helpful to suppress the most common words to reduce the search set when looking for the most relevant images. Also helpful, is to suppress rare words as they probably come from outliers in the image set.

You can control how much the top and bottom end of the visual word distribution affects the search results by tuning the `WordFrequencyRange` property. A good way to set this value is to plot the sorted `WordFrequency` values.

### **BagOfFeatures** — Bag of visual words

`bagOfFeatures` object

Bag of visual words, specified as the `bagOfFeatures` object used in the index.

### **MatchThreshold** — Percentage of similar words required between query and potential image match

0.01 (default) | numeric value in the range [0 1]

Percentage of similar words required between a query and a potential image match, specified as a numeric value in the range [0, 1]. To obtain more search results, lower this threshold.

### WordFrequencyRange — Word frequency range

[0.01 0.9] (default) | two-element vector

Word frequency range, specified as a two-element vector of a lower and upper percentage, [*lower upper*]. Use the word frequency range to ignore common words (the upper percentage range) or rare words (the lower percentage range) within the image index. These words often occur as repeated patterns or outliers and can reduce search accuracy. You can control how much the top and bottom end of the visual word distribution affects the search results by tuning the `WordFrequencyRange` property. A good way to set this value is to plot the sorted `WordFrequency` values.

## Methods

<code>addImages</code>	Add new images to image index
<code>removeImages</code>	Remove images from image index

## Examples

### Search ROI for Object

Define a set of images to search.

```
imageFiles = ...
    {'elephant.jpg', 'cameraman.tif', ...
    'peppers.png', 'saturn.png',...
    'pears.png', 'stapleRemover.jpg', ...
    'football.jpg', 'mandi.tif',...
    'kids.tif', 'liftingbody.png', ...
    'office_5.jpg', 'gantrycrane.png',...
    'moon.tif', 'circuit.tif', ...
    'tape.png', 'coins.png'};

imgSet = imageSet(imageFiles);
```

Learn the visual vocabulary.

```
bag = bagOfFeatures(imgSet, 'PointSelection', 'Detector', 'VocabularySize', 1000);
```

```
Creating Bag-Of-Features from 1 image sets.
```

```
-----
```

```
* Image set 1: .
```

```
* Selecting feature point locations using the Detector method.
```

```
* Extracting SURF features from the selected feature point locations.
```

```
** detectSURFFeatures is used to detect key points for feature extraction.
```

```
* Extracting features from 16 images in image set 1...done. Extracted 3680 features.
```

```
* Keeping 80 percent of the strongest features from each image set.
```

```
* Balancing the number of features across all image sets to improve clustering.
```

```
** Image set 1 has the least number of strongest features: 2944.
```

```
** Using the strongest 2944 features from each of the other image sets.
```

```
* Using K-Means clustering to create a 1000 word visual vocabulary.
```

```
* Number of features          : 2944
```

```
* Number of clusters (K)     : 1000
```

```
* Initializing cluster centers...100.00%.
```

```
* Clustering...completed 16/100 iterations (~0.10 seconds/iteration)...converged in 16
```

```
* Finished creating Bag-Of-Features
```

Create an image search index and add images.

```
imageIndex = invertedImageIndex(bag);
```

```
addImages(imageIndex, imgSet);
```

```
Encoding 1 image sets using Bag-Of-Features.
```

```
-----
```

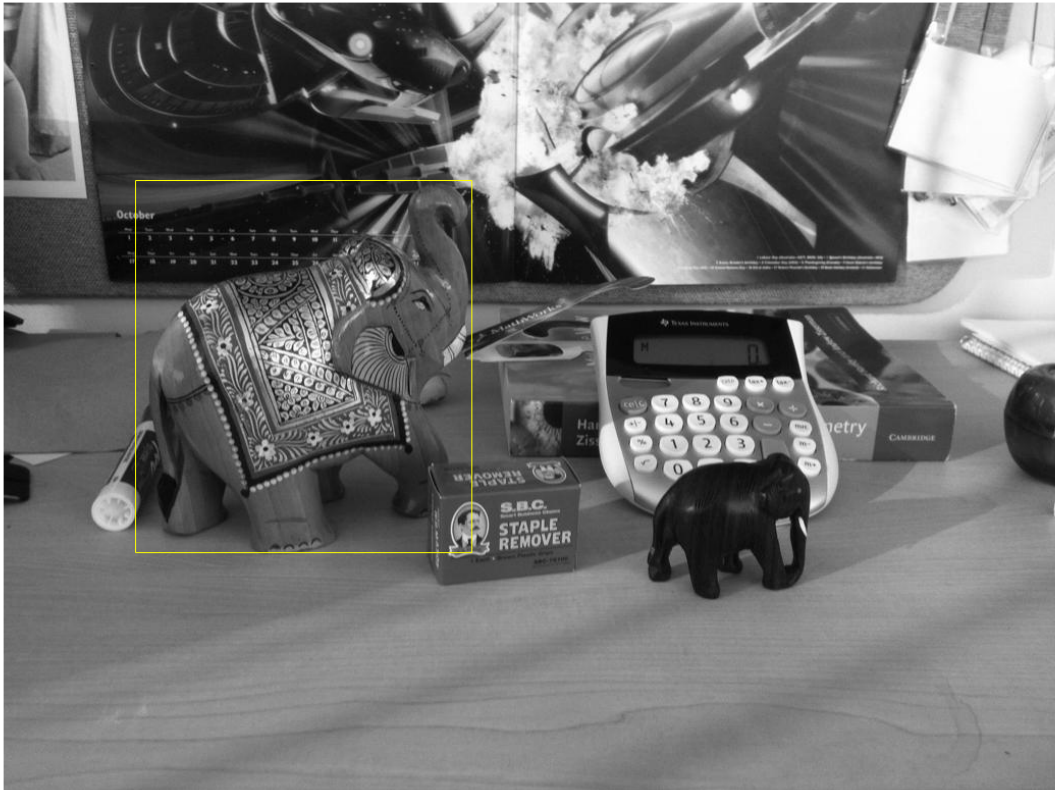
```
* Image set 1: .
```

```
* Encoding 16 images from image set 1...done.
```

```
* Finished encoding images.
```

Specify a query image and an ROI to search for the target object, elephant.

```
queryImage = imread('clutteredDesk.jpg');  
queryROI = [130 175 330 365];  
  
figure  
imshow(queryImage)  
rectangle('Position',queryROI,'EdgeColor','yellow')
```



You can also use the `imrect` function to select an ROI interactively. For example, `queryROI = getPosition(imrect)`.

Find images that contain the object.

```
imageIDs = retrieveImages(queryImage, imageIndex, 'ROI', queryROI)

bestMatch = imageIDs(1);

figure
imshow(imageIndex.ImageLocation{bestMatch})
```

```
imageIDs =
```

```
1
11
2
6
3
12
8
14
16
10
5
13
7
9
15
```





- “Image Retrieval Using Customized Bag of Features”

## References

Sivic, J. and A. Zisserman. *Video Google: A text retrieval approach to object matching in videos*. ICCV (2003) pg 1470-1477.

Philbin, J., O. Chum, M. Isard, J. Sivic, and A. Zisserman. *Object retrieval with large vocabularies and fast spatial matching*. CVPR (2007).

### **See Also**

`imageSet` | `bagOfFeatures` | `evaluateImageRetrieval` | `indexImages` | `retrieveImages`

### **More About**

- “Image Retrieval with Bag of Visual Words”

**Introduced in R2015a**

# addImages

**Class:** `invertedImageIndex`

Add new images to image index

## Syntax

```
addImages(imageIndex, imgSet)
addImages(imageIndex, imgSet, Name, Value)
```

## Description

`addImages(imageIndex, imgSet)` adds the images in `imgSet` into the `imageIndex` object. `imgSet` is an `imageSet` object that contains new images to add to an existing index. Duplicate images are not ignored.

`addImages(imageIndex, imgSet, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments, using any of the preceding syntaxes.

This object supports parallel computing using multiple MATLAB workers. Enable parallel computing from the “Computer Vision System Toolbox Preferences” dialog box. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision System Toolbox**.

## Input Arguments

**imageIndex** — Image search index

`invertedImageIndex` object

Image search index, specified as an `invertedImageIndex` object.

**imgSet** — Image set

`imageSet` object

Image set, specified as an `imageSet` 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`.

Example: `'Verbose',true` sets the `'Verbose'` property set to `true`

#### 'Verbose' — Display progress information

`true` (default) | `false`

Display progress information, specified as a logical scalar. Set `Verbose` to `true` to display progress information.

## Examples

### Add Image to Image Index

#### Define a set of images to search

```
imageFiles = ...
    {'elephant.jpg', 'cameraman.tif', ...
    'peppers.png', 'saturn.png',...
    'pears.png',    'stapleRemover.jpg', ...
    'football.jpg', 'mandi.tif',...
    'kids.tif',     'liftingbody.png', ...
    'office_5.jpg', 'gantrycrane.png',...
    'moon.tif',     'circuit.tif', ...
    'tape.png',    'coins.png'};
```

```
imgSet = imageSet(imageFiles);
```

#### Learn the visual vocabulary.

```
bag = bagOfFeatures(imgSet, 'PointSelection', 'Detector', 'VocabularySize', 1000, 'Ver
```

#### Create an image search index.

```
imageIndex = invertedImageIndex(bag);
```

**Add images.**

```
addImages(imageIndex, imgSet);
```

```
Encoding 1 image sets using Bag-Of-Features.
```

```
-----
```

```
* Image set 1: .
```

```
* Encoding 16 images from image set 1...done.
```

```
* Finished encoding images.
```

# removeImages

**Class:** `invertedImageIndex`

Remove images from image index

## Syntax

```
removeImages(imageIndex,indices)
```

## Description

`removeImages(imageIndex,indices)` removes the images from the `imageIndex` object that correspond to the `indices` input.

## Input Arguments

**imageIndex** — Image search index

`invertedImageIndex` object

Image search index, specified as an `invertedImageIndex` object.

**indices** — Image indices

vector

Image indices, specified as a row or column vector. The indices correspond to the images within `imageIndex.Location`.

## Examples

### Remove Indexed Image

Create image set.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets','cups');  
imgSets = imageSet(setDir, 'recursive');
```

**Index the image set.**

```
imageIndex = indexImages(imgSets, 'Verbose', false)
imageIndex.ImageLocation
```

```
imageIndex =
```

```
invertedImageIndex with properties:
```

```
    ImageLocation: {6x1 cell}
      ImageWords: [6x1 vision.internal.visualWords]
    WordFrequency: [1x1366 double]
    BagOfFeatures: [1x1 bagOfFeatures]
    MatchThreshold: 0.0100
    WordFrequencyRange: [0.0100 0.9000]
```

```
ans =
```

```
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\bigMug.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\blueCup.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\handMade.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\holdingCup.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\plaid.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\plainWhite.jpg'
```

**Remove first and third image.**

```
removeImages(imageIndex, [1 3]);
imageIndex.ImageLocation
```

```
ans =
```

```
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\blueCup.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\holdingCup.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\plaid.jpg'
'B:\matlab\toolbox\vision\visiondata\imageSets\cups\plainWhite.jpg'
```

## cameraCalibrationErrors class

Object for storing standard errors of estimated camera parameters

### Syntax

cameraCalibrationErrors

### Description

cameraCalibrationErrors contains the standard errors of estimated camera parameters. The `estimateCameraParameters` function returns the cameraCalibrationErrors object. You can access the intrinsics and extrinsics standard errors using the object properties. You can display the standard errors using the object's `displayErrors` method.

### Properties

#### IntrinsicsErrors

Standard errors of estimated camera intrinsics and distortion coefficients.

#### ExtrinsicsErrors

Standard errors of estimated camera extrinsics.

### Methods

<code>displayErrors</code>	Display standard errors of camera parameter estimation
----------------------------	--



## Examples

### Estimate and Display Camera Calibration Standard Errors

Create a cell array of file names of calibration images.

```
for i = 1:10
    imageFileName = sprintf('image%02d.jpg',i);
    imageFileNames{i} = fullfile(matlabroot,'toolbox','vision','visiondata','calibrati
end
```

Detect the calibration pattern.

```
[imagePoints,boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate the world coordinates of the corners of the squares.

```
squareSize = 29;
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the camera.

```
[params,~,errors] = estimateCameraParameters(imagePoints,worldPoints);
```

Display the standard errors.

```
displayErrors(errors,params);
```

```
Standard Errors of Estimated Camera Parameters
```

```
Intrinsics
```

```
-----
```

```
Focal length (pixels): [ 716.0020 +/- 3.8920      711.8093 +/- 4.7720 ]
Principal point (pixels):[ 569.4790 +/- 6.3904      357.8015 +/- 3.9323 ]
Radial distortion: [ -0.3444 +/- 0.0100      0.1359 +/- 0.0526 ]
```

```
Extrinsics
```

```
-----
```

```
Rotation vectors:
```

```
[ -0.6044 +/- 0.0064      -0.1869 +/- 0.0087      -0.3864
 [ -0.7264 +/- 0.0059      -0.1086 +/- 0.0085      0.1937
 [ -0.6682 +/- 0.0061      -0.1522 +/- 0.0087      -0.1357
 [ -0.5775 +/- 0.0066      -0.2965 +/- 0.0088      -0.5650
```

[	-0.3115 +/- 0.0078	-0.1513 +/- 0.0088	-0.1081
[	-0.7523 +/- 0.0061	0.0031 +/- 0.0086	0.0317
[	-0.7569 +/- 0.0062	0.1859 +/- 0.0085	0.4291
[	-0.7494 +/- 0.0060	0.0693 +/- 0.0085	0.2038
[	-0.6199 +/- 0.0064	0.0144 +/- 0.0087	0.3643
[	0.3476 +/- 0.0076	-0.2290 +/- 0.0086	-0.0428

Translation vectors (mm):

[	-152.5581 +/- 7.1350	-29.2133 +/- 4.4383	798.2530
[	-216.9255 +/- 8.2291	-62.2290 +/- 5.1827	922.2418
[	-137.0242 +/- 8.3867	-46.9033 +/- 5.2076	939.0848
[	-158.2523 +/- 7.9129	-29.9041 +/- 4.9211	885.5162
[	-182.1781 +/- 7.9274	-26.9372 +/- 4.9501	887.2424
[	-186.2656 +/- 9.1338	-82.6673 +/- 5.7104	1021.9812
[	-142.8862 +/- 9.3289	-106.5762 +/- 5.8204	1043.7253
[	-182.2946 +/- 9.0897	-76.0732 +/- 5.6877	1018.1460
[	-210.9513 +/- 8.7878	-90.8939 +/- 5.5298	984.6058
[	-326.7958 +/- 7.4671	-122.3645 +/- 4.8672	829.4460

- “Evaluating the Accuracy of Single Camera Calibration”

### See Also

[cameraParameters](#) | [stereoParameters](#) | [stereoCalibrationErrors](#) | [intrinsicsEstimationErrors](#) | [extrinsicsEstimationErrors](#) | [Camera Calibrator](#) | [detectCheckerboardPoints](#) | [estimateCameraParameters](#) | [generateCheckerboardPoints](#) | [showExtrinsics](#) | [showReprojectionErrors](#) | [Stereo Camera Calibrator](#) | [undistortImage](#) | [undistortPoints](#)

### More About

- “Single Camera Calibration App”

**Introduced in R2014b**

# displayErrors

**Class:** cameraCalibrationErrors

Display standard errors of camera parameter estimation

## Syntax

```
displayErrors(estimationErrors,cameraParams)
```

## Description

`displayErrors(estimationErrors,cameraParams)` returns the camera parameters and corresponding standard errors. The `estimationErrors` input must be a `cameraCalibrationErrors` object. The `cameraParams` input must be a `cameraParameters` object.

## Input Arguments

**estimationErrors** — Standard errors of estimated parameters

`cameraCalibrationErrors` object

Standard errors of estimated parameters, specified as a `cameraCalibrationErrors` object.

**cameraParams** — Camera parameters

`cameraParameters` object

Object for storing camera parameters, specified as a `cameraParameters` returned by the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

**Introduced in R2014b**

# stereoCalibrationErrors class

Object for storing standard errors of estimated stereo parameters

## Syntax

stereoCalibrationErrors

## Description

stereoCalibrationErrors contains the standard errors of estimated stereo parameters. The `estimateCameraParameters` function returns the stereoCalibrationErrors object. You can access the standard errors for stereo calibration using the object's properties. You can display the standard errors using the object `displayErrors` method.

## Properties

### Camera1IntrinsicsErrors

Standard errors of camera 1 estimated intrinsics and distortion coefficients, specified as an `intrinsicsEstimationErrors` object.

### Camera1ExtrinsicsErrors

Standard errors of camera 1 estimated extrinsics parameters, specified as an `extrinsicsEstimationErrors` object.

### Camera2IntrinsicsErrors

Standard errors of camera 2 estimated intrinsics and distortion coefficients, specified as an `intrinsicsEstimationErrors` object.

### RotationOfCamera2Error

Standard errors of rotated vector of camera 2 relative to camera 1, specified as a 3-element vector.

## TranslationOfCamera2Error

Standard errors of translation of camera 2 relative to camera 1, specified as a 3-element vector.

## Methods

<code>displayErrors</code>	Display standard errors of camera parameter estimation
----------------------------	--

## Examples

### Estimate and Display Stereo Calibration Standard Errors

Specify the calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');
leftImages = imageSet(fullfile(imageDir,'left'));
rightImages = imageSet(fullfile(imageDir,'right'));
images1 = leftImages.ImageLocation;
images2 = rightImages.ImageLocation;
```

Detect the checkerboards.

```
[imagePoints,boardSize] = detectCheckerboardPoints(images1,images2);
```

Specify the world coordinates of the checkerboard keypoints.

```
squareSize = 108; % in millimeters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the stereo camera system.

```
[params,~,errors] = estimateCameraParameters(imagePoints,worldPoints);
```

Display the standard errors.

```
displayErrors(errors, params);
```

```
Standard Errors of Estimated Stereo Camera Parameters
-----
```

```
Camera 1 Intrinsic
```

```

-----
Focal length (pixels): [ 1038.3079 +/- 0.9611      1038.2745 +/- 0.9399 ]
Principal point (pixels): [ 656.6034 +/- 0.5011      485.6363 +/- 0.3892 ]
Radial distortion: [ -0.3609 +/- 0.0011      0.1825 +/- 0.0039 ]

```

### Camera 1 Extrinsics

```

-----
Rotation vectors:
[ -0.1693 +/- 0.0010      0.0276 +/- 0.0008      -3.1613
[ 0.2003 +/- 0.0008      -0.0519 +/- 0.0007      -3.0990
[ 0.4195 +/- 0.0008      -0.1059 +/- 0.0007      -3.1112
[ 0.5247 +/- 0.0008      -0.0607 +/- 0.0006      -3.0551
[ 0.6810 +/- 0.0008      -0.0294 +/- 0.0008      -3.0330
[ 0.3512 +/- 0.0010      -0.0988 +/- 0.0009      -3.0334
[ 0.0220 +/- 0.0010      -0.1170 +/- 0.0010      -3.0834
[ -0.2756 +/- 0.0011      -0.0843 +/- 0.0010      -3.0945
[ -0.4397 +/- 0.0011      -0.1117 +/- 0.0009      -3.0653
[ -0.2525 +/- 0.0012      -0.1347 +/- 0.0010      -3.1040

```

### Translation vectors (mm):

```

[ 707.6594 +/- 0.7226      226.9705 +/- 0.5902      1493.7247
[ 367.6252 +/- 0.7688      191.6252 +/- 0.6036      1590.5733
[ 225.5625 +/- 0.7606      191.0502 +/- 0.5943      1579.0046
[ 48.7366 +/- 0.7621      196.6627 +/- 0.5942      1580.9597
[ -173.5078 +/- 1.0299      150.8890 +/- 0.7971      2119.9396
[ 9.7211 +/- 0.9975      176.7781 +/- 0.7778      2067.2874
[ 294.4454 +/- 0.9730      167.7595 +/- 0.7606      2011.7128
[ 613.2744 +/- 0.9496      166.0783 +/- 0.7599      1969.1923
[ 766.0655 +/- 0.8978      165.4368 +/- 0.7360      1869.3334
[ 952.5847 +/- 1.0786      -14.9333 +/- 0.8903      2256.4651

```

### Camera 2 Intrinsics

```

-----
Focal length (pixels): [ 1042.9245 +/- 0.9776      1042.7038 +/- 0.9615 ]
Principal point (pixels): [ 640.5187 +/- 0.4880      478.8757 +/- 0.3883 ]
Radial distortion: [ -0.3617 +/- 0.0011      0.1840 +/- 0.0033 ]

```

### Position And Orientation of Camera 2 Relative to Camera 1

```

-----
Rotation of camera 2: [ -0.0039 +/- 0.0003      0.0055 +/- 0.0006      -0

```

Translation of camera 2 (mm): [ -119.8668 +/- 0.0590      -0.3419 +/- 0.0609      0

## See Also

[cameraParameters](#) | [cameraCalibrationErrors](#) | [stereoParameters](#) | [stereoCalibrationErrors](#) | [intrinsicEstimationErrors](#) | [extrinsicEstimationErrors](#) | [Camera Calibrator](#) | [detectCheckerboardPoints](#) | [estimateCameraParameters](#) | [generateCheckerboardPoints](#) | [showExtrinsics](#) | [showReprojectionErrors](#) | [Stereo Camera Calibrator](#) | [undistortImage](#)

## More About

- “Single Camera Calibration App”
- “Stereo Calibration App”

**Introduced in R2014b**

# displayErrors

**Class:** stereoCalibrationErrors

Display standard errors of camera parameter estimation

## Syntax

```
displayErrors(estimationErrors, stereoParams)
```

## Description

`displayErrors(estimationErrors, stereoParams)` displays stereo parameters and corresponding standard errors to the screen. The `estimationErrors` input must be a `stereoCalibrationErrors` object. The `stereoParams` input must be a `stereoParameters` object.



# MSERRegions class

Object for storing MSER regions

## Description

This object describes MSER regions and corresponding ellipses that have the same second moments as the regions. It passes data between the `detectMSERFeatures` and `extractFeatures` functions. The object can also be used to manipulate and plot the data returned by these functions.

## Tips

Although `MSERRegions` may hold many regions, it is a scalar object. Therefore, `NUMEL(MSERRegions)` always returns 1. This value may differ from `LENGTH(MSERRegions)`, which returns the true number of regions held by the object.

## Construction

`regions = MSERRegions(pixellist)` constructs an MSER regions object, `regions`, from an  $M$ -by-1 cell array of regions, `pixellist`. Each cell contains a  $P$ -by-2 array of  $[x\ y]$  coordinates for the detected MSER regions, where  $P$  varies based on the number of pixels in a region.

Code Generation Support
Supports Code Generation: Yes
Compile-time constant inputs: No restrictions.
Supports MATLAB Function block: Yes
For code generation, you must specify both the <code>pixellist</code> cell array and the <code>length</code> of each array, as the second input. The object outputs, <code>regions.PixelList</code> as an array. The region sizes are defined in <code>regions.Lengths</code> .

### Code Generation Support

Generated code for this function uses a precompiled platform-specific shared library.

“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### **pixellist** — Cell array of point coordinates for detected MSER regions

Cell array of  $M$ -by-1 regions. Each region contains a  $P$ -by-2 array of  $[x\ y]$  coordinates of the detected MSER regions.

## Properties

The following properties are read-only and are calculated once the input pixel list is specified.

### **Count**

Number of stored regions

**Default:** 0

### **Location**

An  $M$ -by-2 array of  $[x\ y]$  centroid coordinates of ellipses that have the same second moments as the MSER regions.

### **Axes**

A two-element vector,  $[\text{majorAxis}\ \text{minorAxis}]$ . This vector specifies the major and minor axis of the ellipse that have the same second moments as the MSER regions.

### **Orientation**

A value in the range from  $-\pi/2$  to  $+\pi/2$  radians. This value represents the orientation of the ellipse as measured from the  $X$ -axis to the major axis of the ellipse. You can use this property for visualization purposes.

## Methods

<code>isempty</code>	Returns true for empty object
<code>length</code>	Number of stored points
<code>plot</code>	Plot MSER regions
<code>size</code>	Size of the MSERRegions object

## Examples

### Detect MSER Features in an Image

Load an image.

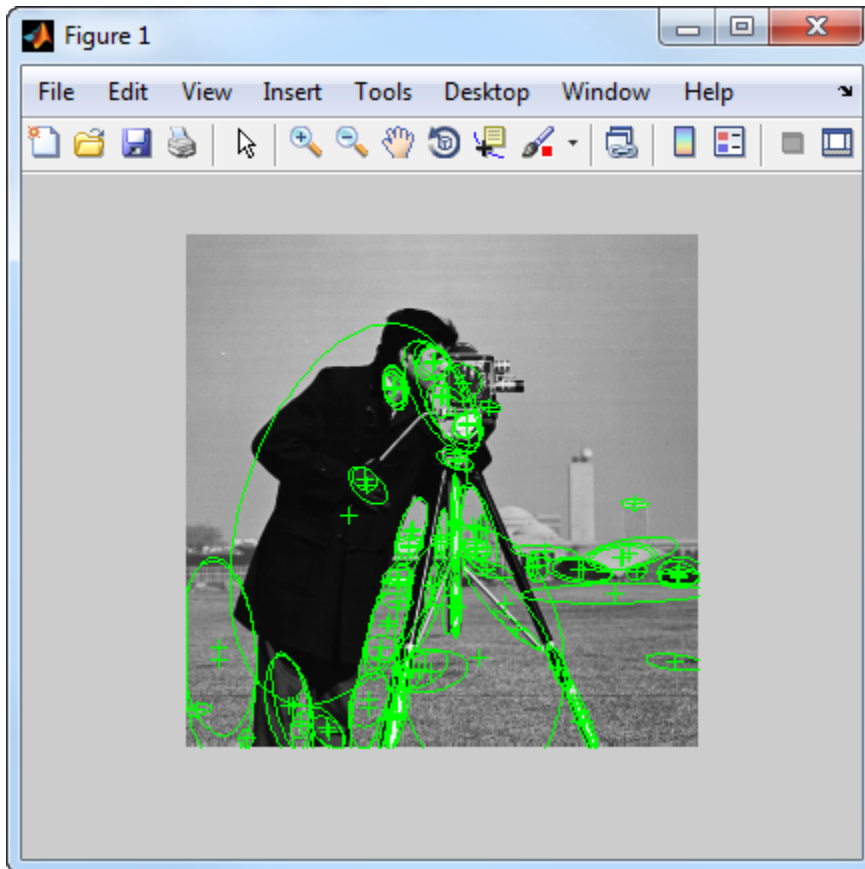
```
I = imread('cameraman.tif');
```

Detect and store regions.

```
regions = detectMSERFeatures(I);
```

Display the centroids and axes of detected regions.

```
imshow(I); hold on;  
plot(regions);
```



### Display MSER Feature Regions from the MSERRegions Object

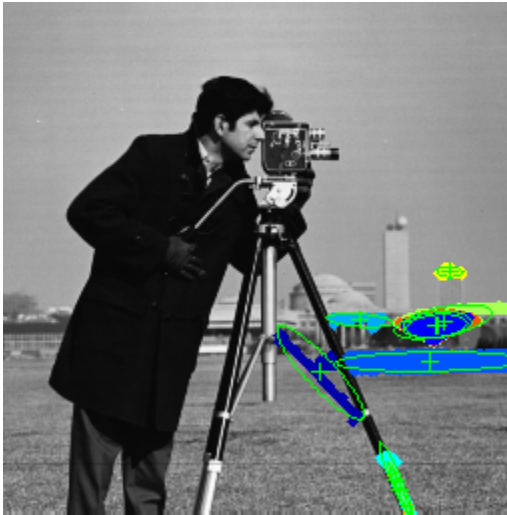
Detect and display the first 10 regions contained in the MSERRegions object

Detect MSER features.

```
I = imread('cameraman.tif');  
regions = detectMSERFeatures(I);
```

Display the first 10 regions in the MSERRegions object.

```
imshow(I); hold on;  
plot(regions(1:10), 'showPixelList', true);
```



### Combine MSER Region Detector with SURF Descriptors

Extract and display SURF descriptors at locations identified by MSER detector.

Read image.

```
I = imread('cameraman.tif');
```

Detect MSER features.

```
regionsObj = detectMSERFeatures(I);
```

Extract and display SURF descriptors.

```
[features, validPtsObj] = extractFeatures(I, regionsObj);  
imshow(I); hold on;  
plot(validPtsObj, 'showOrientation', true);
```



- “Find MSER Regions in an Image” on page 3-134
- “Detect SURF Interest Points in a Grayscale Image” on page 3-150
- “Automatically Detect and Recognize Text in Natural Images”

## References

- [1] Nister, D., and H. Stewenius, "Linear Time Maximally Stable Extremal Regions", *Lecture Notes in Computer Science*. 10th European Conference on Computer Vision, Marseille, France: 2008, no. 5303, pp. 183–196.
- [2] Matas, J., O. Chum, M. Urba, and T. Pajdla. "Robust wide baseline stereo from maximally stable extremal regions." *Proceedings of British Machine Vision Conference*, pages 384-396, 2002.

## See Also

[SURFPoints](#) | [detectMSERFeatures](#) | [detectSURFFeatures](#) | [edge](#) | [extractFeatures](#) | [matchFeatures](#)

**Introduced in R2012a**

## **isempty**

**Class:** MSERRegions

Returns true for empty object

### **Syntax**

`isempty(MSERRegionsObj)`

### **Description**

`isempty(MSERRegionsObj)` returns a true value, if the MSERRegionsObj object is empty.

## length

**Class:** MSERRegions

Number of stored points

## Syntax

`length(MSERRegionsObj)`

## Description

`length(MSERRegionsObj)` returns the number of stored points in the MSERRegionsObj object.



# plot

**Class:** MSERRegions

Plot MSER regions

## Syntax

```
MSERRegions.plot  
MSERRegions.plot(AXES_HANDLE,...)  
MSERRegions.plot(AXES_HANDLE,Name,Value)
```

## Description

`MSERRegions.plot` plots MSER points in the current axis.

`MSERRegions.plot(AXES_HANDLE,...)` plots using axes with the handle `AXES_HANDLE`.

`MSERRegions.plot(AXES_HANDLE,Name,Value)` Additional control for the `plot` method requires specification of parameters and corresponding values. An additional option is specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **AXES\_HANDLE**

Handle for plot method to use for display. You can set the handle using `gca`.

## Name-Value Pair Arguments

### **showEllipses**

Display ellipses around feature. Set this value to `true` or `false`. When you set this value to `true`, the object draws an ellipse with the same 2nd order moments as the region.

When you set this value to `false`, only the ellipses centers are plotted.

**Default:** true

### **showOrientation**

Display a line corresponding to the region's orientation. Set this value to `true` or `false`. When you set this value to `true`, the object draws a line corresponding to the region's orientation. The object draws the line from the ellipse's centroid to the edge of the ellipse, which indicates the region's major axis.

**Default:** false

### **showPixelList**

Display the MSER regions. Set this value to `true` or `false`. When you set this value to `true`, the object plots the MSER regions using the JET colormap.

## Examples

### **Plot MSER Regions**

Extract MSER features and plot the regions

Read image and extract MSER features

```
I = imread('cameraman.tif');
regions = detectMSERFeatures(I);
imshow(I);hold on;
plot(regions);
```

Plot MSER Regions

```
figure; imshow(I);hold on;
plot(regions,'showPixelList',true, 'showEllipses',false);
hold off;
```

## size

**Class:** MSERRegions

Size of the MSERRegions object

## Syntax

```
size(MSERRegionsObj)
```

## Description

`size(MSERRegionsObj)` returns the size of the `MSERRegionsObj` object.

## cornerPoints class

Object for storing corner points

### Description

This object stores information about feature points detected from a 2-D grayscale image.

### Tips

Although `cornerPoints` may hold many points, it is a scalar object. Therefore, `NUMEL(cornerPoints)` always returns 1. This value may differ from `LENGTH(cornerPoints)`, which returns the true number of points held by the object.

### Construction

`points = cornerPoints(Location)` constructs a `cornerPoints` object from an  $M$ -by-2 array of [x y] point coordinates, `Location`.

`points = cornerPoints(Location,Name,Value)` constructs a `cornerPoints` object with optional input properties specified by one or more `Name,Value` pair arguments. Each additional property can be specified as a scalar or a vector whose length matches the number of coordinates in `Location`.

Code Generation Support
Compile-time constant inputs: No restriction.
Supports MATLAB Function block: No
To index locations with this object, use the syntax: <code>points.Location(idx,:)</code> , for <code>points</code> object. See <code>visionRecoverformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### Location

*M*-by-2 array of [x y] point coordinates.

## 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`.

### 'Count'

Number of points held by the object.

**Default:** 0

### 'Metric'

Value describing strength of detected feature.

**Default:** 0.0

## Methods

<code>isempty</code>	Returns true for empty object
<code>length</code>	Number of stored points
<code>plot</code>	Plot corner points
<code>selectStrongest</code>	Return points with strongest metrics
<code>selectUniform</code>	Return a uniformly distributed subset of feature points
<code>size</code>	Size of the cornerPoints object
<code>gather</code>	Retrieve cornerPoints from the GPU

## Examples

### Plot Strongest Features from Detected Feature Points

Read image.

```
I = imread('cameraman.tif');
```

Detect feature points.

```
points = detectHarrisFeatures(I);
```

Display the 10 strongest points.

```
strongest = points.selectStrongest(10);  
imshow(I); hold on;  
plot(strongest);
```



**Display the (x,y) coordinates.**

```
strongest.Location
```

```
ans =
```

```
112.4516 208.4412
108.6510 228.1681
136.6969 114.7962
181.4160 205.9876
135.5823 123.4529
100.4951 174.3253
146.7581 94.7393
135.2899 92.6485
129.8439 110.0350
130.5716 91.0424
```

**Create a Corner Points Object and Display Points****Create a checkerboard image.**

```
I = checkerboard(50,2,2);
```

**Load the locations.**

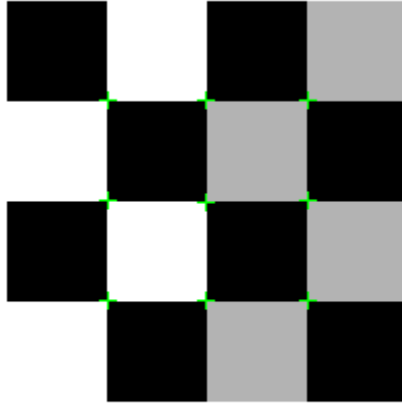
```
location = [51 51 51 100 100 100 151 151 151;...
            50 100 150 50 101 150 50 100 150]';
```

**Save points.**

```
points = cornerPoints(location);
```

**Display points on checkerboard.**

```
imshow(I); hold on;
plot(points);
```



### See Also

[MSERRegions](#) | [SURFPoints](#) | [BRISKPoints](#) | [binaryFeatures](#) | [detectBRISKFeatures](#) | [detectFASTFeatures](#) | [detectHarrisFeatures](#) | [detectMinEigenFeatures](#) | [detectMSERFeatures](#) | [detectSURFFeatures](#) | [extractFeatures](#) | [extractHOGFeatures](#) | [matchFeatures](#)



## **isempty**

**Class:** cornerPoints

Returns true for empty object

### **Syntax**

`isempty(cornerPointsObj)`

### **Description**

`isempty(cornerPointsObj)` returns a true value, if the corner points object is empty.

## length

**Class:** cornerPoints

Number of stored points

## Syntax

`length(cornerPointsObj)`

## Description

`length(cornerPointsObj)` returns the number of stored points in the corner points object.

# plot

**Class:** cornerPoints

Plot corner points

## Syntax

```
cornerPoints.plot  
cornerPoints.plot(axesHandle, ___ )
```

## Description

`cornerPoints.plot` plots corner points in the current axis.

`cornerPoints.plot(axesHandle, ___ )` plots using axes with the handle `axesHandle` instead of the current axes (`gca`).

## Input Arguments

### **axesHandle**

Handle for plot method to use for display. You can set the handle using `gca`.

## Examples

### **Plot corner features**

Read an image.

```
I = imread('cameraman.tif');
```

Detect corner features.

```
featurePoints = detectHarrisFeatures(I);
```

Plot feature image with detected features.

```
imshow(I); hold on;  
plot(featurePoints);
```

# selectStrongest

**Class:** cornerPoints

Return points with strongest metrics

## Syntax

```
cornerPoints = cornerPoints.selectStrongest(N)
```

## Description

`cornerPoints = cornerPoints.selectStrongest(N)` returns N number of points with strongest metrics.

## Examples

### Select Strongest Features

Create object holding 50 points.

```
points = cornerPoints(ones(50,2), 'Metric', 1:50);
```

Keep 2 strongest features.

```
points = points.selectStrongest(2)
```

# selectUniform

**Class:** cornerPoints

Return a uniformly distributed subset of feature points

## Syntax

```
pointsOut = selectUniform(pointsIn,N,imageSize)
```

## Description

`pointsOut = selectUniform(pointsIn,N,imageSize)` returns N uniformly distributed points with the strongest metrics.

## Input Arguments

### **pointsIn** — Corner points

cornerPoints object

SURF points, returned as a cornerPoints object. This object contains information about corner features detected in a grayscale image. You can use one of the detection functions to obtain the corner points.

### **N** — Number of points

integer

Number of points, specified as an integer.

### **imageSize** — Size of image

2-element vector | 3-element vector

Size of image, specified as a 2- or 3-element vector.

## Examples

### Select A Uniformly Distributed Subset of Features From an Image

Load an image.

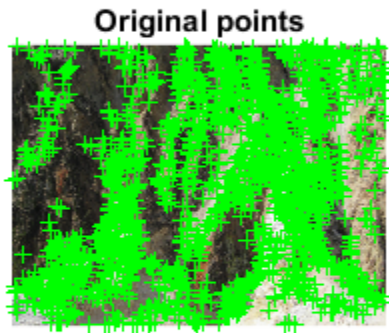
```
im = imread('yellowstone_left.png');
```

Detect many corners by reducing the quality threshold.

```
points1 = detectHarrisFeatures(rgb2gray(im), 'MinQuality', 0.05);
```

Plot image with detected corners.

```
subplot(1,2,1);  
imshow(im);  
hold on  
plot(points1);  
hold off  
title('Original points');
```



Select a uniformly distributed subset of points.

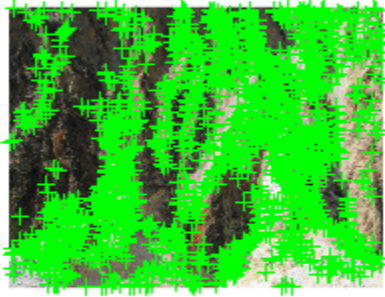
```
numPoints = 100;  
points2 = selectUniform(points1,numPoints,size(im));
```

Plot images showing original and subset of points.

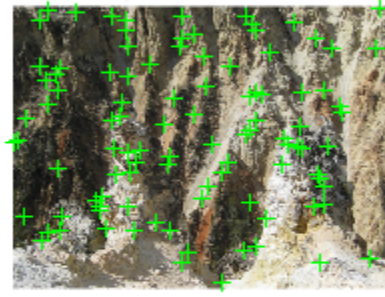
```
subplot(1, 2, 2);  
imshow(im);  
hold on  
plot(points2);  
hold off  
title('Uniformly distributed points');
```



**Original points**



**Uniformly distributed points**



## **size**

**Class:** cornerPoints

Size of the cornerPoints object

## **Syntax**

`size(cornerPointsObj)`

## **Description**

`size(cornerPointsObj)` returns the size of the corner points object.

# gather

**Class:** cornerPoints

Retrieve cornerPoints from the GPU

## Syntax

```
pointsCPU = gather(pointsGPU)
```

## Description

`pointsCPU = gather(pointsGPU)` returns a `cornerPoints` object with data gathered from the GPU for the `Location` and `Metric` properties.

## Examples

This example uses requires the Parallel Computing Toolbox™.

Fine and plot corner points in an image.

```
I = gpuArray(imread('cameraman.tif'));  
pointsGPU = detectHarrisFeatures(I);  
imshow(I);  
hold on;  
plot(pointsGPU.selectStrongest(50));
```

Copy the corner points to the CPU for further processing.

```
pointsCPU = gather(pointsGPU);
```

## SURFPoints class

Object for storing SURF interest points

### Description

This object provides the ability to pass data between the `detectSURFFeatures` and `extractFeatures` functions. It can also be used to manipulate and plot the data returned by these functions. You can use the object to fill the points interactively. You can use this approach in situations where you might want to mix a non-SURF interest point detector with a SURF descriptor.

### Tips

Although `SURFPoints` may hold many points, it is a scalar object. Therefore, `NUMEL(surfPoints)` always returns 1. This value may differ from `LENGTH(surfPoints)`, which returns the true number of points held by the object.

### Construction

`points = SURFPoints(Location)` constructs a `SURFPoints` object from an  $M$ -by-2 array of [x y] point coordinates, `Location`.

`points = SURFPoints(Location,Name,Value)` constructs a `SURFPoints` object with optional input properties specified by one or more `Name,Value` pair arguments. Each additional property can be specified as a scalar or a vector whose length matches the number of coordinates in `Location`.

Code Generation Support
Compile-time constant inputs: No restriction.
Supports MATLAB Function block: No
To index locations with this object, use the syntax: <code>points.Location(idx,:)</code> , for <code>points</code> object. See <code>visionRecoverformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### Location — Point coordinates

*M*-by-2 array of [x y] point coordinates.

Point coordinates, specified as an *M*-by-2 array. The **Location** input contains *M* number of [x y] points.

## Properties

### Count

Number of points held by the object.

**Default:** 0

### Location

Array of [x y] point coordinates

### Scale

Specifies scale at which the interest points were detected. This value must be greater than or equal to 1.6.

**Default:** 1.6

### Metric

Value describing strength of detected feature. The SURF algorithm uses a determinant of an approximated Hessian.

**Default:** 0.0

### SignOfLaplacian

Sign of the Laplacian determined during the detection process. This value must be an integer, -1, 0, or 1. You can use this parameter to accelerate the feature matching process.

Blobs with identical metric values but different signs of Laplacian can differ by their intensity values. For example, a white blob on a black background versus a black blob on a

white background. You can use this parameter to quickly eliminate blobs that do not match.

For non-SURF detectors, this property is not relevant. For example, for corner features, you can simply use the default value of 0.

**Default:** 0

### **Orientation**

Orientation of the detected feature, specified as an angle, in radians. The angle is measured counter-clockwise from the X-axis with the origin specified by the `Location` property. Do not set this property manually. Rely instead, on the call to `extractFeatures` to fill in this value. The `extractFeatures` function modifies the default value of 0.0. The `Orientation` is mainly useful for visualization purposes.

**Default:** 0.0

## **Methods**

<code>isempty</code>	Returns true for empty object
<code>length</code>	Number of stored points
<code>plot</code>	Plot SURF points
<code>selectStrongest</code>	Return points with strongest metrics
<code>selectUniform</code>	Return a uniformly distributed subset of feature points
<code>size</code>	Size of the SURFPoints object

## **Examples**

### **Detect SURF Features**

**Read in image.**

```
I = imread('cameraman.tif');
```

**Detect SURF features.**

```
points = detectSURFFeatures(I);
```

**Display location and scale for the 10 strongest points.**

```
strongest = points.selectStrongest(10);  
imshow(I); hold on;  
plot(strongest);
```



**Display [x y] coordinates for the 10 strongest points on command line.**

```
strongest.Location
```

```
ans =
```

```
139.7482    95.9542  
107.4502   232.0347  
116.6112   138.2446  
105.5152   172.1816  
113.6975    48.7220  
104.4210    75.7348  
111.3914   154.4597
```

```
106.2879 175.2709
131.1298 98.3900
124.2933 64.4942
```

### Detect SURF Features and Display the Last 5 Points

Read in image.

```
I = imread('cameraman.tif');
```

Detect SURF feature.

```
points = detectSURFFeatures(I);
```

Display the last 5 points.

```
imshow(I); hold on;
plot(points(end-4:end));
```



- “Detect SURF Interest Points in a Grayscale Image” on page 3-150



- “Display MSER Feature Regions from the MSERRegions Object” on page 2-152
- “Find MSER Regions in an Image” on page 3-134
- “Detect MSER Features in an Image” on page 2-151

## References

- [1] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. “SURF:Speeded Up Robust Features.” *Computer Vision and Image Understanding (CVIU)*. Vol. 110, No. 3, pp. 346–359, 2008.

## See Also

[MSERRegions](#) | [BRISKPoints](#) | [cornerPoints](#) | [detectFASTFeatures](#) | [detectHarrisFeatures](#) | [detectMinEigenFeatures](#) | [detectMSERFeatures](#) | [detectSURFFeatures](#) | [extractFeatures](#) | [matchFeatures](#)

**Introduced in R2011b**

## **isempty**

**Class:** SURFPoints

Returns true for empty object

### **Syntax**

`isempty(SURFPointsObj)`

### **Description**

`isempty(SURFPointsObj)` returns a true value, if the SURFPointsObj object is empty.

# length

**Class:** SURFPoints

Number of stored points

## Syntax

`length(SURFPointsObj)`

## Description

`length(SURFPointsObj)` returns the number of stored points in the `SURFPointsObj` object.

# plot

**Class:** SURFPoints

Plot SURF points

## Syntax

```
surfPoints/plot  
surfPoints/plot(axesHandle, ___ )  
surfPoints/plot(axesHandle,Name,Value)
```

## Description

`surfPoints/plot` plots SURF points in the current axis.

`surfPoints/plot(axesHandle, ___ )` plots using axes with the handle `axesHandle`.

`surfPoints/plot(axesHandle,Name,Value)` additional control for the `plot` method specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **axesHandle**

Handle for plot method to use for display. You can set the handle using `gca`.

### **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`.

**'ShowScale'** — Display proportional circle around feature  
true (default) | false

Display proportional circle around feature, specified as a comma-separated pair consisting of 'ShowScale' and the logical `true` or `false`. When you set this value to `true`, the object draws a circle proportional to the scale of the detected feature. The circle contains the feature point located at its center. When you set this value to `false`, the object turns the display of the circle off.

The algorithm represents the scale of the feature with a circle of  $6 \times \text{Scale}$  radius. The SURF algorithm uses this equivalent size of circular area to compute the orientation of the feature.

**'ShowOrientation' — Display a line corresponding to feature point orientation**  
`false` (default) | `true`

Display a line corresponding to feature point orientation, specified as a comma-separated pair consisting of 'ShowOrientation' and the logical `true` or `false`. When you set this value to `true`, the object draws a line corresponding to the point's orientation. The object draws the line from the feature point location to the edge of the circle, indicating the scale.

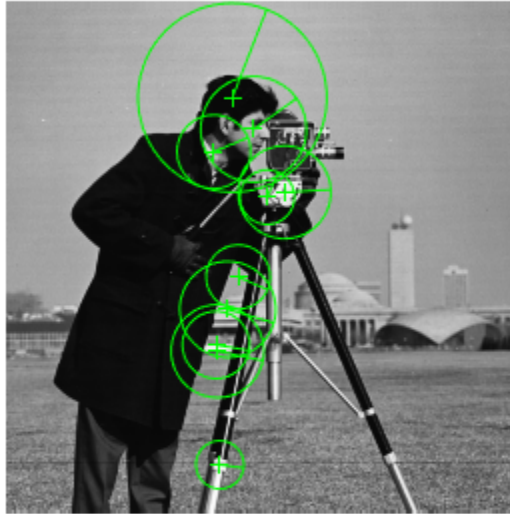
## Examples

### Plot SURF features

Extract SURF features from an image.

```
I = imread('cameraman.tif');
points = detectSURFFeatures(I);
[features, valid_points] = extractFeatures(I, points);

% Visualize 10 strongest SURF features, including their scales and orientation which w
imshow(I); hold on;
strongestPoints = valid_points.selectStrongest(10);
strongestPoints.plot('showOrientation',true);
```



- “Image Retrieval Using Customized Bag of Features”
- “Object Detection in a Cluttered Scene Using Point Feature Matching”
- “Image Category Classification Using Bag of Features”

### **See Also**

`extractFeatures` | `SURFPoints` | `detectSURFFeatures`

# selectStrongest

**Class:** SURFPoints

Return points with strongest metrics

## Syntax

```
strongest = selectStrongest(points,N)
```

## Description

`strongest = selectStrongest(points,N)` returns N number of points with strongest metrics.

## Examples

### Select Strongest Surf Features

Create object holding 50 points.

```
points = SURFPoints(ones(50,2), 'Metric', 1:50);
```

Keep the two strongest features.

```
points = selectStrongest(points,2)
```

```
points =
```

```
    2x1 SURFPoints array with properties:
```

```
        Scale: [2x1 single]
    SignOfLaplacian: [2x1 int8]
        Orientation: [2x1 single]
        Location: [2x2 single]
        Metric: [2x1 single]
        Count: 2
```





# selectUniform

**Class:** SURFPoints

Return a uniformly distributed subset of feature points

## Syntax

```
pointsOut = selectUniform(pointsIn,N,imageSize)
```

## Description

`pointsOut = selectUniform(pointsIn,N,imageSize)` returns N uniformly distributed points with the strongest metrics.

## Input Arguments

### **pointsIn** — SURF points

surfPoints object

SURF points, returned as a SURFPoints object. This object contains information about SURF features detected in a grayscale image. You can use the `detectSURFFeatures` to obtain SURF points.

### **N** — Number of points

integer

Number of points, specified as an integer.

### **imageSize** — Size of image

2-element vector | 3-element vector

Size of image, specified as a 2- or 3-element vector.

# Examples

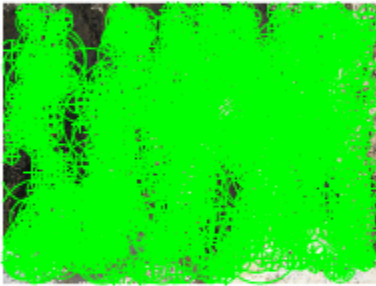
### Select A Uniformly Distributed Subset of Features From an Image

Load an image.

```
im = imread('yellowstone_left.png');
```

Detect and display SURF features.

```
points1 = detectSURFFeatures(rgb2gray(im));  
subplot(1,2,1);  
imshow(im);  
hold on  
plot(points1);  
hold off  
title('Original points');
```

**Original points**

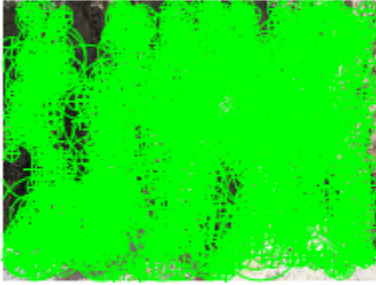
Select a uniformly distributed subset of points.

```
numPoints = 100;  
points2 = selectUniform(points1,numPoints,size(im));
```

Plot images showing original and subset of points.

```
subplot(1, 2, 2);  
imshow(im);  
hold on  
plot(points2);  
hold off  
title('Uniformly distributed points');
```

**Original points**



**Uniformly distributed points**



## size

**Class:** SURFPoints

Size of the SURFPoints object

## Syntax

```
size(SURFPointsObj)
```

## Description

`size(SURFPointsObj)` returns the size of the SURFPointsObj object.

# vision.AlphaBlender System object

**Package:** vision

Combine images, overlay images, or highlight selected pixels

## Description

The AlphaBlender object combines two images, overlays one image over another, or highlights selected pixels.

Use the step syntax below with input images, I1 and I2, the alpha blender object, H, and any optional properties.

`Y = step(H, I1, I2)` performs the alpha blending operation on images I1 and I2.

`Y = step(H, I1, I2, OPACITY)` uses *OPACITY* input to combine pixel values of I1 and I2 when you set the Operation property to Blend and the OpacitySource property to Input port.

`Y = step(H, I1, I2, MASK)` uses *MASK* input to overlay I2 over I1 when you set the Operation property to Binary mask and the MaskSource property to Input port'.

`Y = step(H, I1, MASK)` uses *MASK* input to determine which pixels in I1 are set to the maximum value supported by their data type when you set the Operation property to Highlight selected pixels and the MaskSource property to Input port.

`Y = step(H, I1, I2, ..., LOCATION)` uses *LOCATION* input to specify the upper-left corner position of I2 when you set the LocationSource property to Input port.

## Construction

`H = vision.AlphaBlender` returns an alpha blending System object™, H, that combines the pixel values of two images, using an opacity factor of 0.75.

`H = vision.AlphaBlender(Name, Value)` returns an alpha blending object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

<b>Code Generation Support</b>
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### Operation

Operation to perform

Specify the operation that the object performs as **Blend**, **Binary mask**, or **Highlight selected pixels**. If you set this property to **Blend**, the object linearly combines the pixels of one image with another image. If you set this property to **Binary mask**, the object overwrites the pixel values of one image with the pixel values of another image. If you set this property to **Highlight selected pixel**, the object uses the binary image input, **MASK**, to determine which pixels are set to the maximum value supported by their data type.

### OpacitySource

Source of opacity factor

Specify how to determine any opacity factor as **Property** or **Input port**. This property applies when you set the ??? property to **Blend**. The default is **Property**.

### Opacity

Amount by which the object scales each pixel value before combining them

Specify the amount by which the object scales each pixel value before combining them. Determine this value before combining pixels as a scalar value used for all pixels, or a matrix of values that defines the factor for each pixel. This property applies when you set the ??? property to **Property**. This property is tunable. The default is **0.75**.

### MaskSource

Source of binary mask

Specify how to determine any masking factors as **Property** or **Input port**. This property applies when you set the ??? property to **Binary mask**. The default is **Property**.

### **Mask**

Which pixels are overwritten

Specify which pixels are overwritten as a binary scalar **0** or **1** used for all pixels, or a matrix of **0s** and **1s** that defines the factor for each pixel. This property applies when you set the ??? property to **Property**. This property is tunable. The default is **1**.

### **LocationSource**

Source of location of the upper-left corner of second input image

Specify how to enter location of the upper-left corner of second input image as **Property** or **Input port**. The default is **Property**.

### **Location**

Location [x y] of upper-left corner of second input image relative to first input image

Specify the row and column position of upper-left corner of the second input image relative to upper-left corner of first input image as a two-element vector. This property applies when you set the ??? property to **Property**. The default is [1 1]. This property is tunable.

See “Coordinate Systems” for a discussion on pixel coordinates and spatial coordinates, which are the two main coordinate systems used in the Computer Vision System Toolbox.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Floor**.

#### **OverflowAction**

Overflow action for fixed-point operations



Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

### **OpacityDataType**

Opacity word and fraction lengths

Specify the opacity factor fixed-point data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

### **CustomOpacityDataType**

Opacity word and fraction lengths

Specify the opacity factor fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numerictype([], 16)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input` or `Custom`. The default is `Custom`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numerictype([], 32, 10)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Same as first input`, or `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 10)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as first input` or `Custom`. The default is `Same as first input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 10)`.

## **Methods**

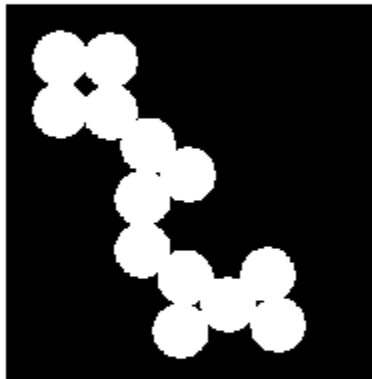
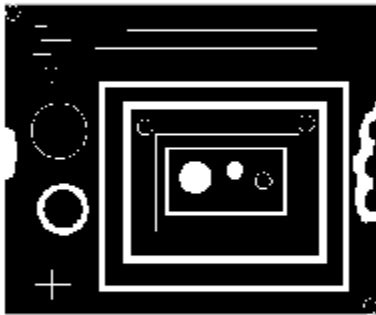
<code>clone</code>	Create alpha blender object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Blend images, overlay images, or highlight selected pixels

## **Examples**

### **Blend Two Images**

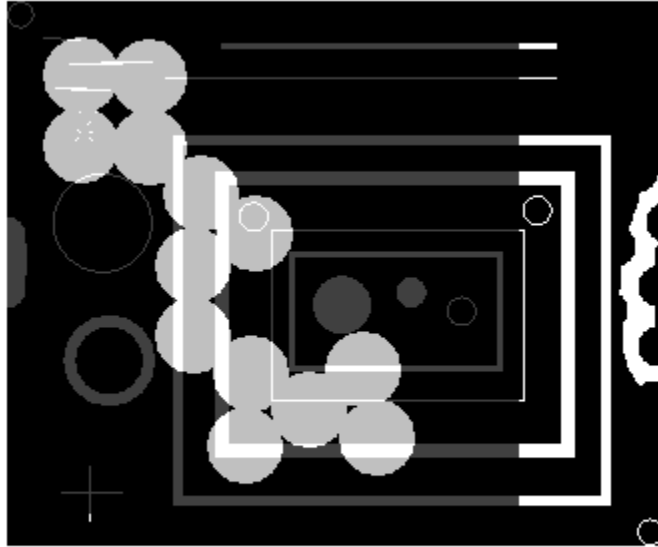
Display the two images.

```
I1 = im2single(imread('blobs.png'));  
I2 = im2single(imread('circles.png'));  
subplot(1,2,1);  
imshow(I1);  
subplot(1,2,2);  
imshow(I2);
```



Blend the two images and display the result.

```
halphablend = vision.AlphaBlender;  
J = step(halphablend,I1,I2);  
figure;  
imshow(J);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Compositing](#) block reference page. The object properties correspond to the block parameters.

## See Also

[insertText](#) | [vision.ShapeInserter](#) | [vision.MarkerInserter](#)

**Introduced in R2012a**

# clone

**System object:** vision.AlphaBlender

**Package:** vision

Create alpha blender object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.AlphaBlender

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.AlphaBlender

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.AlphaBlender

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the AlphaBlender System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



# release

**System object:** vision.AlphaBlender

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.AlphaBlender

**Package:** vision

Blend images, overlay images, or highlight selected pixels

## Syntax

`Y = step(H, I1, I2)`

`Y = step(H, I1, I2, OPACITY)`

`Y = step(H, I1, I2, MASK)`

`Y = step(H, I1, MASK)`

`Y = step(H, I1, I2, ..., LOCATION)`

## Description

`Y = step(H, I1, I2)` performs the alpha blending operation on images *I1* and *I2*.

`Y = step(H, I1, I2, OPACITY)` uses *OPACITY* input to combine pixel values of *I1* and *I2* when you set the *Operation* property to *Blend* and the *OpacitySource* property to *Input port*.

`Y = step(H, I1, I2, MASK)` uses *MASK* input to overlay *I2* over *I1* when you set the *Operation* property to *Binary mask* and the *MaskSource* property to *Input port*'.

`Y = step(H, I1, MASK)` uses *MASK* input to determine which pixels in *I1* are set to the maximum value supported by their data type when you set the *Operation* property to *Highlight selected pixels* and the *MaskSource* property to *Input port*.

`Y = step(H, I1, I2, ..., LOCATION)` uses *LOCATION* input to specify the upper-left corner position of *I2* when you set the *LocationSource* property to *Input port*.

---

**Note:** *H* specifies the System object on which to run this *step* method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.Autocorrelator System object

**Package:** vision

Compute 2-D autocorrelation of input matrix

### Description

The `Autocorrelator` object computes 2-D autocorrelation of input matrix.

### Construction

`H = vision.Autocorrelator` returns a System object, `H`, that performs two-dimensional auto-correlation of an input matrix.

`H = vision.Autocorrelator(Name, Value)` returns a 2-D autocorrelation System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Fixed-Point Properties

#### AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Same as input`, or `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numericType([],32,30)`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numericType([],16,15)`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numericType([],32,30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input`, `Custom`. The default is `Same as input`.

### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

## **Methods**

<code>clone</code>	Create 2-D autocorrelator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute cross-correlation of two input matrices

## **Examples**

### **Compute the 2D Autocorrelation of a Matrix**

```
hac2d = vision.Autocorrelator;  
x = [1 2;2 1];  
Y = step(hac2d, x)
```

Y =

```
1    4    4  
4   10    4  
4    4    1
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [2-D Autocorrelation](#) block reference page. The object properties correspond to the block parameters.

### See Also

[vision.Crosscorrelator](#)

**Introduced in R2012a**

# clone

**System object:** vision.Autocorrelator

**Package:** vision

Create 2-D autocorrelator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



## getNumInputs

**System object:** vision.Autocorrelator

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Autocorrelator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Autocorrelator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Autocorrelator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.Autocorrelator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.Autocorrelator

**Package:** vision

Compute cross-correlation of two input matrices

## Syntax

$Y = \text{step}(H, X)$

## Description

$Y = \text{step}(H, X)$  returns the 2-D autocorrelation,  $Y$ , of input matrix  $X$ .

Calling `step` on an unlocked System object will lock that object.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.Autothresher System object

**Package:** vision

Convert intensity image to binary image

### Description

---

**Note:** The `vision.Autothresher` System object will be removed in a future release. Use the `graythresh` or the `multithresh` function with equivalent functionality instead.

---

Convert intensity images to binary images. Autothresholding uses Otsu's method, which determines the threshold by splitting the histogram of the input image to minimize the variance for each of the pixel groups.

### Construction

`H = vision.Autothresher` returns a System object, `H`, that automatically converts an intensity image to a binary image.

`H = vision.Autothresher(Name, Value)` returns an autothreshold object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Operator

Threshold operator on input matrix values

Specify the condition the object places on the input matrix values as one of `>` | `<=` . The default is `>`.

If you set this property to `>`, and the step method inputs a value greater than the threshold value, the object outputs 1. If the step method inputs a value less than the threshold value, the object outputs 0.

If you set this property to `<=`, and the step method inputs a value less than or equal to the threshold value, the object outputs 1. If the step method inputs a value greater than the threshold value, the object outputs 0.

### **ThresholdOutputPort**

Enable threshold output

Set this property to `true` to enable the output of the calculated threshold values.

The default is `false`.

### **EffectivenessOutputPort**

Enable threshold effectiveness output

Set this property to `true` to enable the output of the effectiveness of the thresholding. The default is `false`. This effectiveness metric ranges from 0 to 1. If every pixel has the same value, the object sets effectiveness metric to 0. If the image has two pixel values, or the histogram of the image pixels is symmetric, the object sets the effectiveness metric to 1.

### **InputRangeSource**

Source of input data range

Specify the input data range as one of `Auto` | `Property`. The default is `Auto`. If you set this property to `Auto`, the object assumes an input range between 0 and 1, inclusive, for floating point data types. For all other data types, the object sets the input range to the full range of the data type. To specify a different input data range, set this property to `Property`.

### **InputRange**

Input data range

Specify the input data range as a two-element numeric row vector. First element of the input data range vector represents the minimum input value while the second element represents the maximum value. This property applies when you set the ??? property to Property.

### **InputRangeViolationAction**

Behavior when input values are out of range

Specify the object's behavior when the input values are outside the expected data range as one of `Ignore` | `Saturate`. The default is `Saturate`. This property applies when you set the ??? property to Property.

### **ThresholdScaleFactor**

Threshold scale factor

Specify the threshold scale factor as a numeric scalar greater than 0. The default is 1. The object multiplies this scalar value with the threshold value computed by Otsu's method. The result becomes the new threshold value. The object does not do threshold scaling. This property is tunable.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

#### **Product1DataType**

Product-1 word and fraction lengths

This is a constant property with value `Custom`.



**CustomProduct1DataType**

Product-1 word and fraction lengths

Specify the product-1 fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. The default is `numericType([],32)`.

**Accumulator1DataType**

Accumulator-1 word and fraction lengths

Specify the accumulator-1 fixed-point data type as `Same as product 1, Custom`. The default is `Same as product 1`.

**CustomAccumulator1DataType**

Accumulator-1 word and fraction lengths

Specify the accumulator-1 fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32)`.

**Product2DataType**

Product-2 word and fraction lengths

This is a constant property with value `Custom`.

**CustomProduct2DataType**

Product-2 word and fraction lengths

Specify the product-2 fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. The default is `numericType([],32)`.

**Accumulator2DataType**

Accumulator-2 word and fraction lengths

Specify the accumulator-2 fixed-point data type as `Same as product 2, Custom`. The default is `Same as product 2`.

### **CustomAccumulator2DataType**

Accumulator-2 word and fraction lengths

Specify the accumulator-2 fixed-point type as a signed `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numerictype([],32)`.

### **Product3DataType**

Product-3 word and fraction lengths

This is a constant property with value `Custom`.

### **CustomProduct3DataType**

Product-3 word and fraction lengths

Specify the product-3 fixed-point type as a signed `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],32)`.

### **Accumulator3DataType**

Accumulator-3 word and fraction lengths

Specify the accumulator-3 fixed-point data type as `Same as product 3, Custom`. This property applies when you set the `???` property to `true`. The default is `Same as product 3`.

### **CustomAccumulator3DataType**

Accumulator-3 word and fraction lengths

Specify the accumulator-3 fixed-point type as a signed `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true`, and when you set the `???` property to `Custom`. The default is `numerictype([],32)`.

### **Product4DataType**

Product-4 word and fraction lengths

Specify the product-4 fixed-point data type as `Same as input`, or `Custom`. The default is `Custom`.

**CustomProduct4DataType**

Product-4 word and fraction lengths

Specify the product-4 fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default value is `numericType([],32,15)`.

**Accumulator4DataType**

Accumulator-4 word and fraction lengths

Specify the accumulator-4 fixed-point data type as `Same as product 4 | Custom`. The default is `Same as product 4`.

**CustomAccumulator4DataType**

Accumulator-4 word and fraction lengths

Specify the accumulator-4 fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],16,4)`.

**QuotientDataType**

Quotient word and fraction lengths

Specify the quotient fixed-point data type as `Custom`.

**CustomQuotientDataType**

Quotient word and fraction lengths

Specify the quotient fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32)`.

**EffectivenessDataType**

Effectiveness metric word and fraction lengths

This is a constant property with value `Custom`. This property applies when you set the `???` property to `true`.

### CustomEffectivenessDataType

Effectiveness metric word and fraction lengths

Specify the effectiveness metric fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true`. The default is `numericType([],16)`.

## Methods

<code>clone</code>	Create autothresher object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Convert input intensity image to binary image

## Examples

### Convert an Image to a Binary Image

```
img = im2single(rgb2gray(imread('peppers.png')));  
imshow(img);  
hautoth = vision.Autothresher;  
bin = step(hautoth,img);  
pause(2);  
figure;imshow(bin);
```

Warning: The `vision.Autothresher` will be removed in a future release. Use the `multithresh` and `imquantize` functions with equivalent functionality instead.





## Algorithms

This object implements the algorithm, inputs, and outputs described on the Autothreshold block reference page. The object properties correspond to the block parameters, except:

You can only specify a value of **Ignore** or **Saturate** for the `InputRangeViolationAction` property of the `System` object. The object does not support the **Error** and **Warn** and **Saturate** options that the corresponding **When data range is exceeded** block parameter offers.

## **See Also**

vision.ColorSpaceConverter

**Introduced in R2012a**

# clone

**System object:** vision.Autothresher

**Package:** vision

Create autothresher object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



## getNumInputs

**System object:** vision.Autothresher

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Autothresher

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.Autothresher

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Autothresher System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.Autothresher

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.Autothresher

**Package:** vision

Convert input intensity image to binary image

## Syntax

```
BW = step(H,I)
[BW,TH] = step(H,I)
[... ,EMETRIC] = step(H,I)
```

## Description

`BW = step(H,I)` converts input intensity image, `I`, to a binary image, `BW`.

`[BW,TH] = step(H,I)` also returns the threshold, `TH`, when the you set the `ThresholdOutputPort` property to `true`.

`[... ,EMETRIC] = step(H,I)` also returns `EMETRIC`, a metric indicating the effectiveness of thresholding the input image when you set the `EffectivenessOutputPort` property to `true`. When the `step` method inputs an image having a single gray level, the object can attain the lower bound of the metric (zero). When the `step` method inputs a two-valued image, the object can attain the upper bound (one).

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.BinaryFileReader System object

**Package:** vision

Read video data from binary files

### Description

---

**Note:** The `vision.BinaryFileReader` System object will be removed in a future release.

---

The `BinaryFileReader` object reads video data from binary files.

### Construction

`H = vision.BinaryFileReader` returns a System object, `H`, that reads binary video data from the specified file in I420 Four Character Code (FOURCC) video format.

`H = vision.BinaryFileReader(Name, Value)` returns a binary file reader System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

`H = vision.BinaryFileReader(FILE, Name, Value)` returns a binary file reader System object, `H`, with the `???` property set to `FILE` and other specified properties set to the specified values.

### Properties

#### Filename

Name of binary file to read from

Specify the name of the binary file as a string. The full path for the file needs to be specified only if the file is not on the MATLAB path. The default is `vipmen.bin`.

**VideoFormat**

Format of binary video data

Specify the format of the binary video data as `Four character codes`, or `Custom`. The default is `Four character codes`.

**FourCharCode**

Four Character Code video format

Specify the binary file format from the available list of Four Character Code video formats. For more information on Four Character Codes, see <http://www.fourcc.org>. This property applies when you set the `???` property to `Four character codes`.

**BitstreamFormat**

Format of data as planar or packed

Specify the data format as `Planar` or `Packed`. This property applies when you set the `???` property to `Custom`. The default is `Planar`.

**OutputSize**

Size of output matrix

Specify the size of the output matrix. This property applies when you set the `???` property to `Packed`.

**VideoComponentCount**

Number of video components in video stream

Specify the number of video components in the video stream as `1`, `2`, `3` or `4`. This number corresponds to the number of video component outputs. This property applies when you set the `???` property to `Custom`. The default is `3`.

**VideoComponentBits**

Bit size of video components

Specify the bit sizes of video components as an integer valued vector of length  $N$ , where  $N$  is the value of the `???` property. This property applies when you set the `???` property to `Custom`. The default is `[8 8 8]`.

### **VideoComponentSizes**

Size of output matrix

Specify the size of the output matrix. This property must be set to an *N*-by-2 array, where *N* is the value of the ??? property. Each row of the matrix corresponds to the size of that video component, with the first element denoting the number of rows and the second element denoting the number of columns. This property applies when you set the ??? property to **Custom** and the **BitstreamFormat** property to **Planar**. The default is [ 120 160; 60 80; 60 80].

### **VideoComponentOrder**

Specify how to arrange the components in the binary file. This property must be set to a vector of length *N*, where *N* is set according to how you set the ??? property. When you set the **BitStreamFormat** property to **Planar**, you must set *N* equal to the value of the ??? property. Otherwise, you can set *N* equal to or greater than the value of the **VideoComponentCount** property.

This property applies when you set the ??? property to **Custom**. The default is [ 1 2 3].

### **InterlacedVideo**

Whether data stream represents interlaced video

Set this property to **true** if the video stream represents interlaced video data. This property applies when you set the ??? property to **Custom**. The default is **false**.

### **LineOrder**

How to fill binary file

Specify how to fill the binary file as **Top line first**, or **Bottom line first**. If this property is set to **Top line first**, the System object first fills the binary file with the first row of the video frame. If it is set to **Bottom line first**, the System object first fills the binary file with the last row of the video frame. The default is **Top line first**.

### **SignedData**

Whether input data is signed

Set this property to **true** if the input data is signed. This property applies when you set the ??? property to **Custom**. The default is **false**.



## ByteOrder

Byte ordering as little endian or big endian

Specify the byte ordering in the output binary file as `Little endian`, `Big endian`. This property applies when you set the `???` property to `Custom`. The default is `Little endian`.

## PlayCount

Number of times to play the file

Specify the number of times to play the file as a positive integer or `inf`. The default is `1`.

## Methods

<code>clone</code>	Create binary file reader object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isDone</code>	End-of-file status (logical)
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>reset</code>	Reset to beginning of file
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Read video components from a binary file

## Examples

### Read Binary Video File and Play Back on Screen

Create a binary file reader and video player object.

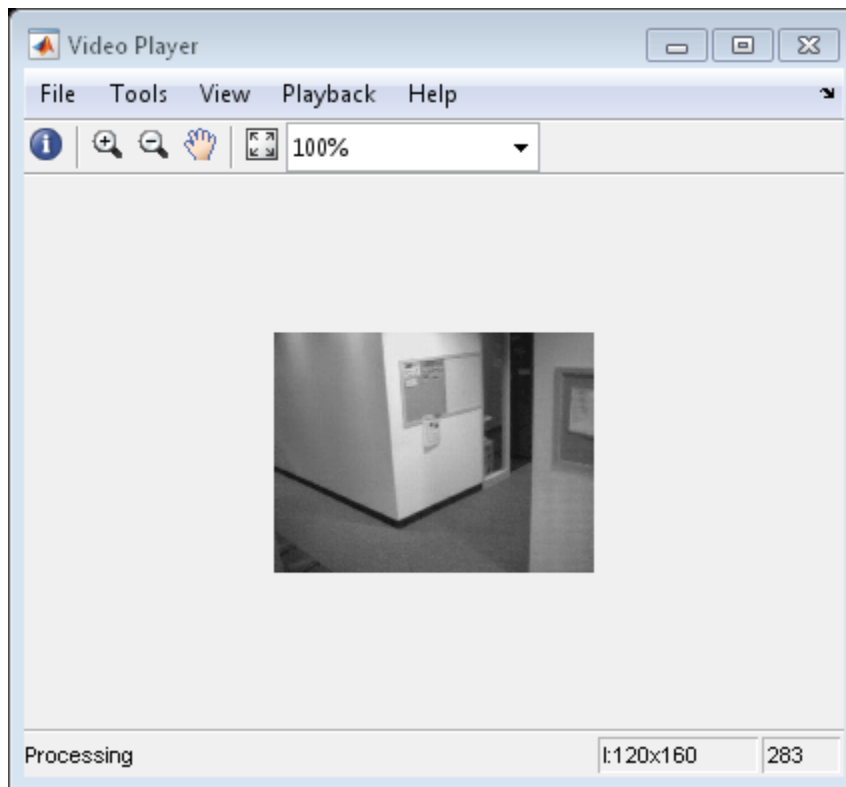
```
hbfr = vision.BinaryFileReader();
```

```
hvp = vision.VideoPlayer;
```

Warning: The `vision.BinaryFileReader` will be removed in a future release. Use the `vision.VideoFileReader` system object with equivalent functionality instead.

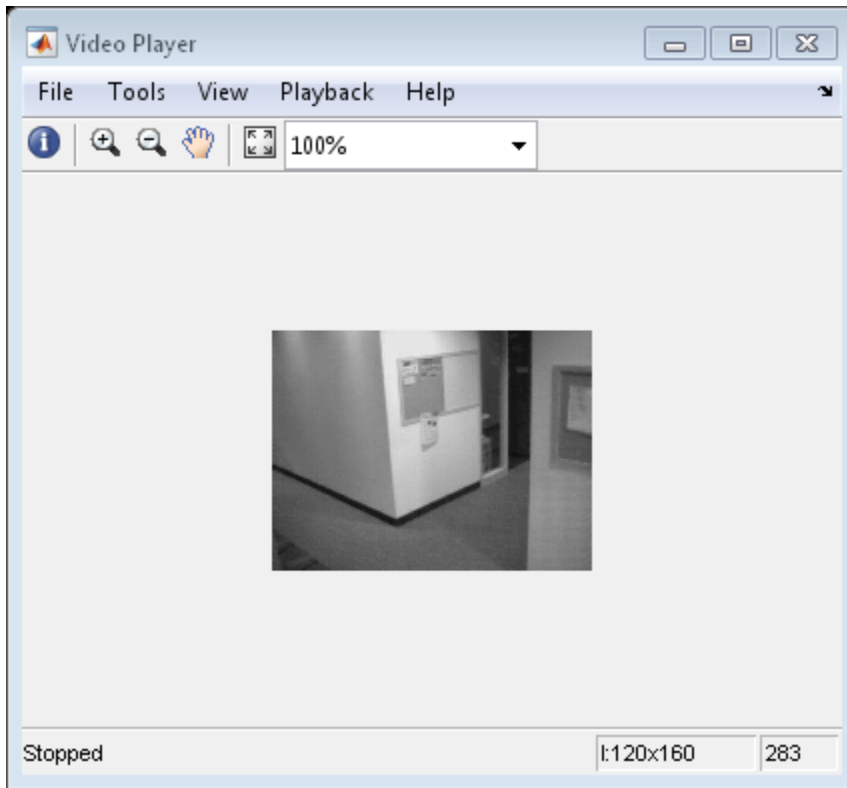
Use the while loop to play the default video.

```
while ~isDone(hbfr)
y = step(hbfr);
step(hvp, y);
end
```



Close the input file and the video display.

```
release(hbfr);
release(hvp);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Read Binary File](#) block reference page. The object properties correspond to the block parameters.

## See Also

[vision.VideoFileReader](#) | [vision.BinaryFileWriter](#)

**Introduced in R2012a**

# clone

**System object:** vision.BinaryFileReader

**Package:** vision

Create binary file reader object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.BinaryFileReader

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.BinaryFileReader

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isDone

**System object:** vision.BinaryFileReader

**Package:** vision

End-of-file status (logical)

## Syntax

TF = isDone(H)

## Description

TF = isDone(H) returns **true** if the **BinaryFileReader** System object, H , has reached the end of the binary file. If **PlayCount** property is set to a value greater than 1 , this method will return **true** every time the end is reached.

## isLocked

**System object:** vision.BinaryFileReader

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the BinaryFileReader System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



## reset

**System object:** vision.BinaryFileReader

**Package:** vision

Reset to beginning of file

## Syntax

reset(H)

## Description

reset(H) System object *H* to the beginning of the specified file.

## release

**System object:** vision.BinaryFileReader

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.BinaryFileReader

**Package:** vision

Read video components from a binary file

## Syntax

```
[Y,Cb,Cr] = step(H)
Y = step(H)
[Y,Cb] = step(H)
[Y,Cb,Cr] = step(H)
[Y,Cb,Cr,Alpha] = step(H)
[... , EOF] = step(H)
```

## Description

`[Y,Cb,Cr] = step(H)` reads the luma, *Y* and chroma, *Cb* and *Cr* components of a video stream from the specified binary file when you set the `VideoFormat` property to 'Four character codes'.

`Y = step(H)` reads the video component *Y* from the binary file when you set the `VideoFormat` property to `Custom` and the `VideoComponentCount` property to 1.

`[Y,Cb] = step(H)` reads video the components *Y* and *Cb* from the binary file when you set the `VideoFormat` property to `Custom` and the `VideoComponentCount` property to 2.

`[Y,Cb,Cr] = step(H)` reads the video components *Y*, *Cb* and *Cr* when you set the `VideoFormat` property to `Custom`, and the `VideoComponentCount` property to 3.

`[Y,Cb,Cr,Alpha] = step(H)` reads the video components *Y*, *Cb*, *Cr* and *Alpha* when you set the `VideoFormat` property to `Custom` and the `VideoComponentCount` property to 4.

`[... , EOF] = step(H)` also returns the end-of-file indicator, *EOF*. *EOF* is set to `true` each time the output contains the last video frame in the file.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.BinaryFileWriter System object

**Package:** vision

Write binary video data to files

## Description

---

**Note:** The `vision.BinaryFileWriter` System object will be removed in a future release.

---

The `BinaryFileWriter` object writes binary video data to files.

## Construction

`H = vision.BinaryFileWriter` returns a System object, `H`, that writes binary video data to an output file, `output.bin` in the I420 Four Character Code format.

`H = vision.BinaryFileWriter(Name, Value)` returns a binary file writer System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

`H = vision.BinaryFileWriter(FILE, Name, Value)` returns a binary file writer System object, `H`, with the `???` property set to `FILE` and other specified properties set to the specified values.

## Properties

### Filename

Name of binary file to write to

Specify the name of the binary file as a string. The default is the file `output.bin`.

### **VideoFormat**

Format of binary video data

Specify the format of the binary video data as **Four character codes**, or **Custom**. The default is **Four character codes**.

### **FourCharCode**

Four Character Code video format

Specify the binary file format from the available list of Four Character Code video formats. For more information on Four Character Codes, see <http://www.fourcc.org>. This property applies when you set the ??? property to **Four character codes**.

### **BitstreamFormat**

Format of data as planar or packed

Specify the data format as **Planar** or **Packed**. This property applies when you set the ??? property to **Custom**. The default is **Planar**.

### **VideoComponentCount**

Number of video components in video stream

Specify the number of video components in the video stream as **1**, **2**, **3** or **4**. This number corresponds to the number of video component outputs. This property applies when you set the ??? property to **Custom**. The default is **3**.

### **VideoComponentBitsSource**

How to specify the size of video components

Indicate how to specify the size of video components as **Auto** or **Property**. If this property is set to **Auto**, each component will have the same number of bits as the input data type. Otherwise, the number of bits for each video component is specified using the ??? property. This property applies when you set the ??? property to **Custom**. The default is **Auto**.

### **VideoComponentBits**

Bit size of video components

Specify the bit size of video components using a vector of length  $N$ , where  $N$  is the value of the ??? property. This property applies when you set the ??? property to `Property`. The default is `[8 8 8]`.

### **VideoComponentOrder**

Arrange video components in binary file

Specify how to arrange the components in the binary file. This property must be set to a vector of length  $N$ , where  $N$  is set according to how you set the ??? property. When you set the `BitStreamFormat` property to `Planar`, you must set  $N$  equal to the value of the ??? property. Otherwise, you can set  $N$  equal to or greater than the value of the `VideoComponentCount` property.

This property applies when you set the ??? property to `Custom`. The default is `[1 2 3]`.

### **InterlacedVideo**

Whether data stream represents interlaced video

Set this property to true if the video stream represents interlaced video data. This property applies when you set the ??? property to `Custom`. The default is `false`.

### **LineOrder**

How to fill binary file

Specify how to fill the binary file as `Top line first`, or `Bottom line first`. If this property is set to `Top line first`, the object first fills the binary file with the first row of the video frame. Otherwise, the object first fills the binary file with the last row of the video frame. The default is `Top line first`.

### **SignedData**

Whether input data is signed

Set this property to true if the input data is signed. This property applies when you set the ??? property to `Custom`. The default is `false`.

### **ByteOrder**

Byte ordering as little endian or big endian

Specify the byte ordering in the output binary file as `Little endian`, or `Big endian`. This property applies when you set the `???` property to `Custom`. The default is `Little endian`.

## Methods

<code>clone</code>	Create binary file writer object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Write video frame to output file

## Examples

Write video to a binary video file

```
filename = fullfile(tempdir,'output.bin');
hbfr = vision.BinaryFileReader;
hbfrw = vision.BinaryFileWriter(filename);

while ~isDone(hbfr)
    [y,cb,cr] = step(hbfr);
    step(hbfrw,y,cb,cr);
end

release(hbfr); % close the input file
release(hbfrw); % close the output file
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Write Binary File` block reference page. The object properties correspond to the block parameters.

### See Also

`vision.VideoFileWriter` | `vision.BinaryFileReader`

**Introduced in R2012a**

# clone

**System object:** vision.BinaryFileWriter

**Package:** vision

Create binary file writer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.BinaryFileWriter

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.BinaryFileWriter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.BinaryFileWriter

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the BinaryFileWriter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.BinaryFileWriter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.BinaryFileWriter

**Package:** vision

Write video frame to output file

## Syntax

```
step(H,Y,Cb,Cr)
step(H,Y)
step(H,Y,Cb)
step(H,Y,Cb,Cr)
step(H,Y,Cb,Cr,Alpha)
```

## Description

`step(H,Y,Cb,Cr)` writes one frame of video to the specified output file. `Y`, `Cb`, `Cr` represent the luma (`Y`) and chroma (`Cb` and `Cr`) components of a video stream. This option applies when you set the `VideoFormat` property to `Four character codes`.

`step(H,Y)` writes video component `Y` to the output file when the `VideoFormat` property is set to `Custom` and the `VideoComponentCount` property is set to 1.

`step(H,Y,Cb)` writes video components `Y` and `Cb` to the output file when the `VideoFormat` property is `Custom` and the `VideoComponentCount` property is set to 2.

`step(H,Y,Cb,Cr)` writes video components `Y`, `Cb` and `Cr` to the output file when the `VideoFormat` property is set to `Custom` and the `VideoComponentCount` property is set to 3.

`step(H,Y,Cb,Cr,Alpha)` writes video components `Y`, `Cb`, `Cr` and `Alpha` to the output file when the `VideoFormat` property is set to `Custom`, and the `VideoComponentCount` property is set to 4.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.BlobAnalysis System object

**Package:** vision

Properties of connected regions

## Description

The `BlobAnalysis` object computes statistics for connected regions in a binary image.

Use the `step` syntax below with input binary image, `BW`, blob analysis object, `H`, and any optional properties. The `step` method computes and returns statistics of the input binary image depending on the property values specified. The order of the returned values when there are multiple outputs are in the order they are described below:

[ `AREA`, `CENTROID`, `BBOX` ] = `step(H,BW)` returns the area, centroid and the bounding box of the blobs when the `AreaOutputPort`, `CentroidOutputPort` and `BoundingBoxOutputPort` properties are set to `true`. These are the only properties that are set to `true` by default. If you set any additional properties to `true`, the corresponding outputs follow the `AREA`, `CENTROID`, and `BBOX` outputs.

[ `___`, `MAJORAXIS` ] = `step(H,BW)` computes the major axis length `MAJORAXIS` of the blobs found in input binary image `BW` when the `MajorAxisLengthOutputPort` property is set to `true`.

[ `___`, `MINORAXIS` ] = `step(H,BW)` computes the minor axis length `MINORAXIS` of the blobs found in input binary image `BW` when the `MinorAxisLengthOutputPort` property is set to `true`.

[ `___`, `ORIENTATION` ] = `step(H,BW)` computes the `ORIENTATION` of the blobs found in input binary image `BW` when the `OrientationOutputPort` property is set to `true`.

[ `___`, `ECCENTRICITY` ] = `step(H,BW)` computes the `ECCENTRICITY` of the blobs found in input binary image `BW` when the `EccentricityOutputPort` property is set to `true`.

[ `___`, `EQDIASQ` ] = `step(H,BW)` computes the equivalent diameter squared `EQDIASQ` of the blobs found in input binary image `BW` when the `EquivalentDiameterSquaredOutputPort` property is set to `true`.

[ \_\_\_\_, EXTENT] = step(H, BW) computes the EXTENT of the blobs found in input binary image BW when the ExtentOutputPort property is set to true.

[ \_\_\_\_, PERIMETER] = step(H, BW) computes the PERIMETER of the blobs found in input binary image BW when the PerimeterOutputPort property is set to true.

[ \_\_\_\_, LABEL] = step(H, BW) returns a label matrix LABEL of the blobs found in input binary image BW when the LabelMatrixOutputPort property is set to true.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

Input/Output	Format	Supported Data Types
BW	Vector or matrix that represents a binary image	Boolean
AREA	Vector that represents the number of pixels in labeled regions	32-bit signed integer
CENTROID	$M$ -by-2 matrix of centroid coordinates, where $M$ represents the number of blobs	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> </ul>
BBOX	$M$ -by-4 matrix of [x y width height] bounding box coordinates, where $M$ represents the number of blobs and [x y] represents the upper left corner of the bounding box.	32-bit signed integer
MAJORAXIS	Vector that represents the lengths of major axes of ellipses	<ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>

Input/Output	Format	Supported Data Types
MINORAXIS	Vector that represents the lengths of minor axes of ellipses	Same as MajorAxis port
ORIENTATION	Vector that represents the angles between the major axes of the ellipses and the <i>x</i> -axis.	Same as MajorAxis port
ECCENTRICITY	Vector that represents the eccentricities of the ellipses	Same as MajorAxis port
EQDIASQ	Vector that represents the equivalent diameters squared	Same as Centroid port
EXTENT	Vector that represents the results of dividing the areas of the blobs by the area of their bounding boxes	Same as Centroid port
PERIMETER	Vector containing an estimate of the perimeter length, in pixels, for each blob	Same as Centroid port
LABEL	Label matrix	8-, 16-, or 32-bit unsigned integer

## Construction

`H = vision.BlobAnalysis` returns a blob analysis System object, `H`, used to compute statistics for connected regions in a binary image.

`H = vision.BlobAnalysis(Name, Value)` returns a blob analysis object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

## Properties

### **AreaOutputPort**

Return blob area

Setting this property to `true` outputs the area of the blobs. The default is `true`.

### **CentroidOutputPort**

Return coordinates of blob centroids

Set this property to `true` to output the coordinates of the centroid of the blobs. The default is `true`.

### **BoundingBoxOutputPort**

Return coordinates of bounding boxes

Set this property to `true` to output the coordinates of the bounding boxes. The default is `true`.

### **MajorAxisLengthOutputPort**

Return vector whose values represent lengths of ellipses' major axes

Set this property to `true` to output a vector whose values represent the lengths of the major axes of the ellipses that have the same normalized second central moments as the labeled regions. This property applies when you set the `OutputDataType` property to `double` or `single`. The default is `false`.

### **MinorAxisLengthOutputPort**

Return vector whose values represent lengths of ellipses' minor axes

Set this property to `true` to output a vector whose values represent the lengths of the minor axes of the ellipses that have the same normalized second central moments as the labeled regions. This property is available when the `OutputDataType` property is `double` or `single`. The default is `false`.

### **OrientationOutputPort**

Return vector whose values represent angles between ellipses' major axes and x-axis

Set this property to `true` to output a vector whose values represent the angles between the major axes of the ellipses and the x-axis. This property applies when you set the `OutputDataType` property to `double` or `single`. The default is `false`.

### **EccentricityOutputPort**

Return vector whose values represent ellipses' eccentricities

Set this property to `true` to output a vector whose values represent the eccentricities of the ellipses that have the same second moments as the region. This property applies when you set the `OutputDataType` property to `double` or `single`. The default is `false`.

### **EquivalentDiameterSquaredOutputPort**

Return vector whose values represent equivalent diameters squared

Set this property to `true` to output a vector whose values represent the equivalent diameters squared. The default is `false`.

### **ExtentOutputPort**

Return vector whose values represent results of dividing blob areas by bounding box areas

Set this property to `true` to output a vector whose values represent the results of dividing the areas of the blobs by the area of their bounding boxes. The default is `false`.

### **PerimeterOutputPort**

Return vector whose values represent estimates of blob perimeter lengths

Set this property to `true` to output a vector whose values represent estimates of the perimeter lengths, in pixels, of each blob. The default is `false`.

### **OutputDataType**

Output data type of statistics

Specify the data type of the output statistics as `double`, `single`, or `Fixed point`. Area and bounding box outputs are always an `int32` data type. Major axis length,

Minor axis length, Orientation and Eccentricity do not apply when you set this property to Fixed point. The default is double.

### **Connectivity**

Which pixels are connected to each other

Specify connectivity of pixels as 4 or 8. The default is 8.

### **LabelMatrixOutputPort**

Return label matrix

Set this property to `true` to output the label matrix. The default is `false`.

### **MaximumCount**

Maximum number of labeled regions in each input image

Specify the maximum number of blobs in the input image as a positive scalar integer. The maximum number of blobs the object outputs depends on both the value of this property, and on the size of the input image. The number of blobs the object outputs may be limited by the input image size. The default is 50.

### **MinimumBlobArea**

Minimum blob area in pixels

Specify the minimum blob area in pixels. The default is 0. This property is tunable.

### **MaximumBlobArea**

Maximum blob area in pixels

Specify the maximum blob area in pixels. The default is `intmax('uint32')`. This property is tunable.

### **ExcludeBorderBlobs**

Exclude blobs that contain at least one border pixel

Set this property to `true` if you do not want to label blobs that contain at least one border pixel. The default is `false`.

## **Fixed-Point Properties**

### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the `OutputDataType` property to `Fixed point`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the `OutputDataType` property to `Fixed point`.

### **ProductDataType**

Product word and fraction lengths

This property is constant and is set to `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `EquivalentDiameterSquaredOutputPort` property to `true`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `EquivalentDiameterSquaredOutputPort` property to `true`. The default is `numericType([],32,16)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

This property is constant and is set to `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point`. The default is `numericType([],32,0)`.

### **CentroidDataType**

Centroid word and fraction lengths

Specify the centroid output's fixed-point data type as `Same as accumulator`, `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `CentroidOutputPort` property to `true`. The default is `Custom`.

### **CustomCentroidDataType**

Centroid word and fraction lengths

Specify the centroid output's fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `CentroidDataType` property to `Custom` and when the `CentroidOutputPort` property to `true`. The default is `numericType([],32,16)`.

### **EquivalentDiameterSquaredDataType**

Equivalent diameter squared word and fraction lengths

Specify the equivalent diameters squared output's fixed-point data type as `Same as accumulator`, `Same as product`, `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `EquivalentDiameterSquaredOutputPort` property to `true`. The default is `Same as product`.

### **CustomEquivalentDiameterSquaredDataType**

Equivalent diameter squared word and fraction lengths

Specify the equivalent diameters squared output's fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `EquivalentDiameterSquaredOutputPort` property to `Custom` and when the `EquivalentDiameterSquaredOutputPort` property to `true`. The default is `numericType([],32,16)`.



### **ExtentDataType**

Extent word and fraction lengths

Specify the extent output's fixed-point data type as `Same as accumulator` or `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `ExtentOutputPort` property to `true`.

### **CustomExtentDataType**

Extent word and fraction lengths

Specify the extent output's fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point`, the `ExtentDataType` property to `Custom` and the `ExtentOutputPort` property to `true`. The default is `numericType([ ], 16, 14)`.

### **PerimeterDataType**

Perimeter word and fraction lengths

Specify the perimeter output's fixed-point data type as `Same as accumulator` or `Custom`. This property applies when you set the `OutputDataType` property to `Fixed point` and the `OutputDataType` property to `true`. The default is `Custom`.

### **CustomPerimeterDataType**

Perimeter word and fraction lengths

Specify the perimeter output's fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Fixed point`, the `PerimeterDataType` property to `Custom` and the `PerimeterOutputPort` property to `true`. The default is `numericType([ ], 32, 16)`.

## **Methods**

<code>clone</code>	Create blob analysis object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Compute and returns statistics of input binary image

## Examples

Find the centroid of a blob.

```
hblob = vision.BlobAnalysis;
hblob.AreaOutputPort = false;
hblob.BoundingBoxOutputPort = false;
img = logical([0 0 0 0 0 0; ...
              0 1 1 1 1 0; ...
              0 1 1 1 1 0; ...
              0 1 1 1 1 0; ...
              0 0 0 0 0 0]);
centroid = step(hblob, img); % [x y] coordinates of the centroid
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the **Blob Analysis** block reference page. The object properties correspond to the block parameters, except:

- The **Warn if maximum number of blobs is exceeded** block parameter does not have a corresponding object property. The object does not issue a warning.
- The **Output blob statistics as a variable-size signal** block parameter does not have a corresponding object property.

## See Also

graythresh | multithresh

Introduced in R2012a

# clone

**System object:** vision.BlobAnalysis

**Package:** vision

Create blob analysis object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.BlobAnalysis

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.BlobAnalysis

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.BlobAnalysis

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the BlobAnalysis System objects.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.BlobAnalysis

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.BlobAnalysis

**Package:** vision

Compute and returns statistics of input binary image

## Syntax

```
AREA = step(H,BW)
[... ,CENTROID] = step(H,BW)
[... ,BBOX] = step(H,BW)
[... ,MAJORAXIS] = step(H,BW)
[... ,MINORAXIS] = step(H,BW)
[... ,ORIENTATION] = step(H,BW)
[... ,ECCENTRICITY] = step(H,BW)
[... ,EQDIASQ] = step(H,BW)
[... ,EXTENT] = step(H,BW)
[... ,PERIMETER] = step(H,BW)
[... ,LABEL] = step(H,BW)
[... ,NUMBLOBS] = step(H,BW)
[AREA,CENTROID,BBOX] = step(H,BW)
```

## Description

`AREA = step(H,BW)` computes the **AREA** of the blobs found in input binary image **BW** when the `AreaOutputPort` property is set to `true`.

`[... ,CENTROID] = step(H,BW)` computes the **CENTROID** of the blobs found in input binary image **BW** when the `CentroidOutputPort` property is set to `true`.

`[... ,BBOX] = step(H,BW)` computes the bounding box **BBOX** of the blobs found in input binary image **BW** when the `BoundingBoxOutputPort` property is set to `true`.

`[... ,MAJORAXIS] = step(H,BW)` computes the major axis length **MAJORAXIS** of the blobs found in input binary image **BW** when the `MajorAxisLengthOutputPort` property is set to `true`.



[...,MINORAXIS] = step(H,BW) computes the minor axis length MINORAXIS of the blobs found in input binary image BW when the MinorAxisLengthOutputPort property is set to true.

[...,ORIENTATION] = step(H,BW) computes the ORIENTATION of the blobs found in input binary image BW when the OrientationOutputPort property is set to true.

[...,ECCENTRICITY] = step(H,BW) computes the ECCENTRICITY of the blobs found in input binary image BW when the EccentricityOutputPort property is set to true.

[...,EQDIASQ] = step(H,BW) computes the equivalent diameter squared EQDIASQ of the blobs found in input binary image BW when the EquivalentDiameterSquaredOutputPort property is set to true.

[...,EXTENT] = step(H,BW) computes the EXTENT of the blobs found in input binary image BW when the ExtentOutputPort property is set to true.

[...,PERIMETER] = step(H,BW) computes the PERIMETER of the blobs found in input binary image BW when the PerimeterOutputPort property is set to true.

[...,LABEL] = step(H,BW) returns a label matrix LABEL of the blobs found in input binary image BW when the LabelMatrixOutputPort property is set to true.

[...,NUMBLOBS] = step(H,BW) returns the number of blobs found in the input binary image BW when the NumBlobsOutputPort property is set to true.

[AREA,CENTROID,BBOX] = step(H,BW) returns the area, centroid and the bounding box of the blobs, when the AreaOutputPort, CentroidOutputPort and BoundingBoxOutputPort properties are set to true. You can use this to calculate multiple statistics.

The **step** method computes and returns statistics of the input binary image depending on the property values specified. The different options can be used simultaneously. The order of the returned values when there are multiple outputs are in the order they are described.

---

**Note:** H specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an

input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.BlockMatcher System object

**Package:** vision

Estimate motion between images or video frames

## Description

The `BlockMatcher` object estimates motion between images or video frames.

## Construction

`H = vision.BlockMatcher` returns a System object, `H`, that estimates motion between two images or two video frames. The object performs this estimation using a block matching method by moving a block of pixels over a search region.

`H = vision.BlockMatcher(Name, Value)` returns a block matcher System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports Code Generation: No
Supports MATLAB Function block: No
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### ReferenceFrameSource

Reference frame source

Specify the source of the reference frame as one of `Input port` | `Property`. When you set the `???` property to `Input port` a reference frame input must be specified to the `step` method of the block matcher object. The default is `Property`.

### **ReferenceFrameDelay**

Number of frames between reference and current frames

Specify the number of frames between the reference frame and the current frame as a scalar integer value greater than or equal to zero. This property applies when you set the ??? property to `Property`.

The default is 1.

### **SearchMethod**

Best match search method

Specify how to locate the block of pixels in frame  $k+1$  that best matches the block of pixels in frame  $k$ . You can specify the search method as `Exhaustive` or `Three-step`. If you set this property to `Exhaustive`, the block matcher object selects the location of the block of pixels in frame  $k+1$ . The block matcher does so by moving the block over the search region one pixel at a time, which is computationally expensive.

If you set this property to `Three-step`, the block matcher object searches for the block of pixels in frame  $k+1$  that best matches the block of pixels in frame  $k$  using a steadily decreasing step size. The object begins with a step size approximately equal to half the maximum search range. In each step, the object compares the central point of the search region to eight search points located on the boundaries of the region and moves the central point to the search point whose values is the closest to that of the central point. The object then reduces the step size by half, and begins the process again. This option is less computationally expensive, though sometimes it does not find the optimal solution.

The default is `Exhaustive`.

### **BlockSize**

Block size

Specify the size of the block in pixels.

The default is [17 17].

### **Overlap**

Input image subdivision overlap

Specify the overlap (in pixels) of two subdivisions of the input image.

The default is [0 0].

### **MaximumDisplacement**

Maximum displacement search

Specify the maximum number of pixels that any center pixel in a block of pixels can move, from image to image or from frame to frame. The block matcher object uses this property to determine the size of the search region.

The default is [7 7].

### **MatchCriteria**

Match criteria between blocks

Specify how the System object measures the similarity of the block of pixels between two frames or images. Specify as one of Mean square error (MSE) | Mean absolute difference (MAD). The default is Mean square error (MSE).

### **OutputValue**

Motion output form

Specify the desired form of motion output as one of Magnitude-squared | Horizontal and vertical components in complex form. The default is Magnitude-squared.

### **Fixed-Point Properties**

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of Same as input | Custom. The default is Custom. This property applies when you set the ??? property to Mean square error (MSE).

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Mean square error (MSE)` and the `ProductDataType` property to `Custom`.

The default is `numericType([],32,0)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Custom`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`.

The default is `numericType([],32,0)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Custom`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as an unscaled `numericType` object with a `Signedness` of `Auto`. The `numericType` object should be unsigned if the `OutputValue` property is `Magnitude-squared` and, signed if it is `Horizontal` and `vertical` components in complex form.

The default is `numericType([],8)`.

### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

## OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Saturate`

## Methods

<code>clone</code>	Create block matcher object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute motion of input image

## Examples

### Estimate Motion Using BlockMatcher

#### Read and convert RGB image to grayscale

```
img1 = im2double(rgb2gray(imread('onion.png')));
```

#### Create objects

```
hbm = vision.BlockMatcher('ReferenceFrameSource', 'Input port', 'BlockSize', [35 35], ...
    'OutputValue', 'Horizontal and vertical components in complex form');
halphablend = vision.AlphaBlender;
```

#### Offset the first image by [5 5] pixels to create second image

```
img2 = imtranslate(img1, [5 5]);
```

#### Compute motion for the two images

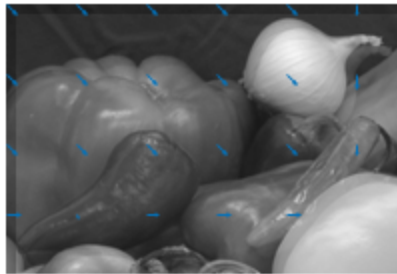
```
motion = step(hbm, img1, img2);
```

### Blend two images

```
img12 = step(halphablend, img2, img1);
```

### Use quiver plot to show the direction of motion on the images

```
[X, Y] = meshgrid(1:35:size(img1, 2), 1:35:size(img1, 1));  
imshow(img12); hold on;  
quiver(X(:), Y(:), real(motion(:)), imag(motion(:)), 0); hold off;
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Block Matching](#) block reference page. The object properties correspond to the block parameters.

### See Also

[opticalFlow](#) | [opticalFlowHS](#) | [opticalFlowLK](#) | [opticalFlowFarneback](#) | [opticalFlowLKDoG](#)

**Introduced in R2012a**



# clone

**System object:** vision.BlockMatcher

**Package:** vision

Create block matcher object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.BlockMatcher

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.BlockMatcher

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.BlockMatcher

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the BlockMatcher System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.BlockMatcher

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.BlockMatcher

**Package:** vision

Compute motion of input image

## Syntax

$V = \text{step}(H, I)$

$C = \text{step}(H, I)$

$Y = \text{step}(H, I, IREF)$

## Description

$V = \text{step}(H, I)$  computes the motion of input image  $I$  from one video frame to another, and returns  $V$  as a matrix of velocity magnitudes.

$C = \text{step}(H, I)$  computes the motion of input image  $I$  from one video frame to another, and returns  $C$  as a complex matrix of horizontal and vertical components, when you set the `OutputValue` property to `Horizontal` and `vertical` components in complex form.

$Y = \text{step}(H, I, IREF)$  computes the motion between input image  $I$  and reference image  $IREF$  when you set the `ReferenceFrameSource` property to `Input` port.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.BoundaryTracer System object

**Package:** vision

Trace object boundary

## Description

---

**Note:** The `vision.BoundaryTracer` System object will be removed in a future release. Use the `bwtraceboundary` or the `bwboundaries` function with equivalent functionality instead.

---

The boundary tracer object traces object boundaries in binary images.

Use the step syntax below with input image `BW`, starting point `STARTPT`, boundary tracer object, `H`, and any optional properties.

`PTS = step(H, BW, STARTPT)` traces the boundary of an object in a binary image, `BW`. The input matrix, `STARTPT`, specifies the starting point for tracing the boundary. `STARTPT` is a two-element vector of `[x y]` coordinates of the initial point on the object boundary. The `step` method outputs `PTS`, an  $M$ -by-2 matrix of `[x y]` coordinates of the boundary points. In this matrix,  $M$  is the number of traced boundary pixels.  $M$  is less than or equal to the value specified by the `MaximumPixelCount` property.

## Construction

`H = vision.BoundaryTracer` returns a System object, `H`, that traces the boundaries of objects in a binary image. In this image nonzero pixels belong to an object and zero-valued pixels constitute the background.

`H = vision.BoundaryTracer(Name, Value)` returns an object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

### Code Generation Support

Supports MATLAB Function block: Yes

### Code Generation Support

“Code Generation Support, Usage Notes, and Limitations”

## Properties

### Connectivity

How to connect pixels to each other

Specify which pixels are connected to each other as one of **4** | **8**. The default is **8**. Set this property to **4** to connect a pixel to the pixels on the top, bottom, left, and right. Set this property to **8** to connect a pixel to the pixels on the top, bottom, left, right, and diagonally.

### InitialSearchDirection

First search direction to find next boundary pixel

Specify the first direction in which to look to find the next boundary pixel that is connected to the starting pixel.

When you set the ??? property to **8**, this property accepts a value of **North**, **Northeast**, **East**, **Southeast**, **South**, **Southwest**, **West**, or **Northwest**.

When you set the ??? property to **4**, this property accepts a value of **North**, **East**, **South**, or **West**

### TraceDirection

Direction in which to trace the boundary

Specify the direction in which to trace the boundary as one of **Clockwise** | **Counterclockwise**. The default is **Clockwise**.

### MaximumPixelCount

Maximum number of boundary pixels

Specify the maximum number of boundary pixels as a scalar integer greater than 1. The object uses this value to preallocate the number of rows of the output matrix, **Y**. This preallocation enables the matrix to hold all the boundary pixel location values.

The default is **500**.



## NoBoundaryAction

How to fill empty spaces in output matrix

Specify how to fill the empty spaces in the output matrix,  $Y$  as one of `None` | `Fill with last point found` | `Fill with user-defined values`. The default is `None`. If you set this property to `None`, the object takes no action. Thus, any element that does not contain a boundary pixel location has no meaningful value. If you set this property to `Fill with last point found`, the object fills the remaining elements with the position of the last boundary pixel. If you set this property to `Fill with user-defined values`, you must specify the values in the `???` property.

## FillValues

Value to fill in remaining empty elements in output matrix

Set this property to a scalar value or two-element vector to fill in the remaining empty elements in the output matrix  $Y$ . This property applies when you set the `???` property to `Fill with user-defined values`.

The default is `[0 0]`.

## Methods

<code>clone</code>	Create boundary tracer object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Trace object boundary in binary image

## Examples

Trace the boundary around an image of a coin:

```
I = imread('coins.png'); % read the image
hautoth = vision.Autothresher;
BW = step(hautoth, I); % threshold the image
[y, x]= find(BW,1); % select a starting point for the trace

% Determine the boundaries
hboundtrace = vision.BoundaryTracer;
PTS = step(hboundtrace, BW, [x y]);

% Display the results
figure, imshow(BW);
hold on; plot(PTS(:,1), PTS(:,2), 'r', 'Linewidth',2);
hold on; plot(x,y,'gx','Linewidth',2); % show the starting point
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Trace Boundary](#) block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**

# clone

**System object:** vision.BoundaryTracer

**Package:** vision

Create boundary tracer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.BoundaryTracer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.BoundaryTracer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.BoundaryTracer

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the BoundaryTracer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.BoundaryTracer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.BoundaryTracer

**Package:** vision

Trace object boundary in binary image

### Syntax

PTS = step(H,BW,STARTPT)

### Description

PTS = step(H,BW,STARTPT) traces the boundary of an object in a binary image BW. The second input matrix, STARTPT, specified the starting point for tracing the boundary. STARTPT is a two-element vector of [x y] coordinates of the initial point on the object boundary. The step method outputs PTS, an  $M$ -by-2 matrix of [x y] coordinates of the boundary points, where  $M$  is the number of traced boundary pixels.  $M$  is less than or equal to the value specified by the MaximumPixelCount property.

---

**Note:** H specifies the System object on which to run this step method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---



# vision.CascadeObjectDetector System object

**Package:** vision

Detect objects using the Viola-Jones algorithm

## Description

The cascade object detector uses the Viola-Jones algorithm to detect people's faces, noses, eyes, mouth, or upper body. You can also use the Training Image Labeler to train a custom classifier to use with this System object. For details on how the function works, see "Train a Cascade Object Detector".

## Construction

`detector = vision.CascadeObjectDetector` creates a System object, `detector`, that detects objects using the Viola-Jones algorithm. The `ClassificationModel` property controls the type of object to detect. By default, the detector is configured to detect faces.

`detector = vision.CascadeObjectDetector(MODEL)` creates a System object, `detector`, configured to detect objects defined by the input string, `MODEL`. The `MODEL` input describes the type of object to detect. There are several valid `MODEL` strings, such as 'FrontalFaceCART', 'UpperBody', and 'ProfileFace'. See the ??? property description for a full list of available models.

`detector = vision.CascadeObjectDetector(XMLFILE)` creates a System object, `detector`, and configures it to use the custom classification model specified with the `XMLFILE` input. The `XMLFILE` can be created using the `trainCascadeObjectDetector` function or OpenCV (Open Source Computer Vision) training functionality. You must specify a full or relative path to the `XMLFILE`, if it is not on the MATLAB path.

`detector = vision.CascadeObjectDetector(Name, Value)` configures the cascade object detector object properties. You specify these properties as one or more name-value pair arguments. Unspecified properties have default values.

### Code Generation Support

Supports MATLAB Function block: No

### Code Generation Support

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries

“Code Generation Support, Usage Notes, and Limitations”

### To detect a feature:

- 1 Define and set up your cascade object detector using the constructor.
- 2 Call the `step` method with the input image, `I`, the cascade object detector object, `detector`, points `PTS`, and any optional properties. See the syntax below for using the `step` method.

Use the `step` syntax with input image, `I`, the selected Cascade object detector object, and any optional properties to perform detection.

`BBOX = step(detector, I)` returns `BBOX`, an  $M$ -by-4 matrix defining  $M$  bounding boxes containing the detected objects. This method performs multiscale object detection on the input image, `I`. Each row of the output matrix, `BBOX`, contains a four-element vector, `[x y width height]`, that specifies in pixels, the upper-left corner and size of a bounding box. The input image `I`, must be a grayscale or truecolor (RGB) image.

`BBOX = step(detector, I, roi)` detects objects within the rectangular search region specified by `roi`. You must specify `roi` as a 4-element vector, `[x y width height]`, that defines a rectangular region of interest within image `I`. Set the 'UseROI' property to `true` to use this syntax.

## Properties

### ClassificationModel — Trained cascade classification model

Trained cascade classification model, specified as a comma-separated pair consisting of 'ClassificationModel' and a string. This value sets the classification model for the detector. You may set this string to an XML file containing a custom classification model, or to one of the valid model strings listed below. You can train a custom classification model using the `trainCascadeObjectDetector` function. The function can train the model using Haar-like features, histograms of oriented gradients (HOG), or local binary

patterns (LBP). For details on how to use the function, see “Train a Cascade Object Detector”.

### Frontal Face (CART)

ClassificationModel String	Image Size Used to Train Model	Model Description
FrontalFaceCART (Default)	[20 20]	Detects faces that are upright and forward facing. This model is composed of weak classifiers, based on the classification and regression tree analysis (CART). These classifiers use Haar features to encode facial features. CART-based classifiers provide the ability to model higher-order dependencies between facial features. [1]

### Frontal Face (LBP)

ClassificationModel String	Image Size Used to Train Model	Model Description
FrontalFaceLBP	[24 24]	Detects faces that are upright and forward facing. This model is composed of weak classifiers, based on a decision stump. These classifiers use local binary patterns (LBP) to encode facial features. LBP features can provide robustness against variation in illumination.[2]

### Upper Body

ClassificationModel String	Image Size Used to Train Model	Model Description
UpperBody	[18 22]	Detects the upper-body region, which is defined as the head and shoulders area. This model uses Haar features to encode the details of the head and shoulder region. Because

ClassificationModel String	Image Size Used to Train Model	Model Description
		it uses more features around the head, this model is more robust against pose changes, e.g. head rotations/tilts. [3]

### Eye Pair

ClassificationModel String	Image Size Used to Train Model	Model Description
EyePairBig EyePairSmall	[11 45] [5 22]	Detects a pair of eyes. The EyePairSmall model is trained using a smaller image. This enables the model to detect smaller eyes than the EyePairBig model can detect.[4]

### Single Eye

ClassificationModel String	Image Size Used to Train Model	Model Description
LeftEye RightEye	[12 18]	Detects the left and right eye separately. These models are composed of weak classifiers, based on a decision stump These classifiers use Haar features to encode details.[4]

### Single Eye (CART)

ClassificationModel String	Image Size Used to Train Model	Model Description
LeftEyeCART RightEyeCART	[20 20]	Detects the left and right eye separately. The weak classifiers that make up these models are CART-trees. Compared to decision stumps, CART-tree-based classifiers are better able to model higher-order dependencies. [5]

**Profile Face**

ClassificationModel String	Image Size Used to Train Model	Model Description
ProfileFace	[20 20]	Detects upright face profiles. This model is composed of weak classifiers, based on a decision stump. These classifiers use Haar features to encode face details.

**Mouth**

ClassificationModel String	Image Size Used to Train Model	Model Description
Mouth	[15 25]	Detects the mouth. This model is composed of weak classifiers, based on a decision stump, which use Haar features to encode mouth details.[4]

**Nose**

ClassificationModel String	Image Size Used to Train Model	Model Description
Nose	[15 18]	This model is composed of weak classifiers, based on a decision stump, which use Haar features to encode nose details.[4]

Default: FrontalFaceCART

**MinSize — Size of smallest detectable object**

Size of smallest detectable object, specified as a comma-separated pair consisting of 'MinSize' and a two-element [*height width*] vector. Set this property in pixels for the minimum size region containing an object. It must be greater than or equal to the image size used to train the model. Use this property to reduce computation time when you know the minimum object size prior to processing the image. When you do not specify a value for this property, the detector sets it to the size of the image used to train the classification model. This property is tunable.

For details explaining the relationship between setting the size of the detectable object and the `ScaleFactor` property, see “Algorithms” on page 2-309 section.

Default: [ ]

### **MaxSize — Size of largest detectable object**

Size of largest detectable object, specified as a comma-separated pair consisting of 'MaxSize' and a two-element [*height width*] vector. Specify the size in pixels of the largest object to detect. Use this property to reduce computation time when you know the maximum object size prior to processing the image. When you do not specify a value for this property, the detector sets it to `size(I)`. This property is tunable.

For details explaining the relationship between setting the size of the detectable object and the `ScaleFactor` property, see the “Algorithms” on page 2-309 section.

Default: [ ]

### **ScaleFactor — Scaling for multiscale object detection**

Scaling for multiscale object detection, specified as a comma-separated pair consisting of 'ScaleFactor' and a value greater than 1.0001. The scale factor incrementally scales the detection resolution between `MinSize` and `MaxSize`. You can set the scale factor to an ideal value using:

`size(I)/(size(I)-0.5)`

The detector scales the search region at increments between `MinSize` and `MaxSize` using the following relationship:

$search\ region = round((Training\ Size) * (ScaleFactor)^N)$

*N* is the current increment, an integer greater than zero, and *Training Size* is the image size used to train the classification model. This property is tunable.

Default: 1.1

### **MergeThreshold — Detection threshold**

Detection threshold, specified as a comma-separated pair consisting of 'MergeDetections' and a scalar integer. This value defines the criteria needed to declare a final detection in an area where there are multiple detections around an object. Groups of colocated detections that meet the threshold are merged to produce one bounding box around the target object. Increasing this threshold may help suppress

false detections by requiring that the target object be detected multiple times during the multiscale detection phase. When you set this property to 0, all detections are returned without performing thresholding or merging operation. This property is tunable.

Default: 4

### **UseROI — Use region of interest**

false (default) | true

Use region of interest, specified as a comma-separated pair consisting of 'UseROI' and a logical scalar. Set this property to true to detect objects within a rectangular region of interest within the input image using the step method.

## Methods

clone	Create cascade object detector object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Detect objects using the Viola-Jones algorithm

## Examples

### **Detect Faces in an Image Using the Frontal Face Classification Model**

**Create a detector object.**

```
faceDetector = vision.CascadeObjectDetector;
```

**Read input image.**

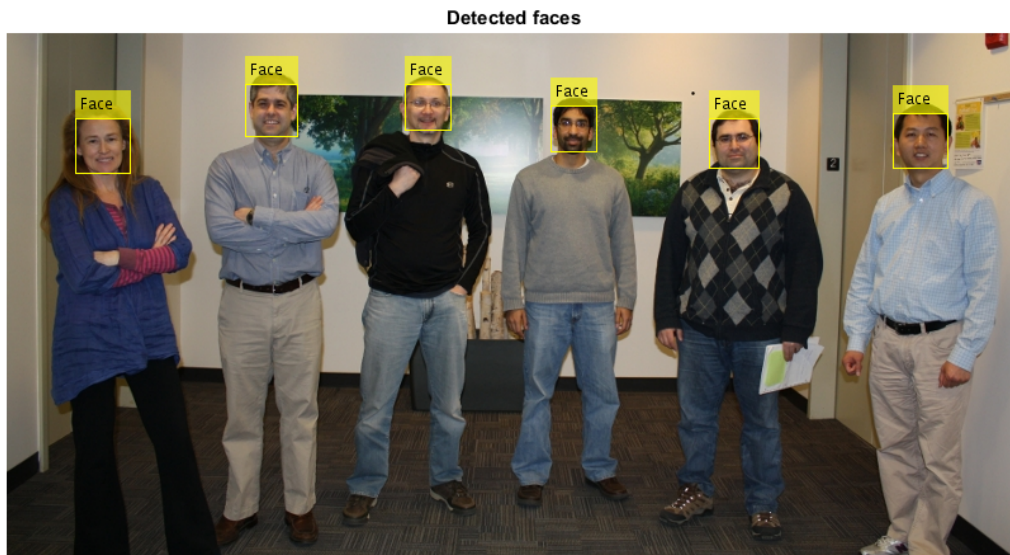
```
I = imread('visionteam.jpg');
```

### Detect faces.

```
bboxes = step(faceDetector, I);
```

### Annotate detected faces.

```
IFaces = insertObjectAnnotation(I, 'rectangle', bboxes, 'Face');  
figure, imshow(IFaces), title('Detected faces');
```



### Detect Upper Body in an Image Using the Upper Body Classification Model.

#### Create a detector object and set properties.

```
bodyDetector = vision.CascadeObjectDetector('UpperBody');  
bodyDetector.MinSize = [60 60];  
bodyDetector.MergeThreshold = 10;
```

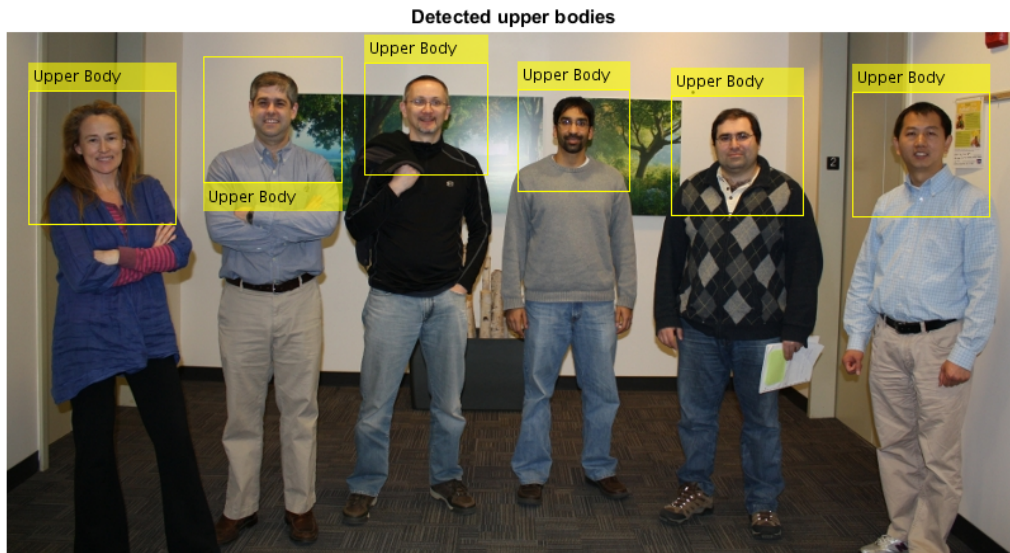
#### Read input image and detect upper body.

```
I2 = imread('visionteam.jpg');  
bboxBody = step(bodyDetector, I2);
```



### Annotate detected upper bodies.

```
Ibody = insertObjectAnnotation(I2, 'rectangle', bboxBody, 'Upper Body');  
figure, imshow(Ibody), title('Detected upper bodies');
```



- “Face Detection and Tracking Using CAMShift”
- “Face Detection and Tracking Using the KLT Algorithm”
- “Face Detection and Tracking Using Live Video Acquisition”

## Algorithms

### Classification Model Training

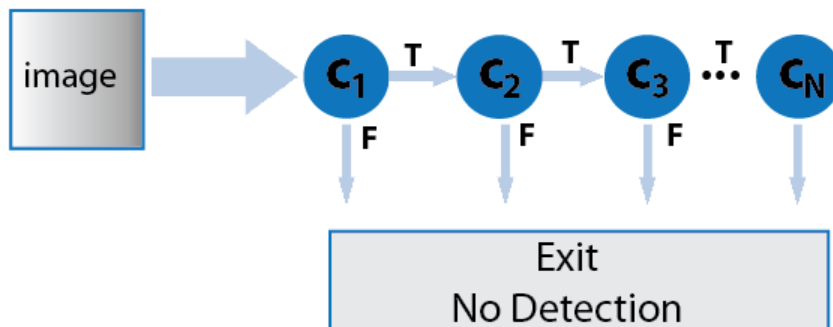
Each model is trained to detect a specific type of object. The classification models are trained by extracting features from a set of known images. These extracted features are then fed into a learning algorithm to train the classification model. Computer Vision

System Toolbox software uses the Viola-Jones cascade object detector. This detector uses HOG[7], LBP[8], and Haar-like [6] features and a cascade of classifiers trained using boosting.

The image size used to train the classifiers defines the smallest region containing the object. Training image sizes vary according to the application, type of target object, and available positive images. You must set the `???` property to a value greater than or equal to the image size used to train the model.

### Cascade of Classifiers

This object uses a cascade of classifiers to efficiently process image regions for the presence of a target object. Each stage in the cascade applies increasingly more complex binary classifiers, which allows the algorithm to rapidly reject regions that do not contain the target. If the desired object is not found at any stage in the cascade, the detector immediately rejects the region and processing is terminated. By terminating, the object avoids invoking computation-intensive classifiers further down the cascade.



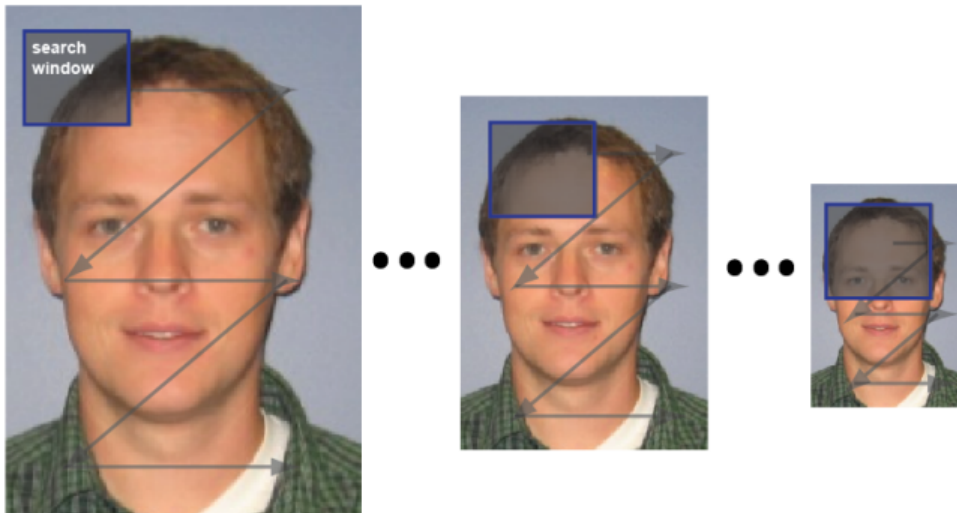
### Multiscale Object Detection

The detector incrementally scales the input image to locate target objects. At each scale increment, a sliding window, whose size is the same as the training image size, scans the scaled image to locate objects. The `???` property determines the amount of scaling between successive increments.

The search region size is related to the `ScaleFactor` in the following way:

$$\text{search region} = \text{round}((\text{ObjectTrainingSize}) * (\text{ScaleFactor}^N))$$

$N$  is the current increment, an integer greater than zero, and *ObjectTrainingSize* is the image size used to train the classification model.



The search window traverses the image for each scaled increment.

### Relationship Between MinSize, MaxSize, and ScaleFactor

Understanding the relationship between the size of the object to detect and the scale factor will help you set the properties accordingly. The `MinSize` and `MaxSize` properties limit the size range of the object to detect. Ideally, these properties are modified to reduce computation time when you know the approximate object size prior to processing the image. They are not designed to provide precise filtering of results, based on object size. The behavior of these properties is affected by the `ScaleFactor`. The scale factor determines the quantization of the search window sizes.

$$\text{search region} = \text{round}((\text{Training Size}) * (???)^N)$$

The actual range of returned object sizes may not be exactly what you select for the `???` and `???` properties. For example, a `ScaleFactor` value of 1.1 with a 24x24 training size, and an `N` value from 1 to 5, the search region calculation would be:

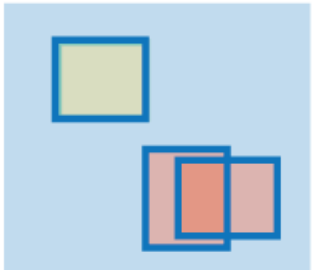
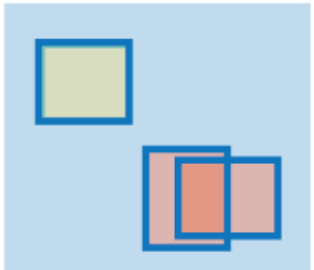
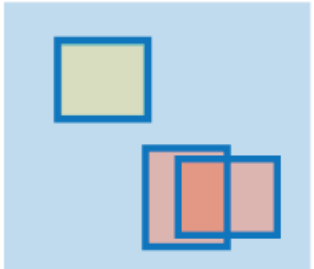
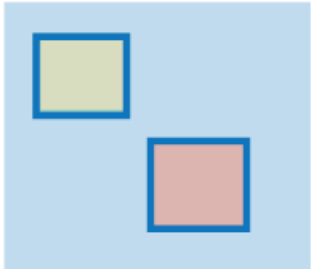
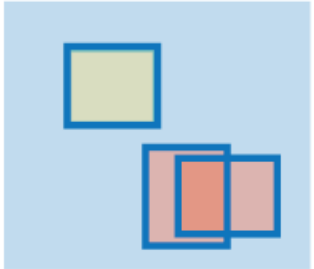
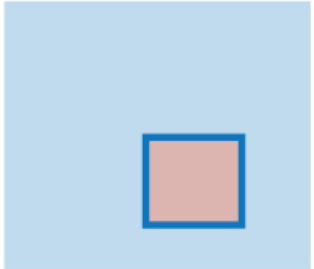
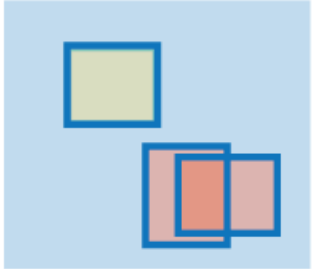

For a `ScaleFactor` value of 1.1 with a 24x24 training size, for 5 increments, the search region calculation would be:

```
>> search region = round(24*1.1.^(1:5))
>> 26 29 32 35 39
```

If you were to set `MaxSize` to 34, due to the search region quantization, the actual maximum object size used by the algorithm would be 32.

### **Merge Detection Threshold**

For each increment in scale, the search window traverses over the image producing multiple detections around the target object. The multiple detections are merged into one bounding box per target object. You can use the `mergeThreshold` property to control the number of detections required before combining or rejecting the detections. The size of the final bounding box is an average of the sizes of the bounding boxes for the individual detections and lies between `MinSize` and `MaxSize`.

MergeThreshold	Detections	Returned Bounding Boxes
0		
1		
2		
3		

## References

- [1] Lienhart R., Kuranov A., and V. Pisarevsky “Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection.”, *Proceedings of the 25th DAGM Symposium on Pattern Recognition*. Magdeburg, Germany, 2003.
- [2] Ojala Timo, Pietikäinen Matti, and Mäenpää Topi, “Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns” . In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2002. Volume 24, Issue 7, pp. 971-987.
- [3] Kruppa H., Castrillon-Santana M., and B. Schiele. “Fast and Robust Face Finding via Local Context”. *Proceedings of the Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, 2003, pp. 157–164.
- [4] Castrillón Marco, Déniz Oscar, Guerra Cayetano, and Hernández Mario, “ENCARA2: Real-time detection of multiple faces at different resolutions in video streams”. In *Journal of Visual Communication and Image Representation*, 2007 (18) 2: pp. 130-140.
- [5] Yu Shiqi “Eye Detection.” Shiqi Yu’s Homepage.
- [6] Viola, Paul and Michael J. Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features”, *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001. Volume: 1, pp.511–518.
- [7] Dalal, N., and B. Triggs, “Histograms of Oriented Gradients for Human Detection”. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Volume 1, (2005), pp. 886–893.
- [8] Ojala, T., M. Pietikainen, and T. Maenpaa, “Multiresolution Gray-scale and Rotation Invariant Texture Classification With Local Binary Patterns”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Volume 24, No. 7 July 2002, pp. 971–987.

## See Also

vision.PeopleDetector | integralImage | trainCascadeObjectDetector | Training Image Labeler | vision.ShapeInserter

## **More About**

- “Label Images for Classification Model Training”
- “Train a Cascade Object Detector”
- “Multiple Object Tracking”

## **External Websites**

- Shiqi Yu’s Homepage
- Detect and Track Multiple Faces in a Live Video Stream

## **Introduced in R2012a**

# clone

**System object:** vision.CascadeObjectDetector

**Package:** vision

Create cascade object detector object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



# getNumInputs

**System object:** vision.CascadeObjectDetector

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.CascadeObjectDetector

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.CascadeObjectDetector

**Package:** vision

Locked status for input attributes and non-tunable properties

### Syntax

TF = isLocked(H)

### Description

TF = isLocked(H) returns the locked status, TF of the CascadeObjectDetector System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.CascadeObjectDetector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You cannot use the release method on System objects in Embedded MATLAB.

---

---

## step

**System object:** vision.CascadeObjectDetector

**Package:** vision

Detect objects using the Viola-Jones algorithm

## Syntax

```
BBOX = step(detector,I)
[ ___ ] = step(detector,I,roi)
```

## Description

`BBOX = step(detector,I)` returns **BBOX**, an  $M$ -by-4 matrix defining  $M$  bounding boxes containing the detected objects. This method performs multi-scale object detection on the input image, **I**.

Each row of the output matrix, **BBOX**, contains a four-element vector, `[x y width height]`, that specifies in pixels, the upper left corner and size of a bounding box. The input image **I**, must be a grayscale or truecolor (RGB) image.

`[ ___ ] = step(detector,I,roi)` detects objects within the rectangular search region specified by **roi**. You must specify **roi** as a 4-element vector, `[x y width height]`, that defines a rectangular region of interest within image **I**. Set the 'UserROI' property to **true** to use this syntax.

---

**Note:** **H** specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.ChromaResampler System object

**Package:** vision

Downsample or upsample chrominance components of images

### Description

The `ChromaResampler` object downsamples or upsamples chrominance components of images.

### Construction

`H = vision.ChromaResampler` returns a chroma resampling System object, `H`, that downsamples or upsamples chroma components of a YCbCr signal to reduce the bandwidth and storage requirements.

`H = vision.ChromaResampler(Name, Value)` returns a chroma resampling System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Resampling

Resampling format

To downsample the chrominance components of images, set this property to one of the following:

[4:4:4 to 4:2:2]

[4:4:4 to 4:2:0 (MPEG1)]  
[4:4:4 to 4:2:0 (MPEG2)]  
[4:4:4 to 4:1:1]  
[4:2:2 to 4:2:0 (MPEG1)]  
[4:2:2 to 4:2:0 (MPEG2)]

To upsample the chrominance components of images, set this property to one of the following:

[4:2:2 to 4:4:4]  
[4:2:0 (MPEG1) to 4:4:4]  
[4:2:0 (MPEG2) to 4:4:4]  
[4:1:1 to 4:4:4]  
[4:2:0 (MPEG1) to 4:2:2]  
[4:2:0 (MPEG2) to 4:2:2]

The default is [4:4:4 to 4:2:2]

### **InterpolationFilter**

Method used to approximate missing values

Specify the interpolation method used to approximate the missing chrominance values as one of `Pixel replication` | `Linear`. The default is `Linear`. When you set this property to `Linear`, the object uses linear interpolation to calculate the missing values. When you set this property to `Pixel replication`, the object replicates the chrominance values of the neighboring pixels to create the upsampled image. This property applies when you upsample the chrominance values.

### **AntialiasingFilterSource**

Lowpass filter used to prevent aliasing

Specify the lowpass filter used to prevent aliasing as one of `Auto` | `Property` | `None`. The default is `Auto`. When you set this property to `Auto`, the object uses a built-in lowpass filter. When you set this property to `Property`, the coefficients of the filters are specified by the `???` and `???` properties. When you set this property to `None`, the object does not filter the input signal. This property applies when you downsample the chrominance values.

### **HorizontalFilterCoefficients**

Horizontal filter coefficients

Specify the filter coefficients to apply to the input signal. This property applies when you set the `???` property to one of `[4:4:4 to 4:2:2]` | `[4:4:4 to 4:2:0 (MPEG1)]` | `[4:4:4 to 4:2:0 (MPEG2)]` | `[4:4:4 to 4:1:1]` and the `???` property to `Property`. The default is `[0.2 0.6 0.2]`.

### **VerticalFilterCoefficients**

Vertical filter coefficients

Specify the filter coefficients to apply to the input signal. This property applies when you set the `Resampling` property to one of `[4:4:4 to 4:2:0 (MPEG1)]` | `[4:4:4 to 4:2:0 (MPEG2)]` | `[4:2:2 to 4:2:0 (MPEG1)]` | `[4:2:2 to 4:2:0 (MPEG2)]` and the `???` property to `Property`. The default is `[0.5 0.5]`.

### **TransposedInput**

Input is row-major format

Set this property to `true` when the input contains data elements from the first row first, then data elements from the second row second, and so on through the last row. Otherwise, the object assumes that the input data is stored in column-major format. The default is `false`.

## **Methods**

<code>clone</code>	Create chroma resampling object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Resample input chrominance components

## **Examples**

Resample the chrominance components of an image.



```
H = vision.ChromaResampler;  
hcsc = vision.ColorSpaceConverter;  
x = imread('peppers.png');  
x1 = step(hcsc, x);  
[Cb, Cr] = step(H, x1(:,:,2), x1(:,:,3));
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Chroma Resampling](#) block reference page. The object properties correspond to the block parameters.

### See Also

[rgb2gray](#) | [rgb2ycbcr](#) | [makecform](#) | [applycform](#)

**Introduced in R2012a**

# clone

**System object:** vision.ChromaResampler

**Package:** vision

Create chroma resampling object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ChromaResampler

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ChromaResampler

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.ChromaResampler

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ChromaResampler System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ChromaResampler

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.ChromaResampler

**Package:** vision

Resample input chrominance components

## Syntax

[Cb1,Cr1] = step(H,Cb,Cr)

## Description

[Cb1,Cr1] = step(H,Cb,Cr) resamples the input chrominance components **Cb** and **Cr** and returns **Cb1** and **Cr1** as the resampled outputs.

---

**Note:** **H** specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.ColorSpaceConverter System object

**Package:** vision

Convert color information between color spaces

### Description

---

**Note:** The `vision.ColorSpaceConverter` System object will be removed in a future release. Use the `rgb2gray`, `rgb2ycbcr`, or the `makecform` and the `applycform` functions with equivalent functionality instead.

---

The `ColorSpaceConverter` object converts color information between color spaces.

### Construction

`H = vision.ColorSpaceConverter` returns a System object, `H`, that converts color information from RGB to YCbCr using the conversion standard Rec. 601 (SDTV).

`H = vision.ColorSpaceConverter(Name, Value)` returns a color space conversion object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Conversion

Color space input and output conversion



Specify the conversion color spaces as one of the following:

[RGB to YCbCr]

[YCbCr to RGB]

[RGB to intensity]

[RGB to HSV]

[HSV to RGB]

[sRGB to XYZ]

[XYZ to sRGB]

[sRGB to L\*a\*b\*]

[L\*a\*b\* to sRGB]

Note that the R, G, B and Y (luma) signal components in the preceding color space conversions are gamma corrected. The default is [RGB to YCbCr].

### **WhitePoint**

Reference white point

Specify the reference white point as one of D50 | D55 | D65. The default is D65. These values are reference white standards known as *tristimulus* values that serve to define the color "white" to correctly map color space conversions. This property applies when you set the ??? property to [sRGB to L\*a\*b\*] or [L\*a\*b\* to sRGB].

### **ConversionStandard**

Standard for RGB to YCbCr conversion

Specify the standard used to convert the values between the RGB and YCbCr color spaces as one of Rec. 601 (SDTV) | Rec. 709 (HDTV). The default is Rec. 601 (SDTV). This property applies when you set the ??? property to [RGB to YCbCr] or [YCbCr to RGB].

### **ScanningStandard**

Scanning standard for RGB-to-YCbCr conversion

Specify the scanning standard used to convert the values between the RGB and YCbCr color spaces as one of [1125/60/2:1] | [1250/50/2:1]. The default is [1125/60/2:1]. This property applies when you set the ??? property to Rec. 709 (HDTV).

### Methods

clone	Create color space converter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert color space of input image

### Examples

Convert an image from an RGB to an intensity color space.

```
i1 = imread('pears.png');
imshow(i1);

hcsc = vision.ColorSpaceConverter;
hcsc.Conversion = 'RGB to intensity';
i2 = step(hcsc, i1);

pause(2);
imshow(i2);
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the **Color Space Conversion** block reference page. The object properties correspond to the block parameters, except:

The **Image signal** block parameter allows you to specify whether the block accepts the color video signal as **One multidimensional signal** or **Separate color signals**. The object does not have a property that corresponds to the **Image signal** block parameter. You must always provide the input image to the **step** method of the object as a single multidimensional signal.

**Introduced in R2012a**

# clone

**System object:** vision.ColorSpaceConverter

**Package:** vision

Create color space converter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ColorSpaceConverter

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ColorSpaceConverter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.ColorSpaceConverter

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ColorSpaceConverter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ColorSpaceConverter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.ColorSpaceConverter

**Package:** vision

Convert color space of input image

## Syntax

$C2 = \text{step}(H, C1)$

## Description

$C2 = \text{step}(H, C1)$  converts a multidimensional input image  $C1$  to a multidimensional output image  $C2$ .  $C1$  and  $C2$  are images in different color spaces.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.ConnectedComponentLabeler System object

**Package:** vision

Label and count the connected regions in a binary image

### Description

---

**Note:** The `vision.ConnectedComponentLabeler` System object will be removed in a future release. Use the `bwlabel` or the `bwlabeln` (n-D Version) functions with equivalent functionality instead.

---

### Construction

`H = vision.ConnectedComponentLabeler` returns a System object, `H`, that labels and counts connected regions in a binary image.

`H = vision.ConnectedComponentLabeler(Name, Value)` returns a label System object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Connectivity

Which pixels are connected to each other

Specify which pixels are connected to each other as either **4** or **8**. If a pixel should be connected to the pixels on the top, bottom, left, and right, set this property to **4**. If a pixel should be connected to the pixels on the top, bottom, left, right, and diagonally, set this property to **8**. The default is **8**.

### **LabelMatrixOutputPort**

Enable output of label matrix

Set to `true` to output the label matrix. Both the `???` and `???` properties cannot be set to `false` at the same time. The default is `true`.

### **LabelCountOutputPort**

Enable output of number of labels

Set to `true` to output the number of labels. Both the `???` and `???` properties cannot be set to `false` at the same time. The default is `true`.

### **OutputDataType**

Output data type

Set the data type of the output to one of `Automatic`, `uint32`, `uint16`, `uint8`. If this property is set to `Automatic`, the System object determines the appropriate data type for the output. If it is set to `uint32`, `uint16`, or `uint8`, the data type of the output is 32-, 16-, or 8-bit unsigned integers, respectively. The default is `Automatic`.

### **OverflowAction**

Behavior if number of found objects exceeds data type size of output

Specify the System object's behavior if the number of found objects exceeds the maximum number that can be represented by the output data type as `Use maximum value of the output data type`, or `Use zero`. If this property is set to `Use maximum value of the output data type`, the remaining regions are labeled with the maximum value of the output data type. If this property is set to `Use zero`, the remaining regions are labeled with zeroes. This property applies when you set the `OutputDataType` property to `uint16` or `uint8`. The default is `Use maximum value of the output data type`.

## **Methods**

<code>clone</code>	Create connected component labeler object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Label and count connected regions in input

## Examples

### Label Connected Regions in an Image

```
img = logical([0 0 0 0 0 0 0 0 0 0 0 0 0; ...
              0 1 1 1 1 0 0 0 0 0 0 1 0; ...
              0 1 1 1 1 1 0 0 0 0 1 1 0; ...
              0 1 1 1 1 1 0 0 0 1 1 1 0; ...
              0 1 1 1 1 0 0 0 1 1 1 1 0; ...
              0 0 0 0 0 0 0 1 1 1 1 1 0; ...
              0 0 0 0 0 0 0 0 0 0 0 0 0])
hlabel = vision.ConnectedComponentLabeler;
hlabel.LabelMatrixOutputPort = true;
hlabel.LabelCountOutputPort = false;
labeled = step(hlabel, img)
```

img =

```

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

Warning: The `vision.ConnectedComponentLabeler` will be removed in a future release. Use the `bwlabel` function with equivalent functionality instead.

labeled =

```

0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 0 0 0 2 0
```

0	1	1	1	1	1	0	0	0	0	2	2	0
0	1	1	1	1	1	0	0	0	2	2	2	0
0	1	1	1	1	0	0	0	2	2	2	2	0
0	0	0	0	0	0	0	2	2	2	2	2	0
0	0	0	0	0	0	0	0	0	0	0	0	0

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Label` block reference page. The object properties correspond to the block parameters, except:

- The `LabelCountOutputPort` and `LabelMatrixOutputPort` object properties correspond to the **Output** block parameter.

## See Also

`vision.BlobAnalysis` | `vision.AutoThresholder`

**Introduced in R2012a**

# clone

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Create connected component labeler object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.



## isLocked

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ConnectedComponentLabeler System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.ConnectedComponentLabeler

**Package:** vision

Label and count connected regions in input

## Syntax

`L = step(H,BW)`

`COUNT = step(H,BW)`

`[L,COUNT] = step(H,BW)`

## Description

`L = step(H,BW)` outputs the matrix, L for input binary image BW when the `LabelMatrixOutputPort` property is true.

`COUNT = step(H,BW)` outputs the number of distinct, connected regions found in input binary image BW when you set the `LabelMatrixOutputPort` property to false and `LabelCountOutputPort` property to true.

`[L,COUNT] = step(H,BW)` outputs both the L matrix and number of distinct, connected regions, COUNT when you set both the `LabelMatrixOutputPort` property and `LabelCountOutputPort` to true.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.ContrastAdjuster System object

**Package:** vision

Adjust image contrast by linear scaling

### Description

---

**Note:** The `vision.ContrastAdjuster` System object will be removed in a future release. Use the `imadjust` or `stretchlim` function with equivalent functionality instead.

---

The `ContrastAdjuster` object adjusts image contrast by linearly scaling pixel values between upper and lower limits. Pixel values that are above or below this range are saturated to the upper or lower limit values.

### Construction

`H = vision.ContrastAdjuster` returns a contrast adjustment object, `H`. This object adjusts the contrast of an image by linearly scaling the pixel values between the maximum and minimum values of the input data.

`H = vision.ContrastAdjuster(Name, Value)` returns a contrast adjustment object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### **InputRange**

How to specify lower and upper input limits

Specify how to determine the lower and upper input limits as one of `Full input data range [min max] | Custom | Range determined by saturating outlier pixels`. The default is `Full input data range [min max]`.

### **CustomInputRange**

Lower and upper input limits

Specify the lower and upper input limits as a two-element vector of real numbers. The first element corresponds to the lower input limit, and the second element corresponds to the upper input limit. This property applies only when you set the `???` property to `Custom`. This property is tunable.

### **PixelSaturationPercentage**

Percentage of pixels to consider outliers

Specify the percentage of pixels to consider outliers, as a two-element vector. This property only applies when you set the `???` property to `Range determined by saturating outlier pixels`.

The contrast adjustment object calculates the lower input limit. This calculation ensures that the maximum percentage of pixels with values smaller than the lower limit can only equal the value of the first element.

Similarly, the object calculates the upper input limit. This calculation ensures that the maximum percentage of pixels with values greater than the upper limit is can only equal the value of the second element.

The default is `[ 1 1 ]`.

### **HistogramNumBins**

Number of histogram bins

Specify the number of histogram bins used to calculate the scaled input values.

The default is `256`.

### **OutputRangeSource**

How to specify lower and upper output limits

Specify how to determine the lower and upper output limits as one of **Auto** | **Property**. The default is **Auto**. If you set the value of this property to **Auto**, the object uses the minimum value of the input data type as the lower output limit. The maximum value of the input data type indicates the upper output limit.

### **OutputRange**

Lower and upper output limits

Specify the lower and upper output limits as a two-element vector of real numbers. The first element corresponds to the lower output limit, and the second element corresponds to the upper output limit. This property only applies when you set the **Property** property to **Property**. This property is tunable.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Floor**.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of **Wrap** | **Saturate**. The default is **Wrap**

#### **ProductInputDataType**

Product input word and fraction lengths

Specify the product input fixed-point data type as **Custom**.

#### **CustomProductInputDataType**

Product input word and fraction lengths

Specify the product input fixed-point type as a scaled **numericType** object with a **Signedness** of **Auto**.

The default is `numericType([],32,16)`.

### **ProductHistogramDataType**

Product histogram word and fraction lengths

Specify the product histogram fixed-point data type as `Custom`. This property only applies when you set the `???` property to `Range` determined by saturating outlier pixels.

### **CustomProductHistogramDataType**

Product histogram word and fraction lengths

Specify the product histogram fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property only applies when you set the `???` property to `Range` determined by saturating outlier pixels.

The default is `numericType([],32,16)`.

## **Methods**

<code>clone</code>	Create contrast adjuster with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Adjust contrast in input image

## **Algorithms**

This object implements the algorithm, inputs, and outputs described on the `Contrast Adjustment` block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**



# clone

**System object:** vision.ContrastAdjuster

**Package:** vision

Create contrast adjuster with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ContrastAdjuster

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .

## getNumOutputs

**System object:** vision.ContrastAdjuster

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.ContrastAdjuster

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ContrastAdjuster System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ContrastAdjuster

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.ContrastAdjuster

**Package:** vision

Adjust contrast in input image

### Syntax

$Y = \text{step}(H, X)$

### Description

$Y = \text{step}(H, X)$  performs contrast adjustment of input  $X$  and returns the adjusted image  $Y$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

# vision.Convolver System object

**Package:** vision

Compute 2-D discrete convolution of two input matrices

## Description

The `Convolver` object computes 2-D discrete convolution of two input matrices.

## Construction

`H = vision.Convolver` returns a System object, `H`, that performs two-dimensional convolution on two inputs.

`H = vision.Convolver(Name, Value)` returns a 2-D convolution System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### OutputSize

Specify dimensions of output

This property controls the size of the output scalar, vector, or matrix produced as a result of the convolution between the two inputs. You can set this property to one of `Full` | `Same as first input` | `Valid`. The default is `Full`. When you set this property to `Full`, the object outputs the full two-dimensional convolution with dimension,

$(Ma+Mb-1, Na+Nb-1)$ . When you set this property to `Same as first input`, the object outputs the central part of the convolution with the same dimensions as the first input. When you set this property to `Valid`, the object outputs only those parts of the convolution that are computed without the zero-padded edges of any input. For this case, the object outputs dimension,  $(Ma-Mb+1, Na-Nb+1)$ .  $(Ma, Na)$  is the size of the first input matrix and  $(Mb, Nb)$  is the size of the second input matrix.

### **Normalize**

Whether to normalize the output

Set to `true` to normalize the output. The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`.

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as first input` | `Custom`. The default is `Custom`.

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 10)`.



### AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as first input` | `Custom`. The default is `Same as product`.

### CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

### OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as first input` | `Custom`. The default is `Custom`.

### CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,12)`.

## Methods

<code>clone</code>	Create 2-D convolver object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes

step

Compute 2-D convolution of input matrices

### Examples

Compute the 2D convolution of two matrices.

```
hconv2d = vision.Convolver;  
x1 = [1 2;2 1];  
x2 = [1 -1;-1 1];  
y = step(hconv2d, x1, x2)
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the [2-D Convolution](#) block reference page. The object properties correspond to the block parameters.

### See Also

[vision.Crosscorrelator](#) | [vision.AutoCorrelator](#)

**Introduced in R2012a**

# clone

**System object:** vision.Convolver

**Package:** vision

Create 2-D convolver object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Convolver

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Convolver

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Convolver

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Convolver System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.Convolver

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.Convolver

**Package:** vision

Compute 2-D convolution of input matrices

### Syntax

`Y = step(HCONV2D,X1,X2)`

### Description

`Y = step(HCONV2D,X1,X2)` computes 2-D convolution of input matrices *X1* and *X2*.

---

**Note:** *H* specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.CornerDetector System object

**Package:** vision

Detect corner features

## Description

---

**Note:** The `vision.CornerDetector` System object will be removed in a future release. Use the `cornermetric` function with equivalent functionality instead.

---

---

**Note:** Use this object for fixed-point support. Otherwise, it is recommended to use one of the new `detectHarrisFeatures`, `detectMinEigenFeatures`, or `detectFASTFeatures` functions in place of `vision.CornerDetector`.

---

The corner detector object finds corners in a grayscale image. It returns corner locations as a matrix of [x y] coordinates. The object finds corners in an image using the Harris corner detection (by Harris & Stephens), minimum eigenvalue (by Shi & Tomasi), or local intensity comparison (Features from Accelerated Segment Test, FAST by Rosten & Drummond) method.

## Construction

`cornerDetector = vision.CornerDetector` returns a corner detector object, `cornerDetector`, that finds corners in an image using a Harris corner detector.

`cornerDetector = vision.CornerDetector(Name, Value)` configures the corner detector object properties. You specify these properties as one or more name-value pair arguments. Unspecified properties have default values.

## To detect corners:

- 1 Define and set up your corner detector using the constructor.

- 2 Call the `step` method with the input image, `I`, the corner detector object, `cornerDetector`, and any optional properties. See the syntax below for using the `step` method.

`LOC = step(cornerDetector, I)` returns corners in the grayscale input image `I`. The `step` method outputs the location of detected corners, specified as an  $M$ -by-2 matrix of  $[x\ y]$  coordinates, where  $M$  is the number of detected corners.  $M$  can be less than or equal to the value specified by the `???` property.

`METRIC = step(cornerDetector, I)` returns a matrix, `METRIC`, with corner metric values. This applies when you set the `???` property to `true`. The output, `METRIC`, represents corner strength. The size of the `METRIC` matrix is the same as that of the input image.

`[LOC, METRIC] = step(cornerDetector, I)` returns the locations of the corners in the  $M$ -by-2 matrix, `LOC`, and the corner metric matrix in `METRIC`. This applies when you set both the `???` and `???` properties to `true`.

## Properties

### Method

Method to find corner values

Specify the method to find the corner values. Specify as one of `Harris corner detection (Harris & Stephens)` | `Minimum eigenvalue (Shi & Tomasi)` | `Local intensity comparison (Rosten & Drummond)`.

Default: `Harris corner detection (Harris & Stephens)`

### Sensitivity

Harris corner sensitivity factor

Specify the sensitivity factor,  $k$ , used in the Harris corner detection algorithm as a scalar numeric value such that  $0 < k < 0.25$ . The smaller the value of  $k$ , the more likely the algorithm will detect sharp corners. This property only applies when you set the `???` property to `Harris corner detection (Harris & Stephens)`. This property is tunable.

Default: `0.04`

### **SmoothingFilterCoefficients**

Smoothing filter coefficients

Specify the filter coefficients for the separable smoothing filter as a real-valued numeric vector. The vector must have an odd number of elements and a length of at least 3. For more information, see `fspecial`. This property applies only when you set the `???` property to either Harris corner detection (Harris & Stephens) or Minimum eigenvalue (Shi & Tomasi).

Default: `Output of fspecial('gaussian',[1 5],1.5)`

### **IntensityThreshold**

Intensity comparison threshold

Specify a positive scalar value. The object uses the intensity threshold to find valid bright or dark surrounding pixels. This property applies only when you set the `???` property to Local intensity comparison (Rosten & Drummond). This property is tunable.

Default: 0.1

### **MaximumAngleThreshold**

Maximum angle in degrees for valid corner

Specify the maximum angle in degrees to indicate a corner as one of 22.5 | 45.0 | 67.5 | 90.0 | 112.5 | 135.0 | 157.5. This property only applies when you set the `???` property to Local intensity comparison (Rosten & Drummond). This property is tunable.

Default: 157.5

### **CornerLocationOutputPort**

Enable output of corner location

Set this property to `true` to output the corner location after corner detection. This property and the `???` property cannot both be set to `false`.

Default: `true`

### **MetricMatrixOutputPort**

Enable output of corner metric matrix

Set this property to `true` to output the metric matrix after corner detection. This property and the `???` property cannot both be set to `false`.

Default: `false`

### **MaximumCornerCount**

Maximum number of corners to detect

Specify the maximum number of corners to detect as a positive scalar integer value. This property only applies when you set the `???` property to `true`.

Default: 200

### **CornerThreshold**

Minimum metric value to indicate a corner

Specify a positive scalar number for the minimum metric value that indicates a corner. This property applies only when you set the `???` property to `true`. This property is tunable.

Default: 0.0005

### **NeighborhoodSize**

Size of suppressed region around detected corner

Specify the size of the neighborhood around the corner metric value over which the object zeros out the values. The neighborhood size is a two element vector of positive odd integers,  $[r\ c]$ . Here,  $r$  indicates the number of rows in the neighborhood, and  $c$  indicates the number of columns. This property only applies when you set the `???` property to `true`.

Default: `[11 11]`

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`.

Default: `Floor`

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`.

Default: `Wrap`

### **CoefficientsDataType**

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as one of `Same word length as input` | `Custom`. This property applies only when you do not set the `???` property to `Local intensity comparison` (Rosten & Drummond).

Default: `Custom`

### **CustomCoefficientsDataType**

Coefficients word and fraction lengths

Specify the coefficients fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. This property applies only when you do not set the `???` property to `Local intensity comparison` (Rosten & Drummond) and you set the `???` property to `Custom`.

Default: `numericType([], 16)`

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as input` | `Custom`. This property applies when you do not set the `Method` property to `Local intensity comparison` (Rosten & Drummond).

Default: `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you do not set the `???` property to `Local intensity comparison` (Rosten & Drummond) and you set the `???` property to `Custom`.

Default: `numerictype([],32,0)`

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as input` | `Custom`.

Default: `Custom`

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object. This property applies when you set the `???` property to `Custom`.

Default: `numerictype([],32,0)`

### **MemoryDataType**

Memory word and fraction lengths

Specify the memory fixed-point data type as one of `Same as input` | `Custom`. This property applies when you do not set the `???` property to `Local intensity comparison` (Rosten & Drummond).

Default: `Custom`

### **CustomMemoryDataType**

Memory word and fraction lengths

Specify the memory fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only you do not set the `???` property to `Local intensity comparison` (Rosten & Drummond) and you set the `???` property to `Custom`.

Default: `numericType([],32,0)`

### **MetricOutputDataType**

Metric output word and fraction lengths

Specify the metric output fixed-point data type as one of `Same as accumulator` | `Same as input` | `Custom`.

Default: `Same as accumulator`

### **CustomMetricOutputDataType**

Metric output word and fraction lengths

Specify the metric output fixed-point type as a signed, scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

Default: `numericType([],32,0)`

## **Methods**

<code>clone</code>	Create corner detector with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Detect corners in input image

## **Examples**

### **Detect Corners in an Input Image**

Detect corners in an input image using the FAST algorithm.

Read the input image.

```
I = imread('circuit.tif');
```

Select FAST algorithm by Rosten & Drummond.

```
cornerDetector = vision.CornerDetector('Method','Local intensity comparison (Rosten & D
```

Find corners

```
pts = step(cornerDetector, I);
```

Create the markers. The color data range must match the data range of the input image. The color format for the marker is [red, green, blue].

```
color = [1 0 0];  
drawMarkers = vision.MarkerInserter('Shape', 'Circle', 'BorderColor', 'Custom', 'Custom
```

Convert the grayscale input image I to an RGB image J before inserting color markers.

```
J = repmat(I,[1 1 3]);
```

Display the image with markers

```
J = step(drawMarkers, J, pts);  
imshow(J); title('Corners detected in a grayscale image');
```

### See Also

[vision.LocalMaximaFinder](#) | [vision.MarkerInserter](#) | [detectHarrisFeatures](#)  
| [detectFASTFeatures](#) | [detectMinEigenFeatures](#) | [insertMarker](#)  
| [detectSURFFeatures](#) | [detectMSERFeatures](#) | [extractFeatures](#) |  
[matchFeatures](#)

**Introduced in R2012a**



# clone

**System object:** vision.CornerDetector

**Package:** vision

Create corner detector with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.CornerDetector

**Package:** vision

Number of expected inputs to step method

### Syntax

getNumInputs(H)

### Description

getNumInputs(H) returns the number of expected inputs to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs(H)**.

## getNumOutputs

**System object:** vision.CornerDetector

**Package:** vision

Number of outputs from step method

### Syntax

getNumOutputs(H)

### Description

getNumOutputs(H) returns the number of outputs from the **step** method.

The **getNumOutputs** method returns a positive integer that is the number of outputs from the **step** method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.CornerDetector

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

isLocked(H)

## Description

isLocked(H) returns the locked state of the corner detector.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, `isLocked` returns a `true` value.

# release

**System object:** vision.CornerDetector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.CornerDetector

**Package:** vision

Detect corners in input image

## Syntax

`LOC = step(cornerDetector, I)`

`METRIC = step(cornerDetector, I)`

`[LOC, METRIC] = step(cornerDetector, I)`

## Description

`LOC = step(cornerDetector, I)` returns corners in the grayscale input image `I`. The `step` method outputs the location of detected corners, specified as an  $M$ -by-2 matrix of  $[x \ y]$  coordinates, where  $M$  is the number of detected corners.  $M$  can be less than or equal to the value specified by the `???` property.

`METRIC = step(cornerDetector, I)` returns a matrix, `METRIC`, with corner metric values. This applies when you set the `???` property to `true`. The output, `METRIC`, represents corner strength. The size of the `METRIC` matrix is the same as that of the input image.

`[LOC, METRIC] = step(cornerDetector, I)` returns the locations of the corners in the  $M$ -by-2 matrix, `LOC`, and the corner metric matrix in `METRIC`. This applies when you set both the `???` and `???` properties to `true`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.Crosscorrelator System object

**Package:** vision

2-D cross-correlation of two input matrices

## Description

The `Crosscorrelator` object computes 2-D cross-correlation of two input matrices.

## Construction

`H = vision.Crosscorrelator` returns a System object, `H`, that performs two-dimensional cross-correlation between two inputs.

`H = vision.Crosscorrelator(Name, Value)` returns a 2-D cross correlation System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### OutputSize

Specify dimensions of output

This property controls the size of the output scalar, vector, or matrix produced as a result of the cross-correlation between the two inputs. This property can be set to one of `Full`, `Same as first input`, `Valid`. If this property is set to `Full`, the output is the full two-dimensional cross-correlation with dimensions  $(Ma+Mb-1, Na+Nb-1)$ . if this property

is set to `same as first input`, the output is the central part of the cross-correlation with the same dimensions as the first input. if this property is set to `valid`, the output is only those parts of the cross-correlation that are computed without the zero-padded edges of any input. this output has dimensions  $(Ma-Mb+1, Na-Nb+1)$ .  $(Ma, Na)$  is the size of the first input matrix and  $(Mb, Nb)$  is the size of the second input matrix. The default is `Full`.

### **Normalize**

Normalize output

Set this property to `true` to normalize the output. If you set this property to `true`, the object divides the output by  $\sqrt{\sum(I_{1p} \cdot I_{1p}) \times \sum(I_2 \cdot I_2)}$ , where  $I_{1p}$  is the portion of the input matrix,  $I_1$  that aligns with the input matrix,  $I_2$ . This property must be set to `false` for fixed-point inputs. The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input`, `Custom`. The default is `Same as first input`.

#### **CustomProductDataType**

Product word and fraction lengths



Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Same as first input`, `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as first input`, `Custom`. The default is `Same as first input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],16,15)`.

## **Methods**

<code>clone</code>	Create 2-D cross correlator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Compute 2-D correlation of input matrices

## Examples

Compute the 2-D correlation of two matrices.

```
hcorr2d = vision.Crosscorrelator;  
x1 = [1 2;2 1];  
x2 = [1 -1;-1 1];  
y = step(hcorr2d,x1,x2);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D Correlation block reference page. The object properties correspond to the block parameters.

## See Also

vision.Autocorrelator

**Introduced in R2012a**

# clone

**System object:** vision.Crosscorrelator

**Package:** vision

Create 2-D cross correlator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Crosscorrelator

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Crosscorrelator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.Crosscorrelator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Crosscorrelator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.Crosscorrelator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.Crosscorrelator

**Package:** vision

Compute 2-D correlation of input matrices

### Syntax

$Y = \text{step}(H, X1, X2)$

### Description

$Y = \text{step}(H, X1, X2)$  computes the 2-D correlation of input matrices  $X1$  and  $X2$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---



# vision.DCT System object

**Package:** vision

Compute 2-D discrete cosine transform

## Description

The DCT object computes a 2-D discrete cosine transform. The number of rows and columns of the input matrix must be a power of 2.

## Construction

`H = vision.DCT` returns a discrete cosine transform System object, `H`, used to compute the two-dimensional discrete cosine transform (2-D DCT) of a real input signal.

`H = vision.DCT(Name, Value)` returns a discrete cosine transform System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### SineComputation

Specify how the System object computes sines and cosines as `Trigonometric` function, or `Table` lookup. This property must be set to `Table` lookup for fixed-point inputs.

### Fixed-Point Properties

#### RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the ??? to `Table lookup`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the ??? to `Table lookup`. The default is `Wrap`.

### **SineTableDataType**

Sine table word-length designation

Specify the sine table fixed-point data type as `Same word length as input`, or `Custom`. This property applies when you set the ??? to `Table lookup`. The default is `Same word length as input`.

### **CustomSineTableDataType**

Sine table word length

Specify the sine table fixed-point type as a signed, unscaled `numericType` object. This property applies when you set the ??? to `Table lookup` and you set the ??? property to `Custom`. The default is `numericType(true,16)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. This property applies when you set the ??? to `Table lookup`. The default is `Custom`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled `numericType` object. This property applies when you set the ??? to `Table lookup`, and the ??? property to `Custom`. The default is `numericType(true,32,30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product`, `Same as first input`, `Custom`. This property applies when you set the `???` property to `Table lookup`. The default is `Full precision`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `???` to `Table lookup`, and `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. This property applies when you set the `???` to `Table lookup`. The default is `Custom`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `???` to `Table lookup`, and the `???` property to `Custom`. The default is `numericType(true,16,15)`.

## **Methods**

<code>clone</code>	Create 2-D discrete cosine transform object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Compute 2-D discrete cosine transform on input

## Examples

Use 2-D discrete cosine transform to analyze the energy content in an image. Set the DCT coefficients lower than a threshold of 0, and then reconstruct the image using the 2-D inverse discrete cosine transform object.

```
hdct2d = vision.DCT;  
I = double(imread('cameraman.tif'));  
J = step(hdct2d, I);  
imshow(log(abs(J)), [], colormap(jet(64)), colorbar)  
  
hidct2d = vision.IDCT;  
J(abs(J) < 10) = 0;  
It = step(hidct2d, J);  
figure, imshow(I, [0 255]), title('Original image')  
figure, imshow(It, [0 255]), title('Reconstructed image')
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D DCT block reference page. The object properties correspond to the block parameters.

### See Also

vision.IDCT

**Introduced in R2012a**

# clone

**System object:** vision.DCT

**Package:** vision

Create 2-D discrete cosine transform object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.DCT

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

# getNumOutputs

**System object:** vision.DCT

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.DCT

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the DCT System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



# release

**System object:** vision.DCT

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.DCT

**Package:** vision

Compute 2-D discrete cosine transform on input

### Syntax

$Y = \text{step}(H,X)$

### Description

$Y = \text{step}(H,X)$  returns the 2-D discrete cosine transform  $Y$  of input  $X$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

# vision.Deinterlacer System object

**Package:** vision

Remove motion artifacts by deinterlacing input video signal

## Description

The `Deinterlacer` object removes motion artifacts by deinterlacing input video signal.

## Construction

`H = vision.Deinterlacer` returns a deinterlacing System object, `H`, that removes motion artifacts from images composed of weaved top and bottom fields of an interlaced signal.

`H = vision.Deinterlacer(Name, Value)` returns a deinterlacing System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### Method

Method used to deinterlace input video

Specify how the object deinterlaces the input video as one of `Line repetition` | `Linear interpolation` | `Vertical temporal median filtering`. The default is `Line repetition`.

### **TransposedInput**

Indicate if input data is in row-major order

Set this property to `true` if the input buffer contains data elements from the first row first, then the second row second, and so on.

The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the `???` property to `Linear Interpolation`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`. This property applies when you set the `???` property to `Linear Interpolation`.

#### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as input` | `Custom`. The default is `Custom`. This property applies when you set the `???` property to `Linear Interpolation`.

#### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property is applicable when the `???` property is `Custom`. This property applies when you set the `???` property to `Linear Interpolation`.

The default is `numericType([], 12, 3)`.

## OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as input` | `Custom`. This property applies when you set the `???` property to `Linear Interpolation`. The default is `Same as input`.

## CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property is applicable when the `???` property is `Custom`. This property applies when you set the `???` property to `Linear Interpolation`.

The default is `numericType([ ],8,0)`.

## Methods

<code>clone</code>	Create deinterlacer object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Deinterlace input video signal

## Examples

### Remove Motion Artifacts From Image

Create a deinterlacer object.

```
hdint = vision.Deinterlacer;
```

Read an image.

```
x = imread('vipinterlace.png');
```

Apply the deinterlacer using the object's step method.

```
y = step(hdint,x);
```

Display the results.

```
imshow(x); title('Original Image');  
figure, imshow(y); title('Image after deinterlacing');
```

Original Image



Image after deinterlacing



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Deinterlacing block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**

# clone

**System object:** vision.Deinterlacer

**Package:** vision

Create deinterlacer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



## getNumInputs

**System object:** vision.Deinterlacer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.Deinterlacer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.Deinterlacer

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Deinterlacer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.Deinterlacer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.Deinterlacer

**Package:** vision

Deinterlace input video signal

## Syntax

$Y = \text{step}(H, X)$

## Description

$Y = \text{step}(H, X)$  deinterlaces input  $X$  according to the algorithm set in the **Method** property.

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## Remove Motion Artifacts From Image

Create a deinterlacer object.

```
hdint = vision.Deinterlacer;
```

Read an image.

```
x = imread('vipinterlace.png');
```

Apply the deinterlacer using the object's **step** method.

```
y = step(hdint,x);
```

Display the results.

```
imshow(x); title('Original Image');  
figure, imshow(y); title('Image after deinterlacing');
```

Original Image



Image after deinterlacing



## vision.DemosaicInterpolator System object

**Package:** vision

Bayer-pattern image conversion to true color

### Description

The `DemosaicInterpolator` object demosaics Bayer's format images. The object identifies the alignment as the sequence of R, G, and B pixels in the top-left four pixels of the image, in row-wise order.

### Construction

`H = vision.DemosaicInterpolator` returns a System object, `H`, that performs demosaic interpolation on an input image in Bayer format with the specified alignment.

`H = vision.DemosaicInterpolator(Name, Value)` returns a System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Method

Interpolation algorithm

Specify the algorithm the object uses to calculate the missing color information as one of `Bilinear` | `Gradient-corrected linear`. The default is `Gradient-corrected linear`.



**SensorAlignment**

Alignment of the input image

Specify the sequence of R, G and B pixels that correspond to the 2-by-2 block of pixels in the top-left corner, of the image. It can be set to one of `RGGB` | `GRBG` | `GBRG` | `BGGR`. The default is `RGGB`. Specify the sequence in left-to-right, top-to-bottom order.

**Fixed-Point Properties****RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

**OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Saturate`.

**ProductDataType**

Product output word and fraction lengths

Specify the product output fixed-point data type as one of `Same as input` | `Custom`. The default is `Custom`.

**CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

**AccumulatorDataType**

Data type of the accumulator

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as input` | `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

## **Methods**

<code>clone</code>	Create demosaic interpolator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform demosaic operation on input

## **Examples**

Demosaic a Bayer pattern encoded-image photographed by a camera with a sensor alignment of `BGGR`.

```
x = imread('mandi.tif');
hdemosaic = ...
    vision.DemosaicInterpolator('SensorAlignment', 'BGGR');
y = step(hdemosaic, x);

imshow(x,'InitialMagnification',20);
title('Original Image');
figure, imshow(y,'InitialMagnification',20);
title('RGB image after demosaic');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the **Demosaic** block reference page. The object properties correspond to the block parameters, except: The **Output image signal** block parameter allows you to specify whether the block outputs the image as **One multidimensional signal** or **Separate color signals**. The object does not have a property that corresponds to the **Output image signal** block parameter. The object always outputs the image as an  $M$ -by- $N$ -by- $P$  color video signal.

## See Also

vision.GammaCorrector

**Introduced in R2012a**

# clone

**System object:** vision.DemosaicInterpolator

**Package:** vision

Create demosaic interpolator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.DemosaicInterpolator

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.DemosaicInterpolator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.DemosaicInterpolator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the DemosaicInterpolator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.DemosaicInterpolator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.DemosaicInterpolator

**Package:** vision

Perform demosaic operation on input

## Syntax

$Y = \text{step}(H, X)$

## Description

$Y = \text{step}(H, X)$  performs the demosaic operation on the input  $X$ . The step method outputs an  $M$ -by- $N$ -by- $P$  color video signal, where  $P$  is the number of color planes.

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.DeployableVideoPlayer System object

**Package:** vision

Display video

### Description

The `DeployableVideoPlayer` object displays video frames. This player is capable of displaying high definition video at high frame rates. This video player object supports C code generation.

### Construction

`depVideoPlayer = vision.DeployableVideoPlayer` returns a video player System object, `depVideoPlayer`, for displaying video frames. Each call to the `step` method, displays the next video frame. This object, unlike the `vision.VideoPlayer` object, can generate C code.

`depVideoPlayer = vision.DeployableVideoPlayer( ____, Name, Value)` configures the video player properties, specified as one or more `Name, Value` pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
Generated code for this function uses a precompiled platform-specific shared library
“Code Generation Support, Usage Notes, and Limitations”.

### To display video:

- 1 Define and set up your deployable video player object using the constructor.
- 2 Call the `step` method with the deployable video player object, `depVideoPlayer`, and any optional properties. See the syntax below for using the `step` method.

`step(depVideoPlayer, I)` displays one grayscale or truecolor RGB video frame, `I`, in the video player.

`step(depVideoPlayer, Y, Cb, Cr)` displays one frame of YCbCr 4:2:2 video in the color components `Y`, `Cb`, and `Cr` when you set the `???` property to `YCbCr 4:2:2`. The number of columns in the `Cb` and `Cr` components must be half the number of columns in the `Y` component.

## Properties

### **Location** — Location of bottom left corner of video window

dependent on the screen resolution (default) | two-element vector

Location of bottom left corner of video window, specified as the comma-separated pair consisting of `'Location'` and a two-element vector. The first and second elements are specified in pixels and represent the horizontal and vertical coordinates respectively. The coordinates `[0 0]` represent the bottom left corner of the screen. The default location depends on the screen resolution, and will result in a window positioned in the center of the screen.

### **Name** — Video window title bar caption

`'Deployable Video Player'`

Video window title bar caption, specified as the comma-separated pair consisting of `'Name'` and a string.

### **Size** — Size of video display window

`True size (1:1)` (default) | `Full-screen` | `Custom`

Size of video display window, specified as the comma-separated pair consisting of `'Size'` and `Full-screen`, `True size (1:1)` or `Custom`. When this property is set to `Full-screen`, use the `ESC` key to exit out of full-screen mode.

### **CustomSize** — Custom size for video player window

`[300 410]` (default) | two-element vector

Custom size for video player window, specified as the comma-separated pair consisting of `'CustomSize'` and a two-element vector. The first and second elements are specified in pixels and represent the horizontal and vertical components respectively. The video data will be resized to fit the window. This property applies when you set the `???` property to `Custom`.

### **InputColorFormat** — Color format of input signal

RGB (default) | 'YCbCr 4:2:2'

Color format of input signal, specified as the comma-separated pair consisting of 'InputColorFormat' and 'RGB' or 'YCbCr 4:2:2'. The number of columns in the Cb and Cr components must be half the number of columns in Y.

## Methods

clone	Create deployable video player object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
isOpen	Visible or hidden status for video player figure
release	Allow property value and input characteristics changes
step	Send multidimensional video to computer screen

## Examples

### **Play a Video From a File**

Set up System objects to read and view a video file.

```
videoFReader = vision.VideoFileReader('atrium.mp4');  
depVideoPlayer = vision.DeployableVideoPlayer;
```

Continue to read frames of video until the last frame is read. Exit the loop if user closes the video player window.

```
cont = ~isDone(videoFReader);  
while cont
```

```
    frame = step(videoFReader);  
    step(depVideoPlayer, frame);  
    cont = ~isDone(videoFReader) && isOpen(depVideoPlayer);  
end
```

Release System objects.

```
release(videoFReader);  
release(depVideoPlayer);
```

## See Also

[vision.VideoFileReader](#) | [vision.VideoFileWriter](#) | [vision.VideoPlayer](#)

**Introduced in R2012a**

# clone

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Create deployable video player object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.



## isLocked

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Locked status for input attributes and nontunable properties

### Syntax

TF = isLocked(H)

### Description

TF = isLocked(H) returns the locked status, TF of the DeployableVideoPlayer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## isOpen

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Visible or hidden status for video player figure

## Syntax

`isOpen(h)`

## Description

`isOpen(h)` returns the visible or hidden status, as a logical, for the video player window. This method is not supported in code generation.

## release

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.DeployableVideoPlayer

**Package:** vision

Send multidimensional video to computer screen

### Syntax

```
step(depVideoPlayer,I)
step(depVideoPlayer,Y,Cb,Cr)
```

### Description

`step(depVideoPlayer,I)` displays one grayscale or truecolor RGB video frame, `I`, in the video player.

`step(depVideoPlayer,Y,Cb,Cr)` displays one frame of YCbCr 4:2:2 video in the color components `Y`, `Cb`, and `Cr` when you set the `???` property to `YCbCr 4:2:2`. The number of columns in the `Cb` and `Cr` components must be half the number of columns in the `Y` component.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.EdgeDetector System object

**Package:** vision

Find object edge

## Description

---

**Note:** The `vision.EdgeDetector` System object will be removed in a future release. Use the `edge` function with equivalent functionality instead.

---

The `EdgeDetector` object finds edges of objects in images. For Sobel, Prewitt and Roberts algorithms, the object finds edges in an input image by approximating the gradient magnitude of the image. The object obtains the gradient as a result of convolving the image with the Sobel, Prewitt or Roberts kernel. For Canny algorithm, the object finds edges by looking for the local maxima of the gradient of the input image. The calculation derives the gradient using a Gaussian filter. This algorithm is more robust to noise and more likely to detect true weak edges.

## Construction

`H = vision.EdgeDetector` returns an edge detection System object, `H`, that finds edges in an input image using Sobel, Prewitt, Roberts, or Canny algorithm.

`H = vision.EdgeDetector(Name, Value)` returns an edge detection object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

# Properties

## Method

Edge detection algorithm

Specify the edge detection algorithm as one of `Sobel` | `Prewitt` | `Roberts` | `Canny`. The default is `Sobel`.

## BinaryImageOutputPort

Output the binary image

Set this property to `true` to output the binary image after edge detection. When you set this property to `true`, the object will output a logical matrix. The nonzero elements of this matrix correspond to the edge pixels and the zero elements correspond to the background pixels. This property applies when you set the `Method` property to `Sobel`, `Prewitt` or `Roberts`.

The default is `true`.

## GradientComponentOutputPorts

Output the gradient components

Set this property to `true` to output the gradient components after edge detection. When you set this property to `true`, and the `Method` property to `Sobel` or `Prewitt`, this System object outputs the gradient components that correspond to the horizontal and vertical edge responses. When you set the `Method` property to `Roberts`, the System object outputs the gradient components that correspond to the 45 and 135 degree edge responses. Both `Method` and `BinaryImageOutputPort` properties cannot be `false` at the same time.

The default is `false`.

## ThresholdSource

Source of threshold value

Specify how to determine threshold as one of `Auto` | `Property` | `Input port`. The default is `Auto`. This property applies when you set the `Method` property to `Canny`. This property also applies when you set the `Method` property to `Sobel`, `Prewitt` or `Roberts` and the `BinaryImageOutputPort` property to `true`.

## Threshold

Threshold value

Specify the threshold value as a scalar of a MATLAB built-in numeric data type. When you set the you set the ??? property to **Sobel**, **Prewitt** or **Roberts**, you must a value within the range of the input data. When you set the **Method** property to **Canny**, specify the threshold as a two-element vector of low and high values that define the weak and strong edges. This property is accessible when the ??? property is **Property**. This property is tunable.

The default is `[0.25 0.6]` when you set the **Method** property to **Canny**. Otherwise, The default is `20`.

## ThresholdScaleFactor

Multiplier to adjust value of automatic threshold

Specify multiplier that is used to adjust calculation of automatic threshold as a scalar MATLAB built-in numeric data type. This property applies when you set the ??? property to **Sobel**, **Prewitt** or **Roberts** and the ??? property to **Auto**. This property is tunable.

The default is `4`.

## EdgeThinning

Enable performing edge thinning

Indicate whether to perform edge thinning. Choosing to perform edge thinning requires additional processing time and resources. This property applies when you set the ??? property to **Sobel**, **Prewitt** or **Roberts** and the ??? property to **true**.

The default is `false`.

## NonEdgePixelsPercentage

Approximate percentage of weak and nonedge pixels

Specify the approximate percentage of weak edge and nonedge image pixels as a scalar between `0` and `100`. This property applies when you set the ??? property to **Canny** and the ??? to **Auto**. This property is tunable.

The default is 70.

### **GaussianFilterStandardDeviation**

Standard deviation of Gaussian

filter Specify the standard deviation of the Gaussian filter whose derivative is convolved with the input image. You can set this property to any positive scalar. This property applies when you set the ??? property to **Canny**.

The default is 1.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Custom**. This property applies when you do not set the ??? property to **Canny**.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as **Wrap** or **Saturate**. This property applies when you do not set the ??? property to **Canny**.

The default is **Wrap**.

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of **Same as first input** | **Custom**. The default is **Custom**. This property applies when you do not set the ??? property to **Canny**.

#### **CustomProductDataType**

Product word and fraction lengths



Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you do not set the `???` property to `Canny`. This property applies when you set the `???` property to `Custom`.

The default is `numericType([],32,8)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as first input` | `Same as product` | `Custom`. The default is `Same as product`. This property applies when you do not set the `???` property to `Canny`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you do not set the `???` property to `Canny`. This property applies when you set the `???` property to `Custom`.

The default is `numericType([],32,8)`.

### **GradientDataType**

Gradient word and fraction lengths

Specify the gradient components fixed-point data type as one of `Same as accumulator` | `Same as first input` | `Same as product` | `Custom`. The default is `Same as first input`. This property applies when you do not set the `???` property to `Canny` and you set the `GradientComponentPorts` property to `true`.

### **CustomGradientDataType**

Gradient word and fraction lengths

Specify the gradient components fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property is accessible when the `???` property is not `Canny`. This property is applicable when the `???` property is `Custom`.

The default is `numericType([],16,4)`.

### Methods

clone	Create edge detector object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Find edges in input image

### Examples

#### Find Edges In An Image

**Create edge detector, color space converter, and image type converter objects.**

```
hedge = vision.EdgeDetector;  
hcsc = vision.ColorSpaceConverter('Conversion', 'RGB to intensity');
```

Warning: The `vision.EdgeDetector` will be removed in a future release. Use the `edge` function with equivalent functionality instead.

Warning: The `vision.ColorSpaceConverter` will be removed in a future release. Use the `rgb2gray`, `rgb2ycbcr`, `ycbcr2rgb`, `rgb2hsv`, `hsv2rgb`, `rgb2xyz`, `rgb2lab`, or `lab2rgb` function with equivalent functionality instead.

**Read in the original image and convert color and data type.**

```
img = step(hcsc, imread('peppers.png'));  
img1 = im2single(img);
```

**Find edges.**

```
edges = step(hedge, img1);
```

**Display original and image with edges.**

```
subplot(1,2,1);  
imshow(imread('peppers.png'));
```

```
subplot(1,2,2);  
imshow(edges);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Edge Detection](#) block reference page. The object properties correspond to the block parameters.

## See Also

[vision.TemplateMatcher](#)

**Introduced in R2012a**

# clone

**System object:** vision.EdgeDetector

**Package:** vision

Create edge detector object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.EdgeDetector

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.EdgeDetector

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.EdgeDetector

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the EdgeDetector System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



# release

**System object:** vision.EdgeDetector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.EdgeDetector

**Package:** vision

Find edges in input image

### Syntax

```
EDGES = step(H,IMG)
[GV,GH] = step(H,IMG)
[EDGES,GV,GH] = step(H,IMG)
```

### Description

`EDGES = step(H,IMG)` finds the edges, `EDGES`, in input `IMG` using the specified algorithm when the `BinaryImageOutputPort` property is `true`. `EDGES` is a boolean matrix with non-zero elements representing edge pixels and zero elements representing background pixels.

`[GV,GH] = step(H,IMG)` finds the two gradient components, `GV` and `GH`, of the input `IMG`, when you set the `Method` property to `Sobel`, `Prewitt` or `Roberts`, and you set the `GradientComponentOutputPorts` property to `true` and the `BinaryImageOutputPort` property to `false`. If you set the `Method` property to `Sobel` or `Prewitt`, then `GV` is a matrix of gradient values in the vertical direction and `GH` is a matrix of gradient values in the horizontal direction. If you set the `Method` property to `Roberts`, then `GV` represents the gradient component at 45 degree edge response, and `GH` represents the gradient component at 135 degree edge response.

`[EDGES,GV,GH] = step(H,IMG)` finds the edges, `EDGES`, and the two gradient components, `GV` and `GH`, of the input `IMG` when you set the `Method` property to `Sobel`, `Prewitt` or `Roberts` and both the `BinaryImageOutputPort` and `GradientComponentOutputPorts` properties are `true`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.FFT System object

**Package:** vision

Two-dimensional discrete Fourier transform

### Description

The `vision.FFT` object computes the 2D discrete Fourier transform (DFT) of a two-dimensional input matrix.

### Construction

`fftObj = vision.FFT` returns a 2D FFT object, `fftObj`, that computes the fast Fourier transform of a two-dimensional input.

`fftObj = vision.FFT(Name, Value)` configures the System object properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### To compute FFT:

- 1 Define and set up your FFT object using the constructor.
- 2 Call the `step` method with the input image, `I` and the FFT object, `fftObj`. See the syntax below for using the `step` method.

`J = step(fftObj, I)` computes the 2-D FFT, `J`, of an  $M$ -by- $N$  input matrix `I`, where  $M$  and  $N$  specify the dimensions of the input. The dimensions  $M$  and  $N$  must be positive integer powers of two when any of the following are true:

The input is a fixed-point data type

You set the `???` property to `true`.  
You set the `???` property to `Radix-2`.

## Properties

### **FFTImplementation**

FFT implementation

Specify the implementation used for the FFT as one of `Auto` | `Radix-2` | `FFTW`. When you set this property to `Radix-2`, the FFT length must be a power of two.

Default: `Auto`

### **BitReversedOutput**

Output in bit-reversed order relative to input

Designates the order of output channel elements relative to the order of input elements. Set this property to `true` to output the frequency indices in bit-reversed order.

Default: `false`

### **Normalize**

Divide butterfly outputs by two

Set this property to `true` if the output of the FFT should be divided by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

Default: `false` with no scaling

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`.

Default: Floor

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`.

Default: `Wrap`

### **SineTableDataType**

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input`, or `Custom`.

Default: `Same word length as input`

### **CustomSineTableDataType**

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numericType` object with a `Signedness` of `Auto`. This property applies only when you set the `???` property to `Custom`.

Default: `numericType([],16)`

### **ProductDataType**

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`.

Default: `Full precision`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when you set the `???` property to `Custom`.

Default: `numericType([],32,30)`

**AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`.

Default: `Full precision`

**CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when you set the `???` property to `Custom`.

Default: `numericType([],32,30)`

**OutputDataType**

Output word and fraction lengths

Specify the output data type as `Full precision`, `Same as input`, or `Custom`.

Default: `Full precision`

**CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

Default: `numericType([],16,15)`

**Methods**

`clone`

Create FFT object with same property values

<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute 2D discrete Fourier transform of input

## Examples

### Use 2-D FFT to View the Frequency Components of an Image

Create the FFT object.

```
fftObj = vision.FFT;
```

Read the image.

```
I = im2single(imread('pout.tif'));
```

Compute the FFT.

```
J = step(fftObj, I);
```

Shift zero-frequency components to the center of spectrum.

```
J_shifted = fftshift(J);
```

Display original image and visualize its FFT magnitude response.

```
figure; imshow(I); title('Input image, I');  
figure; imshow(log(max(abs(J_shifted), 1e-6)), [], colormap(jet(64)));  
title('Magnitude of the FFT of I');
```

## References

[1] FFTW (<http://www.fftw.org>)



- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

### **See Also**

vision.IFFT | vision.IDCT | vision.DCT | `fft2`

**Introduced in R2012a**

# clone

**System object:** vision.FFT

**Package:** vision

Create FFT object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.FFT

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.FFT

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.FFT

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the FFT System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.FFT

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.FFT

**Package:** vision

Compute 2D discrete Fourier transform of input

## Syntax

$Y = \text{step}(H, X)$

## Description

$Y = \text{step}(H, X)$  computes the 2D discrete Fourier transform (DFT),  $Y$ , of an  $M$ -by- $N$  input matrix  $X$ , where the values of  $M$  and  $N$  are integer powers of two.

## vision.ForegroundDetector System object

**Package:** vision

Foreground detection using Gaussian mixture models

### Description

The `ForegroundDetector` System object compares a color or grayscale video frame to a background model to determine whether individual pixels are part of the background or the foreground. It then computes a foreground mask. By using background subtraction, you can detect foreground objects in an image taken from a stationary camera.

### Construction

`detector = vision.ForegroundDetector` returns a foreground detector System object, `detector`. Given a series of either grayscale or color video frames, the object computes and returns the foreground mask using Gaussian mixture models (GMM).

`detector = vision.ForegroundDetector(Name, Value)` returns a foreground detector System object, `detector`, with each specified property name set to the specified value. `Name` can also be a property name and `Value` is the corresponding value. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Code Generation Support
Supports MATLAB Function block: No
Generates platform-dependent library: Yes for MATLAB host. Generates portable C code for non-host target.
“System Objects in MATLAB Code Generation”.
Generates portable C code for non-host target.
Generated code for this function uses a precompiled platform-specific shared library.
“Code Generation Support, Usage Notes, and Limitations”.



## Properties

### **AdaptLearningRate** — Adapt learning rate

'true' (default) | 'false'

Adapt learning rate, specified as the comma-separated pair consisting of 'AdaptLearningRate' and a logical scalar 'true' or 'false'. This property enables the object to adapt the learning rate during the period specified by the NumTrainingFrames property. When you set this property to true, the object sets the ??? property to  $1/(\text{current frame number})$ . When you set this property to false, the LearningRate property must be set at each time step.

### **NumTrainingFrames** — Number of initial video frames for training background model

150 (default) | integer

Number of initial video frames for training background model, specified as the comma-separated pair consisting of 'NumTrainingFrames' and an integer. When you set the ??? to false, this property will not be available.

### **LearningRate** — Learning rate for parameter updates

0.005 (default) | numeric scalar

Learning rate for parameter updates, specified as the comma-separated pair consisting of 'LearningRate' and a numeric scalar. Specify the learning rate to adapt model parameters. This property controls how quickly the model adapts to changing conditions. Set this property appropriately to ensure algorithm stability.

When you set ??? to true, the ??? property takes effect only after the training period specified by ??? is over.

When you set the AdaptLearningRate to false, this property will not be available. This property is tunable.

### **MinimumBackgroundRatio** — Threshold to determine background model

0.7 (default) | numeric scalar

Threshold to determine background model, specified as the comma-separated pair consisting of 'MinimumBackgroundRatio' and a numeric scalar. Set this property to represent the minimum of the apriori probabilities for pixels to be considered background values. Multimodal backgrounds can not be handled, if this value is too small.

### **NumGaussians** — Number of Gaussian modes in the mixture model

5 (default) | positive integer

Number of Gaussian modes in the mixture model

Number of Gaussian modes in the mixture model, specified as the comma-separated pair consisting of 'NumGaussians' and a positive integer. Typically this value is 3, 4 or 5. Set this value to 3 or greater to be able to model multiple background modes.

### **InitialVariance** — Initial mixture model variance

'Auto' (default) | numeric scalar

Initial mixture model variance, specified as the comma-separated pair consisting of 'InitialVariance' and as a numeric scalar or the 'Auto' string.

Image Data Type	Initial Variance
double/single	$(30/255)^2$
uint8	$30^2$

This property applies to all color channels for color inputs.

## Methods

clone	Create foreground detector with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset the GMM model to its initial state
step	Detect foreground using Gaussian mixture models

## Examples

### Detect Moving Cars In Video

Create system objects to read file.

```
videoSource = vision.VideoFileReader('viptraffic.avi','ImageColorSpace','Intensity','V');
```

Setting frames to 5 because it is a short video. Set initial standard deviation.

```
detector = vision.ForegroundDetector(...
    'NumTrainingFrames', 5, ...
    'InitialVariance', 30*30);
```

Perform blob analysis.

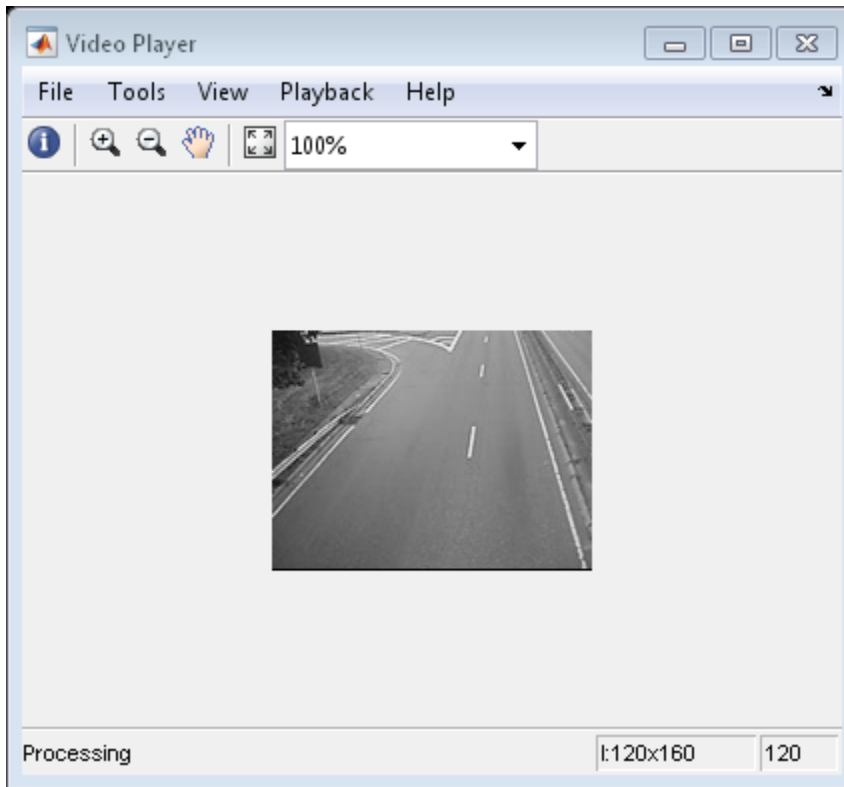
```
blob = vision.BlobAnalysis(...
    'CentroidOutputPort', false, 'AreaOutputPort', false, ...
    'BoundingBoxOutputPort', true, ...
    'MinimumBlobAreaSource', 'Property', 'MinimumBlobArea', 250);
```

Insert a border.

```
shapeInserter = vision.ShapeInserter('BorderColor','White');
```

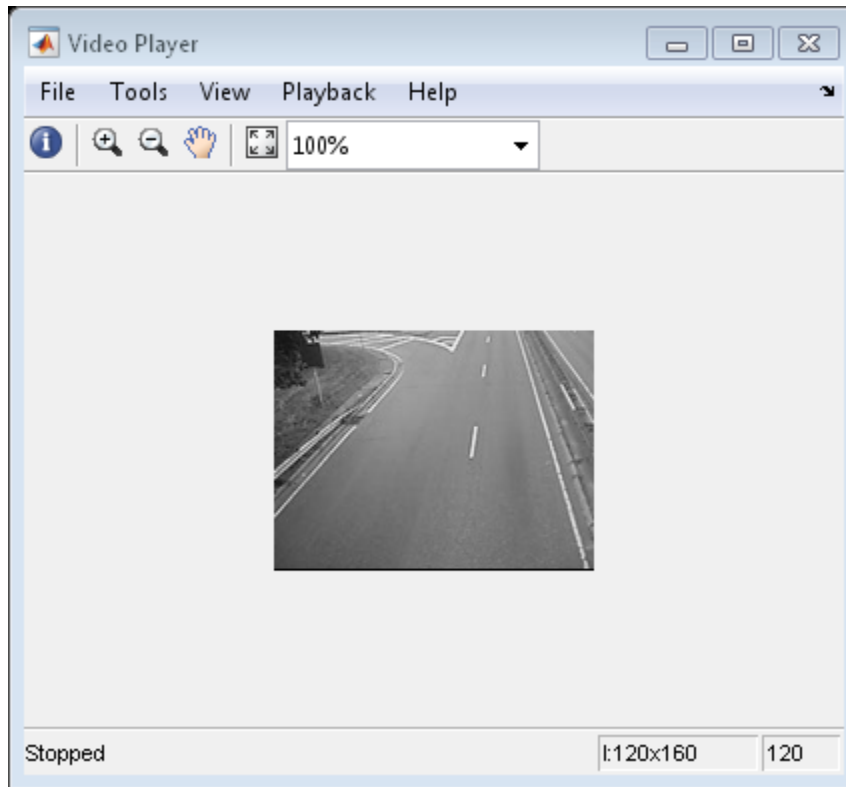
Play results. Draw bounding boxes around cars.

```
videoPlayer = vision.VideoPlayer();
while ~isDone(videoSource)
    frame = step(videoSource);
    fgMask = step(detector, frame);
    bbox = step(blob, fgMask);
    out = step(shapeInserter, frame, bbox);
    step(videoPlayer, out);
end
```



Release objects.

```
release(videoPlayer);  
release(videoSource);
```



## References

- [1] P. Kaewtrakulpong, R. Bowden, *An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection*, In Proc. 2nd European Workshop on Advanced Video Based Surveillance Systems, AVBS01, VIDEO BASED SURVEILLANCE SYSTEMS: Computer Vision and Distributed Processing (September 2001)
- [2] Stauffer, C. and Grimson, W.E.L., *Adaptive Background Mixture Models for Real-Time Tracking*, Computer Vision and Pattern Recognition, IEEE Computer Society Conference on, Vol. 2 (06 August 1999), pp. 2246-252 Vol. 2.

**More About**

- “Multiple Object Tracking”

**Introduced in R2011a**

# clone

**System object:** vision.ForegroundDetector

**Package:** vision

Create foreground detector with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ForegroundDetector

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).



# getNumOutputs

**System object:** vision.ForegroundDetector

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.ForegroundDetector

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ForegroundDetector System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.ForegroundDetector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You cannot use the release method on System objects in code generated from MATLAB.

---

## reset

**System object:** vision.ForegroundDetector

**Package:** vision

Reset the GMM model to its initial state

## Syntax

reset(H)

## Description

reset(H) resets the GMM model to its initial state for the ForegroundDetector object H.

---

## step

**System object:** vision.ForegroundDetector

**Package:** vision

Detect foreground using Gaussian mixture models

## Syntax

```
foregroundMask = step(H,I)
```

```
foregroundMask = step(H,I,learningRate)
```

## Description

`foregroundMask = step(H,I)` computes the foreground mask for input image **I**, and returns a logical mask. When the object returns `foregroundMask` set to 1, it represents foreground pixels. Image **I** can be grayscale or color. When you set the `???` property to `true` (default), the object permits this form of the `step` function call.

`foregroundMask = step(H,I,learningRate)` computes the foreground mask for input image **I** using the `???` you provide. When you set the `???` property to `false`, the object permits this form of the `step` function call.

---

**Note:** **H** specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.GammaCorrector System object

**Package:** vision

Apply or remove gamma correction from images or video streams

### Description

The GammaCorrector object applies gamma correction to input images or video streams.

### Construction

`H = vision.GammaCorrector` returns a System object, HGAMMACORR. This object applies or removes gamma correction from images or video streams.

`H = vision.GammaCorrector(Name, Value)` returns a gamma corrector object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

`H = vision.GammaCorrector(GAMMA, Name, Value)` returns a gamma corrector object, H, with the Gamma property set to GAMMA and other specified properties set to the specified values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### Correction

Specify gamma correction or linearization

Specify the object's operation as one of Gamma | De-gamma. The default is Gamma.

## Gamma

Gamma value of output or input

When you set the ??? property to **Gamma**, this property gives the desired gamma value of the output video stream. When you set the **Correction** property to **De-gamma**, this property indicates the gamma value of the input video stream. You must set this property to a numeric scalar value greater than or equal to 1.

The default is 2.2.

## LinearSegment

Enable gamma curve to have linear portion near origin

Set this property to **true** to make the gamma curve have a linear portion near the origin.

The default is **true**.

## BreakPoint

I-axis value of the end of gamma correction linear segment

Specify the I-axis value of the end of the gamma correction linear segment as a scalar numeric value between 0 and 1. This property applies when you set the ??? property to **true**.

The default is 0.018.

## Methods

clone	Create gamma corrector object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

step

Apply or remove gamma correction from input

## Examples

Improve image contrast.

```
hgamma = ...
    vision.GammaCorrector(2.0, 'Correction', 'De-gamma');
x = imread('pears.png');
y = step(hgamma, x);

imshow(x); title('Original Image');
figure, imshow(y);
title('Enhanced Image after De-gamma Correction');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Gamma Correction](#) block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**



# clone

**System object:** vision.GammaCorrector

**Package:** vision

Create gamma corrector object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.GammaCorrector

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GammaCorrector

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.GammaCorrector

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GammaCorrector System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.GammaCorrector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.GammaCorrector

**Package:** vision

Apply or remove gamma correction from input

### Syntax

$Y = \text{step}(H, X)$

### Description

$Y = \text{step}(H, X)$  applies or removes gamma correction from input  $X$  and returns the gamma corrected or linearized output  $Y$ .

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.GeometricRotator System object

**Package:** vision

Rotate image by specified angle

## Description

---

**Note:** The `vision.GeometricRotator` System object will be removed in a future release. Use the `imrotate` function with equivalent functionality instead.

---

The `GeometricRotator` object rotates an image by a specified angle.

## Construction

`H = vision.GeometricRotator` returns a geometric rotator System object, `H`, that rotates an image by  $\frac{\pi}{6}$  radians.

`H = vision.GeometricRotator(Name, Value)` returns a geometric rotator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

### Code Generation Support

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Properties

### OutputSize

Output size as full or same as input image size

Specify the size of output image as one of `Expanded` to fit rotated input image | `Same as input image`. The default is `Expanded` to fit rotated input image.

When you set this property to `Expanded to fit rotated input image`, the object outputs a matrix that contains all the rotated image values. When you set this property to `Same as input image`, the object outputs a matrix that contains the middle part of the rotated image.

### **AngleSource**

Source of angle

Specify how to specify the rotation angle as one of `Property | Input port`. The default is `Property`.

### **Angle**

Rotation angle value (radians)

Set this property to a real, scalar value for the rotation angle (radians). The default is  $\pi/6$ . This property applies when you set the `AngleSource` property to `Property`.

### **MaximumAngle**

Maximum angle by which to rotate image

Specify the maximum angle by which to rotate the input image as a numeric scalar value greater than 0. The default is  $\pi$ . This property applies when you set the `AngleSource` property to `Input port`.

### **RotatedImageLocation**

How the image is rotated

Specify how the image is rotated as one of `Top-left corner | Center`. The default is `Center`. When you set this property to `Center`, the object rotates the image about its center point. When you set this property to `Top-left corner`, the object rotates the image so that two corners of the input image are always in contact with the top and left side of the output image. This property applies when you set the `OutputSize` property to `Expanded to fit rotated input image`, and, the `AngleSource` property to `Input port`.

### **SineComputation**

How to calculate the rotation



Specify how to calculate the rotation as one of `Trigonometric function` | `Table lookup`. The default is `Table lookup`. When you set this property to `Trigonometric function`, the object computes the sine and cosine values it needs to calculate the rotation of the input image. When you set this property to `Table lookup`, the object computes the trigonometric values it needs to calculate the rotation of the input image, and stores them in a table. The object uses the table, each time it calls the `step` method. In this case, the object requires extra memory.

### **BackgroundFillValue**

Value of pixels outside image

Specify the value of pixels that are outside the image as a numeric scalar value or a numeric vector of same length as the third dimension of input image. The default is `0`. This property is tunable.

### **InterpolationMethod**

Interpolation method used to rotate image

Specify the interpolation method used to rotate the image as one of `Nearest neighbor` | `Bilinear` | `Bicubic`. The default is `Bilinear`. When you set this property to `Nearest neighbor`, the object uses the value of one nearby pixel for the new pixel value. When you set this property to `Bilinear`, the object sets the new pixel value as the weighted average of the four nearest pixel values. When you set this property to `Bicubic`, the object sets the new pixel value as the weighted average of the sixteen nearest pixel values.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Nearest`. This property applies when you set the `SineComputation` property to `Table lookup`. T

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Saturate`. This property applies when you set the `SineComputation` property to `Table lookup`.

### **AngleDataType**

Angle word and fraction lengths

Specify the angle fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property applies when you set the `SineComputation` property to `Table lookup`, and the `AngleSource` property to `Property`.

### **CustomAngleDataType**

Angle word and fraction lengths

Specify the angle fixed-point type as a signed `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup`, the `AngleSource` property is `Property` and the `AngleDataType` property to `Custom`. The default is `numerictype([ ],32,10)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as first input` | `Custom`. The default is `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup`, and the `ProductDataType` property to `Custom`. The default is `numerictype([ ],32,10)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as first input` | `Custom`. The default is `Same as product`. This property applies when you set the `SineComputation` property to `Table lookup`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup`, and the `CustomAccumulatorDataType` property to `Custom`. The default is `numericType([],32,10)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as first input` | `Custom`. The default is `Same as first input`. This property applies when you set the `SineComputation` property to `Table lookup`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup`, and the `OutputDataType` property to `Custom`. The default is `numericType([],32,10)`.

## **Methods**

<code>clone</code>	Create geometric rotator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Return rotated image

## Examples

Rotate an image 30 degrees ( $\pi/6$  radians)

```
img = im2double(imread('peppers.png'));  
hrotate = vision.GeometricRotator;  
hrotate.Angle = pi / 6;
```

```
% Rotate img by pi/6  
rotimg = step(hrotate, img);  
imshow(rotimg);
```

Rotate an image by multiple angles, specifying each rotation angle as an input. The angle can be changed during simulation.

```
hrotate = vision.GeometricRotator;  
hrotate.AngleSource = 'Input port';  
img = im2double(imread('onion.png'));  
figure;  
% Rotate img by multiple angles  
for i = 1: 4  
    rotimg = step(hrotate, img, i*pi/16);  
    subplot(1,4,i);  
    imshow(rotimg);  
end
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Rotate` block reference page. The object properties correspond to the block parameters.

## See Also

`vision.GeometricTranslator` | `vision.GeometricScaler`

**Introduced in R2012a**

# clone

**System object:** vision.GeometricRotator

**Package:** vision

Create geometric rotator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.GeometricRotator

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GeometricRotator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.



## isLocked

**System object:** vision.GeometricRotator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GeometricRotator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.GeometricRotator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.GeometricRotator

**Package:** vision

Return rotated image

## Syntax

$Y = \text{step}(H, \text{IMG})$

$Y = \text{step}(H, \text{IMG}, \text{ANGLE})$

## Description

$Y = \text{step}(H, \text{IMG})$  returns a rotated image  $Y$ , with the rotation angle specified by the `Angle` property.

$Y = \text{step}(H, \text{IMG}, \text{ANGLE})$  uses input *ANGLE* as the angle to rotate the input *IMG* when you set the `AngleSource` property to `Input` port.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.GeometricScaler System object

**Package:** vision

Enlarge or shrink image size

### Description

---

**Note:** The `vision.GeometricScaler` System object will be removed in a future release. Use the `imresize` function with equivalent functionality instead.

---

The `GeometricScaler` object enlarges or shrinks image sizes.

`J = step(H, I)` returns a resized image, `J`, of input image `I`.

`J = step(H, I, ROI)` resizes a particular region of the image `I` defined by the `ROI` specified as `[x y width height]`, where `[x y]` represents the upper left corner of the ROI. This option applies when you set the `SizeMethod` property to `Number` of output rows and columns, the `Antialiasing` property to `false`, the `InterpolationMethod` property to `Bilinear`, `Bicubic` or `Nearest neighbor`, and the `ROIProcessing` property to `true`.

`[J, FLAG] = step(H, I, ROI)` also returns `FLAG` which indicates whether the given region of interest is within the image bounds. This applies when you set the `SizeMethod` property to `Number` of output rows and columns, the `Antialiasing` property to `false`, the `InterpolationMethod` property to `Bilinear`, `Bicubic` or `Nearest neighbor` and, the `ROIProcessing` and the `ROIValidityOutputPort` properties to `true`.

### Construction

`H = vision.GeometricScaler` returns a System object, `H`, that changes the size of an image or a region of interest within an image.

`H = vision.GeometricScaler(Name, Value)` returns a geometric scaler object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

**Code Generation Support**

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Properties

### SizeMethod

Aspects of image to resize

Specify which aspects of the input image to resize as one of `Output size as a percentage of input size` | `Number of output columns and preserve aspect ratio` | `Number of output rows and preserve aspect ratio` | `Number of output rows and columns`. The default is `Output size as a percentage of input size`.

### ResizeFactor

Percentage by which to resize rows and columns

Set this property to a scalar percentage or a two-element vector. The default is `[200 150]`. When you set this property to a scalar percentage, the object applies this value to both rows and columns. When you set this property to a two-element vector, the object uses the first element as the percentage to resize the rows, and the second element as the percentage to resize the columns. This property applies when you set the ??? property to `Output size as a percentage of input size`.

### NumOutputColumns

Number of columns in output image

Specify the number of columns of the output image as a positive, integer scalar value. The default is 25. This property applies when you set the ??? property to `Number of output columns and preserve aspect ratio`.

### NumOutputRows

Number of rows in output image

Specify the number of rows of the output image as a positive integer scalar value. The default is 25. This property applies when you set the ??? property to `Number of output rows` and `preserve aspect ratio`.

### **Size**

Dimensions of output image

Set this property to a two-element vector. The default is [25 35]. The object uses the first element for the number of rows in the output image, and the second element for the number of columns. This property applies when you set the ??? property to `Number of output rows` and `columns`.

### **InterpolationMethod**

Interpolation method used to resize the image

Specify the interpolation method to resize the image as one of `Nearest neighbor` | `Bilinear` | `Bicubic` | `Lanczos2` | `Lanczos3`. The default is `Bilinear`. When you set this property to `Nearest neighbor`, the object uses one nearby pixel to interpolate the pixel value. When you set this property to `Bilinear`, the object uses four nearby pixels to interpolate the pixel value. When you set this property to `Bicubic` or `Lanczos2`, the object uses sixteen nearby pixels to interpolate the pixel value. When you set this property to `Lanczos3`, the object uses thirty six surrounding pixels to interpolate the pixel value.

### **Antialiasing**

Enable low-pass filtering when shrinking image

Set this property to `true` to perform low-pass filtering on the input image before shrinking it. This prevents aliasing when the `ResizeFactor` property value is between 0 and 100 percent.

### **ROIProcessing**

Enable region-of-interest processing

Indicate whether to resize a particular region of each input image. The default is `false`. This property applies when you set the ??? property to `Number of output rows` and `columns`, the ??? parameter to `Nearest neighbor`, `Bilinear`, or `Bicubic`, and the ??? property to `false`.

### **ROIValidityOutputPort**

Enable indication that ROI is outside input image

Indicate whether to return the validity of the specified ROI being completely inside image. The default is `false`. This property applies when you set the `???` property to `true`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Nearest`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Saturate`.

### **InterpolationWeightsDataType**

Interpolation weights word and fraction lengths

Specify the interpolation weights fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property applies when you set:

- The `???` property to `Bicubic`, `Lanczos2`, or `Lanczos3`.
- The `???` property to any value other than `Output size as a percentage of input size`.
- When you set the combination of:
  - The `???` property to `Output size as a percentage of input size`.
  - The `???` property to `Bilinear` or `Nearest neighbor`.
  - Any of the elements of the `ResizeFactor` less than 100, implying shrinking of the input image.
  - The `???` property to `true`.

### **CustomInterpolationWeightsDataType**

Interpolation weights word and

`fraction lengths` Specify the interpolation weights fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numerictype([],32)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as input | Custom`. The default is `Custom`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numerictype([],32,10)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product | Same as input | Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numerictype([],32,10)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as input | Custom`. The default is `Same as input`.

### **CustomOutputDataType**

Output word and fraction lengths



Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

## Methods

<code>clone</code>	Create geometric scaler object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Resize image

## Examples

Enlarge an image. Display the original and the enlarged images.

```
I=imread('cameraman.tif');
hgs=vision.GeometricScaler;
hgs.SizeMethod = ...
    'Output size as a percentage of input size';
hgs.InterpolationMethod='Bilinear';

J = step(hgs,I);
imshow(I); title('Original Image');
figure,imshow(J);title('Resized Image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Resize` block reference page. The object properties correspond to the block parameters.

**See Also**

vision.Pyramid | vision.GeometricRotator | vision.GeometricTranslator

**Introduced in R2012a**

# clone

**System object:** vision.GeometricScaler

**Package:** vision

Create geometric scaler object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.GeometricScaler

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GeometricScaler

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.GeometricScaler

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GeometricScaler System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.GeometricScaler

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.GeometricScaler

**Package:** vision

Resize image

## Syntax

`J = step(H,I)`

`J = step(H,X,ROI)`

`[J,FLAG] = step(H,I,ROI)`

## Description

`J = step(H,I)` returns a resized image, *J*, of input image *I*.

`J = step(H,X,ROI)` resizes a particular region of the image *I* defined by the *ROI* input. This applies when you set the `SizeMethod` property to `Number of output rows and columns`, the `Antialiasing` property to `false`, the `InterpolationMethod` property to `Bilinear`, `Bicubic` or `Nearest neighbor`, and the `ROIProcessing` property to `true`.

`[J,FLAG] = step(H,I,ROI)` also returns *FLAG* which indicates whether the given region of interest is within the image bounds. This applies when you set the `SizeMethod` property to `Number of output rows and columns`, the `Antialiasing` property to `false`, the `InterpolationMethod` property to `Bilinear`, `Bicubic` or `Nearest neighbor` and, the `ROIProcessing` and the `ROIValidityOutputPort` properties to `true`.

---

**Note:** *H* specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.GeometricShearer System object

**Package:** vision

Shift rows or columns of image by linearly varying offset

## Description

The GeometricShearer System object shifts the rows or columns of an image by a gradually increasing distance left or right or up or down.

## Construction

`H = vision.GeometricShearer` creates a System object, *H*, that shifts the rows or columns of an image by gradually increasing distance left or right or up or down.

`H = vision.GeometricShearer(Name, Value, ... 'NameN', ValueN)` creates a geometric shear object, *H*, with each specified property set to the specified value.

Code Generation Support
Supports MATLAB Function block: Yes
“Code Generation Support, Usage Notes, and Limitations”

## Properties

### Direction

Direction to apply offset

Specify the direction of linearly increasing the offset as one of `Horizontal` | `Vertical`. The default is `Horizontal`. Set this property to `Horizontal` to linearly increase the offset of the rows, or `Vertical` to linearly increase the offset of the columns.

### OutputSize

Output size

Specify the size of the output image as one of `Full` | `Same as input image`. The default is `Full`. If you set this property to `Full`, the object outputs a matrix that contains the sheared image values. If you set this property to `Same as input image`, the object outputs a matrix that is the same size as the input image and contains a portion of the sheared image.

### **ValuesSource**

Source of shear values

Specify the source of shear values as one of `Property` | `Input port`. The default is `Property`.

### **Values**

Shear values in pixels

Specify the shear values as a two-element vector that represents the number of pixels by which to shift the first and last rows or columns of the input. The default is `[0 3]`. This property applies when you set the ??? property to `Property`.

### **MaximumValue**

Maximum number of pixels by which to shear image

Specify the maximum number of pixels by which to shear the image as a real, numeric scalar. The default is `20`. This property applies when you set the ??? property to `Input port`.

### **BackgroundFillValue**

Value of pixels outside image

Specify the value of pixels that are outside the image as a numeric scalar or a numeric vector that has the same length as the third dimension of the input image. The default is `0`. This property is tunable.

### **InterpolationMethod**

Interpolation method used to shear image

Specify the interpolation method used to shear the image as one of `Nearest neighbor` | `Bilinear` | `Bicubic`. The default is `Bilinear`. When you set this property to `Nearest`

neighbor, the object uses the value of one nearby pixel for the new pixel value. When you set this property to **Bilinear**, the object sets the new pixel value as the weighted average of the two nearest pixel values. When you set this property to **Bicubic**, the object sets the new pixel value as the weighted average of the four nearest pixel values.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Nearest**.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of **Wrap** | **Saturate**. The default is **Saturate**.

#### **ValuesDataType**

Shear values word and fraction lengths

Specify the shear values fixed-point data type as one of **Same word length as input** | **Custom**. The default is **Same word length as input**. This property applies when you set the ??? property to **Property**.

#### **CustomValuesDataType**

Shear values word and fraction lengths

Specify the shear values fixed-point type as an auto-signed **numericType** object. This property applies when you set the ??? property to **Property** and the ??? property to **Custom**. The default is **numericType([ ],32,10)**.

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of **Same as first input** | **Custom**. The default is **Custom**. This property applies when you set the ??? property to either **Bilinear** or **Bicubic**.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as an auto-signed scaled `numericType` object. This property applies when you set the `???` property to either `Bilinear` or `Bicubic`, and the `???` property to `Custom`. The default is `numericType([],32,10)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as first input` | `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as an auto-signed scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as first input` or `Custom`. The default is `Same as first input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as an auto-signed scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

## **Methods**

`clone`

Create a geometric shear object with same property values

getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Shift rows or columns of image by linearly varying offset

## Examples

Apply a horizontal shear to an image.

```
hshear = vision.GeometricShearer('Values',[0 20]);  
img = im2single(checkerboard);  
outimg = step(hshear,img);  
subplot(2,1,1), imshow(img);  
title('Original image');  
subplot(2,1,2), imshow(outimg);  
title('Output image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the **Shear** block reference page. The object properties correspond to the block parameters.

## See Also

[vision.GeometricTransformer](#) | [vision.GeometricScaler](#)

**Introduced in R2012a**

# clone

**System object:** vision.GeometricShearer

**Package:** vision

Create a geometric shear object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.GeometricShearer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$ , to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.GeometricShearer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$ , from the **step** method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the **step** method. This value will change if you alter any properties that turn outputs on or off.



## isLocked

**System object:** vision.GeometricShearer

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

`TF = isLocked(H)`

## Description

`TF = isLocked(H)` returns the locked status, *TF* of the `GeometricShearer` System object.

`isLocked` returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, `isLocked` returns a `true` value.

## release

**System object:** vision.GeometricShearer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.GeometricShearer

**Package:** vision

Shift rows or columns of image by linearly varying offset

## Syntax

$Y = \text{step}(H, \text{IMG})$

$Y = \text{step}(H, \text{IMG}, S)$

## Description

$Y = \text{step}(H, \text{IMG})$  shifts the input, *IMG*, and returns the shifted image, *Y*, with the shear values specified by the **Values** property.

$Y = \text{step}(H, \text{IMG}, S)$  uses the two-element vector, *S*, as the number of pixels by which to shift the first and last rows or columns of *IMG*. This applies when you set the **ValuesSource** property to **Input port**.

---

**Note:** *H* specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.GeometricTransformer System object

**Package:** vision

Apply projective or affine transformation to image

### Description

---

**Note:** The `vision.GeometricTransformer` System object will be removed in a future release. Use the `imwarp` function with equivalent functionality instead.

---

The `GeometricTransformer` object applies a projective or affine transformation to an image.

### Construction

`H = vision.GeometricTransformer` returns a geometric transformation System object, `H`. This object applies a projective or affine transformation to an image.

`H = vision.GeometricTransformer(Name, Value)` returns a geometric transformation object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

When you specify multiple transforms in a single matrix, each transform is applied separately to the original image. If the individual transforms produce an overlapping area, the result of the transform in the last row of the matrix is overlaid on top.

Code Generation Support
Supports MATLAB Function block: Yes
“Code Generation Support, Usage Notes, and Limitations”

## Properties

### **TransformMatrixSource**

Method to specify transformation matrix

Specify as one of `Property` | `Input port`.

Default: `Input port`.

### **TransformMatrix**

Transformation matrix

Specify the applied transformation matrix as a 3-by-2 or  $Q$ -by-6 affine transformation matrix, or a 3-by-3 or a  $Q$ -by-9 projective transformation matrix.  $Q$  is the number of transformations. This property applies when you set the ??? property to `Property`.

Default: `[1 0 0; 0 1 0; 0 0 1]`

### **InterpolationMethod**

Interpolation method

Specify as one of `Nearest neighbor` | `Bilinear` | `Bicubic` for calculating the output pixel value.

Default: `Bilinear`

### **BackgroundFillValue**

Background fill value

Specify the value of the pixels that are outside of the input image. The value can be either scalar or a  $P$ -element vector, where  $P$  is the number of color planes.

Default: `0`

### **OutputImagePositionSource**

Method to specify output image location and size

Specify the value of this property as one of `Auto` | `Property`. If this property is set to `Auto`, the output image location and size are the same values as the input image.

Default: Auto

### **OutputImagePosition**

Output image position vector

Specify the location and size of output image, as a four-element double vector in pixels, of the form, [x y width height]. This property applies when you set the ??? property to Property.

Default: [1 1 512 512]

### **ROIInputPort**

Enable the region of interest input port

Set this property to `true` to enable the input of the region of interest. When set to `false`, the whole input image is processed.

Default: `false`

### **ROIShape**

Region of interest shape

Specify ROIShape as one of `Rectangle ROI` | `Polygon ROI`. This property applies when you set the ??? property to `true`.

Default: `Rectangle ROI`

### **ROIValidityOutputPort**

Enable output of ROI flag

Set this property to `true` to enable the output of an ROI flag indicating when any part of the ROI is outside the input image. This property applies when you set the ??? property to `true`.

Default: `false`

### **ProjectiveTransformMethod**

Projective transformation method

Method to compute the projective transformation. Specify as one of `Compute exact values` | `Use quadratic approximation`.

Default: `Compute exact values`

### **ErrorTolerance**

Error tolerance (in pixels)

Specify the maximum error tolerance in pixels for the projective transformation. This property applies when you set the `???` property to `Use quadratic approximation`.

Default: 1

### **ClippingStatusOutputPort**

Enable clipping status flag output

Set this property to `true` to enable the output of a flag indicating if any part of the output image is outside the input image.

Default: `false`

## **Methods**

<code>clone</code>	Create geometric transformer object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Apply geometric transform to input image

## **Examples**

Apply a horizontal shear to an intensity image.

```
htrans1 = vision.GeometricTransformer(...
    'TransformMatrixSource', 'Property', ...
    'TransformMatrix',[1 0 0; .5 1 0; 0 0 1],...
    'OutputImagePositionSource', 'Property',...
    'OutputImagePosition', [0 0 400 750]);
img1 = im2single(rgb2gray(imread('peppers.png')));
transimg1 = step(htrans1,img1);
imshow(transimg1);
```

Apply a transform with multiple polygon ROI's.

```
htrans2 = vision.GeometricTransformer;
img2 = checker_board(20,10);

tformMat = [
    1      0      30      0      1      -30; ...
    1.0204 -0.4082      70      0  0.4082      30; ...
    0.4082      0  89.1836 -0.4082      1  10.8164];

polyROI = [
    1    101    99    101    99    199      1    199; ...
    1     1    99     1    99    99      1     99; ...
    101   101   199   101   199   199   101   199];

htrans2.BackgroundFillValue = [0.5 0.5 0.75];
htrans2.ROIInputPort = true;
htrans2.ROIShape = 'Polygon ROI';
transimg2 = step(htrans2,img2,tformMat,polyROI);
figure; imshow(img2);
figure; imshow(transimg2);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Apply Geometric Transformation](#) block reference page. The object properties correspond to the block parameters.

### See Also

`estimateGeometricTransform`

**Introduced in R2012a**



# clone

**System object:** vision.GeometricTransformer

**Package:** vision

Create geometric transformer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.GeometricTransformer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GeometricTransformer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.GeometricTransformer

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GeometricTransformer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.GeometricTransformer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.GeometricTransformer

**Package:** vision

Apply geometric transform to input image

## Syntax

`Y = step(H,X,TFORM)`

`Y = step(H,X)`

`Y = step(H,X,ROI)`

`[Y, ROIFLAG]=(H,X,...)`

`[Y,CLIPFLAG]=(H,X,...)`

`[Y,ROIFLAG,CLIPFLAG] = step(H,X,TFORM,ROI)`

## Description

`Y = step(H,X,TFORM)` outputs the transformed image,  $Y$ , of the input image,  $X$ .  $X$  is either an  $M$ -by- $N$  or an  $M$ -by- $N$ -by- $P$  matrix, where  $M$  is the number of rows,  $N$  is the number of columns, and  $P$  is the number of color planes in the image.  $TFORM$  is the applied transformation matrix.  $TFORM$  can be a 2-by-3 or 6-by- $Q$  affine transformation matrix, or a 3-by-3 or 9-by- $Q$  projective transformation matrix, where  $Q$  is the number of transformations.

`Y = step(H,X)` outputs the transformed image,  $Y$ , of the input image,  $X$ . This applies when you set the `TransformMatrixSource` property to `Property`.

`Y = step(H,X,ROI)` outputs the transformed image of the input image within the region of interest,  $ROI$ . When specifying a rectangular region of interest,  $ROI$  must be a 4-element vector or a 4-by- $R$  matrix. When specifying a polygonal region of interest,  $ROI$  must be a  $2L$ -element vector or a  $2L$ -by- $R$  matrix.  $R$  is the number of regions of interest, and  $L$  is the number of vertices in a polygon region of interest.

`[Y, ROIFLAG]=(H,X,...)` returns a boolean flag,  $ROIFLAG$ , indicating if any part of the region of interest is outside the input image. This applies when you set the `ROIValidityOutputPort` property to `true`.

[Y,CLIPFLAG]=(H,X,...) returns a boolean flag, *CLIPFLAG*, indicating if any transformed pixels were clipped. This applies when you set the *ClippingStatusOutputPort* property to true.

[Y,ROIFLAG,CLIPFLAG] = step(H,X,TFORM,ROI) outputs the transformed image, *Y*, of the input image, *X*, within the region of interest, *ROI*, using the transformation matrix, *TFORM*. *ROIFLAG*, indicates if any part of the region of interest is outside the input image, and *CLIPFLAG*, indicates if any transformed pixels were clipped. This provides all operations simultaneously with all possible inputs. Properties must be set appropriately.

---

**Note:** *H* specifies the System object on which to run this *step* method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the *release* method to unlock the object.

---

## **vision.GeometricTransformEstimator System object**

**Package:** vision

Estimate geometric transformation from matching point pairs

### **Description**

---

**Note:** The `vision.GeometricTransformEstimator` System object will be removed in a future release. Use the `fitgeotrans` function with equivalent functionality instead.

---

The `GeometricTransformEstimator` object estimates geometric transformation from matching point pairs and returns the transform in a matrix. Use this object to compute projective, affine, or nonreflective similarity transformations with robust statistical methods, such as, RANSAC and Least Median of Squares.

Use the step syntax below with input points, `MATCHED_POINTS1` and `MATCHED_POINTS2`, the object, `H`, and any optional properties.

`TFORM = step(H,MATCHED_POINTS1, MATCHED_POINTS2)` calculates the transformation matrix, `TFORM`. The input arrays, `MATCHED_POINTS1` and `MATCHED_POINTS2` specify the locations of matching points in two images. The points in the arrays are stored as a set of  $[x_1 y_1; x_2 y_2; \dots; x_N y_N]$  coordinates, where  $N$  is the number of points. When you set the `Transform` property to `Projective`, the `step` method outputs `TFORM` as a 3-by-3 matrix. Otherwise, the `step` method outputs `TFORM` as a 3-by-2 matrix.

`[TFORM,INLIER_INDEX] = step(H, ...)` additionally outputs the logical vector, `INLIER_INDEX`, indicating which points are the inliers.

### **Construction**

`H = vision.GeometricTransformEstimator` returns a geometric transform estimation System object, `H`. This object finds the transformation matrix that maps the largest number of points between two images.



`H = vision.GeometricTransformEstimator(Name, Value)` returns a geometric transform estimation object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

## Properties

### Transform

Transformation type

Specify transformation type as one of `Nonreflective similarity` | `Affine` | `Projective`.

Default: `Affine`

### ExcludeOutliers

Whether to exclude outliers from input points

Set this property to `true` to find and exclude outliers from the input points and use only the inlier points to calculate the transformation matrix. When you set this property to `false`, all input points are used to calculate the transformation matrix.

Default: `true`

### Method

Method to find outliers

Specify the method to find outliers as one of `Random Sample Consensus (RANSAC)` | `Least Median of Squares`.

Default: `Random Sample Consensus (RANSAC)`.

### AlgebraicDistanceThreshold

Algebraic distance threshold for determining inliers

Specify a scalar threshold value for determining inliers as a positive, scalar value. The threshold controls the upper limit used to find the algebraic distance for the RANSAC

method. This property applies when you set the ??? property to **Projective** and the ??? property to **Random Sample Consensus (RANSAC)**. This property is tunable.

Default: 2.5.

### **PixelDistanceThreshold**

Distance threshold for determining inliers in pixels

Specify the upper limit of the algebraic distance that a point can differ from the projection location of its associating point. Set this property as a positive, scalar value. This property applies when you set the **Transform** property to **Nonreflective similarity** or to **Affine**, and the ??? property to **Random Sample Consensus (RANSAC)**. This property is tunable.

Default: 2.5.

### **NumRandomSamplingsMethod**

How to specify number of random samplings

Indicate how to specify number of random samplings as one of **Specified value** | **Desired confidence**. Set this property to **Desired confidence** to specify the number of random samplings as a percentage and a maximum number. This property applies when you set the ??? property to true and the ??? property to **Random Sample Consensus (RANSAC)**.

Default: **Specified value**.

### **NumRandomSamplings**

Number of random samplings

Specify the number of random samplings as a positive, integer value. This property applies when you set the ??? property to **Specified value**. This property is tunable.

Default: 500.

### **DesiredConfidence**

Probability to find largest group of points

Specify as a percentage, the probability to find the largest group of points that can be mapped by a transformation matrix. This property applies when you set the ??? property to `Desired confidence`. This property is tunable.

Default: 99.

### **MaximumRandomSamples**

Maximum number of random samplings

Specify the maximum number of random samplings as a positive, integer value. This property applies when you set the ??? property to `Desired confidence`. This property is tunable.

Default: 1000.

### **InlierPercentageSource**

Source of inlier percentage

Indicate how to specify the threshold to stop random sampling when a percentage of input point pairs have been found as inliers. You can set this property to one of `Auto` | `Property`. If set to `Auto` then inlier threshold is disabled. This property applies when you set the ??? property to `Random Sample Consensus (RANSAC)`.

Default: `Auto`.

### **InlierPercentage**

Percentage of point pairs to be found to stop random sampling

Specify the percentage of point pairs that needs to be determined as inliers, to stop random sampling. This property applies when you set the ??? property to `Property`. This property is tunable.

Default: 75.

### **RefineTransformMatrix**

Whether to refine transformation matrix

Set this property to `true` to perform additional iterative refinement on the transformation matrix. This property applies when you set the ??? property to `true`.

Default: `false`.

### **TransformMatrixDataType**

Data type of the transformation matrix

Specify transformation matrix data type as one of `single` | `double` when the input points are built-in integers. This property is not used when the data type of points is `single` or `double`.

Default: `single`.

## **Methods**

<code>clone</code>	Create geometric transform estimator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Calculate transformation matrix mapping largest number of valid points from input arrays

## **Examples**

### **Recover Transformed Image Using SURF Feature Points**

Detect, extract, and match SURF features from two images.

Read and transform input images

```
Iin = imread('cameraman.tif'); imshow(Iin); title('Base image');  
Iout = imresize(Iin, 0.7); Iout = imrotate(Iout, 31);  
figure; imshow(Iout); title('Transformed image');
```

Detect and extract features from both images.

```
ptsIn = detectSURFFeatures(Iin);
ptsOut = detectSURFFeatures(Iout);
[featuresIn validPtsIn] = extractFeatures(Iin, ptsIn);
[featuresOut validPtsOut] = extractFeatures(Iout, ptsOut);
```

Match feature vectors.

```
index_pairs = matchFeatures(featuresIn, featuresOut);
```

Get matching points.

```
matchedPtsIn = validPtsIn(index_pairs(:,1));
matchedPtsOut = validPtsOut(index_pairs(:,2));
figure; showMatchedFeatures(Iin,Iout,matchedPtsIn,matchedPtsOut);
title('Matched SURF points, including outliers');
```

Compute the transformation matrix using RANSAC.

```
gte = vision.GeometricTransformEstimator;
gte.Transform = 'Nonreflective similarity';
[tform inlierIdx] = step(gte, matchedPtsOut.Location, matchedPtsIn.Location);
figure; showMatchedFeatures(Iin,Iout,matchedPtsIn(inlierIdx),matchedPtsOut(inlierIdx));
title('Matching inliers'); legend('inliersIn', 'inliersOut');
```

Recover the original image.

```
agt = vision.GeometricTransformer;
Ir = step(agt, im2single(Iout), tform);
figure; imshow(Ir); title('Recovered image');
```

### Transform an Image According to Specified Points

Create and transform a checkerboard image to specified points.

Create a checkerboard input image.

```
input = checkerboard;
[h, w] = size(input);
inImageCorners = [1 1; w 1; w h; 1 h];
outImageCorners = [4 21; 21 121; 79 51; 26 6];
hgte1 = vision.GeometricTransformEstimator('ExcludeOutliers', false);
tform = step(hgte1, inImageCorners, outImageCorners);
```

Use tform to transform the image.

```
hgt = vision.GeometricTransformer;  
output = step(hgt, input, tform);  
figure; imshow(input); title('Original image');  
figure; imshow(output); title('Transformed image');
```

## Troubleshooting

The success of estimating the correct geometric transformation depends heavily on the quality of the input point pairs. If you chose the RANSAC or LMS algorithm, the block will randomly select point pairs to compute the transformation matrix and will use the transformation that best fits the input points. There is a chance that all of the randomly selected point pairs may contain outliers despite repeated samplings. In this case, the output transformation matrix, `TForm`, is invalid, indicated by a matrix of zeros.

To improve your results, try the following:

- Increase the percentage of inliers in the input points.

- Increase the number for random samplings.

- For the RANSAC method, increase the desired confidence.

- For the LMS method, make sure the input points have 50% or more inliers.

- Use features appropriate for the image contents

- Be aware that repeated patterns, for example, windows in office building, will cause false matches when you match the features. This increases the number of outliers.

- Do not use this function if the images have significant parallax. You can use the `estimateFundamentalMatrix` function instead.

- Choose the minimum transformation for your problem.

- If a projective transformation produces the error message, “A portion of the input image was transformed to the location at infinity. Only transformation matrices that do not transform any part of the image to infinity are supported.”, it is usually caused by a transformation matrix and an image that would result in an output distortion that does not fit physical reality. If the matrix was an output of the `GeometricTransformationEstimator` object, then most likely it could not find enough inliers.

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Estimate Geometric Transformation` block reference page. The object properties correspond to the block parameters.

## See Also

`vision.GeometricTransformer` | `detectSURFFeatures` | `extractFeatures` |  
`estimateFundamentalMatrix` | `estimateGeometricTransform`

**Introduced in R2012a**

# clone

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Create geometric transform estimator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



## getNumInputs

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GeometricTransformEstimator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.GeometricTransformEstimator

**Package:** vision

Calculate transformation matrix mapping largest number of valid points from input arrays

## Syntax

```
TFORM = step(H,MATCHED_POINTS1, MATCHED_POINTS2)  
[TFORM,INLIER_INDEX] = step(H, ...)
```

## Description

`TFORM = step(H,MATCHED_POINTS1, MATCHED_POINTS2)` calculates the transformation matrix, `TFORM`. The input arrays, `MATCHED_POINTS1` and `MATCHED_POINTS2` specify the locations of matching points in two images. The points in the arrays are stored as a set of  $[x_1 y_1; x_2 y_2; \dots; x_N y_N]$  coordinates, where  $N$  is the number of points. When you set the `Transform` property to `Projective`, the `step` method outputs `TFORM` as a 3-by-3 matrix. Otherwise, the `step` method outputs `TFORM` as a 3-by-2 matrix.

`[TFORM,INLIER_INDEX] = step(H, ...)` additionally outputs the logical vector, `INLIER_INDEX`, indicating which points are the inliers.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.GeometricTranslator System object

**Package:** vision

Translate image in two-dimensional plane using displacement vector

### Description

---

**Note:** The `vision.GeometricTranslator` System object will be removed in a future release. Use the `imtranslate` function with equivalent functionality instead.

---

The `GeometricTranslator` object translates images in two-dimensional plane using displacement vector.

### Construction

`H = vision.GeometricTranslator` returns a System object, `H`. This object moves an image up or down, and left or right.

`H = vision.GeometricTranslator(Name, Value)` returns a geometric translator System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“Code Generation Support, Usage Notes, and Limitations”

### Properties

#### OutputSize

Output size as full or same as input image size

Specify the size of output image after translation as one of `Full` | `Same as input image`. The default is `Full`. When you set this property to `Full`, the object outputs a matrix that contains the translated image values. When you set this property to `Same as input image`, the object outputs a matrix that is the same size as the input image and contains a portion of the translated image.

### **OffsetSource**

Source of specifying offset values

Specify how the translation parameters are provided as one of `Input port` | `Property`. The default is `Property`. When you set the ??? property to `Input port`, a two-element offset vector must be provided to the object's `step` method.

### **Offset**

Translation values

Specify the number of pixels to translate the image as a two-element offset vector. The default is `[1.5 2.3]`. The first element of the vector represents a shift in the vertical direction and a positive value moves the image downward. The second element of the vector represents a shift in the horizontal direction and a positive value moves the image to the right. This property applies when you set the ??? property to `Property`.

### **MaximumOffset**

Maximum number of pixels by which to translate image

Specify the maximum number of pixels by which to translate the input image as a two-element real vector with elements greater than 0. The default is `[8 10]`. This property must have the same data type as the `Offset` input. This property applies when you set the ??? property to `Full` and ??? property to `Input port`. The system object uses this property to determine the size of the output matrix. If the `Offset` input is greater than this property value, the object saturates to the maximum value.

### **BackgroundFillValue**

Value of pixels outside image

Specify the value of pixels that are outside the image as a numeric scalar value or a numeric vector of same length as the third dimension of input image. The default is 0.

### **InterpolationMethod**

Interpolation method used to translate image

Specify the interpolation method used to translate the image as one of **Nearest neighbor** | **Bilinear** | **Bicubic**. The default is **Bilinear**. When you set this property to **Nearest neighbor**, the object uses the value of the nearest pixel for the new pixel value. When you set this property to **Bilinear**, the object sets the new pixel value as the weighted average of the four nearest pixel values. When you set this property to **Bicubic**, the object sets the new pixel value as the weighted average of the sixteen nearest pixel values.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Nearest**.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of **Wrap** | **Saturate**. The default is **Saturate**.

#### **OffsetValuesDataType**

Offset word and fraction lengths

Specify the offset fixed-point data type as one of **Same word length as input** | **Custom**. The default is **Same word length as input**.

#### **CustomOffsetValuesDataType**

Offset word and fraction lengths

Specify the offset fixed-point type as a signed `numericType` object with a **Signedness** of **Auto**. This property applies when you set the `???` property to **Custom**. The default is `numericType([], 16, 6)`.



**ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as first input` | `Custom`. The default is `Custom`. This property is applies when you set the ??? property to `Bilinear` or `Bicubic`.

**CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. This property is applies when you set the ??? property to `Bilinear` or `Bicubic`, and the ??? property to `Custom`. The default is `numericType([],32,10)`.

**AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as first input` | `Custom`. The default is `Same as product`. This property is applies when you set the ??? property to `Bilinear` or `Bicubic`.

**CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`. The default is `numericType([],32,10)`. This property is applies when you set the ??? property to `Bilinear` or `Bicubic`, and the ??? property to `Custom`

**OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as first input` | `Custom`. The default is `Same as first input`.

**CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`. The default is `numericType([],32,10)`.

## Methods

<code>clone</code>	Create geometric translator object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Return a translated image

## Examples

Translate an image

```
htranslate=vision.GeometricTranslator;  
htranslate.OutputSize='Same as input image';  
htranslate.Offset=[30 30];  
  
I=im2single(imread('cameraman.tif'));  
Y = step(htranslate,I);  
imshow(Y);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Translate` block reference page. The object properties correspond to the block parameters.

## See Also

`vision.GeometricRotator`

**Introduced in R2012a**

# clone

**System object:** vision.GeometricTranslator

**Package:** vision

Create geometric translator object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.GeometricTranslator

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.GeometricTranslator

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.GeometricTranslator

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the GeometricTranslator System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.GeometricTranslator

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.GeometricTranslator

**Package:** vision

Return a translated image

## Syntax

$Y = \text{step}(H, I)$

$Y = \text{step}(H, I, \text{OFFSET})$

## Description

$Y = \text{step}(H, I)$  translates the input image  $I$ , and returns a translated image  $Y$ , with the offset specified by the **Offset** property.

$Y = \text{step}(H, I, \text{OFFSET})$  uses input *OFFSET* as the offset to translate the image  $I$ . This applies when you set the **OffsetSource** property to **Input port**. *OFFSET* is a two-element offset vector that represents the number of pixels to translate the image. The first element of the vector represents a shift in the vertical direction and a positive value moves the image downward. The second element of the vector represents a shift in the horizontal direction and a positive value moves the image to the right.

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.Histogram System object

**Package:** vision

Generate histogram of each input matrix

### Description

---

**Note:** The `vision.Histogram` System object will be removed in a future release. Use the `imhist` function with equivalent functionality instead.

---

The `Histogram` object generates histogram of each input matrix.

### Construction

`H = vision.Histogram` returns a histogram System object, `H`, that computes the frequency distribution of the elements in each input matrix.

`H = vision.Histogram(Name, Value, ...)` returns a histogram object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

`H = vision.Histogram(MIN, MAX, NUMBINS, Name, Value, ...)` returns a histogram System object, `H`, with the `???` property set to `MIN`, `???` property set to `MAX`, `???` property set to `NUMBINS` and other specified properties set to the specified values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### LowerLimit

Lower boundary

Specify the lower boundary of the lowest-valued bin as a real-valued scalar value. `NaN` and `Inf` are not valid values for this property. The default is `0`. This property is tunable.

### UpperLimit

Upper boundary

Specify the upper boundary of the highest-valued bin as a real-valued scalar value. `NaN` and `Inf` are not valid values for this property. The default is `1`. This property is tunable.

### NumBins

Number of bins in the histogram

Specify the number of bins in the histogram. The default is `256`.

### Normalize

Enable output vector normalization

Specify whether the output vector,  $v$ , is normalized such that  $\text{sum}(v) = 1$ . Use of this property is not supported for fixed-point signals. The default is `false`.

### RunningHistogram

Enable calculation over successive calls to `step` method

Set this property to `true` to enable computing the histogram of the input elements over successive calls to the `step` method. Set this property to `false` to enable basic histogram operation. The default is `false`.

### ResetInputPort

Enable resetting in running histogram mode

Set this property to `true` to enable resetting the running histogram. When the property is set to `true`, a reset input must be specified to the `step` method to reset the running histogram. This property applies when you set the `???` property to `true`. When you set this property to `false`, the object does not reset. The default is `false`.

### **ResetCondition**

Condition for running histogram mode

Specify event to reset the running histogram as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. This property applies when you set the `???` property to `true`. The default is `Non-zero`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

#### **ProductDataType**

Data type of product

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled `numerictype` object. This property applies only when you set the `???` property to `Custom`. The default is `numerictype(true,32,30)`.

## AccumulatorDataType

Data type of the accumulator

Specify the accumulator data type as `Same as input`, `Same as product`, or `Custom`. The default is `Same as input`.

## CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numericType` object. This property applies only when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

## Methods

<code>clone</code>	Create 2-D histogram object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>reset</code>	Reset histogram bin values to zero
<code>step</code>	Return histogram for input data

## Examples

Compute histogram of a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));
hhist2d = vision.Histogram;
y = step(hhist2d,img);
bar((0:255)/256, y);
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the 2D-Histogram block reference page. The object properties correspond to the block parameters, except:

- **Reset port** block parameter corresponds to both the `ResetCondition` and the `ResetInputPort` object properties.
- The **Find Dimensions Over** block parameter with `Entire input` or `Each column` options, does not have a corresponding property. The object finds dimensions over the entire input.

### See Also

`dsp.Histogram`

**Introduced in R2012a**

# clone

**System object:** vision.Histogram

**Package:** vision

Create 2-D histogram object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Histogram

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).



# getNumOutputs

**System object:** vision.Histogram

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Histogram

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Histogram System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.Histogram

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## reset

**System object:** vision.Histogram

**Package:** vision

Reset histogram bin values to zero

## Syntax

reset(H)

## Description

reset(H) sets the Histogram object bin values to zero when the RunningHistogram property is true.

---

## step

**System object:** vision.Histogram

**Package:** vision

Return histogram for input data

## Syntax

$Y = \text{step}(H, X)$

$Y = \text{step}(H, X, R)$

## Description

$Y = \text{step}(H, X)$  returns a histogram  $y$  for the input data  $X$ . When you set the `RunningHistogram` property to `true`,  $Y$  corresponds to the histogram of the input elements over successive calls to the `step` method.

$Y = \text{step}(H, X, R)$  computes the histogram of the input  $X$  elements over successive calls to the `step` method, and optionally resets the object's state based on the value of  $R$  and the object's `ResetCondition` property. This applies when you set the `RunningHistogram` and `ResetInputPort` properties to `true`.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.HistogramBasedTracker System object

**Package:** vision

Histogram-based object tracking

### Description

The histogram-based tracker incorporates the continuously adaptive mean shift (CAMShift) algorithm for object tracking. It uses the histogram of pixel values to identify the tracked object.

Use the `step` method with input image, `I`, the histogram-based tracker object `H`, and any optional properties to identify the bounding box of the tracked object.

`BBOX = step(H, I)` returns the bounding box, `BBOX`, of the tracked object. The bounding box output is in the format `[x y width height]`. Before calling the `step` method, you must identify the object to track and set the initial search window. Use the `initializeObject` method to do this.

`[BBOX, ORIENTATION] = step(H, I)` additionally returns the angle between the x-axis and the major axis of the ellipse that has the same second-order moments as the object. The range of the returned angle can be from  $-\pi/2$  to  $\pi/2$ .

`[BBOX, ORIENTATION, SCORE] = step(H, I)` additionally returns the confidence score indicating whether the returned bounding box, `BBOX`, contains the tracked object. `SCORE` is between 0 and 1, with the greatest confidence equal to 1.

Use the `initializeObject` method before calling the `step` method in order to set the object to track and to set the initial search window.

`initializeObject(H, I, R)` sets the object to track by extracting it from the `[x y width height]` region `R` located in the 2-D input image, `I`. The input image, `I`, can be any 2-D feature map that distinguishes the object from the background. For example, the image can be a hue channel of the HSV color space. Typically, `I` will be the first frame in which the object appears. The region, `R`, is also used for the initial search window, in the next call to the `step` method. For best results, the object must occupy the majority of the region, `R`.

`initializeObject(H,I,R,N)` additionally, lets you specify `N`, the number of histogram bins. By default, `N` is set to 16. Increasing `N` enhances the ability of the tracker to discriminate the object. However, this approach also narrows the range of changes to the object's visual characteristics that the tracker can accommodate. Consequently, this narrow range increases the likelihood of losing track.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”

---

**Tip** You can improve the computational speed of the `HistogramBasedTracker` object by setting the class of the input image to `uint8`.

---

## Construction

`H = vision.HistogramBasedTracker` returns a System object, `H`, that tracks an object by using the CAMShift algorithm. It uses the histogram of pixel values to identify the tracked object. To initialize the tracking process, you must use the `initializeObject` method to specify an exemplar image of the object. After you specify the image of the object, use the `step` method to track the object in consecutive video frames.

`BBOX = vision.HistogramBasedTracker(Name, Value)` returns a tracker System object with one or more name-value pair arguments. Unspecified properties have default values.

## Properties

### ObjectHistogram

Normalized pixel value histogram

An  $N$ -element vector. This vector specifies the normalized histogram of the object's pixel values. Histogram values must be normalized to a value between 0 and 1. You can use the `initializeObject` method to set the property. This property is tunable.

Default: [ ]

## Methods

clone	Create histogram-based tracker object with same property values
initializeObject	Set object to track
initializeSearchWindow	Initialize search window
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Histogram-based object tracking

## Examples

### Track a Face

Track and display a face in each frame of an input video.

Create System objects for reading and displaying video and for drawing a bounding box of the object.

```
videoFileReader = vision.VideoFileReader('vipcolorsegmentation.avi');  
videoPlayer = vision.VideoPlayer();  
shapeInserter = vision.ShapeInserter('BorderColor', 'Custom', 'CustomBorderColor', [1 0 0]);
```

Read the first video frame, which contains the object. Convert the image to HSV color space. Then define and display the object region.

```
objectFrame = step(videoFileReader);  
objectHSV = rgb2hsv(objectFrame);  
objectRegion = [40, 45, 25, 25];  
objectImage = step(shapeInserter, objectFrame, objectRegion);  
figure; imshow(objectImage); title('Red box shows object region');
```

Optionally, you can select the object region using your mouse. The object must occupy the majority of the region. Use the following command.



```
figure; imshow(objectFrame); objectRegion=round(getPosition(imrect))
```

Set the object, based on the hue channel of the first video frame.

```
tracker = vision.HistogramBasedTracker;  
initializeObject(tracker, objectHSV(:,:,1) , objectRegion);
```

Track and display the object in each video frame. The while loop reads each image frame, converts the image to HSV color space, then tracks the object in the hue channel where it is distinct from the background. Finally, the example draws a box around the object and displays the results.

```
while ~isDone(videoFileReader)  
    frame = step(videoFileReader);  
    hsv = rgb2hsv(frame);  
    bbox = step(tracker, hsv(:,:,1));  
  
    out = step(shapeInserter, frame, bbox);  
    step(videoPlayer, out);  
end
```

Release the video reader and player.

```
release(videoPlayer);  
release(videoFileReader);
```

## References

Bradski, G. and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly Media Inc.: Sebastopol, CA, 2008.

## See Also

[imrect](#) | [rgb2hsv](#) | [size](#) | [vision.ShapeInserter](#)

**Introduced in R2012a**

# clone

**System object:** vision.HistogramBasedTracker

**Package:** vision

Create histogram-based tracker object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# initializeObject

**System object:** vision.HistogramBasedTracker

**Package:** vision

Set object to track

## Syntax

```
initializeObject(H,I,R)
```

## Description

Use the `initializeObject` method to set the object to track, and to set the initial search window. Use this method before calling the `step` method.

`initializeObject(H,I,R)` sets the object to track by extracting it from the [x y width height] region `R` located in the 2-D input image, `I`. The input image, `I`, can be any 2-D feature map that distinguishes the object from the background. For example, the image can be a hue channel of the HSV color space. Typically, `I` will be the first frame in which the object appears. The region, `R`, is also used for the initial search window, in the next call to the `step` method. For best results, the object must occupy the majority of the region, `R`.

`initializeObject(H,I,R,N)` additionally, lets you specify `N`, the number of histogram bins. By default, `N` is set to **16**. Increasing `N` enhances the ability of the tracker to discriminate the object. However, this approach also narrows the range of changes to the object's visual characteristics that the tracker can accommodate. Consequently, this narrow range increases the likelihood of losing track.

## initializeSearchWindow

**System object:** vision.HistogramBasedTracker

**Package:** vision

Initialize search window

### Syntax

```
initializeSearchWindow(H,R)
```

### Description

`initializeSearchWindow(H,R)` sets the initial search window, `R`, specified in the format, `[x y width height]`. The next call to the `step` method will use `R` as the initial window to search for the object. Use this method when you lose track of the object. You can also use it to re-initialize the object's initial location and size.

## isLocked

**System object:** vision.HistogramBasedTracker

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

isLocked(H)

## Description

isLocked(H) returns the locked state of the histogram-based detector.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.HistogramBasedTracker

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You cannot use the release method on System objects in Embedded MATLAB.

---

## step

**System object:** vision.HistogramBasedTracker

**Package:** vision

Histogram-based object tracking

## Syntax

```
BBOX = step(H,I)
[BBOX,ORIENTATION] = step(H,I)
[BBOX,ORIENTATION, SCORE] = step(H,I)
```

## Description

`BBOX = step(H,I)` returns the bounding box, `BBOX`, of the tracked object. The bounding box output is in the format [x y width height]. Before calling the `step` method, you must identify the object to track, and set the initial search window. Use the `initializeObject` method to do this.

`[BBOX,ORIENTATION] = step(H,I)` additionally returns the angle between the x-axis and the major axis of the ellipse that has the same second-order moments as the object. The returned angle is between  $-\pi/2$  and  $\pi/2$ .

`[BBOX,ORIENTATION, SCORE] = step(H,I)` additionally returns the confidence score for the returned bounding box, `BBOX`, that contains the tracked object. `SCORE` is between 0 and 1, with the greatest confidence equal to 1.

Before calling the `step` method, you must identify the object to track, and set the initial search window. Use the `initializeObject` method to do this.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an

input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.HistogramEqualizer System object

**Package:** vision

Enhance contrast of images using histogram equalization

## Description

---

**Note:** The `vision.HistogramEqualizer` System object will be removed in a future release. Use the `histeq` function with equivalent functionality instead.

---

The `HistogramEqualizer` object enhances contrast of images using histogram equalization.

## Construction

`H = vision.HistogramEqualizer` returns a System object, `H`. This object enhances the contrast of input image using histogram equalization.

`H = vision.HistogramEqualizer(Name, Value)` returns a histogram equalization object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### Histogram

How to specify histogram

Specify the desired histogram of the output image as one of `Uniform` | `Input port` | `Custom`. The default is `Uniform`.

### **CustomHistogram**

Desired histogram of output image

Specify the desired histogram of output image as a numeric vector. This property applies when you set the ??? property to `Custom`.

The default is `ones(1,64)`.

### **BinCount**

Number of bins for uniform histogram

Specify the number of equally spaced bins the uniform histogram has as an integer scalar value greater than 1. This property applies when you set the ??? property to `Uniform`. The default is `64`.

## **Methods**

<code>clone</code>	Create histogram equalizer object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform histogram equalization on input image

## **Examples**

Enhance quality of an image:

```
hhisteq = vision.HistogramEqualizer;  
x = imread('tire.tif');  
y = step(hhisteq, x);  
imshow(x); title('Original Image');  
figure, imshow(y);  
title('Enhanced Image after histogram equalization');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Histogram Equalization](#) block reference page. The object properties correspond to the block parameters, except:

The **Histogram** property for the object, corresponds to both the **Target Histogram** and the **Histogram Source** parameters for the block.

The HistogramEqualizer System object does not support variable size signals.

**Introduced in R2012a**

# clone

**System object:** vision.HistogramEqualizer

**Package:** vision

Create histogram equalizer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.HistogramEqualizer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.HistogramEqualizer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.HistogramEqualizer

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the HistogramEqualizer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.HistogramEqualizer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.HistogramEqualizer

**Package:** vision

Perform histogram equalization on input image

## Syntax

$Y = \text{step}(H, X)$

$Y = \text{step}(H, X, HIST)$

## Description

$Y = \text{step}(H, X)$  performs histogram equalization on input image,  $X$ , and returns the enhanced image,  $Y$ .

$Y = \text{step}(H, X, HIST)$  performs histogram equalization on input image,  $X$  using input histogram,  $HIST$ , and returns the enhanced image,  $Y$  when you set the Histogram property to Input port.

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.HoughLines System object

**Package:** vision

Find Cartesian coordinates of lines that are described by rho and theta pairs

### Description

The `HoughLines` object finds Cartesian coordinates of lines that are described by rho and theta pairs. The object inputs are the theta and rho values of lines and a reference image. The object outputs the one-based row and column positions of the intersections between the lines and two of the reference image boundary lines. The boundary lines are the left and right vertical boundaries and the top and bottom horizontal boundaries of the reference image.

### Construction

`H = vision.HoughLines` returns a Hough lines System object, `H`, that finds Cartesian coordinates of lines that are described by rho and theta pairs.

`H = vision.HoughLines(Name, Value)` returns a Hough lines object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### SineComputation

Method to calculate sine values used to find intersections of lines

Specify how to calculate sine values which are used to find intersection of lines as `Trigonometric function`, or `Table lookup`. If this property is set to `Trigonometric function`, the object computes sine and cosine values it needs to calculate the intersections of the lines. If it is set to `Table lookup`, the object computes and stores the trigonometric values it needs to calculate the intersections of the lines in a table and uses the table for each step call. In this case, the object requires extra memory. For floating-point inputs, this property must be set to `Trigonometric function`. For fixed-point inputs, the property must be set to `Table lookup`. The default is `Table lookup`.

### **ThetaResolution**

Spacing of the theta-axis

Specify the spacing of the theta-axis. This property applies when you set the ??? property to `Table lookup`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the ??? property to `Table lookup`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the ??? property to `Table lookup`. The default is `Wrap`.

#### **SineTableDataType**

Sine table word and fraction lengths

Specify the sine table fixed-point data type as a constant property always set to `Custom`. This property applies when you set the ??? property to `Table lookup`.

#### **CustomSineTableDataType**

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Table lookup`, and the `???` property to `Custom`. The default is `numericType([], 16)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input`, `Custom`. This property applies when you set the `???` property to `Table lookup`. The default is `Custom`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Table lookup`, and the `???` property to `Custom`. The default is `numericType([], 32, 16)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Custom`. This property applies when you set the `???` property to `Table lookup`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Table lookup`, and the `???` property to `Custom`. The default is `numericType([], 32, 16)`.

## **Methods**

<code>clone</code>	Create hough lines object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Output intersection coordinates of a line described by a theta and rho pair and reference image boundary lines

## Examples

### Detect Longest Line In An Image

#### Read the intensity image.

```
I = imread('circuit.tif');
```

#### Create an edge detector, Hough transform, local maxima finder, and Hough lines objects.

```
hedge = vision.EdgeDetector;
hhoughtrans = vision.HoughTransform(pi/360, 'ThetaRhoOutputPort', true);
hfindmax = vision.LocalMaximaFinder(1, 'HoughMatrixInput', true);
houghlines = vision.HoughLines('SineComputation', 'Trigonometric function');
```

Warning: The vision.EdgeDetector will be removed in a future release. Use the edge function with equivalent functionality instead.

Warning: The vision.HoughTransform will be removed in a future release. Use the hough function with equivalent functionality instead.

#### Find the edges in the intensity image

```
BW = step(hedge, I);
```

#### Run the edge output through the transform

```
[ht, theta, rho] = step(hhoughtrans, BW);
```

#### Find the location of the max value in the Hough matrix.

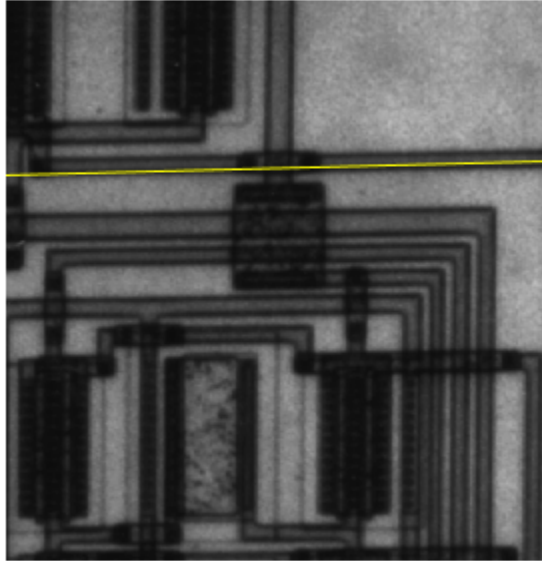
```
idx = step(hfindmax, ht);
```

#### Find the longest line.

```
linepts = step(houghlines, theta(idx(1)-1), rho(idx(2)-1), I);
```

View the image superimposed with the longest line.

```
imshow(I); hold on;  
line(linepts([1 3])-1, linepts([2 4])-1,'color',[1 1 0]);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Hough Lines block reference page. The object properties correspond to the block parameters.

### See Also

[hough](#) | [vision.LocalMaximaFinder](#) | [edge](#)

Introduced in R2012a

# clone

**System object:** vision.HoughLines

**Package:** vision

Create hough lines object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.HoughLines

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).



## getNumOutputs

**System object:** vision.HoughLines

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.HoughLines

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the HoughLines System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

# release

**System object:** vision.HoughLines

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.HoughLines

**Package:** vision

Output intersection coordinates of a line described by a theta and rho pair and reference image boundary lines

### Syntax

PTS = step(H, THETA, RHO, REFIMG)

### Description

PTS = step(H, THETA, RHO, REFIMG) outputs PTS as the zero-based row and column positions of the intersections between the lines described by THETA and RHO and two of the reference image boundary lines.

---

**Note:** H specifies the System object on which to run this step method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

# vision.HoughTransform System object

**Package:** vision

Find lines in images via Hough transform

## Description

---

**Note:** The `vision.HoughTransform` System object will be removed in a future release. Use the `hough` function with equivalent functionality instead.

---

The `HoughTransform` object finds lines in images via Hough transform. The Hough transform maps points in the Cartesian image space to curves in the Hough parameter space using the following equation:

$$\rho = x * \cos(\theta) + y * \sin(\theta)$$

Here,  $\rho$  denotes the distance from the origin to the line along a vector perpendicular to the line, and  $\theta$  denotes the angle between the  $x$ -axis and this vector. This object computes the parameter space matrix, whose rows and columns correspond to the  $\rho$  and  $\theta$  values respectively. Peak values in this matrix represent potential straight lines in the input image.

## Construction

`H = vision.HoughTransform` returns a Hough transform System object, `H`, that implements the Hough transform to detect lines in images.

`H = vision.HoughTransform(Name, Value)` returns a Hough transform object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

`H = vision.HoughTransform(THETARES, RHORES, 'Name', Value, ...)` returns a Hough transform object, `H`, with the `THETARES` property set to `THETARES`, the `RHORES` property set to `RHORES`, and other specified properties set to the specified values.

<b>Code Generation Support</b>
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### **ThetaResolution**

Theta resolution in radians

Specify the spacing of the Hough transform bins along the theta-axis in radians, as a scalar numeric value between 0 and  $\pi/2$ . The default is  $\pi/180$ .

### **RhoResolution**

Rho resolution

Specify the spacing of the Hough transform bins along the rho-axis as a scalar numeric value greater than 0. The default is 1.

### **ThetaRhoOutputPort**

Enable theta and rho outputs

Set this property to `true` for the object to output theta and rho values. The default is `false`.

### **OutputDataType**

Data type of output

Specify the data type of the output signal as `double`, `single`, or `Fixed point`. The default is `double`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the ??? property to `Fixed point`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the ??? property to `Fixed point`. The default is `Saturate`.

### **SineTableDataType**

Sine table word and fraction lengths

This property is constant and is set to `Custom`. This property applies when you set the ??? property to `Fixed point`.

### **CustomSineTableDataType**

Sine table word and fraction lengths

Specify the sine table fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Fixed point`. The default is `numericType([],16,14)`.

### **RhoDataType**

Rho word and fraction lengths

This property is constant and is set to `Custom`. This property applies when you set the ??? property to `Fixed point`.

### **CustomRhoDataType**

Rho word and fraction lengths

Specify the rho fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Fixed point`. The default is `numericType([],32,16)`.

### **ProductDataType**

Product word and fraction lengths

This property is constant and is set to **Custom**. This property applies when you set the ??? property to **Fixed point**.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled **numericType** object with a **Signedness** of **Auto**. This property applies when you set the ??? property to **Fixed point**. The default is `numericType([],32,20)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as **Same as product**, **Custom**. This property applies when you set the ??? property to **Fixed point**. The default is **Custom**.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled **numericType** object with a **Signedness** of **Auto**. This property applies when you set the ??? property to **Fixed point**. The default is `numericType([],32,20)`.

### **HoughOutputDataType**

Hough output word and fraction lengths

This property is constant and is set to **Custom**. This property applies when you set the ??? property to **Fixed point**.

### **CustomHoughOutputDataType**

Hough output word and fraction lengths

Specify the hough output fixed-point data type as an unscaled **numericType** object with a **Signedness** of **Auto**. This property applies when you set the ??? property to **Fixed point**. The default is `numericType(false,16)`.

### **ThetaOutputDataType**

Theta output word and fraction lengths



This property is constant and is set to **Custom**. This property applies when you set the ??? property to **Fixed point**.

### **CustomThetaOutputDataType**

Theta output word and fraction lengths

Specify the theta output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to **Fixed point**. The default is `numericType([], 32, 16)`.

## **Methods**

<code>clone</code>	Create Hough transform object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Output parameter space matrix for binary input image matrix

## **Examples**

### **Detect Longest Line In An Image**

**Read the intensity image.**

```
I = imread('circuit.tif');
```

**Create an edge detector, Hough transform, local maxima finder, and Hough lines objects.**

```
hedge = vision.EdgeDetector;
```

```
hhoughtrans = vision.HoughTransform(pi/360,'ThetaRhoOutputPort', true);  
hfindmax = vision.LocalMaximaFinder(1, 'HoughMatrixInput', true);  
hhoughlines = vision.HoughLines('SineComputation','Trigonometric function');
```

Warning: The vision.EdgeDetector will be removed in a future release. Use the edge function with equivalent functionality instead.

Warning: The vision.HoughTransform will be removed in a future release. Use the hough function with equivalent functionality instead.

### **Find the edges in the intensity image**

```
BW = step(hedge, I);
```

### **Run the edge output through the transform**

```
[ht, theta, rho] = step(hhoughtrans, BW);
```

### **Find the location of the max value in the Hough matrix.**

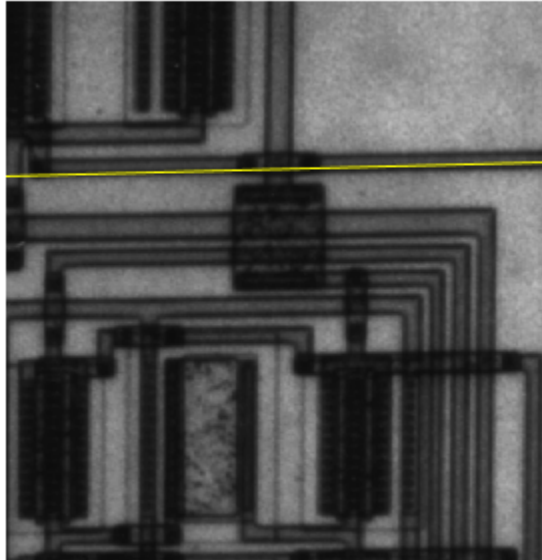
```
idx = step(hfindmax, ht);
```

### **Find the longest line.**

```
linepts = step(hhoughlines, theta(idx(1)-1), rho(idx(2)-1), I);
```

### **View the image superimposed with the longest line.**

```
imshow(I); hold on;  
line(linepts([1 3])-1, linepts([2 4])-1,'color',[1 1 0]);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Hough Transform block reference page. The object properties correspond to the block parameters.

### See Also

[vision.DCT](#) | [vision.LocalMaximaFinder](#) | [edge](#) | [hough](#)

**Introduced in R2012a**

# clone

**System object:** vision.HoughTransform

**Package:** vision

Create Hough transform object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.HoughTransform

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.HoughTransform

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.HoughTransform

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the HoughTransform System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.HoughTransform

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.HoughTransform

**Package:** vision

Output parameter space matrix for binary input image matrix

## Syntax

HT = step(H,BW)

[HT,THETA,RHO] = step(H,BW)

## Description

HT = step(H,BW) outputs the parameter space matrix, HT , for the binary input image matrix BW.

[HT,THETA,RHO] = step(H,BW) also returns the theta and rho values, in vectors THETA and RHO respectively, when you set the `ThetaRhoOutputPort` property to `true`. RHO denotes the distance from the origin to the line along a vector perpendicular to the line, and THETA denotes the angle between the x-axis and this vector. This object computes the parameter space matrix, whose rows and columns correspond to the rho and theta values respectively. Peak values in this matrix represent potential straight lines in the input image.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.IDCT System object

**Package:** vision

Compute 2-D inverse discrete cosine transform

### Description

The IDCT object computes 2-D inverse discrete cosine transform of the input signal. The number of rows and columns of the input matrix must be a power of 2.

### Construction

`H = vision.IDCT` returns a System object, `H`, used to compute the two-dimensional inverse discrete cosine transform (2-D IDCT) of a real input signal.

`H = vision.IDCT(Name, Value)` returns a 2-D inverse discrete cosine transform System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### SineComputation

Specify how the System object computes sines and cosines as `Trigonometric` function, or `Table lookup`. This property must be set to `Table lookup` for fixed-point inputs.

## **Fixed-Point Properties**

### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when you set the ??? to `Table lookup`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the ??? to `Table lookup`. The default is `Wrap`.

### **SineTableDataType**

Sine table word-length designation

Specify the sine table fixed-point data type as `Same word length as input`, or `Custom`. This property applies when you set the ??? to `Table lookup`. The default is `Same word length as input`.

### **CustomSineTableDataType**

Sine table word length

Specify the sine table fixed-point type as a signed, unscaled `numericType` object. This property applies when you set the ??? to `Table lookup` and you set the ??? property to `Custom`. The default is `numericType(true,16)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. This property applies when you set the ??? to `Table lookup`. The default is `Custom`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `???` to `Table lookup`, and the `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product`, `Same as first input`, `Custom`. This property applies when you set the `???` property to `Table lookup`. The default is `Full precision`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `???` to `Table lookup`, and `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. This property applies when you set the `???` to `Table lookup`. The default is `Custom`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `???` to `Table lookup`, and the `???` property to `Custom`. The default is `numericType(true,16,15)`.

## **Methods**

`clone`

Create 2-D inverse discrete cosine transform object with same property values

<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute 2-D inverse discrete cosine transform of the input

## Examples

Use 2-D discrete cosine transform (DCT) to analyze the energy content in an image. Set the DCT coefficients lower than a threshold of 0. Reconstruct the image using 2-D inverse discrete cosine transform (IDCT).

```
hdct2d = vision.DCT;  
I = double(imread('cameraman.tif'));  
J = step(hdct2d, I);  
imshow(log(abs(J)), [], colormap(jet(64)), colorbar)  
  
hidct2d = vision.IDCT;  
J(abs(J) < 10) = 0;  
It = step(hidct2d, J);  
figure, imshow(I, [0 255]), title('Original image')  
figure, imshow(It, [0 255]), title('Reconstructed image')
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D IDCT block reference page. The object properties correspond to the block parameters.

## See Also

`vision.DCT`

**Introduced in R2012a**

# clone

**System object:** vision.IDCT

**Package:** vision

Create 2-D inverse discrete cosine transform object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.IDCT

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.IDCT

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.



# isLocked

**System object:** vision.IDCT

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the IDCT System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.IDCT

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.IDCT

**Package:** vision

Compute 2-D inverse discrete cosine transform of the input

## Syntax

$Y = \text{step}(H,X)$

## Description

$Y = \text{step}(H,X)$  computes the 2-D inverse discrete cosine transform,  $Y$ , of input  $X$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.IFFT System object

**Package:** vision

Two-dimensional inverse discrete Fourier transform

### Description

The `vision.IFFT` object computes the inverse 2D discrete Fourier transform (IDFT) of a two-dimensional input matrix. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- An algorithm chosen by FFTW [1], [2]

### Construction

`H = vision.IFFT` returns a 2D IFFT object, `H`, with the default property and value pair settings.

`H = vision.IFFT(Name, Value)` returns a 2D IFFT object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### FFTImplementation

FFT implementation

Specify the implementation used for the FFT as one of `Auto` | `Radix-2` | `FFTW`. When you set this property to `Radix-2`, the FFT length must be a power of two.

### BitReversedInput

Indicates whether input is in bit-reversed order

Set this property to `true` if the order of 2D FFT transformed input elements are in bit-reversed order. The default is `false`, which denotes linear ordering.

### ConjugateSymmetricInput

Indicates whether input is conjugate symmetric

Set this property to `true` if the input is conjugate symmetric. The 2D DFT of a real valued signal is conjugate symmetric and setting this property to `true` optimizes the 2D IFFT computation method. Setting this property to `false` for conjugate symmetric inputs results in complex output values with nonzero imaginary parts. Setting this property to `true` for non conjugate symmetric inputs results in invalid outputs. This property must be `false` for fixed-point inputs. The default is `true`.

### Normalize

Divide output by FFT length

Specify if the 2D IFFT output should be divided by the FFT length. The value of this property defaults to `true` and divides each element of the output by the product of the row and column dimensions of the input matrix.

### Fixed-Point Properties

#### RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies when you set the ??? property to `false`. The default is `Floor`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the `???` property to `false`. The default is `Wrap`.

### **SineTableDataType**

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input`, `Custom`. This property applies when you set the `???` property to `false`. The default is `Same word length as input`.

### **CustomSineTableDataType**

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the `???` property to `false` and the `???` property is `Custom`. The default is `numericType([], 16)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. This property applies when you set the `???` property to `false`. The default is `Full precision`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `false` and the `???` property to `Custom`. The default is `numericType([], 32, 30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`. This property applies when you set the `???` property to `false`. The default is `Full precision`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `false` and the `???` property to `Custom`. The default is `numericType([], 32, 30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output data type as `Full precision`, `Same as input`, or `Custom`. This property applies when you set the `???` property to `false`. The default is `Full precision`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `false` and the `???` property is `Custom`. The default is `numericType([], 16, 15)`.

## **Methods**

<code>clone</code>	Create IFFT object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes

step

Compute 2D inverse discrete Fourier transform

## Examples

Use the 2D IFFT object to convert an intensity image.

```
hfft2d = vision.FFT;
hifft2d = vision.IFFT;

% Read in the image
xorig = single(imread('cameraman.tif'));

% Convert the image from the spatial
% to frequency domain and back
Y = step(hfft2d, xorig);
xtran = step(hifft2d, Y);

% Display the newly generated intensity image
imshow(abs(xtran), []);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D IFFT block reference page. The object properties correspond to the Simulink block parameters.

## References

- [1] FFTW (<http://www.fftw.org>)
- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## See Also

vision.FFT | vision.IDCT | vision.DCT



**Introduced in R2012a**

# clone

**System object:** vision.IFFT

**Package:** vision

Create IFFT object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.IFFT

**Package:** vision

Number of expected inputs to step method

## Syntax

getNumInputs(H)

## Description

getNumInputs(H) returns the number of expected inputs to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs(H)**.

## getNumOutputs

**System object:** vision.IFFT

**Package:** vision

Number of outputs from step method

### Syntax

getNumOutputs(H)

### Description

getNumOutputs(H) returns the number of output arguments from the **step** method.

The **getNumOutputs** method returns a positive integer that is the number of outputs from the **step** method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.IFFT

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

isLocked(H)

## Description

isLocked(H) returns the locked state of the IFFT object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.IFFT

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.IFFT

**Package:** vision

Compute 2D inverse discrete Fourier transform

## Syntax

$Y = \text{step}(H, X)$

## Description

$Y = \text{step}(H, X)$  computes the 2D inverse discrete Fourier transform (IDFT),  $Y$ , of an  $M$ -by- $N$  input matrix  $X$ , where the values of  $M$  and  $N$  are integer powers of two.

## vision.ImageComplementer System object

**Package:** vision

Complement of pixel values in binary or intensity image

### Description

---

**Note:** The `vision.ImageComplementer` System object will be removed in a future release. Use the `imcomplement` function with equivalent functionality instead.

---

The `ImageComplementer` object computes the complement of pixel values in binary or intensity image. For binary images, the object replaces pixel values equal to 0 with 1, and pixel values equal to 1 with 0. For an intensity image, the object subtracts each pixel value from the maximum value the data input type can represent and then outputs the difference.

### Construction

`H = vision.ImageComplementer` returns an image complement System object, `H`. The object computes the complement of a binary, intensity, or RGB image.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

### Methods

`clone`

Create image complementer object with same property values



<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute complement of input image

## Examples

Compute the complement of an input image.

```
himgcomp = vision.ImageComplementer;
hautoth = vision.Autothresher;

% Read in image
I = imread('coins.png');

% Convert the image to binary
bw = step(hautoth, I);

% Take the image complement
Ic = step(himgcomp, bw);

% Display the results
figure;
subplot(2,1,1), imshow(bw), title('Original Binary image')
subplot(2,1,2), imshow(Ic), title('Complemented image')
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Image Complement](#) block reference page. The object properties correspond to the block parameters.

## See Also

`vision.Autothresher`

**Introduced in R2012a**

# clone

**System object:** vision.ImageComplementer

**Package:** vision

Create image complementer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.ImageComplementer

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ImageComplementer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.ImageComplementer

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ImageComplementer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.ImageComplementer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.ImageComplementer

**Package:** vision

Compute complement of input image

### Syntax

$Y = \text{step}(H,X)$

### Description

$Y = \text{step}(H,X)$  computes the complement of an input image  $X$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---



# vision.ImageDataTypeConverter System object

**Package:** vision

Convert and scale input image to specified output data type

## Description

---

**Note:** The `vision.ImageDataTypeConverter` System object will be removed in a future release. Use the function with equivalent functionality instead.

- `im2double`
  - `im2single`
  - `im2int16`
  - `im2uint16`
  - `im2uint8`
- 

The `ImageDataTypeConverter` object converts and scales an input image to a specified output data type. When converting between floating-point data types, the object casts the input into the output data type and clips values outside the range, to 0 or 1. When converting between all other data types, the object casts the input into the output data type. The object then scales the data type values into the dynamic range of the output data type. For double- and single-precision floating-point data types, the object sets the dynamic range between 0 and 1. For fixed-point data types, the object sets the dynamic range between the minimum and maximum values that the data type can represent.

## Construction

`H = vision.ImageDataTypeConverter` returns a System object, `H`, that converts the input image to a single precision data type.

`H = vision.ImageDataTypeConverter(Name, Value)` returns an image data type conversion object, `H`, with each specified property set to the specified value.

You can specify additional name-value pair arguments in any order as (*Name1*, *Value1*,...,*NameN*,*ValueN*).

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### OutputDataType

Data type of output

Specify the data type of the output signal as one of `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `boolean` | `Custom`. The default is `single`.

### Fixed-Point Properties

#### CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned, scaled `numericType` object. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numericType([], 16, 0)`.

## Methods

<code>clone</code>	Create image data type converter object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Convert data type of input image

## Examples

Convert the image datatype from uint8 to single.

```
x = imread('pout.tif');  
hidtypeconv = vision.ImageDataTypeConverter;  
y = step(hidtypeconv, x);  
imshow(y);  
whos y % Image has been converted from uint8 to single.
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Image Data Type Conversion](#) block reference page. The object properties correspond to the block parameters.

### See Also

[vision.ColorSpaceConverter](#)

**Introduced in R2012a**

# clone

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Create image data type converter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ImageDataTypeConverter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.ImageDataTypeConverter

**Package:** vision

Convert data type of input image

## Syntax

$Y = \text{step}(H,X)$

## Description

$Y = \text{step}(H,X)$  converts the data type of the input image  $X$ . You can set the data type of the converted output image  $Y$  with the **OutputDataType** property.

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.ImageFilter System object

**Package:** vision

Perform 2-D FIR filtering of input matrix

### Description

---

**Note:** The `vision.ImageFilter` System object will be removed in a future release. Use the `imfilter` function with equivalent functionality instead.

---

The `ImageFilter` object performs 2-D FIR filtering of input matrix.

### Construction

`H = vision.ImageFilter` returns a System object, `H`. This object performs two-dimensional FIR filtering of an input matrix using the specified filter coefficient matrix.

`H = vision.ImageFilter(Name, Value)` returns an image filter System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### SeparableCoefficients

Set to `true` if filter coefficients are separable

Using separable filter coefficients reduces the amount of calculations the object must perform to compute the output. The function `isfilterseparable` can be used to check filter separability. The default is `false`.

### **CoefficientsSource**

Source of filter coefficients

Indicate how to specify the filter coefficients as one of `Property | Input port`. The default is `Property`.

### **Coefficients**

Filter coefficients

Specify the filter coefficients as a real or complex-valued matrix. This property applies when you set the `???` property to `false` and the `???` property to `Property`. The default is `[1 0; 0 -1]`.

### **VerticalCoefficients**

Vertical filter coefficients for the separable filter

Specify the vertical filter coefficients for the separable filter as a vector. This property applies when you set the `???` property to `true` and the `???` property to `Property`. The default is `[4 0]`.

### **HorizontalCoefficients**

Horizontal filter coefficients for the separable filter

Specify the horizontal filter coefficients for the separable filter as a vector. This property applies when you set the `???` property to `true` and the `???` property to `Property`. The default is `[4 0]`.

### **OutputSize**

Output size as full, valid or same as input image size

Specify how to control the size of the output as one of `Full | Same as first input | Valid`. The default is `Full`. When you set this property to `Full`, the object outputs the image dimensions in the following way:

output rows = input rows + filter coefficient rows - 1

$\text{output columns} = \text{input columns} + \text{filter coefficient columns} - 1$

When you set this property to **Same as first input**, the object outputs the same dimensions as the input image.

When you set this property to **Valid**, the object filters the input image only where the coefficient matrix fits entirely within it, and no padding is required. In this case, the dimensions of the output image are as follows:

$\text{output rows} = \text{input rows} - \text{filter coefficient rows} - 1$

$\text{output columns} = \text{input columns} - \text{filter coefficient columns} - 1$

### **PaddingMethod**

How to pad boundary of input matrix

Specify how to pad the boundary of input matrix as one of **Constant** | **Replicate** | **Symmetric**, | **Circular**. The default is **Constant**. Set this property to one of the following:

- **Constant** to pad the input matrix with a constant value
- **Replicate** to pad the input matrix by repeating its border values
- **Symmetric** to pad the input matrix with its mirror image
- **Circular** to pad the input matrix using a circular repetition of its elements

This property applies when you set the ??? property to **Full** or to **Same as first input**.

### **PaddingValueSource**

Source of padding value

Specify how to define the constant boundary value as one of **Property** | **Input port**. This property applies when you set the ??? property to **Constant**. The default is **Property**.

### **PaddingValue**

Constant value with which to pad matrix

Specify a constant value with which to pad the input matrix. This property applies when you set the ??? property to **Constant** and the ??? property to **Property**. The default is **0**. This property is tunable.

**Method**

Method for filtering input matrix

Specify the method by which the object filters the input matrix as one of `Convolution` | `Correlation`. The default is `Convolution`.

**Fixed-Point Properties****RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

**OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`.

**CoefficientsDataType**

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as one of `Same word length as input` | `Custom`. This property applies when you set the `???` property to `Property`. The default is `Custom`.

**CustomCoefficientsDataType**

Coefficients word and fraction lengths

Specify the coefficients fixed-point type as a signed or unsigned `numericType` object. This property applies when you set the `???` property to `Property` and the `CoefficientsDataType` property to `Custom`. The default is `numericType([], 16)`.

**ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as input` | `Custom`. The default is `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as an auto-signed, scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 10)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Same as input` | `Custom`. The default is `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as an auto-signed, scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 10)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as input` | `Custom`. The default is `Same as input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType`. This property applies when you set the `???` property to `Custom`. The default is `numericType([], 32, 12)`.

## **Methods**

`clone`

Create image filter object with same property values

getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Filter input image

## Examples

Filter an image to enhance the edges of 45 degree

```
img = im2single(rgb2gray(imread('peppers.png')));  
hfir2d = vision.ImageFilter;  
hfir2d.Coefficients = [1 0; 0 -.5];  
fImg = step(hfir2d, img);  
subplot(2,1,1);imshow(img);title('Original image')  
subplot(2,1,2);imshow(fImg);title('Filtered image')
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D FIR Filter block reference page. The object properties correspond to the block parameters.

### See Also

vision.MedianFilter

**Introduced in R2012a**

# clone

**System object:** vision.ImageFilter

**Package:** vision

Create image filter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



# getNumInputs

**System object:** vision.ImageFilter

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.ImageFilter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.ImageFilter

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ImageFilter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.ImageFilter

**Package:** vision

Allow property value and input characteristics changes

### Syntax

release(H)

### Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.ImageFilter

**Package:** vision

Filter input image

## Syntax

$Y = \text{step}(H, I)$

$Y = \text{step}(H, I, \text{COEFFS})$

$Y = \text{step}(H, I, HV, HH)$

$Y = \text{step}(H, \dots, PVAL)$

## Description

$Y = \text{step}(H, I)$  filters the input image  $I$  and returns the filtered image  $Y$ .

$Y = \text{step}(H, I, \text{COEFFS})$  uses filter coefficients,  $\text{COEFFS}$ , to filter the input image. This applies when you set the `CoefficientsSource` property to `Input port` and the `SeparableCoefficients` property to `false`.

$Y = \text{step}(H, I, HV, HH)$  uses vertical filter coefficients,  $HV$ , and horizontal coefficients,  $HH$ , to filter the input image. This applies when you set the `CoefficientsSource` property to `Input port` and the `SeparableCoefficients` property to `true`.

$Y = \text{step}(H, \dots, PVAL)$  uses the input pad value  $PVAL$  to filter the input image  $I$ . This applies when you set the `OutputSize` property to either `Full` or `Same as first input`, the `PaddingMethod` property to `Constant`, and the `PaddingValueSource` property to `Input port`.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an

input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# integralKernel class

Define filter for use with integral images

## Description

This object describes box filters for use with integral images.

## Construction

`intKernel = integralKernel(bbox, weights)` defines an upright box filter using an  $M$ -by-4 matrix of bounding boxes and their corresponding weights.

`intKernel = integralKernel(bbox, weights, orientation)` the specified orientation.

## Input Arguments

### **bbox** — Bounding boxes

4-element vector |  $M$ -by-4 matrix

Bounding boxes, specified as either a 4-element  $[x, y, width, height]$  vector or an  $M$ -by-4 matrix of individual bounding boxes. The bounding box defines the filter. The  $(x, y)$  coordinates represent the top-most corner of the kernel. The  $(width, height)$  elements represent the width and height accordingly. Specifying the bounding boxes as an  $M$ -by-4 matrix is particularly useful for constructing Haar-like features composed of multiple rectangles.

Sums are computed over regions defined by `bbox`. The bounding boxes can overlap. See “Define an 11-by-11 Average Filter” on page 2-679 for an example of how to specify a box filter.

### **weights** — Weights

$M$ -length vector

Weights, specified as an  $M$ -length vector of weights corresponding to the bounding boxes.

For example, a conventional filter with the coefficients:

```
h = [1  1  1  1;
      1  1  1  1;
      -1 -1 -1 -1;
      -1 -1 -1 -1]
```

and two regions:

region 1: x=1, y=1, width = 4, height = 2

region 2: x=1, y=3, width = 4, height = 2

can be specified as

```
boxH = integralKernel([1 1 4 2; 1 3 4 2], [1, -1]);
```

### **orientation** — Filter orientation

'upright' | 'rotated'

Filter orientation, specified as the string 'upright' or 'rotated'. When you set the orientation to 'rotated', the  $(x,y)$  components refer to the location of the top-left corner of the bounding box. The  $(width,height)$  components refer to a 45-degree line from the top-left corner of the bounding box.

## Properties

These properties are read-only.

### **BoundingBoxes** — Bounding boxes

4-element vector |  $M$ -by-4 matrix

Bounding boxes, stored as either a 4-element  $[x,y,width,height]$  vector or an  $M$ -by-4 matrix of individual bounding boxes.

### **Weights** — Weights

vector

Weights, stored as a vector containing a weight for each bounding box. The weights are used to define the coefficients of the filter.



**Coefficients — Filter coefficients**

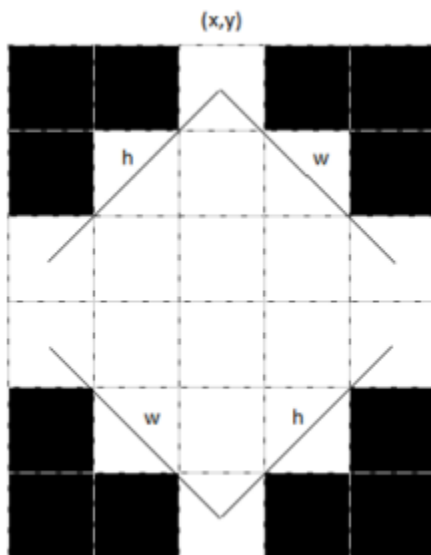
numeric

Filter coefficients, stored as a numeric value.

**Center — Filter center**[*x,y*] coordinates

Filter center, stored as [*x,y*] coordinates. The filter center represents the center of the bounding rectangle. It is calculated by halving the dimensions of the rectangle. For even dimensional rectangles, the center is placed at subpixel locations. Hence, it is rounded up to the next integer.

For example, for this filter, the center is at [3,3].



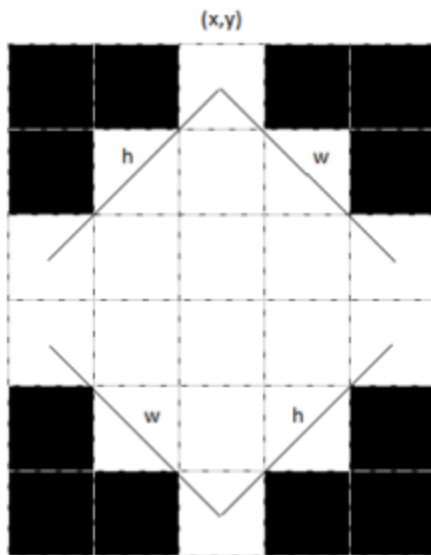
These coordinates are in the kernel space, where the top-left corner is (1,1). To place the center in a different location, provide the appropriate bounding box specification. For this filter, the best workflow would be to construct the upright kernel and then call the `rot45` method to provide the rotated version.

**Size — Filter size**

2-element vector

Filter size, stored as a 2-element vector. The size of the kernel is computed to be the dimensions of the rectangle that bounds the kernel. For a single bounding box vector  $[x,y,width,height]$ , the kernel is bounded within a rectangle of dimensions  $[(width+height)(width+height)-1]$ .

For cascaded rectangles, the lowest corner of the bottom-most rectangle defines the size. For example, a filter with a bounding box specification of  $[3\ 1\ 3\ 3]$ , with weights set to 1, produces a 6-by-5 filter with this kernel:



### Orientation — Filter orientation

'upright' (default) | 'rotated'

Filter orientation, stored as the string 'upright' or 'rotated'.

## Methods

transpose

Transpose filter

rot45

Rotates upright kernel clockwise by 45 degrees

## Examples

### Define an 11-by-11 Average Filter

```
avgH = integralKernel([1 1 11 11], 1/11^2);
```

### Define a Filter to Approximate a Gaussian Second Order Partial Derivative in Y Direction

```
ydH = integralKernel([1,1,5,9;1,4,5,3], [1, -3]);
```

You can also define this filter as `integralKernel([1,1,5,3;1,4,5,3;1,7,5,3], [1, -2, 1]);`. This filter definition is less efficient because it requires three bounding boxes.

Visualize the filter.

```
ydH.Coefficients
```

```
ans =
```

```

1     1     1     1     1
1     1     1     1     1
1     1     1     1     1
-2    -2    -2    -2    -2
-2    -2    -2    -2    -2
-2    -2    -2    -2    -2
1     1     1     1     1
1     1     1     1     1
1     1     1     1     1
```

### Create a Haar-like Wavelet to Detect 45-Degree Edges

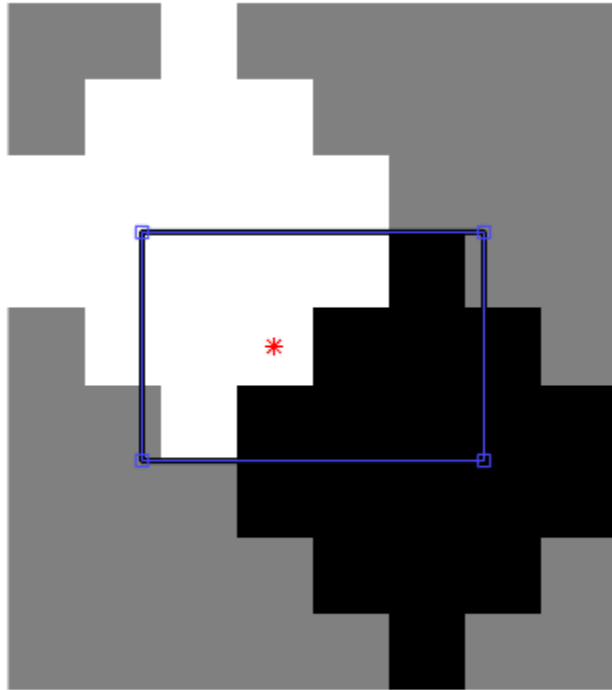
Create the filter.

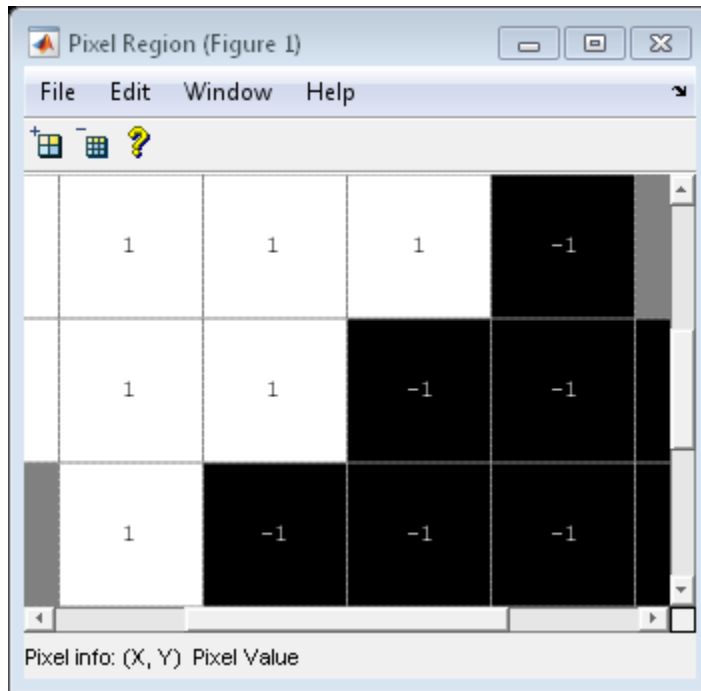
```
K = integralKernel([3,1,3,3;6 4 3 3], [1 -1], 'rotated');
```

Visualize the filter and mark the center.

```

imshow(K.Coefficients, [], 'InitialMagnification', 'fit');
hold on;
plot(K.Center(2),K.Center(1), 'r*');
impixelregion;
```





### Blur an Image Using an Average Filter

Read and display the input image.

```
I = imread('pout.tif');  
imshow(I);
```



Compute the integral image.

```
intImage = integralImage(I);
```

Apply a 7-by-7 average filter.

```
avgH = integralKernel([1 1 7 7], 1/49);  
J = integralFilter(intImage, avgH);
```

Cast the result back to the same class as the input image.

```
J = uint8(J);  
figure  
imshow(J);
```



- “Compute an Integral Image” on page 3-274
- “Find Vertical and Horizontal Edges in Image”

## References

Viola, Paul, and Michael J. Jones. “Rapid Object Detection using a Boosted Cascade of Simple Features”. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1, 2001, pp. 511–518.

## More About

### Computing an Integral Image and Using it for Filtering with Box Filters

The `integralImage` function together with the `integralKernel` object and `integralFilter` function complete the workflow for box filtering based on integral images. You can use this workflow for filtering with box filters.

- Use the `integralImage` function to compute the integral images
- Use the `integralFilter` function for filtering
- Use the `integralKernel` object to define box filters

The `integralKernel` object allows you to transpose the filter. You can use this to aim a directional filter. For example, you can turn a horizontal edge detector into vertical edge detector.

### See Also

`SURFPoints` | `detectMSERFeatures` | `detectSURFFeatures` | `integralFilter` | `integralImage`

**Introduced in R2012a**



# transpose

**Class:** integralKernel

Transpose filter

## Syntax

```
transposedKernel = transpose(intKernel)
```

## Description

`transposedKernel = transpose(intKernel)` transposes the integral kernel. You can use this operation to change the direction of an oriented filter.

## Example

### Construct Haar-like Wavelet Filters

Horizontal filter

```
horiH = integralKernel([1 1 4 3; 1 4 4 3], [-1, 1]);
```

Using the dot and apostrophe create a vertical filter.

```
vertH = horiH.';
```

Using the transpose method.

```
verticalH = transpose(horiH);
```

### **rot45**

Rotates upright kernel clockwise by 45 degrees

### **Syntax**

```
rotKernel = rot45(intKernel)
```

### **Description**

`rotKernel = rot45(intKernel)` rotates upright kernel clockwise by 45 degrees.

### **Example**

#### **Construct and Rotate a Haar-like Wavelet Filter**

Create a horizontal filter.

```
H = integralKernel([1 1 4 3; 1 4 4 3], [-1, 1]);
```

Rotate the filter 45 degrees.

```
rotH = rot45(H);
```

# vision.ImagePadder System object

**Package:** vision

Pad or crop input image along its rows, columns, or both

## Description

---

**Note:** The `vision.ImagePadder` System object will be removed in a future release. Use the `padarray` function with equivalent functionality instead.

---

The `ImagePadder` object pads or crop input image along its rows, columns, or both.

## Construction

`HIMPAD = vision.ImagePadder` returns an image padder System object, `HIMPAD`, that performs two-dimensional padding and/or cropping of an input image.

`HIMPAD = vision.ImagePadder(Name, Value)` returns an image padder object, `HIMPAD`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### Method

How to pad input image

Specify how to pad the input image as **Constant** | **Replicate** | **Symmetric** | **Circular**. The default is **Constant**.

### **PaddingValueSource**

How to specify pad value

Indicate how to specify the pad value as either **Property** | **Input port**. This property applies when you set the ??? property to **Constant**. The default is **Property**.

### **PaddingValue**

Pad value

Specify the constant scalar value with which to pad the image. This property applies when you set the **Method** property to **Constant** and the **PaddingValueSource** property to **Property**. The default is **0**. This property is tunable.

### **SizeMethod**

How to specify output image size

Indicate how to pad the input image to obtain the output image by specifying **Pad size** | **Output size**. When this property is **Pad size**, the size of the padding in the vertical and horizontal directions are specified. When this property is **Output size**, the total number of output rows and output columns are specified. The default is **Pad size**.

### **RowPaddingLocation**

Location at which to add rows

Specify the direction in which to add rows to as **Top** | **Bottom** | **Both top and bottom** | **None**. Set this property to **Top** to add additional rows to the top of the image, **Bottom** to add additional rows to the bottom of the image, **Both top and bottom** to add additional rows to the top and bottom of the image, and **None** to maintain the row size of the input image. The default is **Both top and bottom**.

### **NumPaddingRows**

Number of rows to add

Specify the number of rows to be added to the top, bottom, or both sides of the input image as a scalar value. When the **RowPaddingLocation** property is **Both top and**

bottom, this property can also be set to a two element vector, where the first element controls the number of rows the System object adds to the top of the image and the second element controls the number of rows the System object adds to the bottom of the image. This property applies when you set the `???` property to `Pad size` and the `???` property is not set to `None`. The default is `[2 3]`.

### **NumOutputRowsSource**

How to specify number of output rows

Indicate how to specify the number of output rows as `Property | Next power of two`. If this property is `Next power of two`, the System object adds rows to the input image until the number of rows is equal to a power of two. This property applies when you set the `???` property to `Output size`. The default is `Property`.

### **NumOutputRows**

Total number of rows in output

Specify the total number of rows in the output as a scalar integer. If the specified number is smaller than the number of rows of the input image, then image is cropped. This property applies when you set the `???` property to `Output size` and the `???` property to `Property`. The default is `12`.

### **ColumnPaddingLocation**

Location at which to add columns

Specify the direction in which to add columns one of `Left | Right | Both left and right | None`. Set this property to `Left` to add additional columns on the left side of the image, `Right` to add additional columns on the right side of the image, `Both left and right` to add additional columns on the left and right side of the image, and `None` to maintain the column length of the input image. The default is `Both left and right`.

### **NumPaddingColumns**

Number of columns to add

Specify the number of columns to be added to the left, right, or both sides of the input image as a scalar value. When the `ColumnPaddingLocation` property is `Both left and right`, this property can also be set to a two element vector, where the first element controls the number of columns the System object adds to the left side of the image and the second element controls the number of columns the System object adds to the right

side of the image. This property applies when you set the ??? property to `Pad size` and the `NumPaddingColumns` property is not set to `None`. The default is `2`.

### **NumOutputColumnsSource**

How to specify number of output columns

Indicate how to specify the number of output columns as `Property | Next power of two`. If you set this property to `Next power of two`, the `System` object adds columns to the input until the number of columns is equal to a power of two. This property applies when you set the ??? property to `Output size`. The default is `Property`.

### **NumOutputColumns**

Total number of columns in output

Specify the total number of columns in the output as a scalar integer. If the specified number is smaller than the number of columns of the input image, then image is cropped. This property applies when you set the ??? property to `Output size` and the ??? property to `Property`. The default is `10`.

## **Methods**

<code>clone</code>	Create image padder object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform two-dimensional padding or cropping of input

## **Examples**

Pad two rows to the bottom, and three columns to the right of an image.

Use the value of the last array element as the padding value.

```
himpad = vision.ImagePadder('Method', 'Replicate', ...  
    'RowPaddingLocation', 'Bottom', ...  
    'NumPaddingRows', 2, ...  
    'ColumnPaddingLocation', 'Right', ...  
    'NumPaddingColumns', 3);  
x = [1 2;3 4];  
y = step(himpad,x);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Image Pad](#) block reference page. The object properties correspond to the block parameters.

### See Also

[vision.GeometricScaler](#)

**Introduced in R2012a**

# clone

**System object:** vision.ImagePadder

**Package:** vision

Create image padder object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



# getNumInputs

**System object:** vision.ImagePadder

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.ImagePadder

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.ImagePadder

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ImagePadder System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ImagePadder

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.ImagePadder

**Package:** vision

Perform two-dimensional padding or cropping of input

## Syntax

$Y = \text{step}(H, X)$

$Y = \text{step}(H, X, \text{PAD})$

## Description

$Y = \text{step}(H, X)$  performs two-dimensional padding or cropping of input,  $X$ .

$Y = \text{step}(H, X, \text{PAD})$  performs two-dimensional padding and/or cropping of input,  $X$ , using the input pad value  $\text{PAD}$ . This applies when you set the `Method` property to `Constant` and the `PaddingValueSource` property to `Input port`.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.KalmanFilter class

**Package:** vision

Kalman filter for object tracking

### Description

The Kalman filter object is designed for tracking. You can use it to predict a physical object's future location, to reduce noise in the detected location, or to help associate multiple physical objects with their corresponding tracks. A Kalman filter object can be configured for each physical object for multiple object tracking. To use the Kalman filter, the object must be moving at constant velocity or constant acceleration.

The Kalman filter algorithm involves two steps, prediction and correction (also known as the update step). The first step uses previous states to predict the current state. The second step uses the current measurement, such as object location, to correct the state. The Kalman filter implements a discrete time, linear State-Space System.

---

**Note:** To make configuring a Kalman filter easier, you can use the `configureKalmanFilter` object to configure a Kalman filter. It sets up the filter for tracking a physical object in a Cartesian coordinate system, moving with constant velocity or constant acceleration. The statistics are the same along all dimensions. If you need to configure a Kalman filter with different assumptions, do not use the function, use this object directly.

---

### Construction

`obj = vision.KalmanFilter` returns a Kalman filter object for a discrete time, constant velocity system. In this “State-Space System” on page 2-700, the state transition model,  $A$ , and the measurement model,  $H$ , are set as follows:

Variable	Value
$A$	[1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1]
$H$	[1 0 0 0; 0 0 1 0]

`obj = vision.KalmanFilter(StateTransitionModel,MeasurementModel)`  
 configures the state transition model,  $A$ , and the measurement model,  $H$ .

`obj = vision.KalmanFilter(StateTransitionModel,MeasurementModel,ControlModel)` additionally configures the control model,  $B$ .

`obj = vision.KalmanFilter(StateTransitionModel,MeasurementModel,ControlModel,Name,Value)` configures the Kalman filter object properties, specified as one or more `Name,Value` pair arguments. Unspecified properties have default values.

### Code Generation Support

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”.

## To Track Objects:

Use the `???` and `???` methods based on detection results.

- When the tracked object is detected, use the `predict` and `correct` methods with the Kalman filter object and the detection measurement. Call the methods in the following order:

```
[...] = predict(obj);
[...] = correct(obj,measurement);
```

- When the tracked object is not detected, call the `predict` method, but not the `correct` method. When the tracked object is missing or occluded, no measurement is available. Set the methods up with the following logic:

```
[...] = predict(obj);
If measurement exists
    [...] = correct(obj,measurement);
end
```

- If the tracked object becomes available after missing for the past  $t-1$  contiguous time steps, you can call the `predict` method  $t$  times. This syntax is particularly useful to process asynchronous video. For example,

```
for i = 1:k
    [...] = predict(obj);
end
[...] = correct(obj,measurement)
```

Use the `distance` method to find the best matches. The computed distance values describe how a set of measurements matches the Kalman filter. You can thus select a measurement that best fits the filter. This strategy can be used for matching object detections against object tracks in a multiobject tracking problem. This distance computation takes into account the covariance of the predicted state and the process noise. The `distance` method can only be called after the `predict` method.

**Distance Equation**

$$d(z) = (z - Hx)^T \Sigma^{-1} (z - Hx) + \ln|\Sigma|$$

Where  $\Sigma = HPH^T + R$  and  $|\Sigma|$  is the determinant of  $\Sigma$ . You can then find the best matches by examining the returned distance values.

`d = distance(obj, z_matrix)` computes a distance between the location of a detected object and the predicted location by the Kalman filter object. Each row of the  $N$ -column `z_matrix` input matrix contains a measurement vector. The `distance` method returns a row vector where each distance element corresponds to the measurement input.

**State-Space System**

This object implements a discrete time, linear state-space system, described by the following equations.

State equation:	$x(k) = Ax(k - 1) + Bu(k - 1) + w(k - 1)$
Measurement equation:	$z(k) = Hx(k) + v(k)$

**Variable Definition**

Variable	Description	Dimension
$k$	Time.	Scalar
$x$	State. Gaussian vector with covariance $P$ . [ $x \sim N(\bar{x}, P)$ ]	$M$ -element vector
$P$	State estimation error covariance.	$M$ -by- $M$ matrix
$A$	State transition model.	$M$ -by- $M$ matrix



Variable	Description	Dimension
$B$	Control model.	$M$ -by- $L$ matrix
$u$	Control input.	$L$ -element vector
$w$	Process noise; Gaussian vector with zero mean and covariance $Q$ . [ $w \sim N(0, Q)$ ]	$M$ -element vector
$Q$	Process noise covariance.	$M$ -by- $M$ matrix
$z$	Measurement. For example, location of detected object.	$N$ -element vector
$H$	Measurement model.	$N$ -by- $M$ matrix
$v$	Measurement noise; Gaussian vector with zero mean and covariance $R$ . [ $v \sim N(0, R)$ ]	$N$ -element vector
$R$	Measurement noise covariance.	$N$ -by- $N$ matrix

## Properties

### StateTransitionModel

Model describing state transition between time steps ( $A$ )

Specify the transition of state between times as an  $M$ -by- $M$  matrix. After the object is constructed, this property cannot be changed. This property relates to the  $A$  variable in the “State-Space System” on page 2-700.

**Default:** [1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1]

### MeasurementModel

Model describing state to measurement transformation ( $H$ )

Specify the transition from state to measurement as an  $N$ -by- $M$  matrix. After the object is constructed, this property cannot be changed. This property relates to the  $H$  variable in the “State-Space System” on page 2-700.

**Default:** [1 0 0 0; 0 0 1 0]

### ControlModel

Model describing control input to state transformation ( $B$ )

Specify the transition from control input to state as an  $M$ -by- $L$  matrix. After the object is constructed, this property cannot be changed. This property relates to the  $B$  variable in the “State-Space System” on page 2-700.

**Default:** []

### **State**

State ( $x$ )

Specify the state as a scalar or an  $M$ -element vector. If you specify it as a scalar it will be extended to an  $M$ -element vector. This property relates to the  $x$  variable in the “State-Space System” on page 2-700.

**Default:** [0]

### **StateCovariance**

State estimation error covariance ( $P$ )

Specify the covariance of the state estimation error as a scalar or an  $M$ -by- $M$  matrix. If you specify it as a scalar it will be extended to an  $M$ -by- $M$  diagonal matrix. This property relates to the  $P$  variable in the “State-Space System” on page 2-700.

**Default:** [1]

### **ProcessNoise**

Process noise covariance ( $Q$ )

Specify the covariance of process noise as a scalar or an  $M$ -by- $M$  matrix. If you specify it as a scalar it will be extended to an  $M$ -by- $M$  diagonal matrix. This property relates to the  $Q$  variable in the “State-Space System” on page 2-700.

**Default:** [1]

### **MeasurementNoise**

Measurement noise covariance ( $R$ )

Specify the covariance of measurement noise as a scalar or an  $N$ -by- $N$  matrix. If you specify it as a scalar it will be extended to an  $N$ -by- $N$  diagonal matrix. This property relates to the  $R$  variable in the “State-Space System” on page 2-700.

Default: [1]

## Methods

clone	Create Kalman filter object with same property values
distance	Confidence value of measurement
correct	Correction of measurement, state, and state estimation error covariance
predict	Prediction of measurement

## Examples

### Track Location of An Object

Track the location of a physical object moving in one direction.

Generate synthetic data which mimics the 1-D location of a physical object moving at a constant speed.

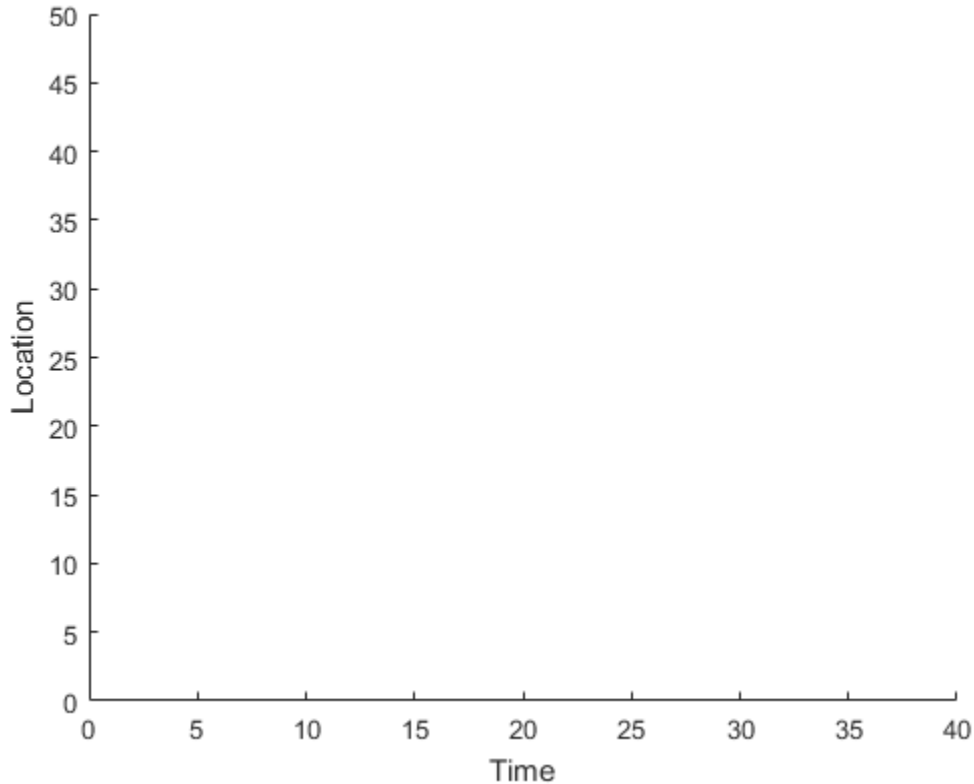
```
detectedLocations = num2cell(2*randn(1,40) + (1:40));
```

Simulate missing detections by setting some elements to empty.

```
detectedLocations{1} = [];  
for idx = 16: 25  
    detectedLocations{idx} = [];  
end
```

Create a figure to show the location of detections and the results of using the Kalman filter for tracking.

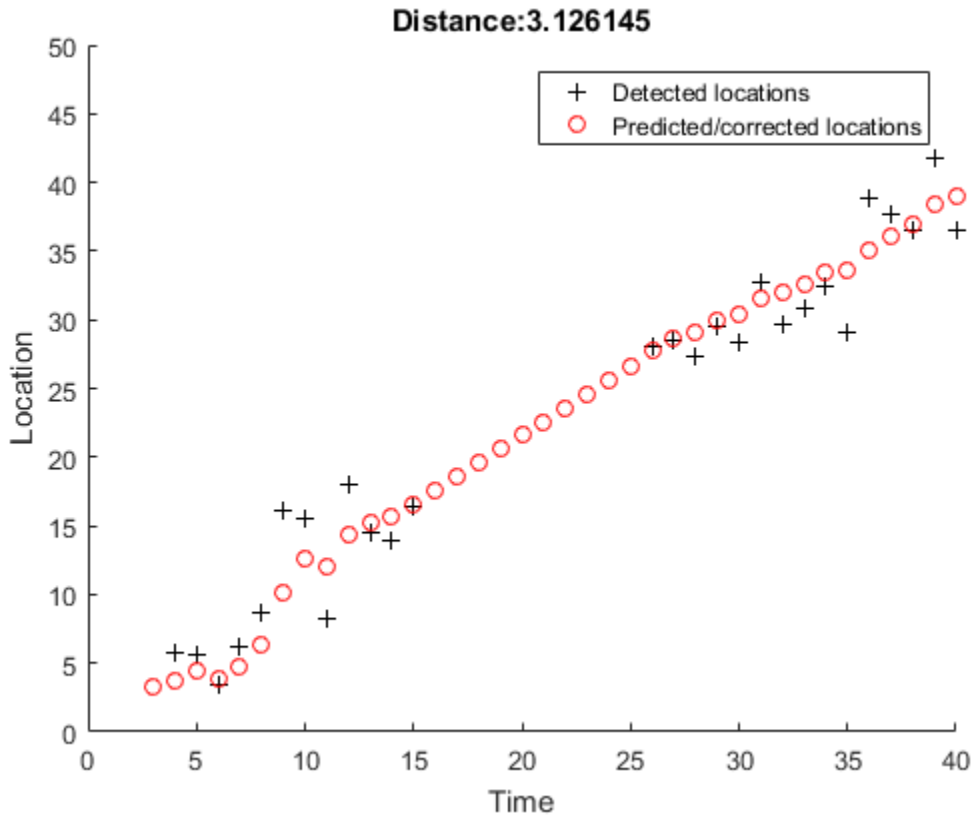
```
figure;  
hold on;  
ylabel('Location');  
ylim([0,50]);  
xlabel('Time');  
xlim([0,length(detectedLocations)]);
```



Create a 1-D, constant speed Kalman filter when the physical object is first detected. Predict the location of the object based on previous states. If the object is detected at the current time step, use its location to correct the states.

```
kalman = [];  
for idx = 1: length(detectedLocations)  
    location = detectedLocations{idx};  
    if isempty(kalman)  
        if ~isempty(location)  
            stateModel = [1 1;0 1];  
            measurementModel = [1 0];  
            kalman = vision.KalmanFilter(stateModel,measurementModel, 'ProcessNoise', 1e-4, 'MeasurementNoise', 1e-4);  
            kalman.State = [location, 0];  
        end  
    end  
end
```

```
    end
else
    trackedLocation = predict(kalman);
    if ~isempty(location)
        plot(idx, location, 'k+');
        d = distance(kalman,location);
        title(sprintf('Distance:%f', d));
        trackedLocation = correct(kalman,location);
    else
        title('Missing detection');
    end
    pause(0.2);
    plot(idx,trackedLocation, 'ro');
end
end
legend('Detected locations', 'Predicted/corrected locations');
```



### Remove Noise From a Signal

Use Kalman filter to remove noise from a random signal corrupted by a zero-mean Gaussian noise.

Synthesize a random signal that has value of 1 and is corrupted by a zero-mean Gaussian noise with standard deviation of 0.1.

```
x = 1;
len = 100;
z = x + 0.1 * randn(1,len);
```

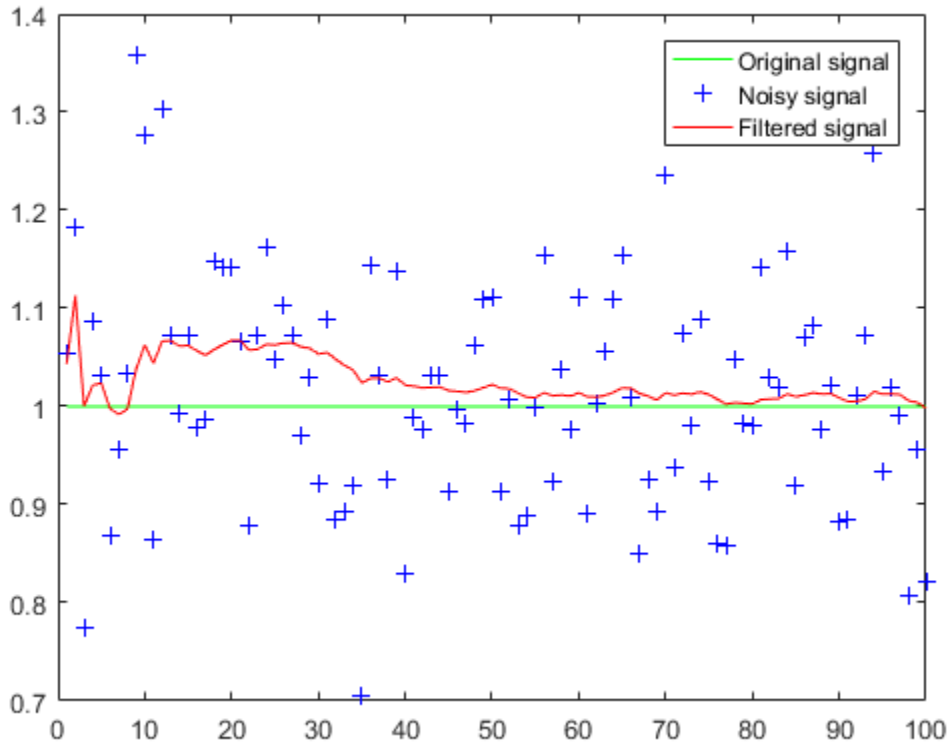
Remove noise from the signal by using a Kalman filter. The state is expected to be constant, and the measurement is the same as state.

```
stateTransitionModel = 1;
measurementModel = 1;
obj = vision.KalmanFilter(stateTransitionModel,measurementModel,'StateCovariance',1,'P

z_corr = zeros(1,len);
for idx = 1: len
    predict(obj);
    z_corr(idx) = correct(obj,z(idx));
end
```

Plot results.

```
figure, plot(x * ones(1,len),'g-');
hold on;
plot(1:len,z,'b+',1:len,z_corr,'r-');
legend('Original signal','Noisy signal','Filtered signal');
```



- “Motion-Based Multiple Object Tracking”

## References

Welch, Greg, and Gary Bishop, *An Introduction to the Kalman Filter*, TR 95-041. University of North Carolina at Chapel Hill, Department of Computer Science.

## See Also

`configureKalmanFilter` | `assignDetectionsToTracks`

**Introduced in R2012b**



# clone

**Class:** vision.KalmanFilter

**Package:** vision

Create Kalman filter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# distance

**Class:** vision.KalmanFilter

**Package:** vision

Confidence value of measurement

## Syntax

```
d = distance(obj, z_matrix)
```

## Description

`d = distance(obj, z_matrix)` computes a distance between the location of a detected object and the predicted location by the Kalman filter object. Each row of the  $N$ -column `z_matrix` input matrix contains a measurement vector. The distance method returns a row vector where each distance element corresponds to the measurement input.

This distance computation takes into account the covariance of the predicted state and the process noise. The `distance` method can only be called after the `predict` method.

## correct

**Class:** vision.KalmanFilter

**Package:** vision

Correction of measurement, state, and state estimation error covariance

## Syntax

```
[z_corr,x_corr,P_corr] = correct(obj,z)
```

## Description

`[z_corr,x_corr,P_corr] = correct(obj,z)` returns the correction of measurement, state, and state estimation error covariance. The correction is based on the current measurement  $z$ , an  $N$ -element vector. The object overwrites the internal state and covariance of the Kalman filter with corrected values.

# predict

**Class:** vision.KalmanFilter

**Package:** vision

Prediction of measurement

## Syntax

```
[z_pred,x_pred,P_pred] = predict(obj)
[z_pred,x_pred,P_pred] = predict(obj,u)
```

## Description

`[z_pred,x_pred,P_pred] = predict(obj)` returns the prediction of measurement, state, and state estimation error covariance at the next time step (e.g., the next video frame). The object overwrites the internal state and covariance of the Kalman filter with the prediction results.

`[z_pred,x_pred,P_pred] = predict(obj,u)` additionally lets you specify the control input,  $u$ , an  $L$ -element vector. This syntax applies when you set the control model,  $B$ .

# vision.LocalMaximaFinder System object

**Package:** vision

Find local maxima in matrices

## Description

The `LocalMaximaFinder` object finds local maxima in matrices.

After constructing the object and optionally setting properties, use the `step` method to find the coordinates of the local maxima in the input image. Use the `step` syntax below with input matrix, `I`, `LocalMaximaFinder` object, `H`, and any optional properties.

`IDX = step(H, I)` returns `[x y]` coordinates of the local maxima in an  $M$ -by-2 matrix, `IDX`.  $M$  represents the number of local maximas found. The maximum value of  $M$  may not exceed the value set in the `MaximumNumLocalMaxima` property.

`[...] = step(H, I, THRESH)` finds the local maxima in input image `I`, using the threshold value `THRESH`, when you set the `ThresholdSource` property to `Input port`.

## Construction

`H = vision.LocalMaximaFinder` returns a local maxima finder System object, `H`, that finds local maxima in input matrices.

`H = vision.LocalMaximaFinder(Name, Value)` returns a local maxima finder object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

`H = vision.LocalMaximaFinder(MAXNUM, NEIGHBORSIZE, Name, Value)` returns a local maxima finder object, `H`, with the `MaximumNumLocalMaxima` property set to `MAXNUM`, `NeighborhoodSize` property set to `NEIGHBORSIZE`, and other specified properties set to the specified values.

### Code Generation Support

Supports MATLAB Function block: Yes

“System Objects in MATLAB Code Generation”.

### Code Generation Support

“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### MaximumNumLocalMaxima

Maximum number of maxima to find

Specify the maximum number of maxima to find as a positive scalar integer value. The default is 2.

### NeighborhoodSize

Neighborhood size for zero-ing out values

Specify the size of the neighborhood around the maxima, over which the System object zeros out values, as a 2-element vector of positive odd integers. The default is [5 7].

### ThresholdSource

Source of threshold

Specify how to enter the threshold value as `Property`, or `Input port`. The default is `Property`.

### Threshold

Value that all maxima should match or exceed

Specify the threshold value as a scalar of MATLAB built-in numeric data type. This property applies when you set the ??? property to `Property`. The default is 10. This property is tunable.

### HoughMatrixInput

Indicator of Hough Transform matrix input

Set this property to `true` if the input is antisymmetric about the rho axis and the theta value ranges from  $-\frac{\pi}{2}$  to  $\frac{\pi}{2}$  radians, which correspond to a Hough matrix.

When you set this property to `true`, the object assumes a Hough matrix input . The block applies additional processing, specific to Hough transform on the right and left boundaries of the input matrix.

The default is `false`.

### **IndexDataType**

Data type of index values

Specify the data type of index values as `double`, `single` , `uint8`, `uint16`, or `uint32`. The default is `uint32`.

## **Methods**

<code>clone</code>	Create local maxima finder object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Find local maxima in input image

## **Examples**

Find a local maxima in an input.

```
I = [0 0 0 0 0 0 0 0 0 0 0; ...
     0 0 0 1 1 2 3 2 1 1 0; ...
     0 0 0 1 2 3 4 3 2 1 0; ...
     0 0 0 1 3 5 7 5 3 1 0; ... % local max at x=7, y=4
     0 0 0 1 2 3 4 3 2 1 0; ...
     0 0 0 1 1 2 3 2 1 1 0; ...
     0 0 0 0 0 0 0 0 0 0 0];
```

```
hLocalMax = vision.LocalMaximaFinder;  
hLocalMax.MaximumNumLocalMaxima = 1;  
hLocalMax.NeighborhoodSize = [3 3];  
hLocalMax.Threshold = 1;  
  
location = step(hLocalMax, I)
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Find Local Maxima](#) block reference page. The object properties correspond to the block parameters.

### See Also

[hough](#) | [vision.Maximum](#) | [vision.HoughLines](#)

**Introduced in R2012a**



# clone

**System object:** vision.LocalMaximaFinder

**Package:** vision

Create local maxima finder object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.LocalMaximaFinder

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .

## getNumOutputs

**System object:** vision.LocalMaximaFinder

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.LocalMaximaFinder

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the LocalMaximaFinder System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.LocalMaximaFinder

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.LocalMaximaFinder

**Package:** vision

Find local maxima in input image

### Syntax

`IDX = step(H,I)`

`[...] = step(H,I,THRESH)`

### Description

`IDX = step(H,I)` returns `[x y]` coordinates of the local maxima in an  $M$ -by-2 matrix, `IDX`, where  $M$  represents the number of local maximas found. The maximum value of  $M$  may not exceed the value set in the `MaximumNumLocalMaxima` property.

`[...] = step(H,I,THRESH)` finds the local maxima in input image `I`, using the threshold value `THRESH`, when you set the `ThresholdSource` property to `Input port`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.MarkerInserter System object

**Package:** vision

Draw markers on output image

## Description

---

**Note:** The `vision.MarkerInserter` System object will be removed in a future release. Use the `insertMarker` function instead.

---

The `MarkerInserter` object can draw circle, x-mark, plus sign, star, or rectangle markers in a 2-D grayscale or truecolor RGB image. The output image can then be displayed or saved to a file.

## Construction

`markerInserter = vision.MarkerInserter` returns a marker inserter System object, `markerInserter`, that draws a circle in an image.

`markerInserter = vision.MarkerInserter(Name, Value)` returns a marker inserter object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

## To insert a marker:

- 1 Define and set up your marker inserter using the constructor.

- 2 Call the `step` method with the input image, `I`, the marker inserter object, `markerInserter`, points `PTS`, and any optional properties. See the syntax below for using the `step` method.

`J = step(markerInserter, I, PTS)` draws the marker specified by the `???` property on input image `I`. The input `PTS` specify the coordinates for the location of the marker. You can specify the `PTS` input as an  $M$ -by-2  $[x\ y]$  matrix of  $M$  number of markers. Each  $[x\ y]$  pair defines the center location of the marker. The markers are embedded on the output image `J`.

`J = step(markerInserter, I, PTS, ROI)` draws the specified marker in a rectangular area defined by the `ROI` input. This applies when you set the `???` property to `true`. The `ROI` input defines a rectangular area as  $[x\ y\ width\ height]$ , where  $[x\ y]$  determine the upper-left corner location of the rectangle, and `width` and `height` specify the size.

`J = step(markerInserter, I, PTS, ..., CLR)` draws the marker with the border or fill color specified by the `CLR` input. This applies when you set the `???` property or the `???` property to 'Input port'.

## Properties

### Shape

Shape of marker

Specify the type of marker to draw as `Circle` | `X-mark` | `Plus`, `Star` | `Square`.

Default: `Circle`

### Size

Size of marker

Specify the size of the marker, in pixels, as a scalar value greater than or equal to 1. This property is tunable.

Default: `3`

### Fill

Enable marker fill



Set this property to true to fill the marker with an intensity value or a color. This property applies when you set the ??? property to **Circle** or **Square**.

Default: `false`

### **BorderColorSource**

Border color source

Specify how the marker's border color is provided as **Input port**, **Property**. This property applies either when you set the ??? property to **X-mark**, **Plus**, or **Star**, or when you set the **Shape** property to **Circle** or **Square**, and the **Fill** property to `false`. When you set ??? to **Input port**, a border color vector must be provided as an input to the System object's **step** method.

Default: `Property`

### **BorderColor**

Border color of marker

Specify the border color of the marker as **Black** | **White** | **Custom**. If this property is set to **Custom**, the ??? property is used to specify the value. This property applies when the ??? property is enabled and set to **Property**.

Default: `Black`

### **CustomBorderColor**

Intensity or color value for marker's border

Specify an intensity or color value for the marker's border. If the input is an intensity image, this property can be set to a scalar intensity value for one marker or  $R$ -element vector where  $R$  is the number of markers. If the input is a color image, this property can be set to a  $P$ -element vector where  $P$  is the number of color planes or a  $P$ -by- $R$  matrix where  $P$  is the number of color planes and  $R$  is the number of markers. This property applies when you set the ??? property to **Custom**. This property is tunable when the ??? property is `false`.

Default: `[200 120 50]`

### **FillColorSource**

Source of fill color

Specify how the marker's fill color is provided as `Input port` | `Property`. This property applies when you set the `???` property to `Circle` or `Square`, and the `???` property to `true`. When this property is set to `Input port`, a fill color vector must be provided as an input to the System object's `step` method.

Default: `Property`

### **FillColor**

Fill color of marker

Specify the color to fill the marker as `Black` | `White` | `Custom`. If this property is set to `Custom`, the `???` property is used to specify the value. This property applies when the `???` property is enabled and set to `Property`.

Default: `Black`

### **CustomFillColor**

Intensity or color value for marker's interior

Specify an intensity or color value to fill the marker. If the input is an intensity image, this property can be set to a scalar intensity value for one marker or  $R$ -element vector where  $R$  is the number of markers. If the input is a color image, this property can be set to a  $P$ -element vector where  $P$  is the number of color planes or a  $P$ -by- $R$  matrix where  $P$  is the number of color planes and  $R$  is the number of markers. This property applies when you set the `???` property to `Custom`. This property is tunable when the `???` property is `false`.

Default: `[200 120 50]`

### **Opacity**

Opacity of shading inside marker

Specify the opacity of the shading inside the marker by a scalar value between 0 and 1, where 0 is transparent and 1 is opaque. This property applies when you set the `???` property to `true`. This property is tunable.

Default: `0.6`

### **ROIInputPort**

Enable defining a region of interest to draw marker

Set this property to `true` to specify a region of interest (ROI) on the input image through an input to the `step` method. If the property is set to `false`, the object uses the entire image.

Default: `false`

### **Antialiasing**

Enable performing smoothing algorithm on marker

Set this property to `true` to perform a smoothing algorithm on the marker. This property applies when you do not set the `???` property to `Square` or `Plus`.

Default: `false`

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Floor`

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` | `Saturate`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Wrap`

#### **OpacityDataType**

Opacity word length

Specify the opacity fixed-point data type as `Same word length as input` or `Custom`. This property applies when you set the `???` property to `true`.

Default: `Custom`

### **CustomOpacityDataType**

Opacity word length

Specify the opacity fixed-point type as an unscaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and the `???` property to `Custom`.

Default: `numericType([],16)`

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input` or `Custom`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`, and the `???` property to `Custom`.

Default: `numericType([],32,14)`

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product | Same as first input | Custom`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Same as product`

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`, and the `???` property to `Custom`.

Default: `numericType([],32,14)`

## Methods

<code>clone</code>	Create marker inserter object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Draw specified marker on input image

## Examples

### Draw White Plus Signs in a Grayscale Image

```
markerInserter = vision.MarkerInserter('Shape','Plus','BorderColor','white');
I = imread('cameraman.tif');
Pts = int32([10 10; 20 20; 30 30]);
J = step(markerInserter, I, Pts);
imshow(J);
```

### Draw Red Circles in a Grayscale Image

```
red = uint8([255 0 0]); % [R G B]; class of red must match class of I
markerInserter = vision.MarkerInserter('Shape','Circle','BorderColor','Custom','Custom');
I = imread('cameraman.tif');
RGB = repmat(I,[1 1 3]); % convert the image to RGB
Pts = int32([60 60; 80 80; 100 100]);
J = step(markerInserter, RGB, Pts);
```

```
imshow(J);
```

### Draw Blue X-marks in a Color Image

```
markerInserter = vision.MarkerInserter('Shape','X-mark','BorderColor','Custom','Custom');  
RGB = imread('autumn.tif');  
Pts = int32([20 20; 40 40; 60 60]);  
J = step(markerInserter, RGB, Pts);  
imshow(J);
```

### See Also

[vision.ShapeInserter](#) | [insertText](#)

**Introduced in R2012a**

# clone

**System object:** vision.MarkerInserter

**Package:** vision

Create marker inserter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MarkerInserter

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).



## getNumOutputs

**System object:** vision.MarkerInserter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the step method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.MarkerInserter

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MarkerInserter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.MarkerInserter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.MarkerInserter

**Package:** vision

Draw specified marker on input image

### Syntax

`J = step(markerInserter, I, PTS)`

`J = step(markerInserter, I, PTS, ROI)`

`J = step(markerInserter, I, PTS, ..., CLR)`

### Description

`J = step(markerInserter, I, PTS)` draws the marker specified by the `???` property on input image *I*. The input `PTS` specify the coordinates for the location of the marker. You can specify the `PTS` input as an *M*-by-2  $[x\ y]$  matrix of *M* number of markers. Each  $[x\ y]$  pair defines the center location of the marker. The markers are embedded on the output image *J*.

`J = step(markerInserter, I, PTS, ROI)` draws the specified marker in a rectangular area defined by the `ROI` input. This applies when you set the `???` property to `true`. The `ROI` input defines a rectangular area as  $[x\ y\ width\ height]$ , where  $[x\ y]$  determine the upper-left corner location of the rectangle, and *width* and *height* specify the size.

`J = step(markerInserter, I, PTS, ..., CLR)` draws the marker with the border or fill color specified by the `CLR` input. This applies when you set the `???` property or the `???` property to 'Input port'.

---

**Note:** *H* specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## Input Arguments

### **H**

Shape inserter object with shape and properties specified.

### **I**

Input image.

### **PTS**

Input  $M$ -by-2 matrix of  $M$  number of [x y] coordinates describing the location for markers. This property must be an integer value. If you enter non-integer value, the object rounds it to the nearest integer.

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_M & y_M \end{bmatrix}$$

Each [x y] pair defines the center of a marker.

The table below shows the data types required for the inputs, **I** and **PTS**.

<b>Input image I</b>	<b>Input matrix PTS</b>
built-in integer	built-in or fixed-point integer
fixed-point integer	built-in or fixed-point integer
double	double, single, or build-in integer
single	double, single, or build-in integer

### **RGB**

Scalar, vector, or matrix describing one plane of the RGB input video stream.

### **ROI**

Input 4-element vector of integers [x y width height], that define a rectangular area in which to draw the shapes. The first two elements represent the one-based coordinates

of the upper-left corner of the area. The second two elements represent the width and height of the area.

- Double-precision floating point
- Single-precision floating point
- 8-, 16-, and 32-bit signed integer
- 8-, 16-, and 32-bit unsigned integer

### **CLR**

This port can be used to dynamically specify shape color.

$P$ -element vector or an  $M$ -by- $P$  matrix, where  $M$  is the number of shapes, and  $P$ , the number of color planes. You can specify a color (RGB), for each shape, or specify one color for all shapes. The data type for the CLR input must be the same as the input image.

## **Output Arguments**

### **J**

Output image. The markers are embedded on the output image.

# vision.Maximum System object

**Package:** vision

Find maximum values in input or sequence of inputs

## Description

The `Maximum` object finds maximum values in an input or sequence of inputs.

## Construction

`H = vision.Maximum` returns an object, `H`, that computes the value and index of the maximum elements in an input or a sequence of inputs.

`H = vision.Maximum(Name, Value)` returns a maximum-finding object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
"System Objects in MATLAB Code Generation".
"Code Generation Support, Usage Notes, and Limitations".

## Properties

### ValueOutputPort

Output maximum value

Set this property to `true` to output the maximum value of the input. This property applies when you set the `???` property to `false`.

Default: `true`.

### **RunningMaximum**

Calculate over single input or multiple inputs

When you set this property to `true`, the object computes the maximum value over a sequence of inputs. When you set this property to `false`, the object computes the maximum value over the current input.

Default: `false`.

### **IndexOutputPort**

Output the index of the maximum value

Set this property to `true` to output the index of the maximum value of the input. This property applies only when you set the ??? property to `false`.

Default: `true`.

### **ResetInputPort**

Additional input to enable resetting of running maximum

Set this property to `true` to enable resetting of the running maximum. When you set this property to `true`, a reset input must be specified to the `step` method to reset the running maximum. This property applies only when you set the ??? property to `true`.

Default: `false`.

### **ResetCondition**

Condition that triggers resetting of running maximum

Specify the event that resets the running maximum as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. This property applies only when you set the ??? property to `true`.

Default: `Non-zero`.

### **IndexBase**

Numbering base for index of maximum value



Specify the numbering used when computing the index of the maximum value as starting from either `One` or `Zero`. This property applies only when you set the `???` property to `true`.

Default: `One`.

### **Dimension**

Dimension to operate along

Specify how the maximum calculation is performed over the data as `All`, `Row`, `Column`, or `Custom`. This property applies only when you set the `???` property to `false`.

Default: `Column`.

### **CustomDimension**

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the maximum. The value of this property cannot exceed the number of dimensions in the input signal. This property only applies when you set the `???` property to `Custom`.

Default: `1`.

### **ROIProcessing**

Enable region-of-interest processing

Set this property to `true` to enable calculation of the maximum value within a particular region of an image. This property applies when you set the `???` property to `All` and the `???` property to `false`.

Default: `false`.

### **ROIForm**

Type of region of interest

Specify the type of region of interest as `Rectangles`, `Lines`, `Label matrix`, or `Binary mask`. This property applies only when you set the `???` property to `true`.

Default: Rectangles.

### **ROIPortion**

Calculate over entire ROI or just perimeter

Specify whether to calculate the maximum over the Entire ROI or the ROI perimeter. This property applies only when you set the ??? property to Rectangles.

Default: Entire ROI.

### **ROIStatistics**

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate Individual statistics for each ROI or a Single statistic for all ROIs. This property applies only when you set the ??? property to Rectangles, Lines, or Label matrix.

### **ValidityOutputPort**

Output flag indicating if any part of ROI is outside input image

Set this property to true to return the validity of the specified ROI as completely or partially inside of the image. This applies when you set the ??? property to Lines or Rectangles.

Set this property to true to return the validity of the specified label numbers. This applies when you set the ??? property to Label matrix.

Default: false.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero.

Default: Floor.

**OverflowAction**

Action to take when integer input is out-of-range

Specify the overflow action as `Wrap` or `Saturate`.

Default: `Wrap`.

**ProductDataType**

Data type of product

Specify the product fixed-point data type as `Same as input` or `Custom`.

Default: `Same as input`.

**CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`.

Default: `numericType(true,32,30)`.

**AccumulatorDataType**

Data type of accumulator

Specify the accumulator fixed-point data type as `Same as product`, `Same as input`, or `Custom`.

Default: `Same as product`.

**CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`.

Default: `numericType(true,32,30)`.

### Methods

clone	Create maximum object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
reset	Reset computation of running maximum
release	Allow property value and input characteristics changes
step	Compute maximum value

### Examples

Determine the maximum value and its index in a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hmax = vision.Maximum;  
[m, ind] = step(hmax, img);
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D Maximum block reference page.

### See Also

[vision.Mean](#) | [vision.Minimum](#)

**Introduced in R2012a**

# clone

**System object:** vision.Maximum

**Package:** vision

Create maximum object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Maximum

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .

# getNumOutputs

**System object:** vision.Maximum

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.Maximum

**Package:** vision

Locked status for input attributes and nontunable properties

### Syntax

TF = isLocked(H)

### Description

TF = isLocked(H) returns the locked status, TF of the Maximum System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



## reset

**System object:** vision.Maximum

**Package:** vision

Reset computation of running maximum

## Syntax

reset(H)

## Description

reset(H) resets the computation of the running maximum for the Maximum object H.

## release

**System object:** vision.Maximum

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.Maximum

**Package:** vision

Compute maximum value

## Syntax

[VAL,IND] = step(H,X)

VAL = step(H,X)

IND = step(H,X)

VAL = step(H,X,R)

[...] = step(H,I,ROI)

[...] = step(H,I,LABEL,LABELNUMBERS)

[...,FLAG] = step(H,I,ROI)

[...,FLAG] = step(H,I,LABEL,LABELNUMBERS)

## Description

[VAL,IND] = step(H,X) returns the maximum value, VAL, and the index or position of the maximum value, IND, along a dimension of X specified by the value of the Dimension property.

VAL = step(H,X) returns the maximum value, VAL, of the input X. When the RunningMaximum property is true, VAL corresponds to the maximum value over a sequence of inputs.

IND = step(H,X) returns the zero- or one-based index IND of the maximum value. To enable this type of processing, set the IndexOutputPort property to true and the ValueOutputPort and RunningMaximum properties to false.

VAL = step(H,X,R) computes the maximum value, VAL, over a sequence of inputs, and resets the state of H based on the value of reset signal, R, and the ResetCondition property. To enable this type of processing, set the RunningMaximum property to true and the ResetInputPort property to true.

[...] = `step(H,I,ROI)` computes the maximum of an input image, `I`, within the given region of interest, `ROI`. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

[...] = `step(H,I,LABEL,LABELNUMBERS)` computes the maximum of an input image, `I`, for a region whose labels are specified in the vector `LABELNUMBERS`. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Label matrix`.

[...,FLAG] = `step(H,I,ROI)` also returns `FLAG`, indicating whether the given region of interest is within the image bounds. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

[...,FLAG] = `step(H,I,LABEL,LABELNUMBERS)` also returns `FLAG`, indicating whether the input label numbers are valid. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Label matrix`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.Mean System object

**Package:** vision

Find mean value of input or sequence of inputs

## Description

The Mean object finds the mean of an input or sequence of inputs.

## Construction

`H = vision.Mean` returns an object, `H`, that computes the mean of an input or a sequence of inputs.

`H = vision.Mean(Name, Value)` returns a mean-finding object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### RunningMean

Calculate over single input or multiple inputs

When you set this property to `true`, the object calculates the mean over a sequence of inputs. When you set this property to `false`, the object computes the mean over the current input. The default is `false`.

### ResetInputPort

Additional input to enable resetting of running mean

Set this property to **true** to enable resetting of the running mean. When you set this property to **true**, a reset input must be specified to the **step** method to reset the running mean. This property applies only when you set the ??? property to **true**. The default is **false**.

### **ResetCondition**

Condition that triggers resetting of running mean

Specify the event that resets the running mean as **Rising edge**, **Falling edge**, **Either edge**, or **Non-zero**. This property applies only when you set the ??? property to **true**. The default is **Non-zero**.

### **Dimension**

Dimension to operate along

Specify how the mean calculation is performed over the data as **All**, **Row**, **Column**, or **Custom**. This property applies only when you set the ??? property to **false**. The default is **All**.

### **CustomDimension**

Numerical dimension to calculate over

Specify the integer dimension, indexed from one, of the input signal over which the object calculates the mean. The value of this property cannot exceed the number of dimensions in the input signal. This property only applies when you set the ??? property to **Custom**. The default is 1.

### **ROIProcessing**

Enable region-of-interest processing

Set this property to **true** to enable calculation of the mean within a particular region of an image. This property applies when you set the ??? property to **All** and the ??? property to **false**. The default is **false**.

### **ROIForm**

Type of region of interest

Specify the type of region of interest as `Rectangles`, `Lines`, `Label matrix`, or `Binary mask`. This property applies only when you set the `???` property to `true`. The default is `Rectangles`.

### **ROIPortion**

Calculate over entire ROI or just perimeter

Specify whether to calculate the mean over the `Entire ROI` or the `ROI perimeter`. This property applies only when you set the `???` property to `Rectangles`. The default is `Entire ROI`.

### **ROIStatistics**

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate `Individual statistics for each ROI` or a `Single statistic for all ROIs`. This property applies only when you set the `???` property to `Rectangles`, `Lines`, or `Label matrix`. The default is `Individual statistics for each ROI`.

### **ValidityOutputPort**

Output flag indicating if any part of ROI is outside input image

Set this property to `true` to return the validity of the specified ROI as completely or partially inside of the image. This applies when you set the `???` property to `Lines` or `Rectangles`.

Set this property to `true` to return the validity of the specified label numbers. This applies when you set the `???` property to `Label matrix`.

The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

### **OverflowAction**

Action to take when integer input is out-of-range

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

### **AccumulatorDataType**

Data type of accumulator

Specify the accumulator fixed-point data type as `Same as input`, or `Custom`. The default is `Same as input`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **OutputDataType**

Data type of output

Specify the output fixed-point data type as `Same as accumulator`, `Same as input`, or `Custom`. The default is `Same as accumulator`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

## **Methods**

<code>clone</code>	Create mean object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method



getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
reset	Reset computation of running mean
release	Allow property value and input characteristics changes
step	Compute mean of input

## Examples

Determine the mean of a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hmean = vision.Mean;  
m = step(hmean,img);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [2-D Mean block reference page](#).

## See Also

[vision.Minimum](#) | [vision.Maximum](#)

**Introduced in R2012a**

# clone

**System object:** vision.Mean

**Package:** vision

Create mean object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.Mean

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Mean

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Mean

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Mean System object..

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## reset

**System object:** vision.Mean

**Package:** vision

Reset computation of running mean

## Syntax

reset(H)

## Description

reset(H) resets the computation of the running mean for the Mean object H.

# release

**System object:** vision.Mean

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.Mean

**Package:** vision

Compute mean of input

## Syntax

```
Y = step(H,X)
Y = step(H,X,R)
Y = step(H,X,ROI)
Y = step(H,X,LABEL,LABELNUMBERS)
[Y,FLAG] = step(H,X,ROI)
[Y,FLAG] = step(H,X,LABEL,LABELNUMBERS)
```

## Description

`Y = step(H,X)` computes the mean of input image elements *X*. When you set the `RunningMean` property to `true`, the output *Y* corresponds to the mean of the input elements over successive calls to the `step` method.

`Y = step(H,X,R)` computes the mean value of the input image elements *X* over successive calls to the `step` method, and optionally resets the computation of the running mean based on the value of reset signal, *R* and the value of the `ResetCondition` property. To enable this type of processing, set the `RunningMean` property to `true` and the `ResetInputPort` property to `true`.

`Y = step(H,X,ROI)` computes the mean of input image elements *X* within the given region of interest specified by the input *ROI*. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

`Y = step(H,X,LABEL,LABELNUMBERS)` computes the mean of the input image elements, *X*, for the region whose labels are specified in the vector *LABELNUMBERS*. The regions are defined and labeled in the matrix *LABEL*. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Label matrix`.



---

[Y, FLAG] = step(H, X, ROI) also returns the output *FLAG*, indicating whether the given region of interest *ROI*, is within the image bounds. To enable this type of processing, set the *ROIProcessing* and *ValidityOutputPort* properties to `true` and the *ROIForm* property to `Lines`, `Rectangles` or `Binary mask`.

[Y, FLAG] = step(H, X, LABEL, LABELNUMBERS) also returns the output *FLAG* which indicates whether the input label numbers are valid. To enable this type of processing, set the *ROIProcessing* and *ValidityOutputPort* properties to `true` and the *ROIForm* property to `Label matrix`.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.Median System object

**Package:** vision

Find median values in an input

## Description

The `Median` object finds median values in an input.

## Construction

`H = vision.Median` returns a System object, `H`, that computes the median of the input or a sequence of inputs.

`H = vision.Median(Name, Value)` returns a median System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### SortMethod

Sort method

Specify the sort method used for calculating the median as `Quick sort` or `Insertion sort`.

### Dimension

Dimension with which to operate along

Specify how the calculation is performed over the data as `All`, `Row`, `Column`, or `Custom`. The default is `All`.

### **CustomDimension**

Numerical dimension over which to calculate

Specify the integer dimension of the input signal over which the object calculates the mean. The value of this property cannot exceed the number of dimensions in the input signal. This property only applies when you set the `???` property to `Custom`. The default is 1.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Action to take when integer input is out-of-range

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

#### **ProductDataType**

Product output word and fraction lengths

Specify the product output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **AccumulatorDataType**

Data type of accumulator

Specify the accumulator fixed-point data type as `Same as input`, or `Custom`. The default is `Same as input`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

### **OutputDataType**

Data type of output

Specify the output fixed-point data type as `Same as accumulator`, `Same as input`, or `Custom`. The default is `Same as accumulator`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

## **Methods**

<code>clone</code>	Create median object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes

step

Compute median of input

## Examples

Determine the median in a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hmdn = vision.Median;  
med = step(hmdn,img);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [2-D Median](#) block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**

# clone

**System object:** vision.Median

**Package:** vision

Create median object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.Median

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.Median

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.



# isLocked

**System object:** vision.Median

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Median System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.Median

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.Median

**Package:** vision

Compute median of input

## Syntax

$Y = \text{step}(H,X)$

## Description

$Y = \text{step}(H,X)$  computes median of input  $X$ .

---

**Note:**  $H$  specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

## vision.MedianFilter System object

**Package:** vision

2D median filtering

### Description

---

**Note:** The `vision.MedianFilter` System object will be removed in a future release. Use the `medfilt2` function with equivalent functionality instead.

---

The `MedianFilter` object performs 2D median filtering.

### Construction

`H = vision.MedianFilter` returns a 2D median filtering object, `H`. This object performs two-dimensional median filtering of an input matrix.

`H = vision.MedianFilter(Name, Value)` returns a median filter object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

`H = vision.MedianFilter(SIZE, Name, Value)` returns a median filter object, `H`, with the `???` property set to `SIZE` and other properties set to the specified values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### NeighborhoodSize

Size of neighborhood to compute the median

Specify the size of the neighborhood over which the median filter computes the median. When you set this property to a positive integer, the integer represents the number of rows and columns in a square matrix. When you set this property to a two-element vector, the vector represents the number of rows and columns in a rectangular matrix. The median filter does not support fixed-point properties when both neighborhood dimensions are odd. The default is `[3 3]`.

### **OutputSize**

Output matrix size

Specify the output size as one of `Same as input size` | `Valid`. The default is `Same as input size`. When you set this property to `Valid`, the 2D median filter only computes the median where the neighborhood fits entirely within the input image and requires no padding. In this case, the dimensions of the output image are:

output rows = input rows - neighborhood rows + 1

output columns = input columns - neighborhood columns + 1

Otherwise, the object outputs the same dimensions as the input image.

### **PaddingMethod**

Input matrix boundary padding method

Specifies how to pad the boundary of the input matrix as one of `Constant` | `Replicate` | `Symmetric` | `Circular`. When you set this property to `Constant`, the object pads the matrix with a constant value. When you set this property to `Replicate`, the object pads the input matrix by repeating its border values. When you set this property to `Symmetric`, the object pads the input matrix with its mirror image. When you set this property to `Circular` the object pads the input matrix using a circular repetition of its elements. This property applies only when you set the `???` property to `Same as input size`. The default is `Constant`.

### **PaddingValueSource**

Source of constant boundary value

Specifies how to define the constant boundary value as one of `Property` | `Input port`. This property applies only when you set the `???` property to `Constant`. The default is `Property`.

### **PaddingValue**

Constant padding value for input matrix

Specifies a constant value to pad the input matrix. The default is 0. This property applies only when you set the ??? property to **Constant** and the ??? property to **Property**. This property is tunable.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of **Ceiling** | **Convergent** | **Floor** | **Nearest** | **Round** | **Simplest** | **Zero**. The default is **Floor**. This property applies only when the ??? property corresponds to even neighborhood options.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of **Wrap** | **Saturate**. The default is **Wrap**. This property is applies only when the ??? property corresponds to even neighborhood options.

#### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of **Same as input** | **Custom**. The default is **Same as input**. This property applies only when the ??? property corresponds to even neighborhood options.

#### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled **numericType** object with a **Signedness** of **Auto**. This property applies only when the ??? property is **Custom** and the ??? property corresponds to even neighborhood options. The default is `numericType([ ],32,30)`.

#### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. This property applies only when the `???` property corresponds to even neighborhood options. The default is `Same as input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `???` property is `Custom` and the `???` property corresponds to even neighborhood options. The default is `numerictype([], 16, 15)`.

## **Methods**

<code>clone</code>	Create 2D median filter with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform median filtering on input image

## **Examples**

### **Perform Median Filtering on an Image with Additive Salt and Pepper Noise**

Read the image and add noise.

```
I = imread('pout.tif');
I = imnoise(I, 'salt & pepper')
```

Create a median filter object.

```
medianFilter = vision.MedianFilter([5 5]);
```

Filter the image.

```
I_filtered = step(medianFilter, I);
```

Display the results with a title.

```
figure; imshowpair(I, I_filtered, 'montage');  
title('Noisy image on the left and filtered image on the right');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Median Filter](#) block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**



# clone

**System object:** vision.MedianFilter

**Package:** vision

Create 2D median filter with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MedianFilter

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs**( $H$ ).

## getNumOutputs

**System object:** vision.MedianFilter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.MedianFilter

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MedianFilter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.MedianFilter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.MedianFilter

**Package:** vision

Perform median filtering on input image

### Syntax

I2 = step(H,I1)

I2 = step(H,I1,PVAL)

### Description

I2 = step(H,I1) performs median filtering on the input image I1 and returns the filtered image I2.

I2 = step(H,I1,PVAL) performs median filtering on input image I1, using PVAL for the padding value. This option applies when you set the **OutputSize** property to **Same as input size**, the **PaddingChoice** property to **Constant**, and the **PaddingValueSource** property to **Input port**.

---

**Note:** H specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

# vision.Minimum System object

**Package:** vision

Find minimum values in input or sequence of inputs

## Description

The `Minimum` object finds minimum values in an input or sequence of inputs.

## Construction

`H = vision.Minimum` returns an object, `H`, that computes the value and index of the minimum elements in an input or a sequence of inputs.

`H = vision.Minimum(Name, Value)` returns a minimum-finding object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
"System Objects in MATLAB Code Generation".
"Code Generation Support, Usage Notes, and Limitations".

## Properties

### ValueOutputPort

Output minimum value

Set this property to `true` to output the minimum value of the input. This property applies when you set the `???` property to `false`.

Default: `true`

### **RunningMinimum**

Calculate over single input or multiple inputs

When you set this property to `true`, the object computes the minimum value over a sequence of inputs. When you set this property to `false`, the object computes the minimum value over the current input.

Default: `false`

### **IndexOutputPort**

Output the index of the minimum value

Set this property to `true` to output the index of the minimum value of the input. This property applies only when you set the ??? property to `false`.

Default: `true`

### **ResetInputPort**

Additional input to enable resetting of running minimum

Set this property to `true` to enable resetting of the running minimum. When you set this property to `true`, a reset input must be specified to the `step` method to reset the running minimum. This property applies only when you set the ??? property to `true`.

Default: `false`

### **ResetCondition**

Condition that triggers resetting of running minimum

Specify the event that resets the running minimum as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. This property applies only when you set the ??? property to `true`.

Default: `Non-zero`

### **Dimension**

Dimension to operate along



Specify how the minimum calculation is performed over the data as **All**, **Row**, **Column**, or **Custom**. This property applies only when you set the ??? property to **false**.

Default: **Column**.

### **CustomDimension**

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the minimum. The value of this property cannot exceed the number of dimensions in the input signal. This property only applies when you set the ??? property to **Custom**.

Default: **1**.

### **ROIProcessing**

Enable region-of-interest processing

Set this property to **true** to enable calculation of the minimum value within a particular region of an image. This property applies when you set the ??? property to **All** and the ??? property to **false**.

Default: **false**.

### **ROIForm**

Type of region of interest

Specify the type of region of interest as **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**. This property applies only when you set the ??? property to **true**.

Default: **Rectangles**.

### **ROIPortion**

Calculate over entire ROI or just perimeter

Specify whether to calculate the minimum over the **Entire ROI** or the **ROI perimeter**. This property applies only when you set the ??? property to **Rectangles**.

Default: **Entire ROI**.

### **ROIStatistics**

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate `Individual` statistics for each ROI or a `Single` statistic for all ROIs. This property applies only when you set the `???` property to `Rectangles`, `Lines`, or `Label matrix`.

### **ValidityOutputPort**

Output flag indicating if any part of ROI is outside input image

Set this property to `true` to return the validity of the specified ROI as completely or partially inside of the image. This applies when you set the `???` property to `Lines` or `Rectangles`.

Set this property to `true` to return the validity of the specified label numbers. This applies when you set the `???` property to `Label matrix`.

Default: `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`.

Default: `Floor`.

#### **OverflowAction**

Action to take when integer input is out-of-range

Specify the overflow action as `Wrap` or `Saturate`.

Default: `Wrap`.

#### **ProductDataType**

Data type of product

Specify the product fixed-point data type as `Same as input` or `Custom`.

Default:`Same as input`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`.

Default:`numericType(true,32,30)`.

### **AccumulatorDataType**

Data type of accumulator

Specify the accumulator fixed-point data type as `Same as product`, `Same as input`, or `Custom`.

Default: `Same as product`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object. This property applies only when you set the `???` property to `Custom`.

Default:`numericType(true,32,30)`.

## **Methods**

<code>clone</code>	Create minimum object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties

release	Allow property value and input characteristics changes
reset	Reset computation of running minimum
step	Compute minimum value

## Examples

Determine the minimum value and its index in a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hmax = vision.Minimum;  
[m, ind] = step(hmax, img);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the 2-D Minimum block reference page.

## See Also

| [vision.Maximum](#) | [vision.Mean](#)

**Introduced in R2012a**

# clone

**System object:** vision.Minimum

**Package:** vision

Create minimum object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Minimum

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

# getNumOutputs

**System object:** vision.Minimum

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.Minimum

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Minimum System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



# release

**System object:** vision.Minimum

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## reset

**System object:** vision.Minimum

**Package:** vision

Reset computation of running minimum

## Syntax

reset(H)

## Description

reset(H) resets computation of the running minimum for the Minimum object H.

## step

**System object:** vision.Minimum

**Package:** vision

Compute minimum value

## Syntax

[VAL,IND] = step(H,X)

VAL = step(H,X)

IND = step(H,X)

VAL = step(H,X,R)

[...] = step(H,I,ROI)

[...] = step(H,I,LABEL,LABELNUMBERS)

[...,FLAG] = step(H,I,ROI)

[...,FLAG] = step(H,I,LABEL,LABELNUMBERS)

## Description

[VAL,IND] = step(H,X) returns the minimum value, VAL, and the index or position of the minimum value, IND, along a dimension of X specified by the value of the Dimension property.

VAL = step(H,X) returns the minimum value, VAL, of the input X. When the RunningMinimum property is true, VAL corresponds to the minimum value over a sequence of inputs.

IND = step(H,X) returns the zero- or one-based index IND of the minimum value. To enable this type of processing, set the IndexOutputPort property to true and the ValueOutputPort and RunningMinimum properties to false.

VAL = step(H,X,R) computes the minimum value, VAL, over a sequence of inputs, and resets the state of H based on the value of reset signal, R, and the ResetCondition property. To enable this type of processing, set the RunningMinimum property to true and the ResetInputPort property to true.

[...] = `step(H,I,ROI)` computes the minimum of an input image, `I`, within the given region of interest, `ROI`. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

[...] = `step(H,I,LABEL,LABELNUMBERS)` computes the minimum of an input image, `I`, for a region whose labels are specified in the vector `LABELNUMBERS`. To enable this type of processing, set the `ROIProcessing` property to `true` and the `ROIForm` property to `Label matrix`.

[...,FLAG] = `step(H,I,ROI)` also returns `FLAG`, indicating whether the given region of interest is within the image bounds. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

[...,FLAG] = `step(H,I,LABEL,LABELNUMBERS)` also returns `FLAG`, indicating whether the input label numbers are valid. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Label matrix`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.MorphologicalBottomHat System object

**Package:** vision

Bottom-hat filtering on image

## Description

---

**Note:** The `vision.MorphologicalBottomHat` System object will be removed in a future release. Use the `imbothat` function with equivalent functionality instead.

---

The `MorphologicalBottomHat` object performs bottom-hat filtering on an intensity or binary image. Bottom-hat filtering is the equivalent of subtracting the input image from the result of performing a morphological closing operation on the input image. The bottom-hat filtering object uses flat structuring elements only.

## Construction

`H = vision.MorphologicalBottomHat` returns a bottom-hat filtering object, `H`, that performs bottom-hat filtering on an intensity or binary image using a predefined neighborhood or structuring element.

`H = vision.MorphologicalBottomHat(Name, Value)` returns a bottom-hat filtering object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

### Code Generation Support

Supports MATLAB Function block: Yes

“System Objects in MATLAB Code Generation”.

“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### ImageType

Specify type of input image or video stream

Specify the type of the input image as **Intensity** or **Binary**. The default is **Intensity**.

### NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as one of **Property** or **Input port**. If set to **Property**, use the ??? property to specify the neighborhood or structuring element values. Otherwise, specify the neighborhood using an input to the **step** method. Note that you can specify structuring elements only by using the **Neighborhood** property. You can not specify structuring elements as inputs to the **step** method. The default is **Property**.

### Neighborhood

Neighborhood or structuring element values

This property applies only when you set the ??? property to **Property**. If specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If specifying a structuring element, use the **strel** function. The default is **strel('octagon', 15)**.

## Methods

clone	Create morphological bottom hat filter with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes

step	Perform bottom-hat filtering on input image
------	---

## Examples

Perform bottom-hat filtering on an image.

```
I = im2single(imread('blobs.png'));  
hbot = vision.MorphologicalBottomHat('Neighborhood',strel('disk', 5));  
J = step(hbot,I);  
figure;  
subplot(1,2,1),imshow(I); title('Original image');  
subplot(1,2,2),imshow(J);  
title('Bottom-hat filtered image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the **Bottom-hat** block reference page. The object properties correspond to the block parameters.

## See Also

strel | vision.MorphologicalTopHat | vision.MorphologicalClose

**Introduced in R2012a**

# clone

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Create morphological bottom hat filter with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



## getNumInputs

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalBottomHat System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.MorphologicalBottomHat

**Package:** vision

Perform bottom-hat filtering on input image

## Syntax

`Y = step(H,I)`

`Y = step(H,I,NHOOD)`

## Description

`Y = step(H,I)` performs bottom-hat filtering on the input image, *I*, and returns the filtered image *Y*.

`Y = step(H,I,NHOOD)` performs bottom-hat filtering on the input image, *I* using *NHOOD* as the neighborhood when you set the `NeighborhoodSource` property to `Input port`. The object returns the filtered image in the output *Y*.

## vision.MorphologicalClose System object

**Package:** vision

Perform morphological closing on image

### Description

---

**Note:** The `vision.MorphologicalClose` System object will be removed in a future release. Use the `imclose` function with equivalent functionality instead.

---

The `MorphologicalClose` object performs morphological closing on an intensity or binary image. The `MorphologicalClose` System object performs a dilation operation followed by an erosion operation using a predefined neighborhood or structuring element. This System object uses flat structuring elements only.

### Construction

`H = vision.MorphologicalClose` returns a System object, `H`, that performs morphological closing on an intensity or binary image.

`H = vision.MorphologicalClose(Name, Value)` returns a morphological closing System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as **Property** or **Input port**. If set to **Property**, use the ??? property to specify the neighborhood or structuring element values. Otherwise, specify the neighborhood using an input to the **step** method. Note that structuring elements can only be specified using **Neighborhood** property and they cannot be used as input to the **step** method. The default is **Property**.

### Neighborhood

Neighborhood or structuring element values

This property is applicable when the ??? property is set to **Property**. If you are specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function. The default is `strel('line',5,45)`.

## Methods

<code>clone</code>	Create morphological close object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform morphological closing on input image

## Examples

Perform morphological closing on an image.

```
img = im2single(imread('blobs.png'));  
hclosing = vision.MorphologicalClose;  
hclosing.Neighborhood = strel('disk', 10);  
closed = step(hclosing, img);  
figure;  
  
subplot(1,2,1),imshow(img); title('Original image');  
subplot(1,2,2),imshow(closed); title('Closed image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the **Closing** block reference page. The object properties correspond to the block parameters.

### See Also

[vision.MorphologicalOpen](#) | [vision.ConnectedComponentLabeler](#) | [vision.Autothresher](#) | [strel](#)

**Introduced in R2012a**



# clone

**System object:** vision.MorphologicalClose

**Package:** vision

Create morphological close object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MorphologicalClose

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .

## getNumOutputs

**System object:** vision.MorphologicalClose

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.MorphologicalClose

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalClose System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.MorphologicalClose

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.MorphologicalClose

**Package:** vision

Perform morphological closing on input image

### Syntax

IC = step(H,I)

IC = step(H,I,NHOOD)

### Description

IC = step(H,I) performs morphological closing on input image *I*.

IC = step(H,I,NHOOD) performs morphological closing on input image *I* using the input *NHOOD* as the neighborhood when you set the **NeighborhoodSource** property to **Input port**.

---

**Note:** H specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

# vision.MorphologicalDilate System object

**Package:** vision

Perform morphological dilation on an image

## Description

---

**Note:** The `vision.MorphologicalDilate` System object will be removed in a future release. Use the `imdilate` function with equivalent functionality instead.

---

## Construction

`H = vision.MorphologicalDilate` returns a System object, `H`, that performs morphological dilation on an intensity or binary image.

`H = vision.MorphologicalDilate(Name, Value)` returns a morphological dilation System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as `Property` or `Input port`. If set to `Property`, use the `???` property to specify the neighborhood or structuring

element values. Otherwise, specify the neighborhood using an input to the `step` method. Note that structuring elements can only be specified using `Neighborhood` property and they cannot be used as input to the `step` method. The default is `Property`.

### Neighborhood

Neighborhood or structuring element values

This property is applicable when the ??? property is set to `Property`. If you are specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function. The default is `[1 1; 1 1]`.

## Methods

<code>clone</code>	Create morphological dilate object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform morphological dilate on input

## Examples

Fuse fine discontinuities on images.

```
x = imread('peppers.png');
hcsc = vision.ColorSpaceConverter;
hcsc.Conversion = 'RGB to intensity';
hautothresh = vision.Autothresher;
hdilate = vision.MorphologicalDilate('Neighborhood', ones(5,5));
x1 = step(hcsc, x);
x2 = step(hautothresh, x1);
y = step(hdilate, x2);
figure;
```



```
subplot(3,1,1),imshow(x); title('Original image');  
subplot(3,1,2),imshow(x2); title('Thresholded Image');  
subplot(3,1,3),imshow(y); title('Dilated Image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Dilation](#) block reference page. The object properties correspond to the block parameters.

## See Also

[vision.MorphologicalErode](#) | [vision.MorphologicalOpen](#) | [vision.MorphologicalClose](#) | [strel](#)

**Introduced in R2012a**

# clone

**System object:** vision.MorphologicalDilate

**Package:** vision

Create morphological dilate object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MorphologicalDilate

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.MorphologicalDilate

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.MorphologicalDilate

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalDilate System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.MorphologicalDilate

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.MorphologicalDilate

**Package:** vision

Perform morphological dilate on input

## Syntax

ID = step(H,I)

ID = step(H,I,NHOOD)

## Description

ID = step(H,I) performs morphological dilation on input image I and returns the dilated image ID.

ID = step(H,I,NHOOD) uses input NHOOD as the neighborhood when the NeighborhoodSource property is set to Input port.

---

**Note:** H specifies the System object on which to run this step method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

## vision.MorphologicalErode System object

**Package:** vision

Perform morphological erosion on an image

### Description

---

**Note:** The `vision.MorphologicalErode` System object will be removed in a future release. Use the `imerode` function with equivalent functionality instead.

---

### Construction

`H = vision.MorphologicalErode` returns a System object, `H`, that performs morphological erosion on an intensity or binary image.

`H = vision.MorphologicalErode(Name, Value)` returns a morphological erosion System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as **Property** or **Input port**. If set to **Property**, use the `???` property to specify the neighborhood or structuring



element values. Otherwise, specify the neighborhood using an input to the `step` method. Note that structuring elements can only be specified using `Neighborhood` property and they cannot be used as input to the `step` method. The default is `Property`.

## Neighborhood

Neighborhood or structuring element values

This property is applicable when the `???` property is set to `Property`. If you are specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function. The default is `strel('square',4)`.

## Methods

<code>clone</code>	Create morphological erode object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform morphological erosion on input

## Examples

Erode an input image.

```
x = imread('peppers.png');
hcsc = vision.ColorSpaceConverter;
hcsc.Conversion = 'RGB to intensity';
hautothresh = vision.Autothresher;
herode = vision.MorphologicalErode('Neighborhood', ones(5,5));
x1 = step(hcsc, x); % convert input to intensity
x2 = step(hautothresh, x1); % convert input to binary
y = step(herode, x2); % Perform erosion on input
```

```
figure;  
  
subplot(3,1,1),imshow(x); title('Original image');  
subplot(3,1,2),imshow(x2); title('Thresholded Image');  
subplot(3,1,3),imshow(y); title('Eroded Image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the `Erosion` block reference page. The object properties correspond to the block parameters.

### See Also

`vision.MorphologicalDilate` | `vision.MorphologicalOpen` | `vision.MorphologicalClose` | `strel`

**Introduced in R2012a**

# clone

**System object:** vision.MorphologicalErode

**Package:** vision

Create morphological erode object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MorphologicalErode

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.MorphologicalErode

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.MorphologicalErode

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalErode System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.MorphologicalErode

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.MorphologicalErode

**Package:** vision

Perform morphological erosion on input

### Syntax

IE = step(H,I)

IE = step(H,I,NHOOD)

### Description

IE = step(H,I) performs morphological erosion on input image I and returns the eroded image IE.

IE = step(H,I,NHOOD) uses input NHOOD as the neighborhood when the NeighborhoodSource property is set to Input port.

---

**Note:** H specifies the System object on which to run this step method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---



# vision.MorphologicalOpen System object

**Package:** vision

Perform morphological opening on an image

## Description

---

**Note:** The `vision.MorphologicalOpen` System object will be removed in a future release. Use the `imopen` function with equivalent functionality instead.

---

## Construction

`H = vision.MorphologicalOpen` returns a System object, `H`, that performs morphological opening on an intensity or binary image.

`H = vision.MorphologicalOpen(Name, Value)` returns a morphological opening System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as `Property` or `Input port`. If set to `Property`, use the `???` property to specify the neighborhood or structuring

element values. Otherwise, specify the neighborhood using an input to the `step` method. Note that structuring elements can only be specified using `Neighborhood` property and they cannot be used as input to the `step` method. The default is `Property`.

### Neighborhood

Neighborhood or structuring element values

This property applies when you set the `???` property to `Property`. If you are specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If you are specifying a structuring element, use the `strel` function. The default is `strel('disk',5)`.

## Methods

<code>clone</code>	Create morphological open object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform morphological opening on input image

## Examples

Perform opening on an image

```
img = im2single(imread('blobs.png'));
hopening = vision.MorphologicalOpen;
hopening.Neighborhood = strel('disk', 5);
opened = step(hopening, img);
figure;

subplot(1,2,1),imshow(img); title('Original image');
subplot(1,2,2),imshow(opened); title('Opened image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Opening](#) block reference page. The object properties correspond to the block parameters.

### See Also

[vision.MorphologicalClose](#) | [vision.ConnectedComponentLabeler](#) | [vision.Autothresher](#) | [strel](#)

**Introduced in R2012a**

# clone

**System object:** vision.MorphologicalOpen

**Package:** vision

Create morphological open object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MorphologicalOpen

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.MorphologicalOpen

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.MorphologicalOpen

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalOpen System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

## release

**System object:** vision.MorphologicalOpen

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



---

## step

**System object:** vision.MorphologicalOpen

**Package:** vision

Perform morphological opening on input image

## Syntax

`IO = step(H,I)`

`IO = step(H,I,NHOOD)`

## Description

`IO = step(H,I)` performs morphological opening on binary or intensity input image `I`.

`IO = step(H,I,NHOOD)` uses input `NHOOD` as the neighborhood when the `NeighborhoodSource` property is set to `Input port`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.MorphologicalTopHat System object

**Package:** vision

Top-hat filtering on image

### Description

---

**Note:** The `vision.MorphologicalTopHat` System object will be removed in a future release. Use the `imtophat` function with equivalent functionality instead.

---

### Construction

`H = vision.MorphologicalTopHat` returns a top-hat filtering object, `H`, that performs top-hat filtering on an intensity or binary image using a predefined neighborhood or structuring element.

`H = vision.MorphologicalTopHat(Name, Value)` returns a top-hat filtering object, `H`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

### Properties

#### ImageType

Specify type of input image or video stream

Specify the type of input image or video stream as `Intensity` or `Binary`. The default is `Intensity`.

## NeighborhoodSource

Source of neighborhood values

Specify how to enter neighborhood or structuring element values as **Property** or **Input port**. If set to **Property**, use the `???` property to specify the neighborhood or structuring element values. Otherwise, specify the neighborhood using an input to the `step` method. Note that structuring elements can only be specified using the **Neighborhood** property. You cannot use the structuring elements as an input to the `step` method. The default is **Property**.

## Neighborhood

Neighborhood or structuring element values

This property applies only when you set the `???` property to **Property**. If specifying a neighborhood, this property must be a matrix or vector of 1s and 0s. If specifying a structuring element, use the `strel` function. The default is `strel('square',4)`.

## Methods

<code>clone</code>	Create morphological top hat filter with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Perform top-hat filtering on input image

## Examples

Perform top-hat filtering to correct uneven illumination.

```
I = im2single(imread('rice.png'));
htop = vision.MorphologicalTopHat('Neighborhood',strel('disk', 12));
```

```
% Improve contrast of output image
hc = vision.ContrastAdjuster; J = step(htop,I);
J = step(hc,J);
figure;

subplot(1,2,1),imshow(I); title('Original image');
subplot(1,2,2),imshow(J);
title('Top-hat filtered image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Top-hat block reference page. The object properties correspond to the block parameters.

## See Also

[strel](#) | [vision.MorphologicalBottomHat](#) | [vision.MorphologicalOpen](#)

**Introduced in R2012a**

# clone

**System object:** vision.MorphologicalTopHat

**Package:** vision

Create morphological top hat filter with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.MorphologicalTopHat

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.MorphologicalTopHat

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.MorphologicalTopHat

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the MorphologicalTopHat System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.



# release

**System object:** vision.MorphologicalTopHat

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.MorphologicalTopHat

**Package:** vision

Perform top-hat filtering on input image

## Syntax

`Y = step(H,I)`

`Y = step(H,I,NHOOD)`

## Description

`Y = step(H,I)` top-hat filters the input image, *I*, and returns the filtered image *Y*.

`Y = step(H,I,NHOOD)` filters the input image, *I* using the input *NHOOD* as the neighborhood when you set the `NeighborhoodSource` property to `Input port`. The object returns the filtered image in the output *Y*.

---

**Note:** *H* specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.VideoFileReader System object

**Package:** vision

Read video frames and audio samples from video file

## Description

The `VideoFileReader` object reads video frames, images, and audio samples from a video file. The object can also read image files.

## Supported Platforms and File Types

The supported file formats available to you depend on the codecs installed on your system.

Platform	Supported File Name Extensions
All Platforms	AVI (.avi)
Windows	<p><b>Image:</b> .jpg, .bmp</p> <p><b>Video:</b> MPEG (.mpeg) MPEG-2 (.mp2) MPEG-1 .mpg MPEG-4, including H.264 encoded video (.mp4, .m4v) Motion JPEG 2000 (.mj2) Windows Media Video (.wmv, .asf, .asx, .asx) and any format supported by Microsoft DirectShow® 9.0 or higher.</p> <p><b>Audio:</b> WAVE (.wav) Windows Media Audio File (.wma) Audio Interchange File Format (.aif, .aiff) Compressed Audio Interchange File Format (.aifc), MP3 (.mp3) Sun Audio (.au) Apple (.snd)</p>

Platform	Supported File Name Extensions
Macintosh	<b>Video:</b> .avi Motion JPEG 2000 (.mj2) MPEG-4, including H.264 encoded video (.mp4, .m4v) Apple QuickTime Movie (.mov) and any format supported by QuickTime as listed on <a href="http://support.apple.com/kb/HT3775">http://support.apple.com/kb/HT3775</a> .
	<b>Audio:</b> Uncompressed .avi
Linux	Motion JPEG 2000 (.mj2) Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on <a href="http://gstreamer.freedesktop.org/documentation/plugins.html">http://gstreamer.freedesktop.org/documentation/plugins.html</a> , including Ogg Theora (.ogg).

Windows XP and Windows 7 x64 platform ships with a limited set of 64-bit video and audio codecs. If a compressed multimedia file fails to play, try one of the two alternatives:

- Run the 32-bit version of MATLAB on your Windows XP x64 platform. Windows XP x64 ships with many 32-bit codecs.
- Save the multimedia file to a supported file format listed in the table above.

If you use Windows, use Windows Media player Version 11 or later.

---

**Note:** MJ2 files with bit depth higher than 8-bits is not supported by `vision.VideoFileReader`. Use `VideoReader` and `VideoWriter` for higher bit depths.

---

## Construction

`videoFReader = vision.VideoFileReader(FileName)` returns a video file reader System object, `videoFReader`. The object can sequentially read video frames and/or audio samples from the input video file, `FILENAME`. Every call to the `step` method returns the next video frame.

`videoFReader = vision.VideoFileReader(FileName,Name,Value)` configures the video file reader properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
Generated code for this function uses a precompiled platform-specific shared library.
“Code Generation Support, Usage Notes, and Limitations”.
<code>vision.VideoFileReader</code> does not generate code for reading compressed images on the Mac.

## To read a file:

- 1 Define and set up your video file reader object using the constructor.
- 2 Call the `step` method with the input filename, `FILENAME`, the video file reader object, `videoFReader`, and any optional properties. See the syntax below for using the `step` method.

`I = step(videoFReader)` outputs next video frame.

`[Y,Cb,Cr] = step(videoFReader)` outputs the next frame of YCbCr 4:2:2 format video in the color components Y, Cb, and Cr. This syntax requires that you set the `???` property to `'YCbCr 4:2:2'`.

`[I,AUDIO] = step(videoFReader)` outputs the next video frame, `I`, and one frame of audio samples, `AUDIO`. This syntax requires that you set the `???` property to `true`.

`[Y,Cb,Cr,AUDIO] = step(videoFReader)` outputs next frame of YCbCr 4:2:2 video in the color components Y, Cb, and Cr, and one frame of audio samples in `AUDIO`. This syntax requires that you set the `???` property to `true`, and the `???` property to `'YCbCr 4:2:2'`.

`[..., EOF] = step(videoFReader)` returns the end-of-file indicator, `EOF`. The object sets `EOF` to `true` each time the output contains the last audio sample and/or video frame.

## Properties

### Filename

Name of video file

Specify the name of the video file as a string. The full path for the file needs to be specified only if the file is not on the MATLAB path.

Default: `vipmen.avi`

### **PlayCount**

Number of times to play file

Specify a positive integer or `inf` to represent the number of times to play the file.

Default: 1

### **AudioOutputPort**

Output audio data

Use this property to control the audio output from the video file reader. This property applies only when the file contains both audio and video streams.

Default: `false`

### **ImageColorSpace**

RGB, YCbCr, or intensity video

Specify whether you want the video file reader object to output RGB, YCbCr 4:2:2 or intensity video frames. This property applies only when the file contains video. This property can be set to `RGB`, `Intensity`, or `YCbCr 4:2:2`.

Default: `RGB`

### **VideoOutputDataType**

Output video data type

Set the data type of the video data output from the video file reader object. This property applies if the file contains video. This property can be set to `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `Inherit`.

Default: `single`

## AudioOutputDataType

Output audio samples data type

Set the data type of the audio data output from the video file reader object. This property applies only if the file contains audio. This property can be set to `double`, `single`, `int16`, or `uint8`.

Default: `int16`

## Methods

<code>clone</code>	Create multimedia file reader object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>info</code>	Information about specified video file
<code>isDone</code>	End-of-file status (logical)
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>reset</code>	Reset internal states of multimedia file reader to read from beginning of file
<code>step</code>	Output video frame

## Examples

### Read and Play a Video File

Load the video using a video reader object.

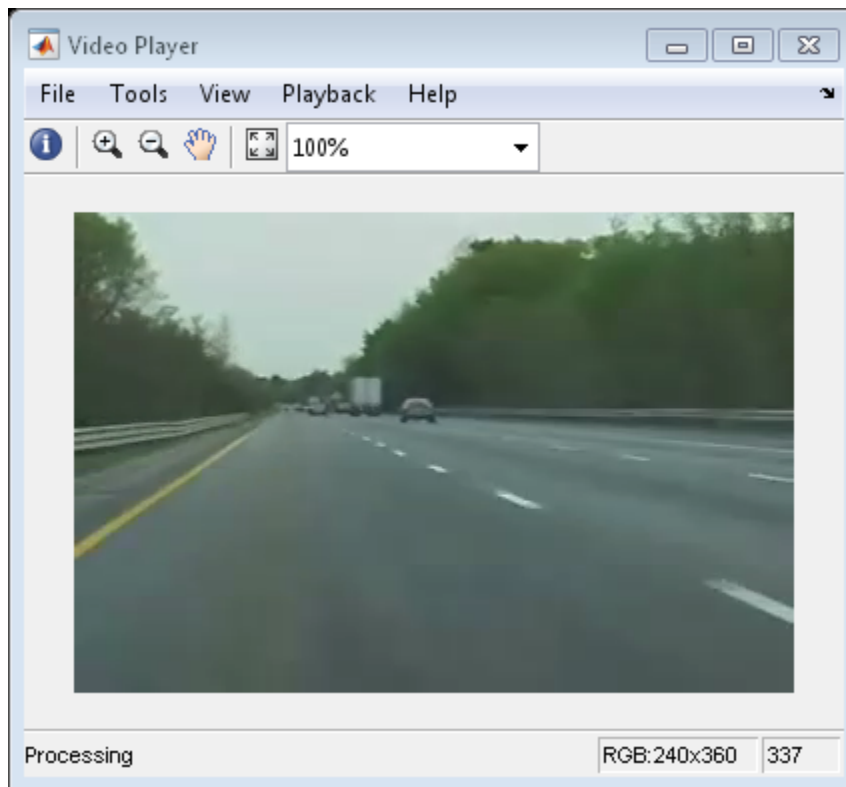
```
videoFReader = vision.VideoFileReader('viplanedeparture.mp4');
```

Create a video player object to play the video file.

```
videoPlayer = vision.VideoPlayer;
```

Use a while loop to read and play the video frames.

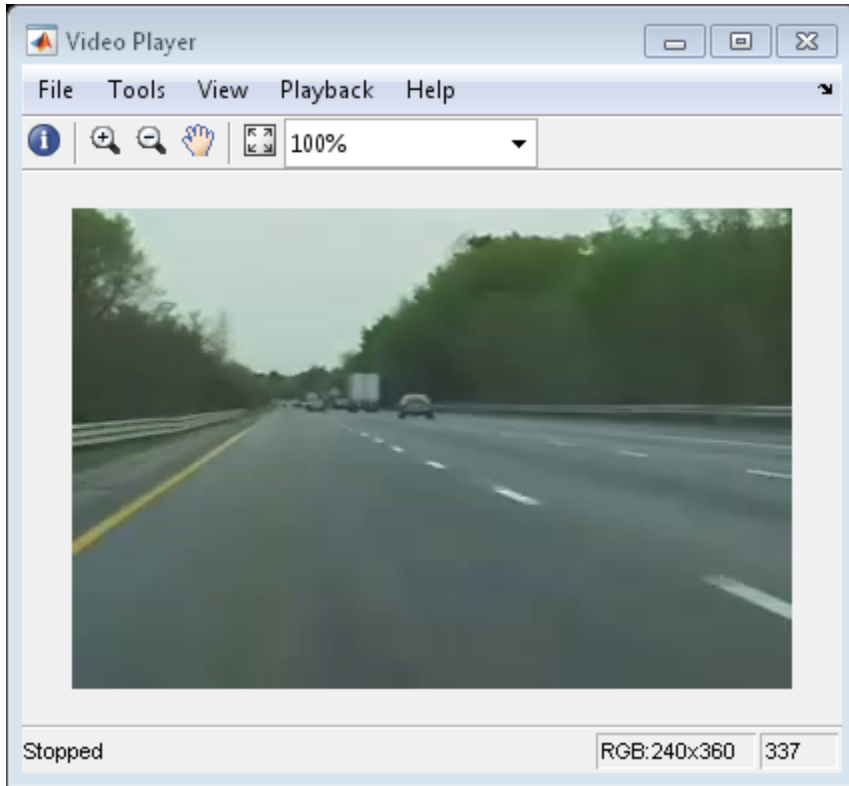
```
while ~isDone(videoFReader)  
    videoFrame = step(videoFReader);  
    step(videoPlayer, videoFrame);  
end
```



Release the objects.

```
release(videoPlayer);  
release(videoFReader);
```





## See Also

[vision.VideoPlayer](#) | [vision.VideoFileWriter](#) | [VideoWriter](#) | [VideoReader](#) | [implay](#)

**Introduced in R2012a**

# clone

**System object:** vision.VideoFileReader

**Package:** vision

Create multimedia file reader object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.VideoFileReader

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.VideoFileReader

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## info

**System object:** vision.VideoFileReader

**Package:** vision

Information about specified video file

## Syntax

`S = info(H)`

## Description

`S = info(H)` returns a MATLAB structure, `S`, with information about the video file specified in the `Filename` property. The fields and possible values for the structure `S` are described below:

Audio	Logical value indicating if the file has audio content.
Video	Logical value indicating if the file has video content.
VideoFrameRate	Frame rate of the video stream in frames per second. The value may vary from the actual frame rate of the recorded video, and takes into consideration any synchronization issues between audio and video streams when the file contains both audio and video content. This implies that video frames may be dropped if the audio stream leads the video stream by more than 1/(actual video frames per second).
VideoSize	Video size as a two-element numeric vector of the form: [VideoWidthInPixels, VideoHeightInPixels]
VideoFormat	Video signal format.

## isDone

**System object:** vision.VideoFileReader

**Package:** vision

End-of-file status (logical)

## Syntax

```
status = isDone(H)
```

## Description

`status = isDone(H)` returns a logical value indicating that the `VideoFileReader` System object, `H`, has reached the end of the multimedia file after playing it `PlayCount` number of times. After the object plays the file the number of times set by the `PlayCount` property, it sets the status to `true`.

## isLocked

**System object:** vision.VideoFileReader

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the VideoFileReader System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.VideoFileReader

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



## **reset**

Reset internal states of multimedia file reader to read from beginning of file

### **Syntax**

reset(H)

### **Description**

reset(H) resets the `VideoFileReader` object to read from the beginning of the file.

# step

**System object:** vision.VideoFileReader

**Package:** vision

Output video frame

## Syntax

```
I = step(H)
[Y,Cb,Cr] = step(H)
[I,AUDIO] = step(H)
[Y,Cb,Cr,AUDIO] = step(H)
[... , EOF] = step(H)
```

## Description

Use the **step** method with the video file reader object and any optional properties to output the next video frame.

**I = step(H)** outputs next video frame.

**[Y,Cb,Cr] = step(H)** outputs next frame of YCbCr 4:2:2 video in the color components Y, Cb, and Cr. This syntax requires that you set the **ImageColorSpace** property to 'YCbCr 4:2:2'.

**[I,AUDIO] = step(H)** outputs next video frame, I, and one frame of audio samples, AUDIO. This syntax requires that you set the **AudioOutputPort** property to **true**.

**[Y,Cb,Cr,AUDIO] = step(H)** outputs next frame of YCbCr 4:2:2 video in the color components Y, Cb, and Cr, and one frame of audio samples in AUDIO. This syntax requires that you set the **AudioOutputPort** property to **true**, and the **ImageColorSpace** property to 'YCbCr 4:2:2'.

**[... , EOF] = step(H)** returns the end-of-file indicator, EOF. The object sets EOF to **true** each time the output contains the last audio sample and/or video frame.

---

**Note:** H specifies the System object on which to run this **step** method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.VideoFileWriter System object

**Package:** vision

Write video frames and audio samples to video file

### Description

The `VideoFileWriter` object writes video frames and audio samples to a video file. `.mwv` files can be written only on Windows. The video and audio can be compressed. The available compression types depend on the encoders installed on the platform.

---

**Note** This block supports code generation for platforms that have file I/O available. You cannot use this block with Simulink Desktop Real-Time software, because that product does not support file I/O.

This object performs best on platforms with Version 11 or later of Windows Media Player software. This object supports only uncompressed RGB24 AVI files on Linux and Mac platforms.

Windows 7 UAC (User Account Control), may require administrative privileges to encode `.mwv` and `.wma` files.

---

The generated code for this object relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from an object, you may need to add precompiled shared library files to your system path. See “MATLAB Coder” and “Simulink Shared Library Dependencies” for details.

This object allows you to write `.wma` / `.wmv` streams to disk or across a network connection. Similarly, the `vision.VideoFileReader` object allows you to read `.wma` / `.wmv` streams to disk or across a network connection. If you want to play an `.mp3` / `.mp4` file, but you do not have the codecs, you can re-encode the file as `.wma` / `.wmv`, which is supported by the Computer Vision System Toolbox.

## Construction

`videoFWriter = vision.VideoFileWriter` returns a video file writer System object, `videoFWriter`. It writes video frames to an uncompressed 'output.avi' video file. Every call to the `step` method writes a video frame.

`videoFWriter = vision.VideoFileWriter(FILENAME)` returns a video file writer object, `videoFWriter` that writes video to a file, `FILENAME`. The file type can be `.avi` or `.wmv`, specified by the `???` property.

`videoFWriter = vision.VideoFileWriter(..., 'Name', Value)` configures the video file writer properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
Generated code for this function uses a precompiled platform-specific shared library.
“Code Generation Support, Usage Notes, and Limitations”.

## To write to a file:

- 1 Define and set up your video file writer object using the constructor.
- 2 Call the `step` method with the optional output filename, `FILENAME`, the video file writer object, `videoFWriter`, and any optional properties. See the syntax below for using the `step` method.

`step(videoFWriter, I)` writes one frame of video, `I`, to the output file. The input video can be an  $M$ -by- $N$ -by-3 truecolor RGB video frame, or an  $M$ -by- $N$  grayscale video frame.

`step(videoFWriter, Y, Cb, Cr)` writes one frame of YCbCr 4:2:2 video. The width of Cb and Cr color components must be half of the width of Y. You must set the value of the `???` property to 'YCbCr 4:2:2'.

`step(videoFWriter, I, AUDIO)` writes one frame of the input video, `I`, and one frame of audio samples, `AUDIO`, to the output file. This applies when you set the `???` property to `true`.

`step(videoFWriter, Y, Cb, Cr, AUDIO)` writes one frame of YCbCr 4:2:2 video, and one frame of audio samples, `AUDIO`, to the output file. This applies when you set the `???` to `true` and the `???` property to `'YCbCr 4:2:2'`. The width of Cb and Cr color components must be half of the width of Y.

## Properties

### Filename

Video output file name

Specify the name of the video file as a string. The file extension you give for `Filename` must match the `FileFormat`. The supported file extensions are, `'.wmv'`, `'.avi'`, `'.mj2'`, `'.mp4'`, and `'.m4v'`.

Default: `output.avi`

### FileFormat

Output file format

Specify the format of the file that is created. Supported formats and abbreviations:

File Format	Description	File Extension	Supported Platform
AVI	Audio-Video Interleave file	<code>.avi</code>	All platforms
MJ2000	Motion JPEG 2000 file	<code>.mj2</code>	All platforms
WMV	Windows Media Video	<code>.wmv</code>	Windows
MPEG4	MPEG-4/H.264 Video	<code>.mp4</code> , <code>.m4v</code>	Windows and Mac

Default: `AVI`

### AudioInputPort

Write audio data

Use this property to control whether the object writes audio samples to the video file. Set this value to `true` to write audio data.

Default: `false`

### **FrameRate**

Video frame rate

Specify the frame rate of the video data in frames per second as a positive numeric scalar.

For videos which also contain audio data, the rate of the audio data will be determined as the rate of the video multiplied by the number of audio samples passed in each invocation of the `step` method. For example, if you use a frame rate of 30, and pass 1470 audio samples to the `step` method, the object sets the audio sample to 44100, ( $1470 \times 30 = 44100$ ).

Default: 30

### **AudioCompressor**

Audio compression encoder

Specify the type of compression algorithm to implement for audio data. This compression reduces the size of the video file. Choose `None` (`uncompressed`) to save uncompressed audio data to the video file. The other options reflect the available audio compression algorithms installed on your system. This property applies only when writing AVI files on Windows platforms.

### **VideoCompressor**

Video compression encoder

Specify the type of compression algorithm to use to compress the video data. This compression reduces the size of the video file. Choose `None` (`uncompressed`) to save uncompressed video data to the video file. The `VideoCompressor` property can also be set to one of the compressors available on your system. To obtain a list of available video compressors, you can use tab completion. Follow these steps:

- 1 Instantiate the object:

```
y = vision.VideoFileWriter
```

- 2 To launch the tab completion functionality, type the following up to the open quote.

```
y.VideoCompressor='
```

A list of compressors available on your system will appear after you press the Tab key. For example:

```
>>
```

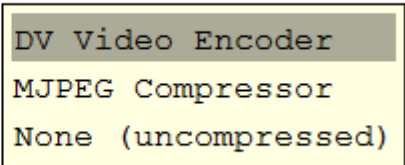
```
>>
```

```
>>
```

```
>>
```

```
>>
```

```
>> y.VideoCompressor='
```



```
DV Video Encoder
MJPEG Compressor
None (uncompressed)
```

This property applies only when writing AVI files on Windows platforms.

### **AudioDataType**

Uncompressed audio data type

Specify the compressed output audio data type. This property only applies when you write uncompressed WAV files.

### **FileColorSpace**

Color space for output file

Specify the color space of AVI files as RGB or YCbCr 4:2:2. This property applies when you set the ??? property to AVI and only on Windows platforms.

Default: RGB

### **Quality**

Control size of output video file

Specify the output video quality as an integer in the range [0,100]. Increase this value for greater video quality. However, doing so increases the file size. Decrease the value to lower video quality with a smaller file size.



The **Quality** property only applies when you are writing MPEG4 video files (on Windows or Mac) or when you are writing MJPEG-AVI video only files on a Mac or Linux.

### **CompressionFactor**

Target ratio between number of bytes in input image and compressed image

Specify the compression factor as an integer greater than 1 to indicate the target ratio between the number of bytes in the input image and the compressed image. The data is compressed as much as possible, up to the specified target. This property applies only when writing Lossy MJ2000 files.

## **Methods**

## **Examples**

### **Write a Video to an AVI File**

**Set up the reader and writer objects.**

```
videoFReader = vision.VideoFileReader('viplanedeparture.mp4');  
videoFWriter = vision.VideoFileWriter('myFile.avi', 'FrameRate', videoFReader.info.VideoRate);
```

**Write the first 50 frames from original file into a newly created AVI file.**

```
for i=1:50  
    videoFrame = step(videoFReader);  
    step(videoFWriter, videoFrame);  
end
```

**Close the input and output files.**

```
release(videoFReader);  
release(videoFWriter);
```

### **See Also**

vision.VideoPlayer | vision.VideoFileReader | VideoWriter | VideoReader

**Introduced in R2012a**

# clone

**System object:** vision.VideoFileWriter

**Package:** vision

Create video file writer object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.VideoFileWriter

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.VideoFileWriter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.VideoFileWriter

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the VideoFileWriter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.VideoFileWriter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

---

## step

**System object:** vision.VideoFileWriter

**Package:** vision

Write input video data to file

## Syntax

```
step(videoFWriter,I)
step(videoFWriter,Y,Cb,Cr)
step(videoFWriter,I,AUDIO)
step(videoFWriter,Y,Cb,Cr,AUDIO)
```

## Description

`step(videoFWriter,I)` writes one frame of the input video, `I`, to the output file. The input video can be an  $M$ -by- $N$ -by-3 truecolor RGB video frame, or an  $M$ -by- $N$  grayscale video frame where  $M$ -by- $N$  represents the size of the image.

`step(videoFWriter,Y,Cb,Cr)` writes one frame of YCbCr 4:2:2 video. The width of Cb and Cr color components must be half of the width of Y. You must set the value of the ??? property to 'YCbCr 4:2:2'.

`step(videoFWriter,I,AUDIO)` writes one frame of the input video, `I`, and one frame of audio samples, `AUDIO`, to the output file. This applies when you set the ??? property to true.

`step(videoFWriter,Y,Cb,Cr,AUDIO)` writes one frame of YCbCr 4:2:2 video, and one frame of audio samples, `AUDIO`, to the output file. This applies when you set the ??? to true and the ??? property to 'YCbCr 4:2:2'. The width of Cb and Cr color components must be half of the width of Y.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---



# ocrText class

Object for storing OCR results

## Description

`ocrText` contains recognized text and metadata collected during optical character recognition (OCR). The `ocr` function returns the `ocrText` object. You can access the information contained in the object with the `ocrText` properties. You can also locate text that matches a specific pattern with the object's `locateText` method.

Code Generation Support
Compile-time constant input: No restrictions.
Supports MATLAB Function block: No
“Code Generation Support, Usage Notes, and Limitations”

## Properties

### Text — Text recognized by OCR

array of characters

Text recognized by OCR, specified as an array of characters. The text includes white space and new line characters.

### CharacterBoundingBoxes — Bounding box locations

*M*-by-4 matrix

Bounding box locations, stored as an *M*-by-4 matrix. Each row of the matrix contains a four-element vector,  $[x \ y \ width \ height]$ . The  $[x \ y]$  elements correspond to the upper-left corner of the bounding box. The  $[width \ height]$  elements correspond to the size of the rectangular region in pixels. The bounding boxes enclose text found in an image using the `ocr` function. Bounding boxes width and height that correspond to new line characters are set to zero. Character modifiers found in languages, such as Hindi, Tamil, and Bangalese, are also contained in a zero width and height bounding box.

### CharacterConfidences — Character recognition confidence

array

Character recognition confidence, specified as an array. The confidence values are in the range [0, 1]. A confidence value, set by the `ocr` function, should be interpreted as a probability. The `ocr` function sets confidence values for spaces between words and sets new line characters to NaN. Spaces and new line characters are not explicitly recognized during OCR. You can use the confidence values to identify the location of misclassified text within the image by eliminating characters with low confidence.

### **Words — Recognized words**

cell array

Recognized words, specified as a cell array.

### **WordBoundingBoxes — Bounding box location and size**

*M*-by-4 matrix

Bounding box location and size, stored as an *M*-by-4 matrix. Each row of the matrix contains a four-element vector, [*x y width height*], that specifies the upper left corner and size of a rectangular region in pixels.

### **WordConfidences — Recognition confidence**

array

Recognition confidence, specified as an array. The confidence values are in the range [0, 1]. A confidence value, set by the `ocr` function, should be interpreted as a probability. The `ocr` function sets confidence values for spaces between words and sets new line characters to NaN. Spaces and new line characters are not explicitly recognized during OCR. You can use the confidence values to identify the location of misclassified text within the image by eliminating words with low confidence.

## Methods

`locateText`

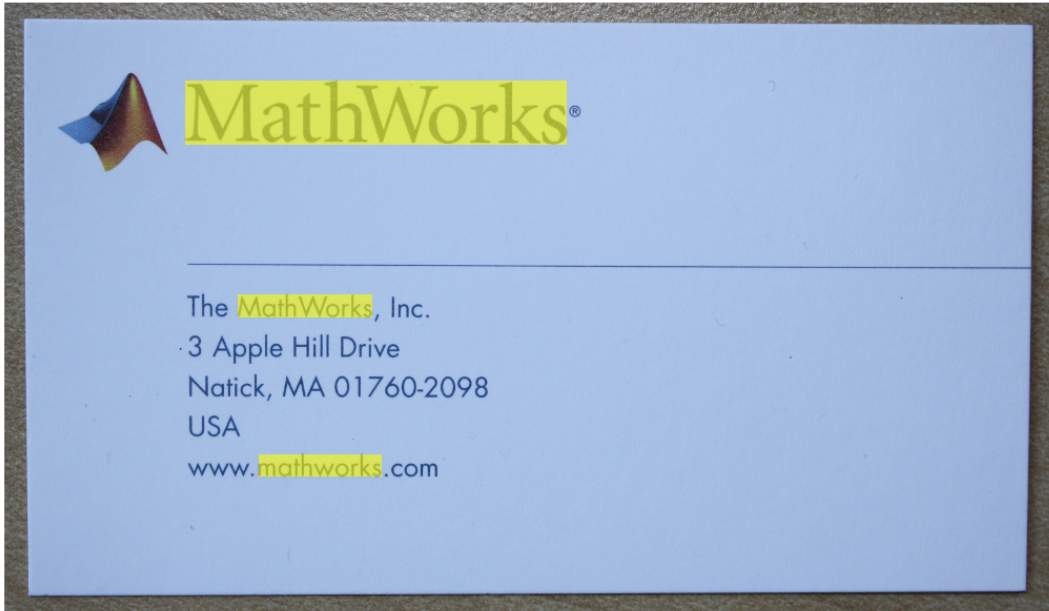
Locate string pattern

## Examples

### **Find and Highlight Text in an Image**

```
businessCard = imread('businessCard.png');
```

```
ocrResults = ocr(businessCard);  
bboxes = locateText(ocrResults, 'MathWorks', 'IgnoreCase', true);  
Iocr = insertShape(businessCard, 'FilledRectangle', bboxes);  
figure; imshow(Iocr);
```



### Find Text Using Regular Expressions

```
businessCard = imread('businessCard.png');  
ocrResults = ocr(businessCard);  
bboxes = locateText(ocrResults, 'www.*com', 'UseRegexp', true);  
img = insertShape(businessCard, 'FilledRectangle', bboxes);  
figure; imshow(img);
```



**See Also**

`insertShape` | `ocr` | `regexp` | `strfind`

**Introduced in R2014a**

# locateText

**Class:** ocrText

Locate string pattern

## Syntax

```
bboxes = locateText(ocrText,pattern)
bboxes = locateText(ocrText,pattern,Name, Value)
```

## Description

`bboxes = locateText(ocrText,pattern)` returns the location and size of bounding boxes stored in the `ocrText` object. The `locateText` method returns only the locations of bounding boxes which correspond to text within an image that exactly match the input pattern.

`bboxes = locateText(ocrText,pattern,Name, Value)` uses additional options specified by one or more `Name,Value` arguments.

## Input Arguments

### **ocrText** — Object containing OCR results

ocrText object

Recognized text and metrics, returned as an `ocrText` object. The object contains the recognized text, the location of the recognized text within the input image, and the metrics indicating the confidence of the results. The confidence values range between 0 and 100 and represent a percent probability. When you specify an *M*-by-4 `roi`, the function returns `ocrText` as an *M*-by-1 array of `ocrText` objects. Confidence values range between 0 and 1. Interpret the confidence values as probabilities.

### **pattern** — OCR string pattern

single string | cell array of strings

OCR string pattern, specified as a single string or a cell array of strings. The method returns only the locations of bounding boxes which correspond to text within an image that exactly match the input `pattern`.

### 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`.

#### 'UseRegex' — Regular expression

`false` (default) | logical scalar

Regular expression, specified as a logical scalar. When you set this property to `true`, the method treats the pattern as a regular expression. For more information about regular expressions, see `regex`.

#### 'IgnoreCase' — Case sensitivity

`false` (default) | logical scalar

Case sensitivity, specified as a logical scalar. When you set this property to `true`, the method performs case-insensitive text location.

## Output Arguments

### `bbbox` — Text bounding boxes

method |  $M$ -by-4 matrix

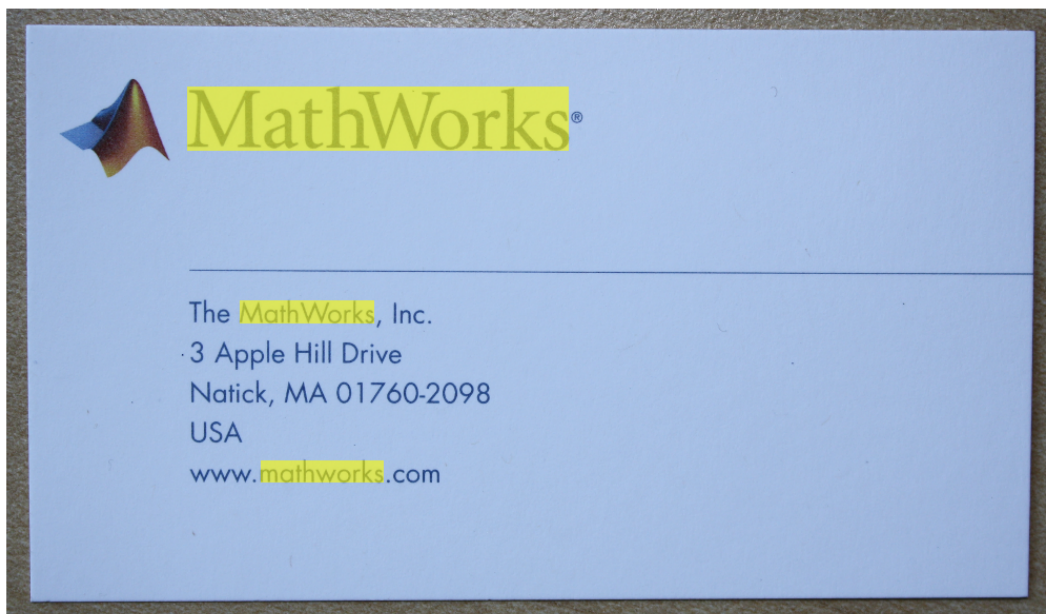
Text bounding boxes, specified as an  $M$ -by-4 matrix. Each row of the matrix contains a four-element vector,  $[x\ y\ width\ height]$ . The  $[x\ y]$  elements correspond to the upper-left corner of the bounding box. The  $[width\ height]$  elements correspond to the size of the rectangular region in pixels. The bounding boxes enclose text found in an image using the `ocr` function. The `ocr` function stores OCR results in the `ocrText` object.

## Examples

### Find and Highlight Text in an Image

```
businessCard = imread('businessCard.png');
```

```
ocrResults = ocr(businessCard);  
bboxes = locateText(ocrResults, 'MathWorks', 'IgnoreCase', true);  
Iocr = insertShape(businessCard, 'FilledRectangle', bboxes);  
figure; imshow(Iocr);
```



### Find Text Using Regular Expressions

```
businessCard = imread('businessCard.png');  
ocrResults = ocr(businessCard);  
bboxes = locateText(ocrResults, 'www.*com', 'UseRegexp', true);  
img = insertShape(businessCard, 'FilledRectangle', bboxes);  
figure; imshow(img);
```







# pointCloud class

Object for storing a 3-D point cloud

## Syntax

```
ptCloud = pointCloud(xyzPoints)
ptCloud = pointCloud(xyzPoints,Name,Value)
```

## Description

`ptCloud = pointCloud(xyzPoints)` returns a point cloud object with coordinates specified by `xyzPoints`.

`ptCloud = pointCloud(xyzPoints,Name,Value)` returns a point cloud object, with additional properties specified by one or more `Name,Value` pair arguments.

## Tips

The `pointCloud` object is a handle object. If you want to create a separate copy of a point cloud, you can use the `MATLAB` `copy` method.

```
ptCloudB = copy(ptCloudA)
```

If you want to preserve a single copy of a point cloud, which can be modified by point cloud functions, use the same point cloud variable name for the input and output.

```
ptCloud = pcFunction(ptCloud)
```

## Input Arguments

### **xyzPoints** — Point cloud *x*, *y*, and *z* locations

*M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud *x*, *y*, and *z* locations, specified as either an *M*-by-3 or an *M*-by-*N*-by-3 matrix. The *M*-by-*N*-by-3 matrix is commonly referred to as an *organized point cloud*. The `xyzPoints` matrix contains *M* or *M*-by-*N* [*x,y,z*] points. The *z* values in the matrix, which generally correspond to depth or elevation, determine the color of each point. The `xyzPoints` input must be numeric.

### Name-Value Pair Arguments

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

Example: `'Color',[0 0 255]`

#### 'Color' — Point cloud color

[ ] (default) | *M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud color of points, specified as the comma-separated pair of 'Color' and an *M*-by-3, or *M*-by-*N*-by-3 matrix. RGB values range between [0, 255]. Single or double inputs are rescaled to [0, 255].

Coordinates	Valid Values of Color	
<i>M</i> -by-3 matrix	<i>M</i> -by-3 matrix	
<i>M</i> -by- <i>N</i> -by-3 matrix	<i>M</i> -by- <i>N</i> -by-3 matrix containing RGB values for each point.	

**'Normal'** — Normal vector[ ] (default) |  $M$ -by-3 matrix |  $M$ -by- $N$ -by-3 matrix

Normal vector at each point, specified as the comma-separated pair consisting of 'Normal' and a matrix. The normal matrix contains the coordinates for each point and their normal vectors.

Coordinates	Normal Matrix
$M$ -by-3 matrix	$M$ -by-3 matrix, where each row contains a corresponding normal vector.
$M$ -by- $N$ -by-3 matrix	$M$ -by- $N$ -by-3 matrix containing a 1-by-1-by-3 normal vector for each point.

## Properties

**Location** — Point coordinates $M$ -by-3 matrix |  $M$ -by- $N$ -by-3 matrix

Point coordinates, stored as an  $M$ -by-3 or  $M$ -by- $N$ -by-3 matrix. Each entry specifies the  $x$ ,  $y$ ,  $z$  coordinates of a point. This property is read-only.

**Color** — RGB color of point $M$ -by-3 matrix |  $M$ -by- $N$ -by-3 matrix

RGB color of point, specified as an  $M$ -by-3 or  $M$ -by- $N$ -by-3 matrix. Each entry specifies the RGB color of a point.

Data Types: `uint8`**Normal** — Normal vector $M$ -by-3 matrix |  $M$ -by- $N$ -by-3 matrix

Normal vector, specified as an  $M$ -by-3 matrix or  $M$ -by- $N$ -by-3 matrix. Each entry specifies the  $x$ ,  $y$ ,  $z$  component of a normal vector of a point. The data type must be the same as the `Location` data type.

**Count** — Number of points

positive integer

Number of points, stored as a positive integer. This property is read-only.

### **XLimits** — Range of x-axis coordinates

1-by-2 vector

Range of coordinates along x-axis, stored as a 1-by-2 vector. This property is read-only.

### **YLimits** — Range of y-axis coordinates

1-by-2 vector

Range of coordinates along y-axis, stored as a 1-by-2 vector. This property is read-only.

### **ZLimits** — Range of z-axis coordinates

1-by-2 vector

Range of coordinates along z-axis, stored as a 1-by-2 vector. This property is read-only.

## Methods

findNearestNeighbors	Find nearest neighbors of a point
findNeighborsInRadius	Find neighbors within a radius
findPointsInROI	Find points within ROI
removeInvalidPoints	Remove invalid points
select	Select points by index

## Examples

### **Find Shortest Distance Between Two Point Clouds**

Create two point clouds.

```
ptCloud1 = pointCloud(rand(100,3,'single'));  
ptCloud2 = pointCloud(1+rand(100,3,'single'));  
  
minDist = inf;
```

Find the nearest neighbor in ptCloud2 for each point in ptCloud1.

```
for i = 1 : ptCloud1.Count  
    point = ptCloud1.Location(i,:);
```

```
[~,dist] = findNearestNeighbors(ptCloud2,point,1);  
if dist < minDist  
    minDist = dist;  
end  
end
```

- “3-D Point Cloud Registration and Stitching”

## See Also

[pcplayer](#) | [affine3d](#) | [pcdenoise](#) | [pcdownsample](#) | [pcmerge](#) | [pcread](#) | [pcregrigid](#)  
| [pcshow](#) | [pctransform](#) | [pcwrite](#)

## More About

- “Coordinate Systems”

**Introduced in R2015a**

# findNearestNeighbors

**Class:** pointCloud

Find nearest neighbors of a point

## Syntax

```
[indices,dists] = findNearestNeighbors(ptCloud,point,K)  
[indices,dists] = findNearestNeighbors(ptCloud,point,K,Name, Value)
```

## Description

[indices,dists] = findNearestNeighbors(ptCloud,point,K) returns the K nearest neighbors of a query point within the input point cloud. The point input is an [X,Y,Z] vector.

[indices,dists] = findNearestNeighbors(ptCloud,point,K,Name, Value) uses additional options specified by one or more Name,Value arguments.

## Input Arguments

### **ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **point** — Query point

[X,Y,Z] vector.

Query point, specified as an [X,Y,Z] vector.

### **K** — Number of nearest neighbors

positive integer

Number of nearest neighbors, specified as a positive integer.

## 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`.

### 'Sort' — Sort indices

`false` (default) | `true`

Sort indices, specified as a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn sorting off, set `Sort` to `false`.

### 'MaxLeafChecks' — Number of leaf nodes

`inf` (default) | `integer`

Number of leaf nodes, specified as an integer. Set `MaxLeafChecks` to the number of leaf nodes to search in the Kdtree. When you set this value to `inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

## Output Arguments

### `indices` — Indices of stored points

column vector

Indices of stored points, returned as a column vector. The `indices` output contains K linear indices to the stored points in the point cloud.

### `dists` — Distances to query point

column vector

Distances to query point, returned as a column vector. The `dists` contains K Euclidean distances to the query point.

## Examples

### Find K-Nearest Neighbors in a Point Cloud

Create a point cloud object with randomly generated points.

```
ptCloud = pointCloud(1000*rand(100,3, 'single'));
```

Define a query point and the number of neighbors.

```
point = [50,50,50];  
K = 10;
```

Get the indices and the distances of 10 nearest points.

```
[indices,dists] = findNearestNeighbors(ptCloud,point,K);
```

## References

Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". In *VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

## See Also

pointCloud

**Introduced in R2015a**



# findNeighborsInRadius

**Class:** pointCloud

Find neighbors within a radius

## Syntax

```
[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)
[indices,dists] = findNeighborsInRadius(ptCloud,point,radius,Name,
Value)
```

## Description

`[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)` returns the neighbors within a radius of a query point.

`[indices,dists] = findNeighborsInRadius(ptCloud,point,radius,Name,Value)` uses additional options specified by one or more `Name,Value` arguments.

## Input Arguments

### **ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **point** — Query point

$[X,Y,Z]$  vector.

Query point, specified as an  $[X,Y,Z]$  vector.

### **radius** — Radius

scalar

Radius, specified as a scalar. The function finds the neighbors within the `radius` of a query point.

### 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`.

#### 'Sort' — Sort indices

`false` (default) | `true`

Sort indices, specified as a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn sorting off, set `Sort` to `false`.

#### 'MaxLeafChecks' — Number of leaf nodes

`inf` (default) | `integer`

Number of leaf nodes, specified as an integer. Set `MaxLeafChecks` to the number of leaf nodes to search in the `Kdtree`. When you set this value to `inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

### Output Arguments

#### `indices` — Indices of stored points

column vector

Indices of stored points, returned as a column vector. The `indices` output contains `K` linear indices to the stored points in the point cloud.

#### `dists` — Distances to query point

column vector

Distances to query point, returned as a column vector. The `dists` contains `K` Euclidean distances to the query point.

### Examples

#### Find Neighbors Within A Given Radius Using `Kdtree`

Create a point cloud object with randomly generated points.

```
ptCloud = pointCloud(100*rand(1000,3, 'single'));
```

Define a query point and set the radius.

```
point = [50,50,50];  
radius = 5;
```

Get all of the points within the radius.

```
[indices, dists] = findNeighborsInRadius(ptCloud,point,radius)
```

```
indices =
```

```
Empty matrix: 0-by-1
```

```
dists =
```

```
Empty matrix: 0-by-1
```

## References

Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". In *VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

**Introduced in R2015a**

# findPointsInROI

**Class:** pointCloud

Find points within ROI

## Syntax

```
indices = findPointsInROI(ptCloud,roi)
```

## Description

`indices = findPointsInROI(ptCloud,roi)` returns the points within a region of interest.

## Input Arguments

**ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

**roi** — Region of interest

3-by-2 matrix

Region of interest, specified as a 3-by-2 matrix. The format defining the region of interest cuboid matrix is [*xmin*, *xmax*; *ymin*, *ymax*; *zmin*, *zmax*].

## Output Arguments

**indices** — Linear indices to stored points

column vector

Linear indices to stored points, returned as a column vector. The indices vector contains the stored points in the `ptCloud` object.

## Examples

### Find Points Within Cuboid

Create a point cloud object with randomly generated points.

```
ptCloudA = pointCloud(100*rand(1000,3,'single'));
```

Define a cuboid.

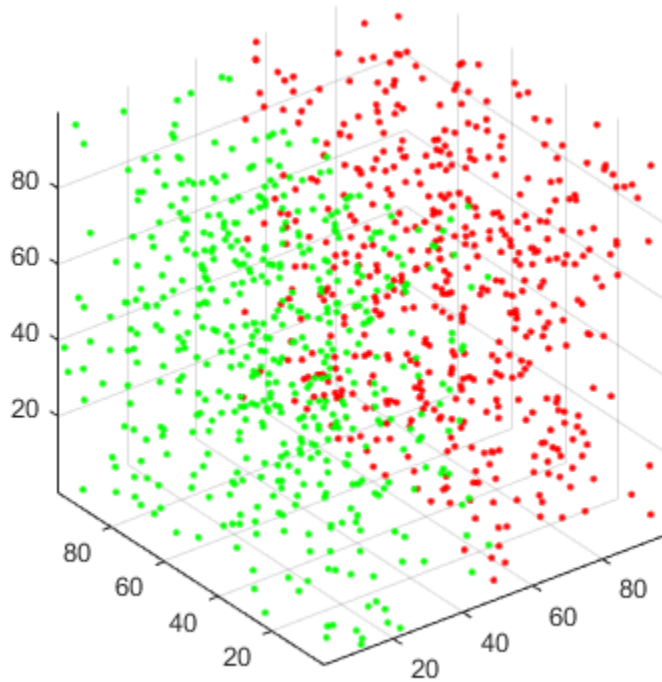
```
roi = [0,50;0,inf;0,inf];
```

Find all the points within the cuboid.

```
indices = findPointsInROI(ptCloudA, roi);  
ptCloudB = select(ptCloudA,indices);
```

Display points within the ROI.

```
pcshow(ptCloudA.Location,'r');  
hold on;  
pcshow(ptCloudB.Location,'g');  
hold off;
```



## References

Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". In *VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

**Introduced in R2015a**

# removeInvalidPoints

**Class:** pointCloud

Remove invalid points

## Syntax

```
[ptCloudOut,indices]= removeInvalidPoints(ptCloud)
```

## Description

[ptCloudOut,indices]= removeInvalidPoints(ptCloud) removes points with Inf or NaN coordinates from point cloud and provides indices of valid points.

## Input Arguments

**ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

## Output Arguments

**ptCloudOut** — Point cloud with points removed

pointCloud object

Point cloud, returned as a pointCloud object with Inf or NaN coordinates removed.

---

**Note:** After calling this function, an organized point cloud ( $M$ -by- $N$ -by-3) becomes an unorganized ( $X$ -by-3) point cloud.

---

**indices** — Indices of valid points

vector

Indices of valid points in the point cloud, specified as a vector.

## Examples

### Remove Infinite-valued Points From a Point Cloud

Create a point cloud object.

```
ptCloud = pointCloud(nan(100,3))
% Remove invalid points.
ptCloud = removeInvalidPoints(ptCloud)
```

```
ptCloud =
```

```
pointCloud with properties:
```

```
Location: [100x3 double]
Color: []
Normal: []
Count: 100
XLimits: [0x2 double]
YLimits: [0x2 double]
ZLimits: [0x2 double]
```

```
ptCloud =
```

```
pointCloud with properties:
```

```
Location: [0x3 double]
Color: []
Normal: []
Count: 0
XLimits: [0x2 double]
YLimits: [0x2 double]
ZLimits: [0x2 double]
```

**Introduced in R2015a**



# select

**Class:** pointCloud

Select points by index

## Syntax

```
ptCloudOut = select(ptCloud,indices)
ptCloudOut = select(ptCloud,row,column)
```

## Description

`ptCloudOut = select(ptCloud,indices)` returns a pointCloud object containing the points selected using linear indices.

`ptCloudOut = select(ptCloud,row,column)` returns a pointCloud object containing the points selected using row and column subscripts.

## Input Arguments

### **ptCloud** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **indices** — Indices of stored points

vector

Indices of stored points, specified as a vector.

### **row** — row subscript

vector

This syntax applies only to organized point cloud in the format (*M*-by-*N*-by-3).

**column** — column subscript

vector

This syntax applies only to organized point cloud in the format (*M*-by-*N*-by-3).

## Output Arguments

**ptCloudOut** — Selected point cloud

pointCloud object

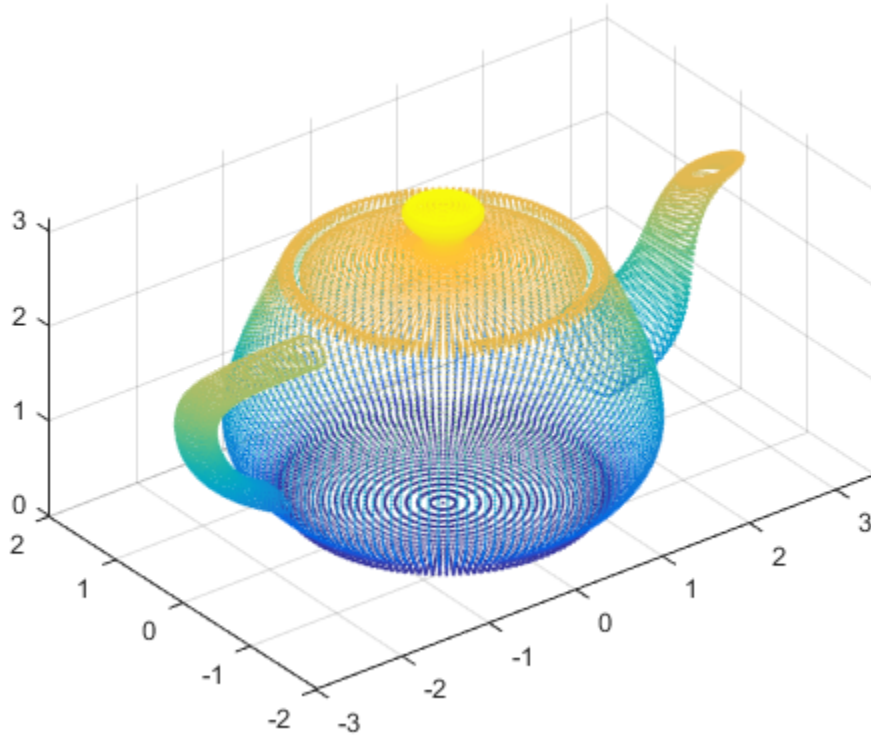
Point cloud, returned as a pointCloud object.

## Examples

### Downsample Point Cloud With Fixed Step

Read and display a point cloud file.

```
ptCloud = pcread('teapot.ply');  
figure  
pcshow(ptCloud);
```



Downsample a point cloud with fixed step size

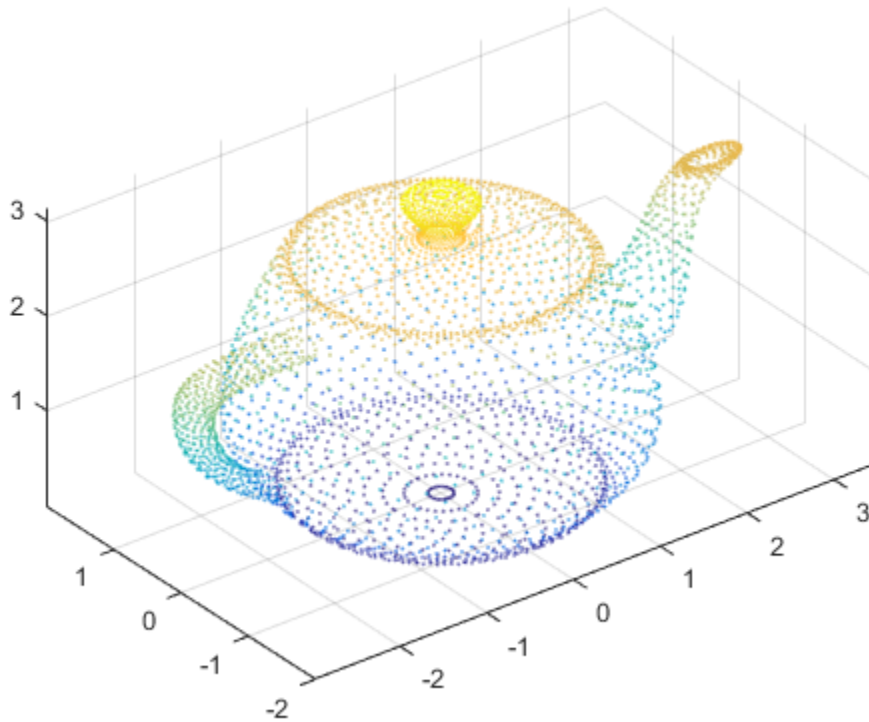
```
stepSize = 10;  
indices = 1:stepSize:ptCloud.Count;
```

Select indexed point cloud.

```
ptCloudOut = select(ptCloud, indices);
```

Display the selected point cloud.

```
figure  
pcshow(ptCloudOut);
```



Introduced in R2015a

# cylinderModel class

Object for storing a parametric cylinder model

## Syntax

```
model = cylinderModel(params)
```

## Description

`model = cylinderModel(params)` constructs a parametric cylinder model from the 1-by-7 `params` input vector that describes a cylinder.

## Input Arguments

### **params** — cylinder parameters

1-by-7 scalar vector

Cylinder parameters, specified as a 1-by-7 scalar vector containing seven parameters  $[x1,y1,z1,x2,y2,z2,r]$  that describe a cylinder.

- $[x1,y1,z1]$  and  $[x2,y2,z2]$  are the centers of each end-cap surface of the cylinder.
- $r$  is the radius of the cylinder.

## Properties

These properties are read-only.

### **Parameters** — Cylinder model parameters

1-by-7 scalar vector

Cylinder model parameters, stored as a 1-by-7 scalar vector that describes a cylinder  $[x1,y1,z1,x2,y2,z2,r]$  that describe a cylinder.

- $[x1,y1,z1]$  and  $[x2,y2,z2]$  are the centers of each end-cap surface of the cylinder.
- $r$  is the radius of the cylinder.

### **Center — Center of cylinder**

1-by-3 vector

Center of cylinder, stored as a 1-by-3 vector.

### **Height — Height of cylinder**

scalar

Height of cylinder, stored as a scalar.

### **Radius — Radius of cylinder**

scalar

Radius of cylinder, stored as a scalar.

## Methods

plot

Plot cylinder in a figure window

## Examples

### **Detect Cylinder in Point Cloud**

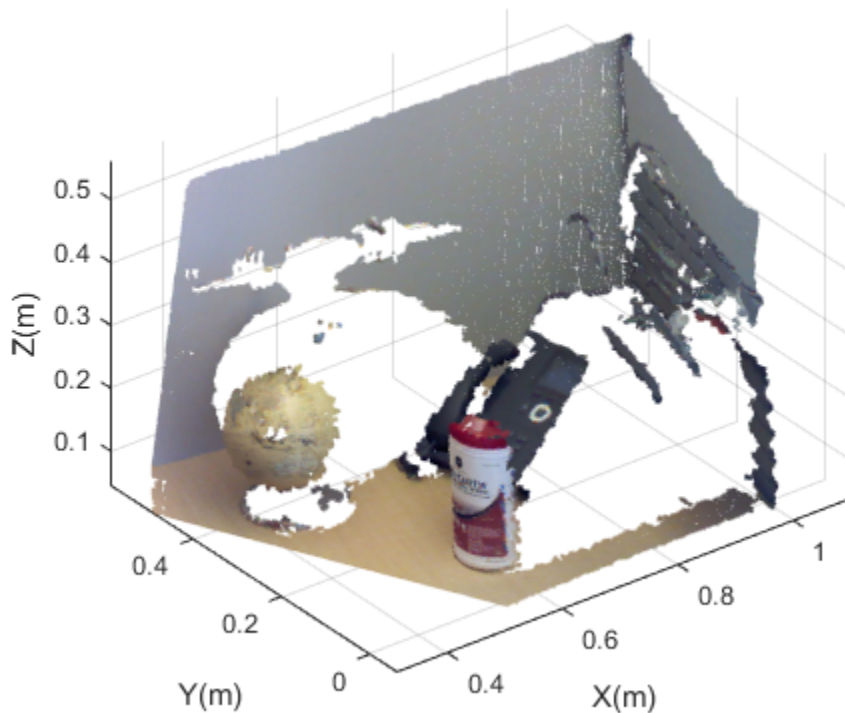
Load the point cloud.

```
load('object3d.mat');
```

Display point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a Cylinder in a Point Cloud')
```

## Detect a Cylinder in a Point Cloud



Set the maximum point-to-cylinder distance (5 mm) for the cylinder fitting.

```
maxDistance = 0.005;
```

Set the region of interest to constrain the search.

```
roi = [0.4,0.6;-inf,0.2;0.1,inf];  
sampleIndices = findPointsInROI(ptCloud,roi);
```

Set the orientation constraint.

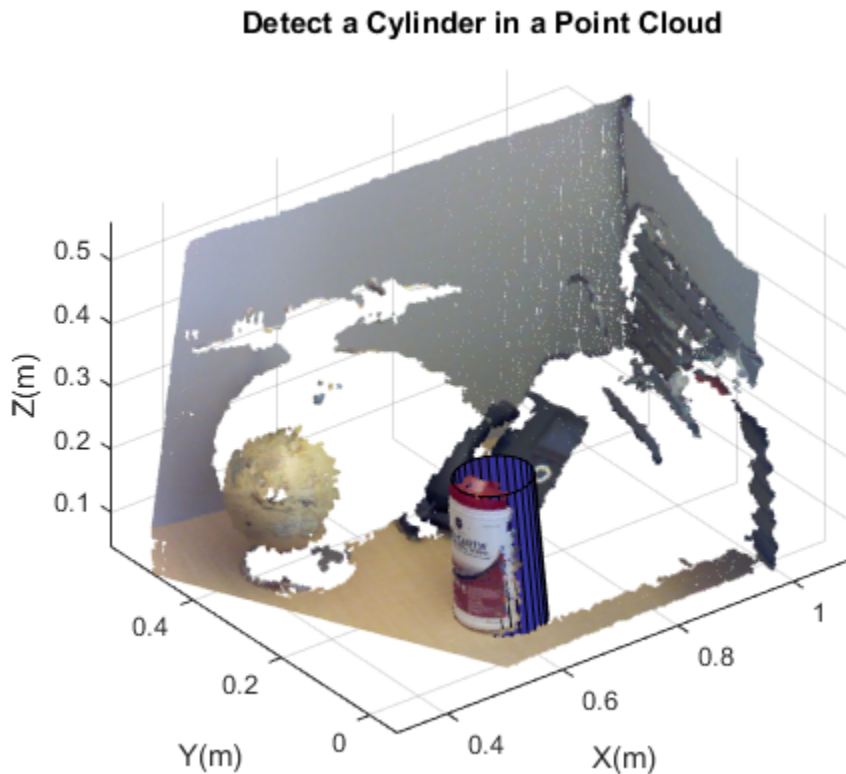
```
referenceVector = [0,0,1];
```

Detect the cylinder in the point cloud and extract it.

```
model = pcfitcylinder(ptCloud,maxDistance,referenceVector,'SampleIndices',sampleIndices)
```

Plot the cylinder.

```
hold on  
plot(model)
```



- “3-D Point Cloud Registration and Stitching”

### See Also

[pointCloud](#) | [planeModel](#) | [sphereModel](#) | [pcplayer](#) | [affine3d](#) | [pcdenoise](#) | [pcdownsample](#) | [pcfitcylinder](#) | [pcfitplane](#) | [pcfitsphere](#) | [pcmerge](#) | [pcread](#) | [pcregister](#) | [pcshow](#) | [pctransform](#) | [pcwrite](#)



**Introduced in R2015b**

# plot

**Class:** cylinderModel

Plot cylinder in a figure window

## Syntax

```
plot(model)
plot(model, 'Parent', ax)
```

## Description

`H = plot(model)` plots a cylinder within the axis limits of the current figure. `H` is the handle to `surf`, a 3-D shaded surface plot.

`H = plot(model, 'Parent', ax)` additionally specifies an output axes.

## Input Arguments

**model** — Parametric cylinder model

cylinder model

Parametric cylinder model returned by `cylinderModel`.

**'ax'** — Output axes

`gca` (default) | axes

Output axes, specified as the current axes for displaying the cylinder.

## Examples

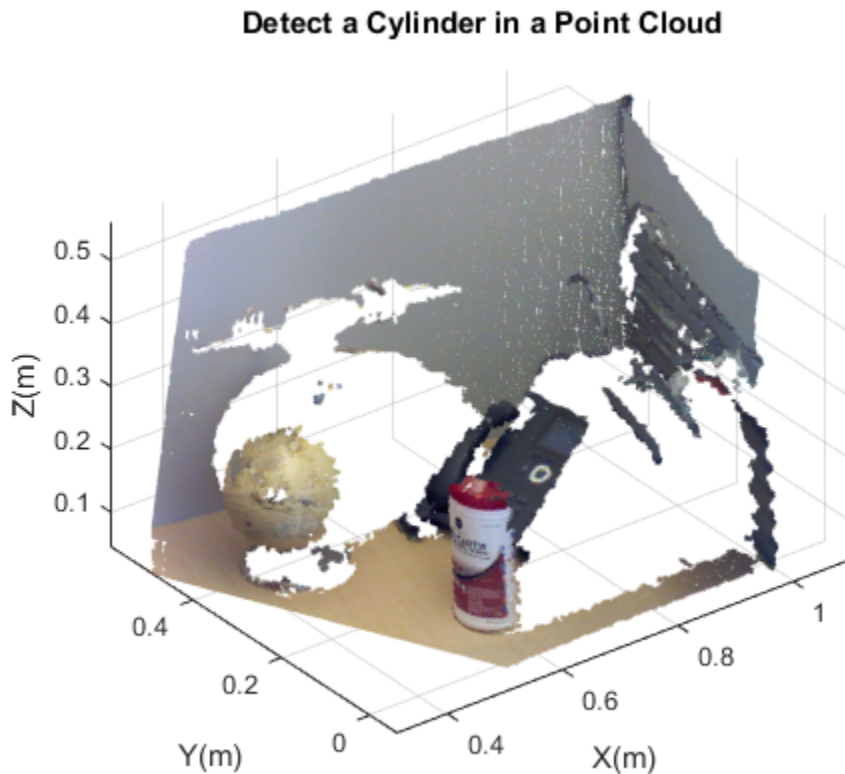
### Detect Cylinder in Point Cloud

Load the point cloud.

```
load('object3d.mat');
```

Display point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a Cylinder in a Point Cloud')
```



Set the maximum point-to-cylinder distance (5 mm) for the cylinder fitting.

```
maxDistance = 0.005;
```

Set the region of interest to constrain the search.

```
roi = [0.4,0.6;-inf,0.2;0.1,inf];
sampleIndices = findPointsInROI(ptCloud,roi);
```

Set the orientation constraint.

```
referenceVector = [0,0,1];
```

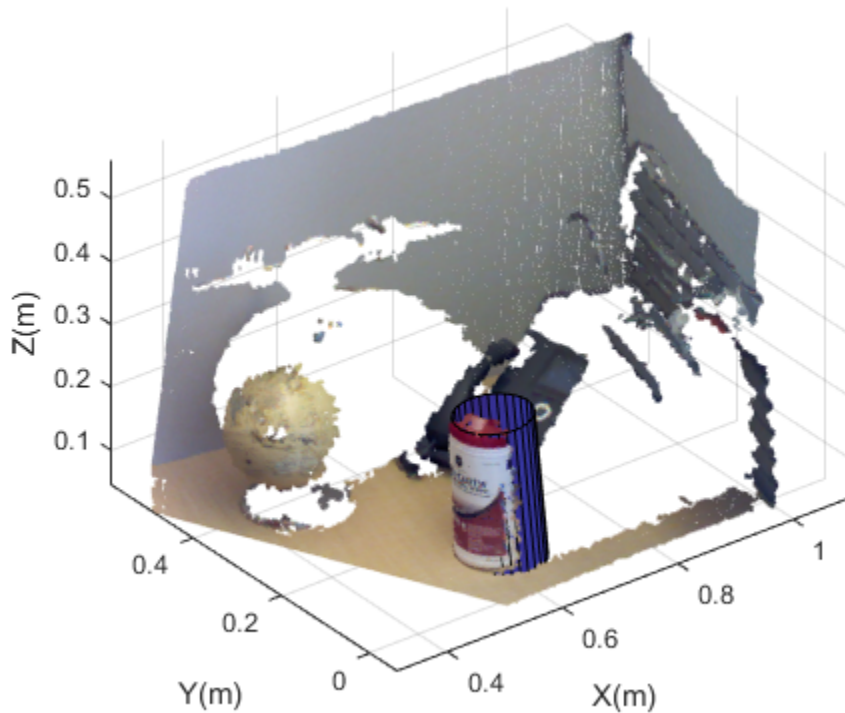
Detect the cylinder in the point cloud and extract it.

```
model = pcfitcylinder(ptCloud,maxDistance,referenceVector,'SampleIndices',sampleIndices)
```

Plot the cylinder.

```
hold on  
plot(model)
```

### Detect a Cylinder in a Point Cloud



### See Also

[pcfitcylinder](#) | [cylinderModel](#)

**Introduced in R2015b**

## planeModel class

Object for storing a parametric plane model

### Syntax

```
model = planeModel(params)
```

### Description

`model = planeModel(params)` constructs a parametric plane model from the 1-by-4 `params` input vector that describes a plane.

### Input Arguments

**params** — Plane parameters

1-by-4 scalar vector

Plane parameters, specified as a 1-by-4 vector. This input specifies the `Parameters` property. The four parameters  $[a,b,c,d]$  describe the equation for a plane:

$$ax + by + cz + d = 0$$

### Properties

These properties are read-only.

**Parameters** — Plane model parameters

1-by-4 vector

Plane model parameters, stored as a 1-by-4 vector. These parameters are specified by the `params` input argument.

### Normal — Normal vector of the plane

1-by-3 vector

Normal vector of the plane, stored as a 1-by-3 vector. The  $[a,b,c]$  vector specifies the unnormalized normal vector of the plane.

## Methods

plot Plot plane in a figure window

## Examples

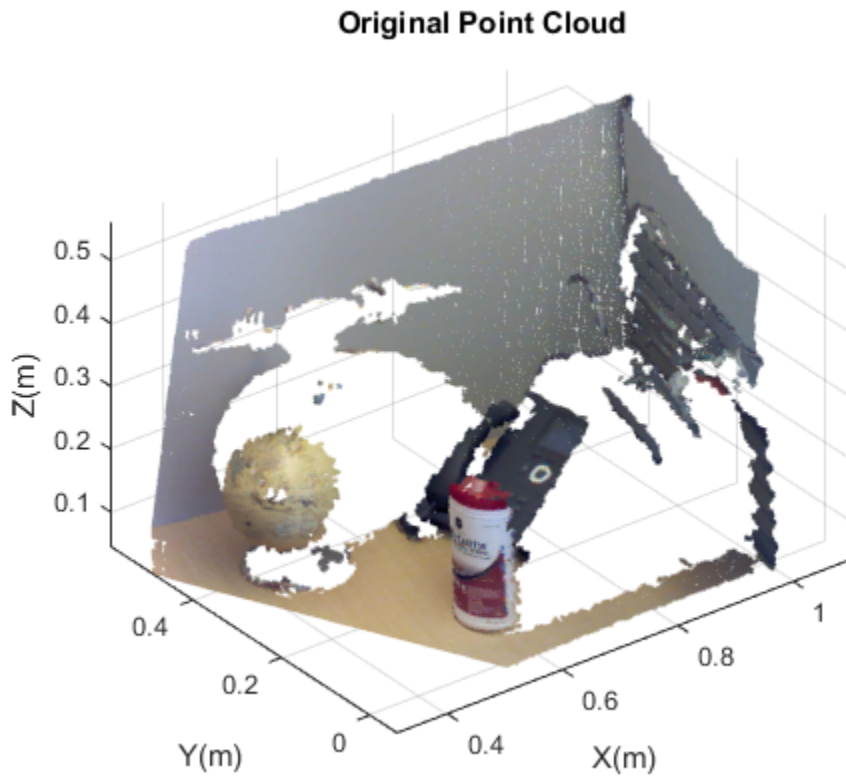
### Detect Multiple Planes from Point Cloud

Load the point cloud.

```
load('object3d.mat')
```

Display and label the point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Original Point Cloud')
```



Set the maximum point-to-plane distance (2cm) for plane fitting.

```
maxDistance = 0.02;
```

Set the normal vector of the plane.

```
referenceVector = [0,0,1];
```

Set the maximum angular distance to 5 degrees.

```
maxAngularDistance = 5;
```

Detect the first plane, the table, and extract it from the point cloud.

```
[model1,inlierIndices,outlierIndices] = pcfplane(ptCloud,maxDistance,referenceVector,
```



```
plane1 = select(ptCloud,inlierIndices);
remainPtCloud = select(ptCloud,outlierIndices);
```

Set the region of interest to constrain the search for the second plane, left wall.

```
roi = [-inf,inf;0.4,inf;-inf,inf];
sampleIndices = findPointsInROI(ptCloud,roi);
```

Detect the left wall and extract it from the remaining point cloud.

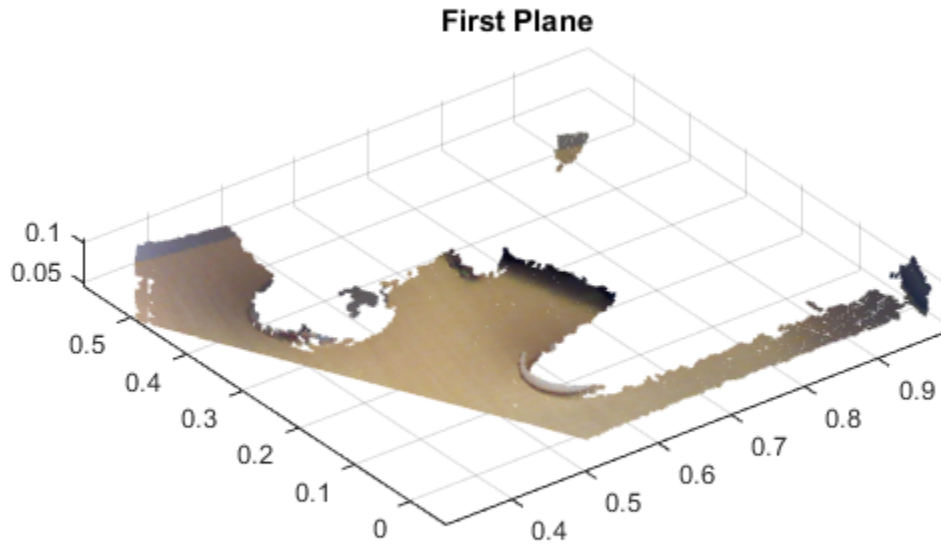
```
[model2,inlierIndices,outlierIndices] = pcfitplane(remainPtCloud,maxDistance, 'SampleIndices');
plane2 = select(remainPtCloud,inlierIndices);
remainPtCloud = select(remainPtCloud,outlierIndices);
```

Plot the two planes and the remaining points.

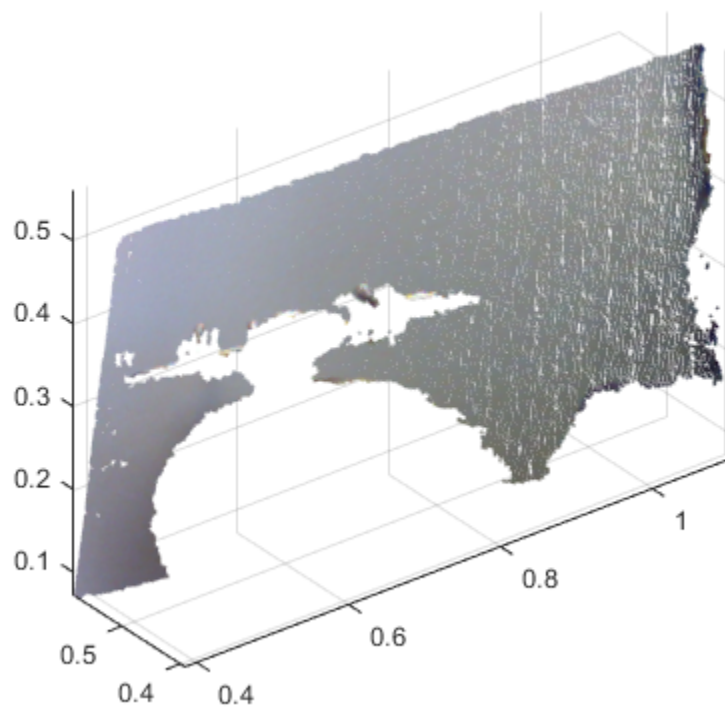
```
figure
pcshow(plane1)
title('First Plane')
```

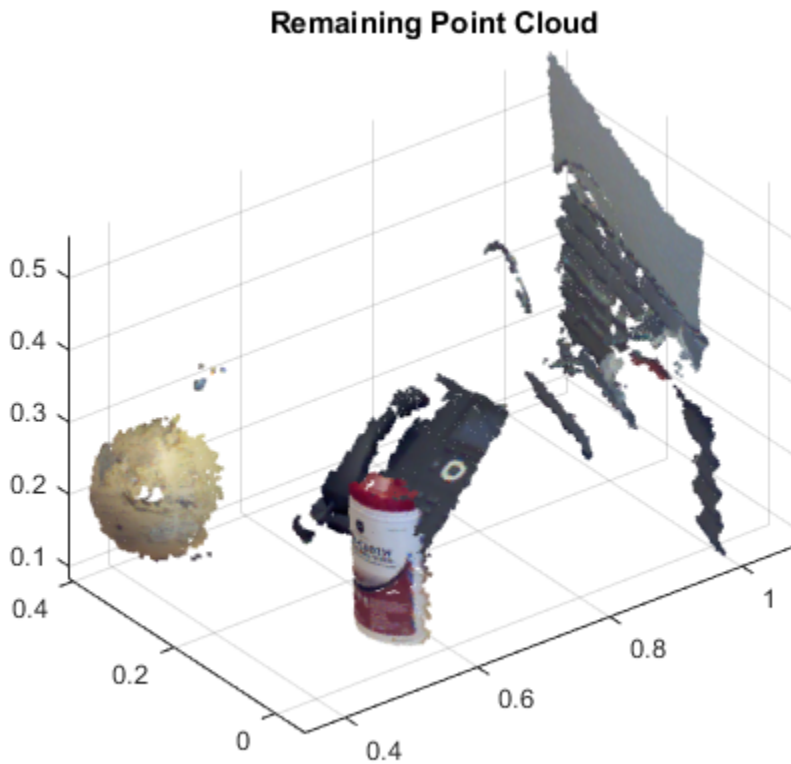
```
figure
pcshow(plane2)
title('Second Plane')
```

```
figure
pcshow(remainPtCloud)
title('Remaining Point Cloud')
```



### Second Plane





- “3-D Point Cloud Registration and Stitching”

**See Also**

`pointCloud` | `sphereModel` | `cylinderModel` | `pcplayer` | `affine3d` | `pcdenoise` | `pcdownsample` | `pcfitcylinder` | `pcfitplane` | `pcfitsphere` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcstransform` | `pcwrite`

**Introduced in R2015b**

# plot

**Class:** planeModel

Plot plane in a figure window

## Syntax

```
plot(model)
plot(model,Name,Value)
```

## Description

`H = plot(model)` plots a plane within the axis limits of the current figure. `H` is the handle to the patch.

`H = plot(model,Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**model** — Parametric plane model

plane model

Parametric plane model returned by planeModel.

## 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`.

**'Parent'** — Output axes

gca (default) | axes

Output axes, specified as the comma-separated pair of 'Parent' and the current axes for displaying the visualization.

### 'Color' — Color of the plane

'red' | color string | [R G B] vector

Color of the plane, specified as the comma-separated pair of 'Color' and either a color string or an [R G B] vector. The [R G B] values must be in the range of [0 1]. Supported color strings are listed in ColorSpec specifications.

## Examples

### Detect Plane in Point Cloud

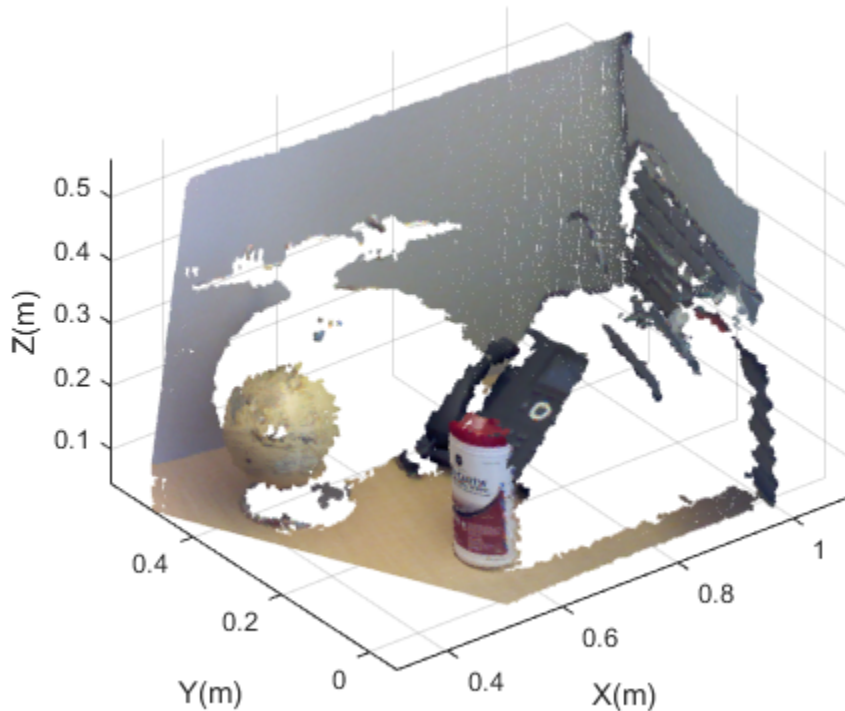
Load point cloud.

```
load('object3d.mat');
```

Display the point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a plane in a point cloud')
```

### Detect a plane in a point cloud



Set the maximum point-to-plane distance (2cm) for plane fitting.

```
maxDistance = 0.02;
```

Set the normal vector of a plane.

```
referenceVector = [0, 0, 1];
```

Set the maximum angular distance (5 degrees).

```
maxAngularDistance = 5;
```

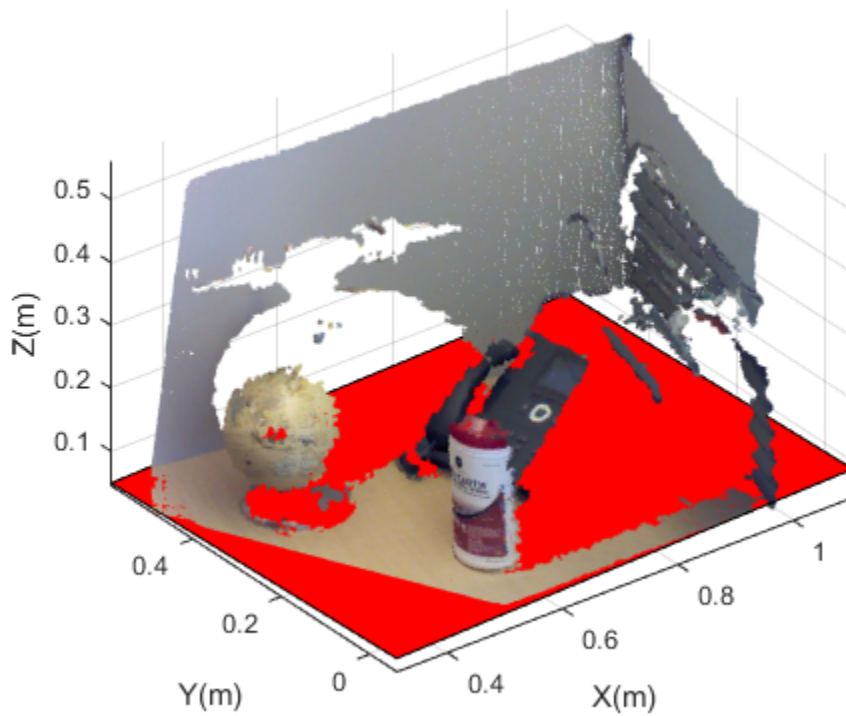
Detect the table in the point cloud and extract it.

```
model = pcfitplane(ptCloud,maxDistance,referenceVector,maxAngularDistance);
```

Plot the plane.

```
hold on  
plot(model)
```

### Detect a plane in a point cloud



### See Also

[pcfitplane](#) | [planeModel](#)

Introduced in R2015b



# sphereModel class

Object for storing a parametric sphere model

## Syntax

```
model = sphereModel(params)
```

## Description

`model = sphereModel(params)` constructs a parametric sphere model from the 1-by-4 `params` input vector that describes a sphere.

## Input Arguments

### **params** — Sphere parameters

1-by-4 scalar vector

Sphere parameters, specified as a 1-by-4 scalar vector. This input specifies the `Parameters` property. The four parameters  $[a,b,c,d]$  satisfy the equation for a sphere:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = d^2$$

## Properties

These properties are read-only.

### **Parameters** — Sphere model parameters

1-by-4 vector

Sphere model parameters, stored as a 1-by-4 vector. These parameters are specified by the `params` input argument. The four parameters  $[a,b,c,d]$  satisfy the equation for a sphere:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = d^2$$

**Center — Center of the sphere**

1-by-3 vector

Center of the sphere, stored as a 1-by-3 vector  $[xc,yc,zc]$  that specifies the center coordinates of the sphere.

**Radius — Radius of sphere**

scalar

Radius of sphere, stored as a scalar value.

## Methods

plot Plot sphere in a figure window

## Examples

**Detect Sphere in Point Cloud**

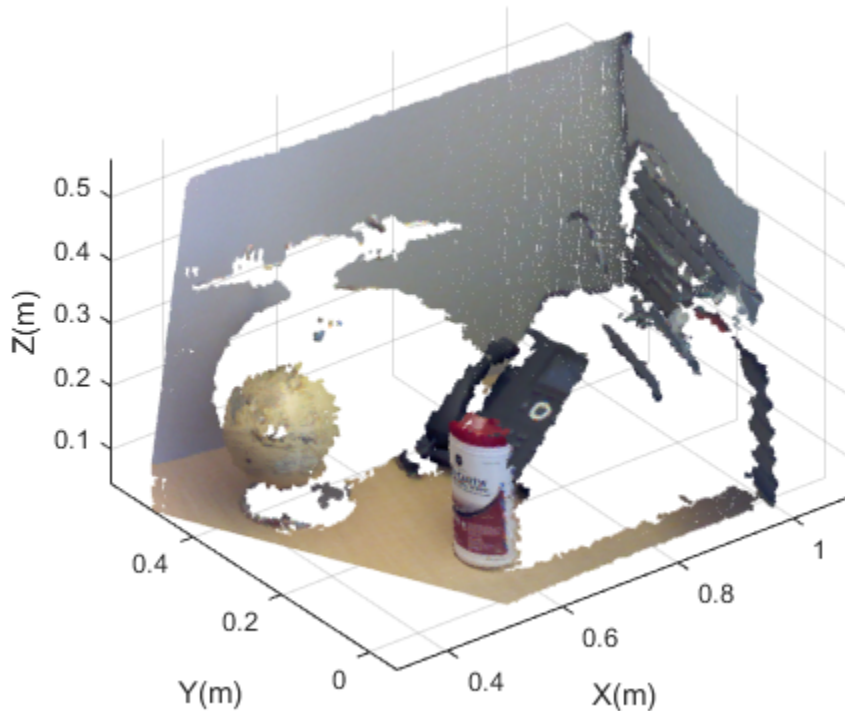
Load point cloud.

```
load('object3d.mat');
```

Display point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a sphere in a point cloud')
```

## Detect a sphere in a point cloud



Set the maximum point-to-sphere distance (1cm), for sphere fitting.

```
maxDistance = 0.01;
```

Set the region of interest to constrain the search.

```
roi = [-inf, 0.5; 0.2, 0.4; 0.1, inf];  
sampleIndices = findPointsInROI(ptCloud, roi);
```

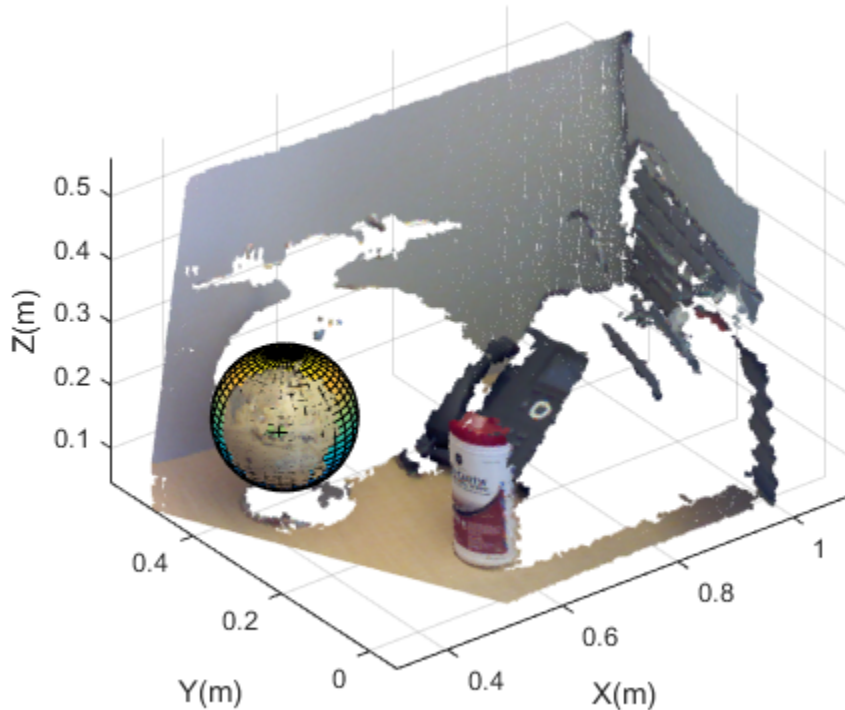
Detect the globe in the point cloud and extract it.

```
model = pcfitsphere(ptCloud, maxDistance, 'SampleIndices', sampleIndices);
```

Plot the sphere.

```
hold on  
plot(model)
```

### Detect a sphere in a point cloud



- “3-D Point Cloud Registration and Stitching”

### See Also

`pointCloud` | `planeModel` | `cylinderModel` | `pcplayer` | `affine3d` | `pcdenoise` |  
`pcdownsample` | `pcfitcylinder` | `pcfitplane` | `pcfitsphere` | `pcmerge` | `pcread` |  
`pcregister` | `pcshow` | `pctransform` | `pcwrite`

**Introduced in R2015b**

# plot

**Class:** sphereModel

Plot sphere in a figure window

## Syntax

```
plot(model)
plot(model, 'Parent', ax)
```

## Description

`H = plot(model)` plots a sphere in the current figure. `H` is the handle to `surf`, a 3-D shaded surface plot.

`H = plot(model, 'Parent', ax)` additionally allows you to specify an output axes.

## Input Arguments

**model** — Parametric sphere model

sphere model

Parametric sphere model returned by `sphereModel`.

**'Parent'** — Output axes

`gca` (default) | axes

Output axes, specified as the comma-separated pair of `'Parent'` and the current axes for displaying the visualization.

## Examples

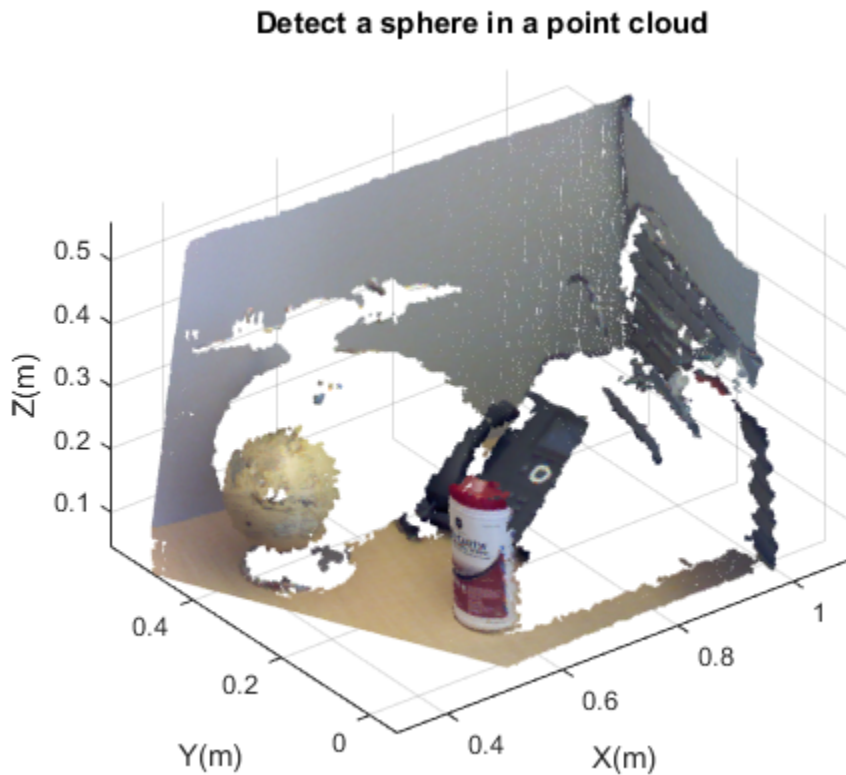
**Detect Sphere in Point Cloud**

Load point cloud.

```
load('object3d.mat');
```

Display point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a sphere in a point cloud')
```



Set the maximum point-to-sphere distance (1cm), for sphere fitting.

```
maxDistance = 0.01;
```

Set the region of interest to constrain the search.

```
roi = [-inf, 0.5; 0.2, 0.4; 0.1, inf];  
sampleIndices = findPointsInROI(ptCloud, roi);
```

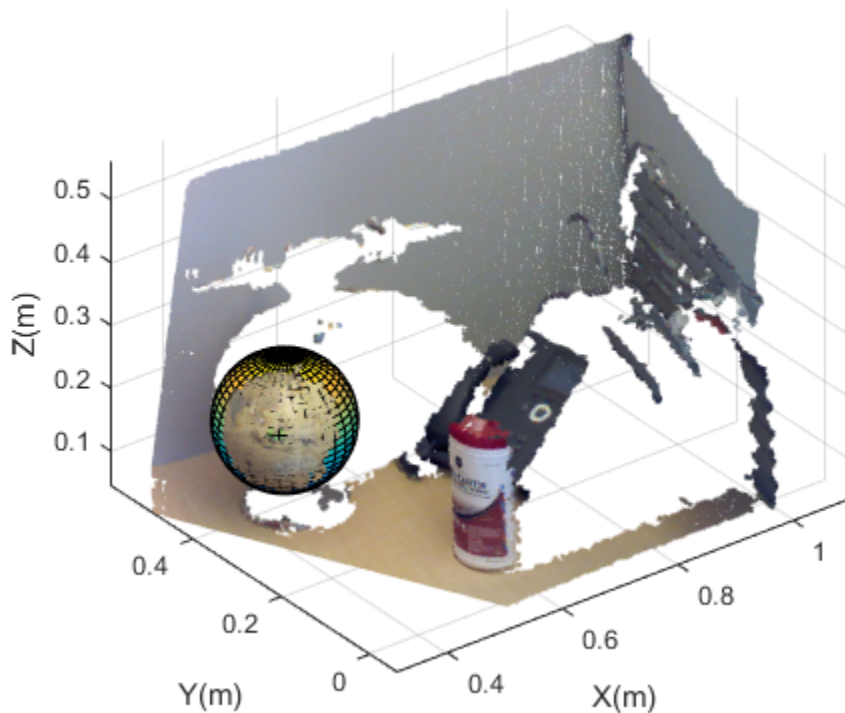
Detect the globe in the point cloud and extract it.

```
model = pcfitsphere(ptCloud, maxDistance, 'SampleIndices', sampleIndices);
```

Plot the sphere.

```
hold on  
plot(model)
```

### Detect a sphere in a point cloud



### See Also

[pcfitsphere](#) | [sphereModel](#)

**Introduced in R2015b**



# opticalFlow class

Object for storing optical flow matrices

## Description

---

**Note:** The `vision.opticalFlow` System object will be removed in a future release. Use `opticalFlowHS`, `opticalFlowLKDoG`, `opticalFlowLK`, or `opticalFlowFarneback` with equivalent functionality instead.

---

The `OpticalFlow` object stores the direction and speed of a moving object from one image or video frame to another.

## Construction

`flow = opticalFlow(Vx,Vy)` constructs an optical flow object from two equal-sized matrices, `Vx` and `Vy`, which are the  $x$  and  $y$  components of velocity.

`flow = opticalFlow( ___,Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

Code Generation Support
Supports Code Generation: Yes
Supports MATLAB Function block: Yes
“Code Generation Support, Usage Notes, and Limitations”.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: 'Vx', randn(100, 100)

### **'Vx' — X component of velocity**

*M*-by-*N* matrix

X component of velocity, specified as the comma-separated pair consisting of 'Vx' and an *M*-by-*N* matrix.

### **'Vy' — Y component of velocity**

*M*-by-*N* matrix

Y component of velocity, specified as the comma-separated pair consisting of 'Vy' and an *M*-by-*N* matrix.

### **'Orientation' — Phase angles of optical flow**

*M*-by-*N* matrix

Phase angles of optical flow in radians, specified as the comma-separated pair consisting of 'Orientation' and an *M*-by-*N* matrix of size *Vx* or *Vy*.

### **'Magnitude' — Magnitude of optical flow**

*M*-by-*N* matrix

Magnitude of optical flow, specified as the comma-separated pair consisting of 'Magnitude' and an *M*-by-*N* matrix.

## Methods

plot Plot optical flow

## Examples

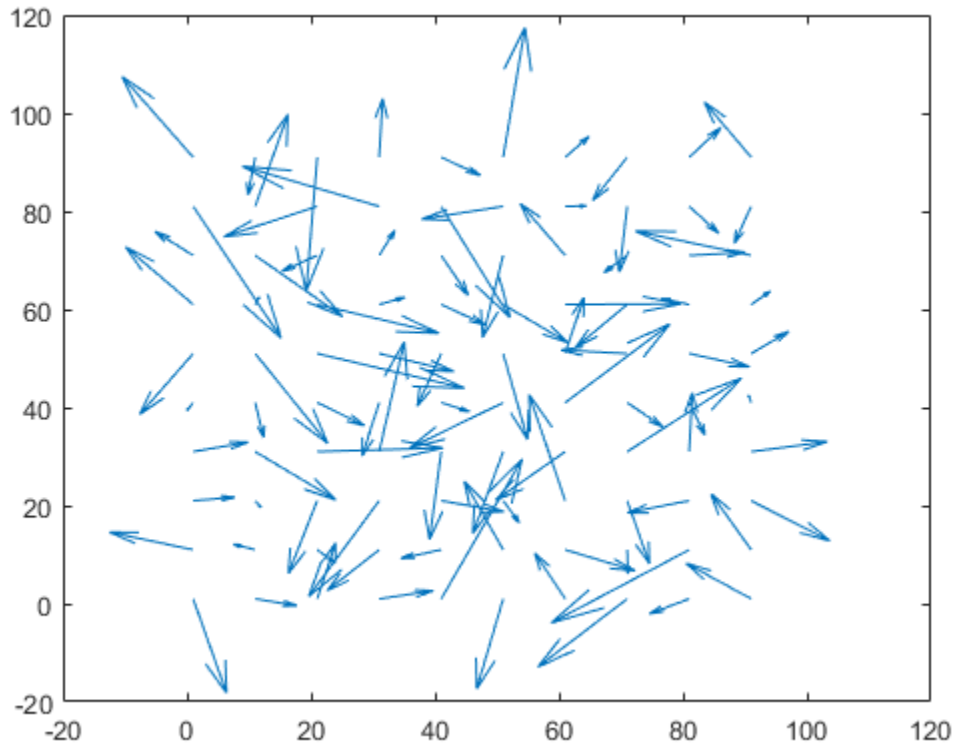
### **Construct Optical Flow Object and Plot Its Velocity**

Construct an optical flow object.

```
opflow = opticalFlow(randn(100,100),randn(100,100));
```

Plot the velocity of the object as a quiver plot.

```
plot(opflow, 'DecimationFactor', [10 10], 'ScaleFactor', 10);
```



## See Also

[quiver](#) | [opticalFlowHS](#) | [opticalFlowLK](#) | [opticalFlowFarneback](#) | [opticalFlowLKDoG](#)

**Introduced in R2015a**

# plot

**Class:** opticalFlow

Plot optical flow

## Syntax

```
plot(flow)
plot(flow, Name, Value)
```

## Description

`plot(flow)` plots the optical flow vectors.

`plot(flow, Name, Value)` includes additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**flow** — Object for storing optical flow matrices

opticalFlow object

Object for storing optical flow velocity matrices, specified as an opticalFlow 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`.

**'DecimationFactor'** — Decimation factor of velocity vectors

[1 1] | 2-element vector

Decimation factor of velocity vectors, specified as the comma-separated pair consisting of 'DecimationFactor' and a 2-element vector. The 2-element vector, [*XDecimFactor* *YDecimFactor*], specifies the decimation factor of velocity vectors along the *x* and *y* directions. *XDecimFactor* and *YDecimFactor* are positive scalar integers. Increase the value of this property to get a less cluttered quiver plot.

**'ScaleFactor' — Scaling factor for velocity vector display**

1 | positive integer-valued scalar

Scaling factor for velocity vector display, specified as the comma-separated pair consisting of 'ScaleFactor' and a positive integer-valued scalar. Increase this value to display longer vectors.

**'Parent' — Display axes**

gca handle

Display axes, specified as the comma-separated pair consisting of 'Parent' and the handle to an axes.

## Examples

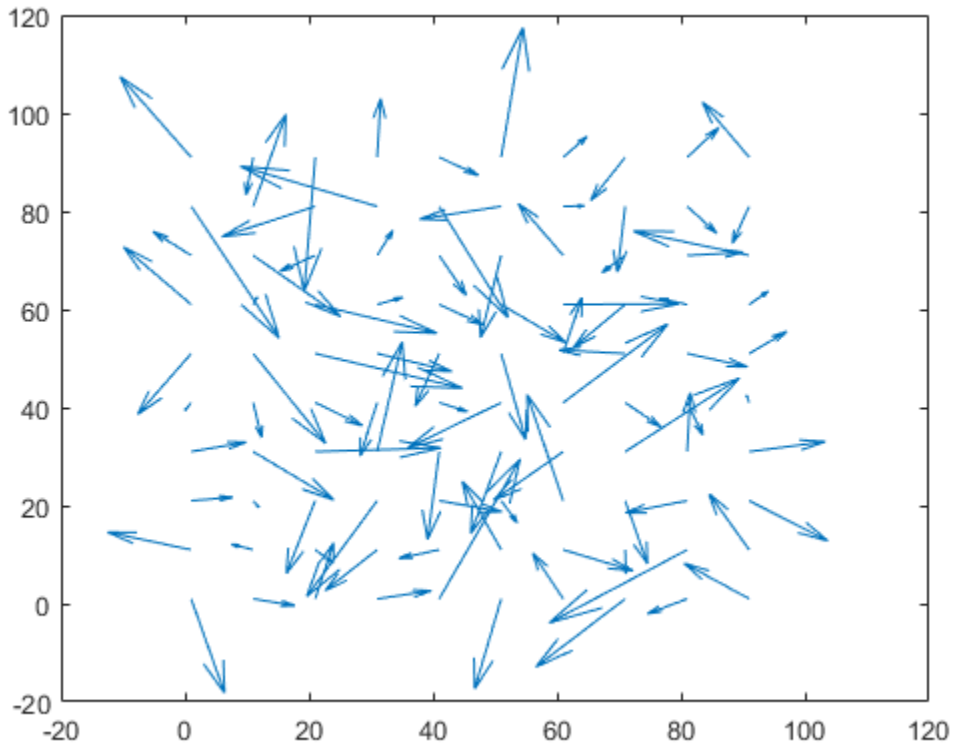
### Construct Optical Flow Object and Plot Its Velocity

Construct an optical flow object.

```
opflow = opticalFlow(randn(100,100),randn(100,100));
```

Plot the velocity of the object as a quiver plot.

```
plot(opflow,'DecimationFactor',[10 10],'ScaleFactor',10);
```



### See Also

`quiver` | `opticalFlow`

Introduced in R2015a

# opticalFlowHS class

Estimate optical flow using Horn-Schunck method

## Description

Estimate the direction and speed of a moving object from one image or video frame to another using the Horn-Schunck method.

## Construction

`opticFlow = opticalFlowHS` returns an optical flow object that is used to estimate the direction and speed of an object's motion. `opticalFlowHS` uses the Horn-Schunck algorithm.

`opticFlow = opticalFlowHS(Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

Code Generation Support
Supports Code Generation: Yes
Supports MATLAB Function block: No
“Code Generation Support, Usage Notes, and Limitations”.

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'MaxIteration',10`

**'Smoothness'** — Expected smoothness

1 (default) | positive scalar

Expected smoothness of optical flow, specified as the comma-separated pair consisting of 'Smoothness' and a positive scalar. Increase this value when there is increased motion between consecutive frames. A typical value for 'Smoothness' is around 1.

### 'MaxIteration' — Maximum number of iterations

10 (default) | positive integer-valued scalar

Maximum number of iterations to perform in the optical flow iterative solution, specified as the comma-separated pair consisting of 'MaxIteration' and a positive integer-valued scalar. Increase this value to estimate objects with low velocity.

Computation stops when the number of iterations equals `MaxIteration` or when the algorithm reaches the value set by `VelocityDifference`. To use only `MaxIteration` to stop computation, set `VelocityDifference` to 0.

### 'VelocityDifference' — Minimum absolute velocity difference

0 (default) | positive scalar

Minimum absolute velocity difference at which to stop iterative computation, specified as the comma-separated pair consisting of 'VelocityDifference' and a positive scalar. This value depends on the input data type. Decrease this value to estimate the optical flow of objects that have low velocity.

Computation stops when the number of iterations equals `MaxIteration` or when the algorithm reaches the value set by `VelocityDifference`. To only use `MaxIteration` to stop computation, set `VelocityDifference` to 0. To use only `VelocityDifference`, set `MaxIteration` to `Inf`

## Methods

`reset`

Reset the internal state of the object

`estimateFlow`

Estimate optical flow

## Examples

### Compute Optical Flow Using Horn-Schunck Method

Read in a video file.



```
vidReader = VideoReader('viptraffic.avi');  
  
Create optical flow object.  
opticFlow = opticalFlowHS;  
  
Estimate the optical flow of objects in the video.  
while hasFrame(vidReader)  
    frameRGB = readFrame(vidReader);  
    frameGray = rgb2gray(frameRGB);  
  
    flow = estimateFlow(opticFlow,frameGray);  
  
    imshow(frameRGB)  
    hold on  
    plot(flow, 'DecimationFactor',[5 5], 'ScaleFactor',25)  
    hold off  
end
```



## Algorithms

To compute the optical flow between two images, you must solve the following optical flow constraint equation:

$$I_x u + I_y v + I_t = 0$$

- $I_x$ ,  $I_y$ , and  $I_t$  are the spatiotemporal image brightness derivatives.
- $u$  is the horizontal optical flow.
- $v$  is the vertical optical flow.

## Horn-Schunck Method

By assuming that the optical flow is smooth over the entire image, the Horn-Schunck method computes an estimate of the velocity field,  $[u \ v]^T$ , that minimizes this equation:

$$E = \iint (I_x u + I_y v + I_t)^2 dx dy + \alpha \iint \left\{ \left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 \right\} dx dy$$

In this equation,  $\frac{\partial u}{\partial x}$  and  $\frac{\partial u}{\partial y}$  are the spatial derivatives of the optical velocity component,  $u$ , and  $\alpha$  scales the global smoothness term. The Horn-Schunck method minimizes the previous equation to obtain the velocity field,  $[u \ v]$ , for each pixel in the image. This method is given by the following equations:

$$u_{x,y}^{k+1} = u_{x,y}^{-k} - \frac{I_x [I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

$$v_{x,y}^{k+1} = v_{x,y}^{-k} - \frac{I_y [I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

In these equations,  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$  is the velocity estimate for the pixel at  $(x,y)$ , and

$\begin{bmatrix} \bar{u}_{x,y}^{-k} & \bar{v}_{x,y}^{-k} \end{bmatrix}$  is the neighborhood average of  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$ . For  $k = 0$ , the initial velocity is 0.

To solve  $u$  and  $v$  using the Horn-Schunck method:

- 1 Compute  $I_x$  and  $I_y$  using the Sobel convolution kernel,  $[-1 \ -2 \ -1; 0 \ 0 \ 0; 1 \ 2 \ 1]$ , and its transposed form, for each pixel in the first image.
- 2 Compute  $I_t$  between images 1 and 2 using the  $[-1 \ 1]$  kernel.
- 3 Assume the previous velocity to be 0, and compute the average velocity for each pixel using  $[0 \ 1 \ 0; 1 \ 0 \ 1; 0 \ 1 \ 0]$  as a convolution kernel.
- 4 Iteratively solve for  $u$  and  $v$ .

## References

- [1] Barron, J. L., D. J. Fleet, S. S. Beauchemin, and T. A. Burkitt. "Performance of optical flow techniques". *CVPR*, 1992.

## See Also

[opticalFlowLKDoG](#) | [opticalFlowLK](#) | [quiver](#) | [opticalFlow](#) | [opticalFlowFarneback](#)

**Introduced in R2015a**

## **reset**

**Class:** opticalFlowHS

Reset the internal state of the object

## **Syntax**

```
reset(opticFlow)
```

## **Description**

`reset(opticFlow)` resets the internal state of the optical flow object. Doing this sets the previous frame to black.

**Introduced in R2015a**

# estimateFlow

**Class:** opticalFlowHS

Estimate optical flow

## Syntax

```
flow = estimateFlow(opticFlow,I)
```

## Description

`flow = estimateFlow(opticFlow,I)` estimates the optical flow of the input image, `I`, with respect to the previous image. The function sets the previous image for the first run to a black image. After the first run, each new image, `I`, becomes the current image, and the image from the last run becomes the previous one. You can also use flow estimation to predict a future position of a moving object for image reconstruction.

## Input Arguments

**opticFlow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, specified as an opticalFlow object.

**I** — Input image

grayscale image

Input image, specified as a grayscale image. To process the first frame, the function sets the previous frame to black.

## Output Arguments

**flow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, returned as an opticalFlow object.

# Examples

## Compute Optical Flow Using Horn-Schunck Method

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowHS;
```

Estimate the optical flow of objects in the video.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25)
    hold off
end
```



**Introduced in R2015a**

# opticalFlowLK class

Estimate optical flow using Lucas-Kanade method

## Description

Estimate the direction and speed of a moving object from one image or video frame to another using the Lucas-Kanade method.

## Construction

`opticFlow = opticalFlowLK` returns an optical flow object used to estimate the direction and speed of an object's motion. `opticalFlowLK` uses the Lucas-Kanade algorithm and a difference filter,  $[-1 \ 1]$ , for temporal smoothing.

`opticFlow = opticalFlowLK(Name, Value)` includes additional options specified by one or more `Name, Value` pair arguments.

Code Generation Support
Supports Code Generation: Yes
Supports MATLAB Function block: No
“Code Generation Support, Usage Notes, and Limitations”.

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'NoiseThreshold', 0.0039`

### 'NoiseThreshold' — Threshold for noise reduction

0.0039 (default) | positive scalar

Threshold for noise reduction, specified as the comma-separated pair consisting of 'NoiseThreshold' and a positive scalar. The more you increase this number, the less an object's movement affects the optical flow calculation.

## Methods

<code>estimateFlow</code>	Estimate optical flow
<code>reset</code>	Reset the internal state of the object

## Examples

### Compute Optical Flow Using Lucas-Kanade Algorithm

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowLK('NoiseThreshold',0.009);
```

Estimate and display the optical flow of objects in the video.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',10)
    hold off
end
```





## Algorithms

To compute the optical flow between two images, you must solve the following optical flow constraint equation:

$$I_x u + I_y v + I_t = 0$$

.

- $I_x$ ,  $I_y$ , and  $I_t$  are the spatiotemporal image brightness derivatives.
- $u$  is the horizontal optical flow.
- $v$  is the vertical optical flow.

## Lucas-Kanade Method

To solve the optical flow constraint equation for  $u$  and  $v$ , the Lucas-Kanade method divides the original image into smaller sections and assumes a constant velocity in each section. Then, it performs a weighted least-square fit of the optical flow constraint equation to a constant model for  $[u \ v]^T$  in each section  $\Omega$ . The method achieves this fit by minimizing the following equation:

$$\sum_{x \in \Omega} W^2 [I_x u + I_y v + I_t]^2$$

$W$  is a window function that emphasizes the constraints at the center of each section. The solution to the minimization problem is

$$\begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 I_x I_t \\ \sum W^2 I_y I_t \end{bmatrix}$$

The Lucas-Kanade method computes  $I_t$  using a difference filter,  $[-1 \ 1]$ .

$u$  and  $v$  are solved as follows:

- 1 Compute  $I_x$  and  $I_y$  using the kernel  $[-1 \ 8 \ 0 \ -8 \ 1]/12$  and its transposed form.
- 2 Compute  $I_t$  between images 1 and 2 using the  $[-1 \ 1]$  kernel.
- 3 Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a separable and isotropic 5-by-5 element kernel whose effective 1-D coefficients are  $[1 \ 4 \ 6 \ 4 \ 1]/16$ .
- 4 Solve the 2-by-2 linear equations for each pixel using the following method:

$$\bullet \text{ If } A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$$

$$\text{Then the eigenvalues of } A \text{ are } \lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$$

- The eigenvalues are compared to the threshold,  $\tau$ , that corresponds to the value you enter for the threshold for noise reduction. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

A is nonsingular, the system of equations are solved using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

A is singular (noninvertible), the gradient flow is normalized to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.

## References

- [1] Barron, J. L., D. J. Fleet, S. S. Beauchemin, and T. A. Burkitt. "Performance of optical flow techniques". *CVPR*, 1992.

## See Also

[opticalFlowHS](#) | [opticalFlowLKDoG](#) | [quiver](#) | [opticalFlow](#) | [opticalFlowFarneback](#)

**Introduced in R2015a**

# estimateFlow

**Class:** opticalFlowHS

Estimate optical flow

## Syntax

```
flow = estimateFlow(opticFlow,I)
```

## Description

`flow = estimateFlow(opticFlow,I)` estimates the optical flow of the input image, `I`, with respect to the previous image. The function sets the previous image for the first run to a black image. After the first run, each new image, `I`, becomes the current image, and the image from the last run becomes the previous one. You can also use flow estimation to predict a future position of a moving object for image reconstruction.

## Input Arguments

**opticFlow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, specified as an opticalFlow object.

**I** — Input image

grayscale image

Input image, specified as a grayscale image. To process the first frame, the function sets the previous frame to black.

## Output Arguments

**f1ow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, returned as an opticalFlow object.

## Examples

### Compute Optical Flow Using Lucas-Kanade Algorithm

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowLK('NoiseThreshold',0.009);
```

Estimate and display the optical flow of objects in the video.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',10)
    hold off
end
```



Introduced in R2015a

## **reset**

**Class:** opticalFlowLK

Reset the internal state of the object

## **Syntax**

```
reset(opticFlow)
```

## **Description**

`reset(opticFlow)` resets the internal state of the optical flow object. Doing this sets the previous frame to black.

**Introduced in R2015a**

# opticalFlowLKDoG class

Estimate optical flow using Lucas-Kanade derivative of Gaussian method

## Description

Estimate the direction and speed of a moving object from one image or video frame to another using the Lucas-Kanade derivative of Gaussian (DoG) method.

## Construction

`opticFlow = opticalFlowLKDoG` returns an optical flow object that is used to estimate the direction and speed of an object's motion. `opticalFlowLKDoG` uses the Lucas-Kanade method and a derivative of Gaussian (DoG) filter for temporal smoothing.

`opticFlow = opticalFlowLKDoG(Name, Value)` includes additional options specified by one or more `Name, Value` pair arguments.

Code Generation Support
Supports Code Generation: Yes
Supports MATLAB Function block: No
“Code Generation Support, Usage Notes, and Limitations”.

## Input Arguments

### Name-Value Pair Arguments

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

Example: `'NumFrames', 3`

**'NumFrames'** — Number of buffered frames  
3 (default) | positive integer-valued scalar

Number of buffered frames for temporal smoothing, specified as the comma-separated pair consisting of 'NumFrames' and a positive integer-valued scalar. The more you increase this number, the less an object's abrupt movement affects the optical flow calculation. The amount of delay in flow estimation depends on the value of NumFrames. The output flow corresponds to the image at  $t_{flow} = (t_{current} - (NumFrames - 1) / 2)$ , where  $t_{current}$  is the time of the current image.

**'ImageFilterSigma' — Standard deviation for image smoothing filter**

1.5 | positive scalar

Standard deviation for image smoothing filter, specified as the comma-separated pair consisting of 'ImageFilterSigma' and a positive scalar.

**'GradientFilterSigma' — Standard deviation for gradient smoothing filter**

1 | positive scalar

Standard deviation for gradient smoothing filter, specified as the comma-separated pair consisting of 'GradientFilterSigma' and a positive scalar.

**'NoiseThreshold' — Threshold for noise reduction**

0.0039 (default) | positive scalar

Threshold for noise reduction, specified as the comma-separated pair consisting of 'NoiseThreshold' and a positive scalar. As you increase this number, the less an object's movement impacts the optical flow calculation.

## Methods

estimateFlow

Estimate optical flow

reset

Reset the internal state of the object

## Examples

### Compute Optical Flow Using Lucas-Kanade derivative of Gaussian

Read in a video file.



```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowLKDoG('NumFrames',3);
```

Estimate the optical flow of the objects and display the flow vectors.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB);
    hold on;
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25);
    hold off;
end
```



## Algorithms

To compute the optical flow between two images, you must solve the following optical flow constraint equation:

$$I_x u + I_y v + I_t = 0$$

- $I_x$ ,  $I_y$ , and  $I_t$  are the spatiotemporal image brightness derivatives.
- $u$  is the horizontal optical flow.
- $v$  is the vertical optical flow.

## Lucas-Kanade Derivative of Gaussian Method

To solve the optical flow constraint equation for  $u$  and  $v$ , the Lucas-Kanade method divides the original image into smaller sections and assumes a constant velocity in each section. Then, it performs a weighted least-square fit of the optical flow constraint equation to a constant model for  $[u \ v]^T$  in each section  $\Omega$ . The method achieves this fit by minimizing the following equation:

$$\sum_{x \in \Omega} W^2 [I_x u + I_y v + I_t]^2$$

$W$  is a window function that emphasizes the constraints at the center of each section. The solution to the minimization problem is

$$\begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 I_x I_t \\ \sum W^2 I_y I_t \end{bmatrix}$$

The Lucas-Kanade method computes  $I_t$  using a derivative of Gaussian filter.

To solve the optical flow constraint equation for  $u$  and  $v$ :

- 1 Compute  $I_x$  and  $I_y$  using the following steps:
  - a Use a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the `NumFrames` property.

- b** Use a Gaussian filter and the derivative of a Gaussian filter to smooth the image using spatial filtering. Specify the standard deviation and length of the image smoothing filter using the `ImageFilterSigma` property.
- 2** Compute  $I_t$  between images 1 and 2 using the following steps:
- a** Use the derivative of a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the `NumFrames` property.
- b** Use the filter described in step 1b to perform spatial filtering on the output of the temporal filter.
- 3** Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a gradient smoothing filter. Use the `GradientFilterSigma` property to specify the standard deviation and the number of filter coefficients for the gradient smoothing filter.
- 4** Solve the 2-by-2 linear equations for each pixel using the following method:

$$\bullet \text{ If } A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$$

Then the eigenvalues of A are  $\lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$

- When the algorithm finds the eigenvalues, it compares them to the threshold,  $\tau$ , that corresponds to the value you enter for the `NoiseThreshold` property. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

A is nonsingular, so the algorithm solves the system of equations using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

A is singular (noninvertible), so the algorithm normalizes the gradient flow to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.

### References

- [1] Barron, J.L., D.J. Fleet, S.S. Beauchemin, and T.A. Burkitt. *Performance of optical flow techniques*. CVPR, 1992.

### See Also

`opticalFlowHS` | `opticalFlowLK` | `quiver` | `opticalFlow` | `opticalFlowFarneback`

**Introduced in R2015a**

# estimateFlow

**Class:** opticalFlowLKDoG

Estimate optical flow

## Syntax

```
flow = estimateFlow(opticFlow,I)
```

## Description

`flow = estimateFlow(opticFlow,I)` estimates optical flow using the current image, `I`, and the previous images. The function sets the previous images for the first run to a black image. After the first run, each new image, `I`, becomes the current image, and the image from the last run becomes the previous one. You can also use flow estimation to predict a future position of a moving object for image reconstruction.

The amount of delay in flow estimation depends on the value of `NumFrames`. The output flow corresponds to the image at  $t_{flow} = (t_{current} - (NumFrames - 1) / 2)$ , where  $t_{current}$  is the time of the current image.

## Input Arguments

**opticFlow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, specified as an opticalFlow object.

**I** — Input image

grayscale image

Input image, specified as a grayscale image. To process the first frame, the function sets the previous frame to black.

# Output Arguments

### **flow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, returned as an opticalFlow object.

# Examples

### **Compute Optical Flow Using Lucas-Kanade derivative of Gaussian**

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowLKDoG('NumFrames',3);
```

Estimate the optical flow of the objects and display the flow vectors.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB);
    hold on;
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25);
    hold off;
end
```



**Introduced in R2015a**

### **reset**

**Class:** `opticalFlowLKDoG`

Reset the internal state of the object

### **Syntax**

```
reset(opticFlow)
```

### **Description**

`reset(opticFlow)` resets the internal state of the optical flow object. Doing this sets the previous frame to black.

**Introduced in R2015a**



# opticalFlowFarneback class

Estimate optical flow using Farneback method

## Description

Estimate the direction and speed of a moving object from one image or video frame to another using the Farneback method.

## Construction

`opticFlow = opticalFlowFarneback` returns an optical flow object that you can use to estimate the direction and speed of an object's motion. `estimateFlow` method of this class uses the Farneback algorithm to estimate the optical flow.

`opticFlow = opticalFlowFarneback(Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

Code Generation Support
Supports Code Generation: Yes
Supports MATLAB Function block: No
Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries.
“Code Generation Support, Usage Notes, and Limitations”.

## Input Arguments

### Name-Value Pair Arguments

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

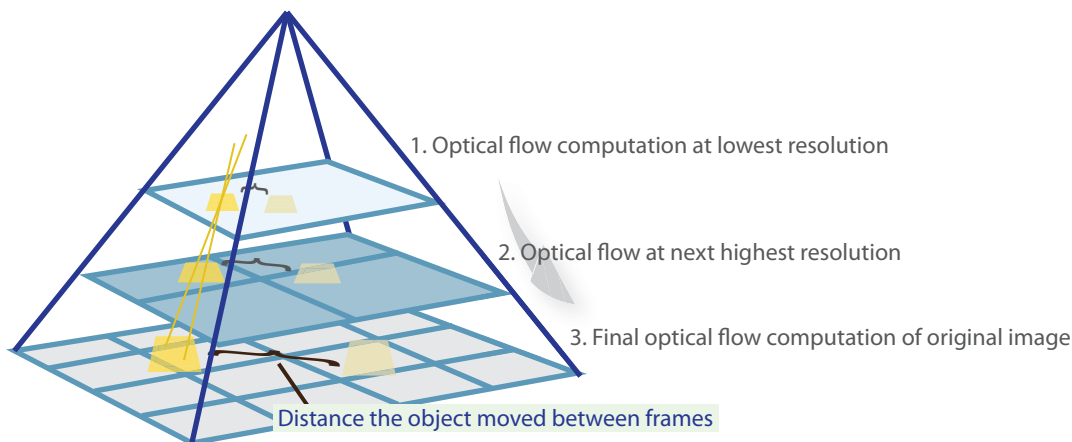
Example: 'NumPyramidLevels',3

### 'NumPyramidLevels' – Number of pyramid layers

3 (default) | positive scalar

Number of pyramid layers, specified as the comma-separated pair consisting of 'NumPyramidLevels' and a positive scalar. The value includes the initial image as one of the layers. When you set this value to 1, `opticalFlowFarneback` uses the original image only. It does not add any pyramid layers.

The `opticalFlowFarneback` algorithm generates an image pyramid, where each level has a lower resolution compared to the previous level. When you select a pyramid level greater than 1, the algorithm can track the points at multiple levels of resolution, starting at the lowest level. Increasing the number of pyramid levels enables the algorithm to handle larger displacements of points between frames. However, the number of computations also increases. Recommended values are between 1 and 4. The diagram shows an image pyramid with 3 levels.



The algorithm forms each pyramid level by downsampling the previous level. The tracking begins in the lowest resolution level, and continues tracking until convergence. The optical flow algorithm propagates the result of that level to the next level as the initial guess of the point locations. In this way, the algorithm refines the tracking with each level, ending with the original image. Using the pyramid levels enables the optical flow algorithm to handle large pixel motions, which can be distances greater than the neighborhood size.

**'PyramidScale' — Image scale**

0.5 (default) | positive scalar in the range (0,1)

Image scale, specified as the comma-separated pair consisting of 'PyramidScale' and a positive scalar in the range (0,1). The pyramid scale is applied to each image at every pyramid level. A value of 0.5 creates a classical pyramid, where each level reduces in resolution by a factor of two compared to the previous level.

**'NumIterations' — Number of search iterations per pyramid level**

3 (default) | positive integer

Number of search iterations per pyramid level, specified as the comma-separated pair consisting of 'NumIterations' and a positive integer. The Farneback algorithm performs an iterative search for the new location of each point until convergence.

**'NeighborhoodSize' — Size of the pixel neighborhood**

5 (default) | positive integer

Size of the pixel neighborhood, specified as the comma-separated pair consisting of 'NeighborhoodSize' and a positive integer. Increase the neighborhood size to increase blurred motion. The blur motion yields a more robust estimation of optical flow. A typical value for NeighborhoodSize is 5 or 7.

**'FilterSize' — Averaging filter size**

15 (default) | positive integer in the range [2, Inf)

Averaging filter size, specified as the comma-separated pair consisting of 'FilterSize' and a positive integer in the range [2, Inf). After the algorithm computes the displacement (flow), the averaging over neighborhoods is done using a Gaussian filter of size (FilterSize \* FilterSize). Additionally, the pixels close to the borders are given a reduced weight because the algorithm assumes that the polynomial expansion coefficients are less reliable there. Increasing the filter size increases the robustness of the algorithm to image noise. The larger the filter size, the greater the algorithm handles image noise and fast motion detection, making it more robust.

## Methods

estimateFlow

Estimate optical flow

reset

Reset the internal state of the object

# Examples

## Optical Flow Estimation Using the Farneback Algorithm

Load a video.

```
vidReader = VideoReader('visiontraffic.avi','CurrentTime',11);
```

Set up an optical flow object to do the estimate.

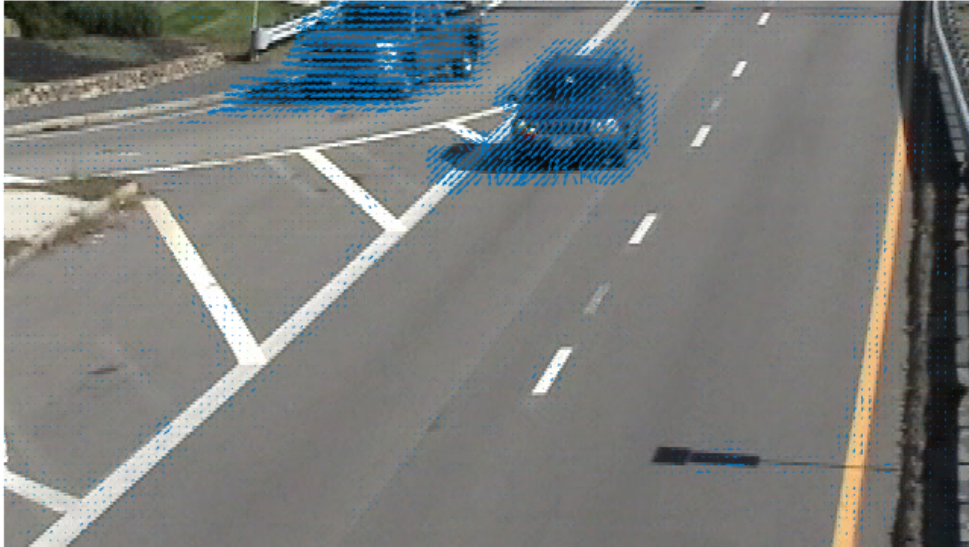
```
opticFlow = opticalFlowFarneback;
```

Read in video frames and estimate optical flow of each frame. Display the video frames with flow vectors.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',2)
    hold off
end
```



## References

- [1] Farneback, G. "Two-Frame Motion Estimation Based on Polynomial Expansion."  
*Proceedings of the 13th Scandinavian Conference on Image Analysis*. Gothenburg, Sweden, 2003.

## See Also

`opticalFlowLKDoG` | `opticalFlowHS` | `quiver` | `opticalFlow` | `opticalFlowLK`

Introduced in R2015b

# estimateFlow

**Class:** opticalFlowFarneback

Estimate optical flow

## Syntax

```
flow = estimateFlow(opticFlow,I)
```

## Description

`flow = estimateFlow(opticFlow,I)` estimates the optical flow of the input image, `I`, with respect to the previous image. The function sets the previous image for the first run to a black image. After the first run, each new image, `I`, becomes the current image, and the image from the last run becomes the previous one. You can also use flow estimation to predict a future position of a moving object for image reconstruction.

## Input Arguments

**opticFlow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, specified as an opticalFlow object.

**I** — Input image

grayscale image

Input image, specified as a grayscale image. To process the first frame, the function sets the previous frame to black.

## Output Arguments

**flow** — Object for storing optical flow velocity matrices

opticalFlow object

Object for storing optical flow velocity matrices, returned as an opticalFlow object.

## Examples

**Introduced in R2015b**

### **reset**

**Class:** `opticalFlowFarneback`

Reset the internal state of the object

### **Syntax**

```
reset(opticFlow)
```

### **Description**

`reset(opticFlow)` resets the internal state of the optical flow object. Doing this sets the previous frame to black.

**Introduced in R2015b**



# vision.OpticalFlow System object

**Package:** vision

Estimate object velocities

## Description

The `OpticalFlow` System object estimates object velocities from one image or video frame to another. It uses either the Horn-Schunck or the Lucas-Kanade method.

## Construction

`opticalFlow = vision.OpticalFlow` returns an optical flow System object, `opticalFlow`. This object estimates the direction and speed of object motion from one image to another or from one video frame to another.

`opticalFlow = vision.OpticalFlow(Name, Value)` returns an optical flow System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports Code Generation: No
“Code Generation Support, Usage Notes, and Limitations”.

## To estimate velocity:

- 1 Define and set up your text inserter using the constructor.
- 2 Call the `step` method with the input image, `I`, the optical flow object, `opticalFlow`, and any optional properties. See the syntax below for using the `step` method.

`VSQ = step(opticalFlow, I)` computes the optical flow of input image, `I`, from one video frame to another, and returns `VSQ`, specified as a matrix of velocity magnitudes.

`V = step(opticalFlow, I)` computes the optical flow of input image, `I`, from one video frame to another, and returns `V`, specified as a complex matrix of horizontal and vertical

components. This applies when you set the ??? property to 'Horizontal and vertical components in complex form'.

[...] = step(opticalFlow, I1, I2) computes the optical flow of the input image I1, using I2 as a reference frame. This applies when you set the ??? property to 'Input port'.

[..., IMV] = step(opticalFlow, I) outputs the delayed input image, IMV. The delay is equal to the latency introduced by the computation of the motion vectors. This property applies when you set the ??? property to 'Lucas-Kanade', the ??? property to 'Derivative of Gaussian', and the ??? property to true.

## Properties

### Method

Optical flow computation algorithm

Specify the algorithm to compute the optical flow as one of Horn-Schunck | Lucas-Kanade.

Default: Horn-Schunck

### ReferenceFrameSource

Source of reference frame for optical flow calculation

Specify computing optical flow as one of Property | Input port. When you set this property to Property, you can use the ReferenceFrameDelay property to determine a previous frame with which to compare. When you set this property to Input port, supply an input image for comparison.

This property applies when you set the ??? property to Horn-Schunck. This property also applies when you set the Method property to Lucas-Kanade and the ??? property to Difference filter [-1 1].

Default: Property

### ReferenceFrameDelay

Number of frames between reference frame and current frame

Specify the number of frames between the reference and current frame as a positive scalar integer. This property applies when you set the ??? property to `Current frame` and `N-th frame back`.

Default: 1

### **Smoothness**

Expected smoothness of optical flow

Specify the smoothness factor as a positive scalar number. If the relative motion between the two images or video frames is large, specify a large positive scalar value. If the relative motion is small, specify a small positive scalar value. This property applies when you set the ??? property to `Horn-Schunck`. This property is tunable.

Default: 1

### **IterationTerminationCondition**

Condition to stop iterative solution computation

Specify when the optical flow iterative solution stops. Specify as one of `Maximum iteration count` | `Velocity difference threshold` | `Either`. This property applies when you set the ??? property to `Horn-Schunck`.

Default: `Maximum iteration count`

### **MaximumIterationCount**

Maximum number of iterations to perform

Specify the maximum number of iterations to perform in the optical flow iterative solution computation as a positive scalar integer. This property applies when you set the ??? property to `Horn-Schunck` and the ??? property to either `Maximum iteration count` or `Either`. This property is tunable.

Default: 10

### **VelocityDifferenceThreshold**

Velocity difference threshold

Specify the velocity difference threshold to stop the optical flow iterative solution computation as a positive scalar number. This property applies when you set the ??? property to `Horn-Schunck` and the ??? property to either `Maximum iteration count` or `Either`. This property is tunable.

Default: `eps`

### **OutputValue**

Form of velocity output

Specify the velocity output as one of `Magnitude-squared` | `Horizontal` and `vertical` components in complex form.

Default: `Magnitude-squared`

### **TemporalGradientFilter**

Temporal gradient filter used by Lucas-Kanade algorithm

Specify the temporal gradient filter used by the Lucas-Kanade algorithm as one of `Difference filter [-1 1]` | `Derivative of Gaussian`. This property applies when you set the ??? property to `Lucas-Kanade`.

Default: `Difference filter [-1 1]`

### **BufferedFramesCount**

Number of frames to buffer for temporal smoothing

Specify the number of frames to buffer for temporal smoothing as an odd integer from `3` to `31`, both inclusive. This property determines characteristics such as the standard deviation and the number of filter coefficients of the Gaussian filter used to perform temporal filtering. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Derivative of Gaussian`.

Default: `3`

### **ImageSmoothingFilterStandardDeviation**

Standard deviation for image smoothing filter

Specify the standard deviation for the Gaussian filter used to smooth the image using spatial filtering. Use a positive scalar number. This property applies when you set the ??? property to Lucas-Kanade and the ??? property to Derivative of Gaussian.

Default: 1.5

### **GradientSmoothingFilterStandardDeviation**

Standard deviation for gradient smoothing filter

Specify the standard deviation for the filter used to smooth the spatiotemporal image gradient components. Use a positive scalar number. This property applies when you set the ??? property to Lucas-Kanade and the ??? property to Derivative of Gaussian.

Default: 1

### **DiscardIllConditionedEstimates**

Discard normal flow estimates when constraint equation is ill-conditioned

When the optical flow constraint equation is ill conditioned, set this property to `true` so that the motion vector is set to `0`. This property applies when you set the ??? property to Lucas-Kanade and the ??? property to Derivative of Gaussian. This property is tunable.

Default: `false`

### **MotionVectorImageOutputport**

Return image corresponding to motion vectors

Set this property to `true` to output the image that corresponds to the motion vector being output by the System object. This property applies when you set the ??? property to Lucas-Kanade and the ??? property to Derivative of Gaussian.

Default: `false`

### **NoiseReductionThreshold**

Threshold for noise reduction

Specify the motion threshold between each image or video frame as a positive scalar number. The higher the number, the less small movements impact the optical flow

calculation. This property applies when you set the ??? property to `Lucas-Kanade`. This property is tunable.

Default: `0.0039`

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding mode for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

Default: `Nearest`

#### **OverflowAction**

Overflow mode for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

Default: `Saturate`

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Custom`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled `numericType` object. You can apply this property when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`. This property applies when you set the ??? property to `Custom`.

Default: `numericType(true,32,20)`

**AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product` | `Custom`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

Default: `Same as product`

**CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numericType` object. You can apply this property when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`. This property applies when you set the ??? property to `Custom`.

Default: `numericType(true,32,20)`

**GradientDataType**

Gradients word and fraction lengths

Specify the gradient components fixed-point data type as one of `Same as accumulator` | `Same as accumulator` | `Same as product` | `Custom`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

Default: `Same as accumulator`

**CustomGradientDataType**

Gradients word and fraction lengths

Specify the gradient components fixed-point type as a signed, scaled `numericType` System object. You can apply this property when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`. This property applies when you set the ??? property to `Custom`.

The default is `numericType(true,32,20)`.

Default: 1

### **ThresholdDataType**

Threshold word and fraction lengths

Specify the threshold fixed-point data type as one of `Same word length as first input` | `Custom`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

Default: `Same word length as first input`

### **CustomThresholdDataType**

Threshold word and fraction lengths

Specify the threshold fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. You can apply this property when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`. This property applies when you set the `ThresholdMode` property to `Custom`.

Default: `numericType([],16,12)`

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Custom`. This property applies when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. The `numericType` object should be unsigned if you set the `OutputValue` property to `Magnitude-squared`. It should be signed if set to `Horizontal` and `vertical components in complex form`. You can apply this property when you set the ??? property to `Lucas-Kanade` and the ??? property to `Difference filter [-1 1]`. This property applies when you set the ??? property to `Custom`.

Default: `numericType(false,32,20)`



## Methods

clone	Create optical flow object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Estimate direction and speed of object motion between video frames

## Examples

### Track Cars Using Optical Flow

Set up objects.

```
videoReader = vision.VideoFileReader('viptraffic.avi','ImageColorSpace','Intensity','V
converter = vision.ImageDataTypeConverter;
opticalFlow = vision.OpticalFlow('ReferenceFrameDelay', 1);
opticalFlow.OutputValue = 'Horizontal and vertical components in complex form';
shapeInserter = vision.ShapeInserter('Shape','Lines','BorderColor','Custom','CustomBo
videoPlayer = vision.VideoPlayer('Name','Motion Vector');
```

Convert the image to single precision, then compute optical flow for the video. Generate coordinate points and draw lines to indicate flow. Display results.

```
while ~isDone(videoReader)
    frame = step(videoReader);
    im = step(converter, frame);
    of = step(opticalFlow, im);
    lines = videooptflowlines(of, 20);
    if ~isempty(lines)
        out = step(shapeInserter, im, lines);
        step(videoPlayer, out);
    end
end
```

end

Close the video reader and player

```
release(videoPlayer);  
release(videoReader);
```

## Algorithms

To compute the optical flow between two images, you must solve the following optical flow constraint equation:

$$I_x u + I_y v + I_t = 0$$

.

- $I_x$ ,  $I_y$ , and  $I_t$  are the spatiotemporal image brightness derivatives.
- $u$  is the horizontal optical flow.
- $v$  is the vertical optical flow.

## Horn-Schunck Method

By assuming that the optical flow is smooth over the entire image, the Horn-Schunck method computes an estimate of the velocity field,  $[u \ v]^T$ , that minimizes this equation:

$$E = \iint (I_x u + I_y v + I_t)^2 dx dy + \alpha \iint \left\{ \left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 \right\} dx dy$$

.

In this equation,  $\frac{\partial u}{\partial x}$  and  $\frac{\partial u}{\partial y}$  are the spatial derivatives of the optical velocity component,  $u$ , and  $\alpha$  scales the global smoothness term. The Horn-Schunck method minimizes the previous equation to obtain the velocity field,  $[u \ v]$ , for each pixel in the image. This method is given by the following equations:

$$u_{x,y}^{k+1} = u_{x,y}^{-k} - \frac{I_x[I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

$$v_{x,y}^{k+1} = v_{x,y}^{-k} - \frac{I_y[I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t]}{\alpha^2 + I_x^2 + I_y^2}$$

In these equations,  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$  is the velocity estimate for the pixel at  $(x,y)$ , and

$\begin{bmatrix} \bar{u}_{x,y}^{-k} & \bar{v}_{x,y}^{-k} \end{bmatrix}$  is the neighborhood average of  $\begin{bmatrix} u_{x,y}^k & v_{x,y}^k \end{bmatrix}$ . For  $k = 0$ , the initial velocity is 0.

To solve  $u$  and  $v$  using the Horn-Schunck method:

- 1 Compute  $I_x$  and  $I_y$  using the Sobel convolution kernel,  $\begin{bmatrix} -1 & -2 & -1; & 0 & 0 & 0; & 1 & 2 & 1 \end{bmatrix}$ , and its transposed form, for each pixel in the first image.
- 2 Compute  $I_t$  between images 1 and 2 using the  $\begin{bmatrix} -1 & 1 \end{bmatrix}$  kernel.
- 3 Assume the previous velocity to be 0, and compute the average velocity for each pixel using  $\begin{bmatrix} 0 & 1 & 0; & 1 & 0 & 1; & 0 & 1 & 0 \end{bmatrix}$  as a convolution kernel.
- 4 Iteratively solve for  $u$  and  $v$ .

## Lucas-Kanade Method

To solve the optical flow constraint equation for  $u$  and  $v$ , the Lucas-Kanade method divides the original image into smaller sections and assumes a constant velocity in each section. Then, it performs a weighted least-square fit of the optical flow constraint equation to a constant model for  $\begin{bmatrix} u & v \end{bmatrix}^T$  in each section  $\Omega$ . The method achieves this fit by minimizing the following equation:

$$\sum_{x \in \Omega} W^2 [I_x u + I_y v + I_t]^2$$

$W$  is a window function that emphasizes the constraints at the center of each section. The solution to the minimization problem is

$$\begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 I_x I_t \\ \sum W^2 I_y I_t \end{bmatrix}$$

### Lucas-Kanade Difference Filter

When you set the **Temporal gradient filter** to Difference filter  $[-1 \ 1]$ ,  $u$  and  $v$  are solved as follows:

- 1 Compute  $I_x$  and  $I_y$  using the kernel  $[-1 \ 8 \ 0 \ -8 \ 1]/12$  and its transposed form.

If you are working with fixed-point data types, the kernel values are signed fixed-point values with word length equal to 16 and fraction length equal to 15.

- 2 Compute  $I_t$  between images 1 and 2 using the  $[-1 \ 1]$  kernel.
- 3 Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a separable and isotropic 5-by-5 element kernel whose effective 1-D coefficients are  $[1 \ 4 \ 6 \ 4 \ 1]/16$ . If you are working with fixed-point data types, the kernel values are unsigned fixed-point values with word length equal to 8 and fraction length equal to 7.
- 4 Solve the 2-by-2 linear equations for each pixel using the following method:

$$\text{If } A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$$

$$\text{Then the eigenvalues of } A \text{ are } \lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$$

$$\text{In the fixed-point diagrams, } P = \frac{a+c}{2}, Q = \frac{\sqrt{4b^2 + (a-c)^2}}{2}$$

- The eigenvalues are compared to the threshold,  $\tau$ , that corresponds to the value you enter for the threshold for noise reduction. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

A is nonsingular, the system of equations are solved using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

A is singular (noninvertible), the gradient flow is normalized to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.

### Derivative of Gaussian

If you set the temporal gradient filter to **Derivative of Gaussian**,  $u$  and  $v$  are solved using the following steps. You can see the flow chart for this process at the end of this section:

- 1 Compute  $I_x$  and  $I_y$  using the following steps:
  - a Use a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the ??? property.
  - b Use a Gaussian filter and the derivative of a Gaussian filter to smooth the image using spatial filtering. Specify the standard deviation and length of the image smoothing filter using the ??? property.
- 2 Compute  $I_t$  between images 1 and 2 using the following steps:
  - a Use the derivative of a Gaussian filter to perform temporal filtering. Specify the temporal filter characteristics such as the standard deviation and number of filter coefficients using the ??? parameter.
  - b Use the filter described in step 1b to perform spatial filtering on the output of the temporal filter.

- 3 Smooth the gradient components,  $I_x$ ,  $I_y$ , and  $I_t$ , using a gradient smoothing filter. Use the ??? property to specify the standard deviation and the number of filter coefficients for the gradient smoothing filter.
- 4 Solve the 2-by-2 linear equations for each pixel using the following method:

- If  $A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 I_x^2 & \sum W^2 I_x I_y \\ \sum W^2 I_y I_x & \sum W^2 I_y^2 \end{bmatrix}$

Then the eigenvalues of A are  $\lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1, 2$

- When the object finds the eigenvalues, it compares them to the threshold,  $\tau$ , that corresponds to the value you enter for the ??? parameter. The results fall into one of the following cases:

Case 1:  $\lambda_1 \geq \tau$  and  $\lambda_2 \geq \tau$

A is nonsingular, so the object solves the system of equations using Cramer's rule.

Case 2:  $\lambda_1 \geq \tau$  and  $\lambda_2 < \tau$

A is singular (noninvertible), so the object normalizes the gradient flow to calculate  $u$  and  $v$ .

Case 3:  $\lambda_1 < \tau$  and  $\lambda_2 < \tau$

The optical flow,  $u$  and  $v$ , is 0.

## References

- [1] Barron, J.L., D.J. Fleet, S.S. Beauchemin, and T.A. Burkitt. *Performance of optical flow techniques*. CVPR, 1992.

## See Also

vision.Pyramid | quiver | opticalFlowLKDoG | opticalFlowLK | opticalFlow

**Introduced in R2012a**

# clone

**System object:** vision.OpticalFlow

**Package:** vision

Create optical flow object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.



# getNumInputs

**System object:** vision.OpticalFlow

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

For many System objects, this method is a no-op. Objects that have internal states will describe in their help what the reset method does for that object.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.OpticalFlow

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.OpticalFlow

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

isLocked(h)

## Description

isLocked(h) returns the locked status, TF of the OpticalFlow System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.OpticalFlow

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.OpticalFlow

**Package:** vision

Estimate direction and speed of object motion between video frames

## Syntax

```
VSQ = step(opticalFlow,I)
V = step(opticalFlow,I)
[...] = step(opticalFlow,I1,I2)
[... , IMV] = step(opticalFlow,I)
```

## Description

`VSQ = step(opticalFlow,I)` computes the optical flow of input image, `I`, from one video frame to another, and returns `VSQ`, specified as a matrix of velocity magnitudes.

`V = step(opticalFlow,I)` computes the optical flow of input image, `I`, from one video frame to another, and returns `V`, specified as a complex matrix of horizontal and vertical components. This applies when you set the `???` property to 'Horizontal and vertical components in complex form'.

`[...] = step(opticalFlow,I1,I2)` computes the optical flow of the input image `I1`, using `I2` as a reference frame. This applies when you set the `???` property to 'Input port'.

`[... , IMV] = step(opticalFlow,I)` outputs the delayed input image, `IMV`. The delay is equal to the latency introduced by the computation of the motion vectors. This property applies when you set the `???` property to 'Lucas-Kanade', the `???` property to 'Derivative of Gaussian', and the `???` property to `true`.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the **release** method to unlock the object.

---

# vision.PeopleDetector System object

**Package:** vision

Detect upright people using HOG features

## Description

The people detector object detects people in an input image using the Histogram of Oriented Gradient (HOG) features and a trained Support Vector Machine (SVM) classifier. The object detects unoccluded people in an upright position.

## Construction

`peopleDetector = vision.PeopleDetector` returns a System object, `peopleDetector`, that tracks a set of points in a video.

`peopleDetector = vision.PeopleDetector(MODEL)` creates a `peopleDetector` System object and sets the `ClassificationModel` property to `MODEL`. The input `MODEL` can be either `'UprightPeople_128x64'` or `'UprightPeople_96x48'`.

`peopleDetector = vision.PeopleDetector(Name, Value)` configures the tracker object properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: No
Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries.
“Code Generation Support, Usage Notes, and Limitations”

## To detect people:

- 1 Define and set up your people detector object using the constructor.
- 2 Call the `step` method with the input image, `I`, and the people detector object, `peopleDetector`. See the syntax below for using the `step` method.

`BBOXES = step(peopleDetector, I)` performs multiscale object detection on the input image, `I`. The method returns an  $M$ -by-4 matrix defining  $M$  bounding boxes, where  $M$  represents the number of detected people. Each row of the output matrix, `BBOXES`, contains a four-element vector, `[x y width height]`. This vector specifies, in pixels, the upper-left corner and size, of a bounding box. When no people are detected, the `step` method returns an empty vector. The input image, `I`, must be a grayscale or truecolor (RGB) image.

`[BBOXES, SCORES] = step(peopleDetector, I)` returns a confidence value for the detections. The  $M$ -by-1 vector, `SCORES`, contain positive values for each bounding box in `BBOXES`. Larger score values indicate a higher confidence in the detection. The `SCORES` value depends on how you set the `MergeDetections` property. When you set the property to `true`, the people detector algorithm evaluates classification results to produce the `SCORES` value. When you set the property to `false`, the detector returns the unaltered classification `SCORES`.

`[ ___ ] = step(peopleDetector, I, roi)` detects people within the rectangular search region specified by `roi`. You must specify `roi` as a 4-element vector, `[x y width height]`, that defines a rectangular region of interest within image `I`. Set the `'UseROI'` property to `true` to use this syntax.

## Properties

### **ClassificationModel** — Name of classification model

'UprightPeople\_128x64' (default) | 'UprightPeople\_96x48'

Name of classification model, specified as a comma-separated pair consisting of 'ClassificationModel' and the string 'UprightPeople\_128x64' or 'UprightPeople\_96x48'. The pixel dimensions indicate the image size used for training.

The images used to train the models include background pixels around the person. Therefore, the actual size of a detected person is smaller than the training image size.

### **ClassificationThreshold** — People classification threshold

1 (default) | nonnegative scalar value

People classification threshold, specified as a comma-separated pair consisting of 'ClassificationThreshold' and a nonnegative scalar value. Use this threshold to control the classification of individual image subregions during multiscale detection. The



threshold controls whether a subregion gets classified as a person. You can increase this value when there are many false detections. The higher the threshold value, the more stringent the requirements are for the classification. Vary the threshold over a range of values to find the optimum value for your data set. Typical values range from 0 to 4. This property is tunable.

### **MinSize — Smallest region containing a person**

[ ] (default) | two-element vector

Smallest region containing a person, specified as a comma-separated pair consisting of 'MinSize' and a two-element [*height width*] vector. Set this property in pixels for the minimum size region containing a person. When you know the minimum person size to detect, you can reduce computation time. To do so, set this property to a value larger than the image size used to train the classification model. When you do not specify this property, the detector sets it to the image size used to train the classification model. This property is tunable.

### **MaxSize — Largest region containing a person**

[ ] (default) | two-element vector

Largest region that contains a person, specified as a comma-separated pair consisting of 'MaxSize' and a two-element [*height width*] vector. Set this property in pixels for the largest region containing a person. When you know the maximum person size to detect, you can reduce computation time. To do so, set this property to a value smaller than the size of the input image. When you do not specify this property, the detector sets it to the input image size. This property is tunable.

### **ScaleFactor — Multiscale object detection scaling**

1.05 (default) | numeric value greater than 1.0001

Multiscale object detection scaling, specified as a comma-separated pair consisting of 'ScaleFactor' and a value greater than 1.0001. The scale factor incrementally scales the detection resolution between MinSize and MaxSize. You can set the scale factor to an ideal value using:

$$\text{size(I)} / (\text{size(I)} - 0.5)$$

The object calculates the detection resolution at each increment.

$$\text{round}(\text{TrainingSize} * (\text{ScaleFactor}^N))$$

In this case, the *TrainingSize* is [128 64] for the 'UprightPeople\_128x64' model and [96 48] for the 'UprightPeople\_96x48' model. *N* is the increment. Decreasing the scale factor can increase the detection accuracy. However, doing so increases the computation time. This property is tunable.

### **WindowStride** — Detection window stride

[8 8] (default) | scalar | two-element vector

Detection window stride in pixels, specified as a comma-separated pair consisting of 'WindowStride' and a scalar or a two-element [x y] vector. The object uses the window stride to slide the detection window across the image. When you specify this value as a vector, the first and second elements are the stride size in the *x* and *y* directions. When you specify this value as a scalar, the stride is the same for both *x* and *y*. Decreasing the window stride can increase the detection accuracy. However, doing so increases computation time. Increasing the window stride beyond [8 8] can lead to a greater number of missed detections. This property is tunable.

### **MergeDetections** — Merge detection control

true | logical scalar

Merge detection control, specified as a comma-separated pair consisting of 'MergeDetections' and a logical scalar. This property controls whether similar detections are merged. Set this property to **true** to merge bounding boxes using a mean-shift based algorithm. Set this property to **false** to output the unmerged bounding boxes.

For more flexibility and control of merging parameters, you can use the `selectStrongestBbox` function in place of the `MergeDetections` algorithm. To do this, set the `MergeDetections` property to **false**. See the “Tracking Pedestrians from a Moving Car” example, which shows the use of the people detector and the `selectStrongestBbox` function.

### **UseROI** — Use region of interest

false (default) | true

Use region of interest, specified as a comma-separated pair consisting of 'UseROI' and a logical scalar. Set this property to **true** to detect objects within a rectangular region of interest within the input image.

## Methods

clone

Create alpha blender object with same property values

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Detect upright people using HOG features

## Examples

### Detect People

Create a people detector and load the input image.

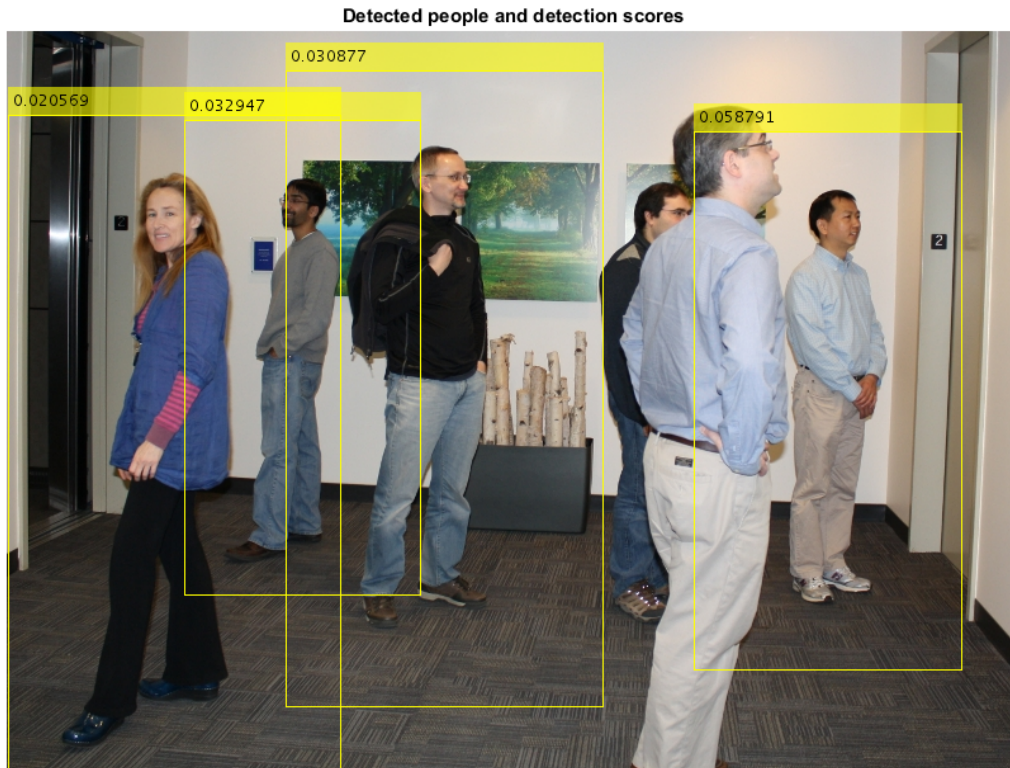
```
peopleDetector = vision.PeopleDetector;  
I = imread('visionteam1.jpg');
```

Detect people using the people detector object.

```
[bboxes,scores] = step(peopleDetector,I);
```

Annotate detected people.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);  
figure, imshow(I)  
title('Detected people and detection scores');
```



- “Tracking Pedestrians from a Moving Car”

## References

Dalal, N. and B. Triggs. “Histograms of Oriented Gradients for Human Detection,” *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, June 2005, pp. 886-893.

## See Also

`vision.ShapeInserter` | `vision.CascadeObjectDetector` | `insertShape` | `insertText` | `round` | `size`

## **More About**

- “Multiple Object Tracking”

**Introduced in R2012b**

# clone

**System object:** vision.PeopleDetector

**Package:** vision

Create alpha blender object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## isLocked

**System object:** vision.PeopleDetector

**Package:** vision

Locked status for input attributes and nontunable properties

### Syntax

TF = isLocked(H)

### Description

TF = isLocked(H) returns the locked status, TF of the PeopleDetector System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.PeopleDetector

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You cannot use the release method on System objects in Embedded MATLAB.

---



## step

**System object:** vision.PeopleDetector

**Package:** vision

Detect upright people using HOG features

## Syntax

```
BBOXES = step(peopleDetector,I)
[BBOXES,SCORES] = step(peopleDetector,I)
[ ___ ] = step(peopleDetector,I,roi)
```

## Description

`BBOXES = step(peopleDetector,I)` performs multi-scale object detection on the input image, `I`. The method returns an  $M$ -by-4 matrix defining  $M$  bounding boxes, where  $M$  represents the number of detected people. Each row of the output matrix, `BBOXES`, contains a four-element vector, `[x y width height]`, that specifies in pixels, the upper-left corner and size of a bounding box. When no people are detected, the `step` method returns an empty vector. The input image `I`, must be a grayscale or truecolor (RGB) image.

`[BBOXES,SCORES] = step(peopleDetector,I)` additionally returns a confidence value for the detections. The  $M$ -by-1 vector, `SCORES`, contain positive values for each bounding box in `BBOXES`. Larger score values indicate a higher confidence in the detection. The `SCORES` value depends on how you set the `MergeDetections` property. When you set the property to `true`, the people detector algorithm evaluates classification results to produce the `SCORES` value. When you set the property to `false`, the detector returns the unaltered classification `SCORES`.

`[ ___ ] = step(peopleDetector,I,roi)` detects people within the rectangular search region specified by `roi`. You must specify `roi` as a 4-element vector, `[x y width height]`, that defines a rectangular region of interest within image `I`. Set the `'UseROI'` property to `true` to use this syntax.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.PointTracker System object

**Package:** vision

Track points in video using Kanade-Lucas-Tomasi (KLT) algorithm

## Description

The point tracker object tracks a set of points using the Kanade-Lucas-Tomasi (KLT), feature-tracking algorithm. You can use the point tracker for video stabilization, camera motion estimation, and object tracking. It works particularly well for tracking objects that do not change shape and for those that exhibit visual texture. The point tracker is often used for short-term tracking as part of a larger tracking framework.

As the point tracker algorithm progresses over time, points can be lost due to lighting variation, out of plane rotation, or articulated motion. To track an object over a long period of time, you may need to reacquire points periodically.

## Construction

`pointTracker = vision.PointTracker` returns a System object, `pointTracker`, that tracks a set of points in a video.

`pointTracker = vision.PointTracker(Name,Value)` configures the tracker object properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: No
“System Objects in MATLAB Code Generation”.
Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries.
“Code Generation Support, Usage Notes, and Limitations”.

### Initialize Tracking Process

To initialize the tracking process, you must use the `initialize` method to specify the initial locations of the points and the initial video frame.

`initialize(pointTracker, points, I)` initializes points to track and sets the initial video frame. The initial locations `POINTS`, must be an  $M$ -by-2 array of  $[x\ y]$  coordinates. The initial video frame, `I`, must be a 2-D grayscale or RGB image and must be the same size and data type as the video frames passed to the `step` method.

The `detectFASTFeatures`, `detectSURFFeatures`, `detectHarrisFeatures`, and `detectMinEigenFeatures` functions are few of the many ways to obtain the initial points for tracking.

### To track a set of points:

- 1 Define and set up your point tracker object using the constructor.
- 2 Call the `step` method with the input image, `I`, and the point tracker object, `pointTracker`. See the following syntax for using the `step` method.

After initializing the tracking process, use the `step` method to track the points in subsequent video frames. You can also reset the points at any time by using the `setPoints` method.

`[points, point_validity] = step(pointTracker, I)` tracks the points in the input frame, `I` using the point tracker object, `pointTracker`. The output `points` contain an  $M$ -by-2 array of  $[x\ y]$  coordinates that correspond to the new locations of the points in the input frame, `I`. The output, `point_validity` provides an  $M$ -by-1 logical array, indicating whether or not each point has been reliably tracked.

A point can be invalid for several reasons. The point can become invalid if it falls outside of the image. Also, it can become invalid if the spatial gradient matrix computed in its neighborhood is singular. If the bidirectional error is greater than the `MaxBidirectionalError` threshold, this condition can also make the point invalid.

`[points, point_validity, scores] = step(pointTracker, I)` additionally returns the confidence score for each point. The  $M$ -by-1 output array, `scores`, contains values between 0 and 1. These values correspond to the degree of similarity between the neighborhood around the previous location and new location of each point. These values are computed as a function of the sum of squared differences between the previous and

new neighborhoods. The greatest tracking confidence corresponds to a perfect match score of 1.

`setPoints(pointTracker, points)` sets the points for tracking. The method sets the  $M$ -by-2 `points` array of  $[x\ y]$  coordinates with the points to track. You can use this method if the points need to be redetected because too many of them have been lost during tracking.

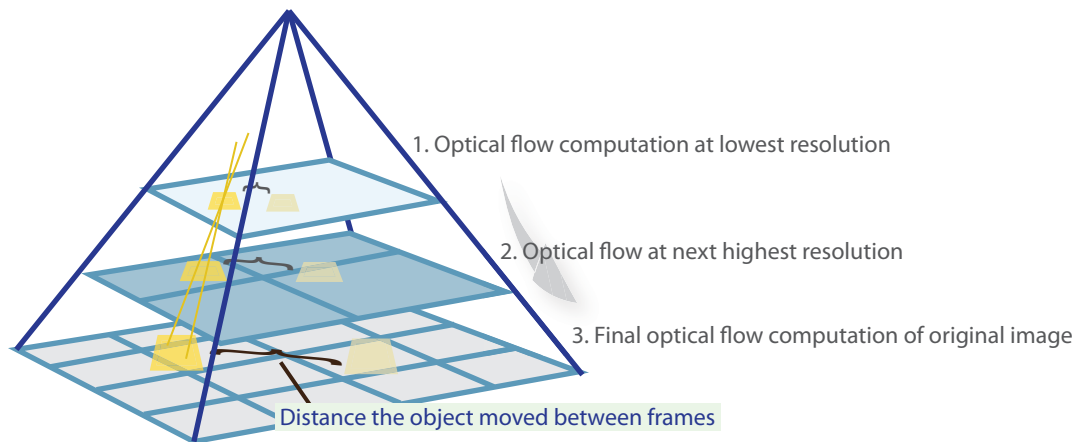
`setPoints(pointTracker, points, point_validity)` additionally lets you mark points as either valid or invalid. The input logical vector `point_validity` of length  $M$ , contains the true or false value corresponding to the validity of the point to be tracked. The length  $M$  corresponds to the number of points. A false value indicates an invalid point that should not be tracked. For example, you can use this method with the `estimateGeometricTransform` function to determine the transformation between the point locations in the previous and current frames. You can mark the outliers as invalid.

## Properties

### **NumPyramidLevels**

Number of pyramid levels

Specify an integer scalar number of pyramid levels. The point tracker implementation of the KLT algorithm uses image pyramids. The object generates an image pyramid, where each level is reduced in resolution by a factor of two compared to the previous level. Selecting a pyramid level greater than 1, enables the algorithm to track the points at multiple levels of resolution, starting at the lowest level. Increasing the number of pyramid levels allows the algorithm to handle larger displacements of points between frames. However, computation cost also increases. Recommended values are between 1 and 4.



Each pyramid level is formed by down-sampling the previous level by a factor of two in width and height. The point tracker begins tracking each point in the lowest resolution level, and continues tracking until convergence. The object propagates the result of that level to the next level as the initial guess of the point locations. In this way, the tracking is refined with each level, up to the original image. Using the pyramid levels allows the point tracker to handle large pixel motions, which can comprise distances greater than the neighborhood size.

**Default: 3**

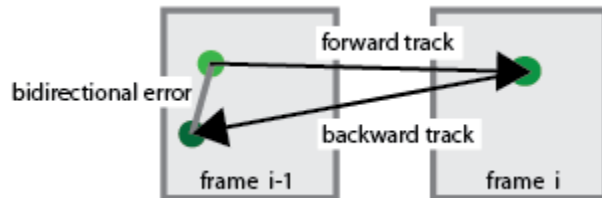
### **MaxBidirectionalError**

Forward-backward error threshold

Specify a numeric scalar for the maximum bidirectional error. If the value is less than `inf`, the object tracks each point from the previous to the current frame. It then tracks the same points back to the previous frame. The object calculates the bidirectional error. This value is the distance in pixels from the original location of the points to the final location after the backward tracking. The corresponding points are considered invalid when the error is greater than the value set for this property. Recommended values are between 0 and 3 pixels.

Using the bidirectional error is an effective way to eliminate points that could not be reliably tracked. However, the bidirectional error requires additional computation. When

you set the `MaxBidirectionalError` property to `inf`, the object does not compute the bidirectional error.



**Default:** `inf`

### **BlockSize**

Size of neighborhood

Specify a two-element vector, `[height, width]` to represent the neighborhood around each point being tracked. The height and width must be odd integers. This neighborhood defines the area for the spatial gradient matrix computation. The minimum value for `BlockSize` is `[5 5]`. Increasing the size of the neighborhood, increases the computation time.

**Default:** `[31 31]`

### **MaxIterations**

Maximum number of search iterations

Specify a positive integer scalar for the maximum number of search iterations for each point. The KLT algorithm performs an iterative search for the new location of each point until convergence. Typically, the algorithm converges within 10 iterations. This property sets the limit on the number of search iterations. Recommended values are between 10 and 50.

**Default:** 30

## **Methods**

`initialize`

Initialize video frame and points to track

setPoints

Set points to track

step

Track points in video using Kanade-Lucas-Tomasi (KLT) algorithm

## Examples

### Track a Face

**Create System objects for reading and displaying video and for drawing a bounding box of the object.**

```
videoFileReader = vision.VideoFileReader('visionface.avi');  
videoPlayer = vision.VideoPlayer('Position', [100, 100, 680, 520]);
```

**Read the first video frame, which contains the object, define the region.**

```
objectFrame = step(videoFileReader);  
objectRegion = [264, 122, 93, 93];
```

**As an alternative, you can use the following commands to select the object region using a mouse. The object must occupy the majority of the region.**

```
figure; imshow(objectFrame); objectRegion=round(getPosition(imrect))
```

**Show initial frame with a red bounding box.**

```
objectImage = insertShape(objectFrame, 'Rectangle', objectRegion, 'Color', 'red');  
figure; imshow(objectImage); title('Red box shows object region');
```



Red box shows object region



**Detect interest points in the object region.**

```
points = detectMinEigenFeatures(rgb2gray(objectFrame), 'ROI', objectRegion);
```

**Display the detected points.**

```
pointImage = insertMarker(objectFrame, points.Location, '+', 'Color', 'white');  
figure, imshow(pointImage), title('Detected interest points');
```

Detected interest points



### Create a tracker object.

```
tracker = vision.PointTracker('MaxBidirectionalError', 1);
```

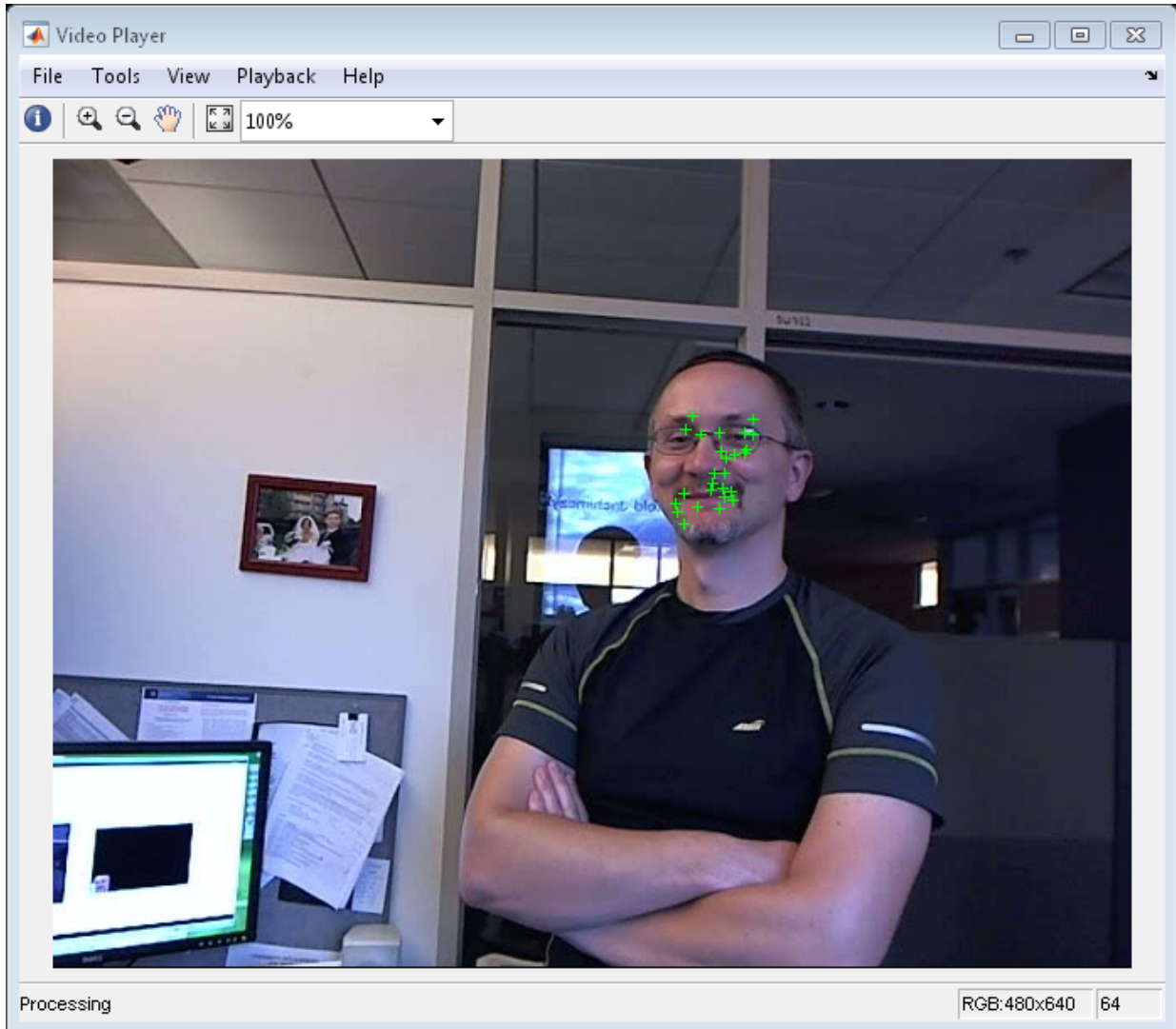
### Initialize the tracker.

```
initialize(tracker, points.Location, objectFrame);
```

### Read, track, display points, and results in each video frame.

```
while ~isDone(videoFileReader)  
    frame = step(videoFileReader);  
    [points, validity] = step(tracker, frame);  
    out = insertMarker(frame, points(validity, :), '+');
```

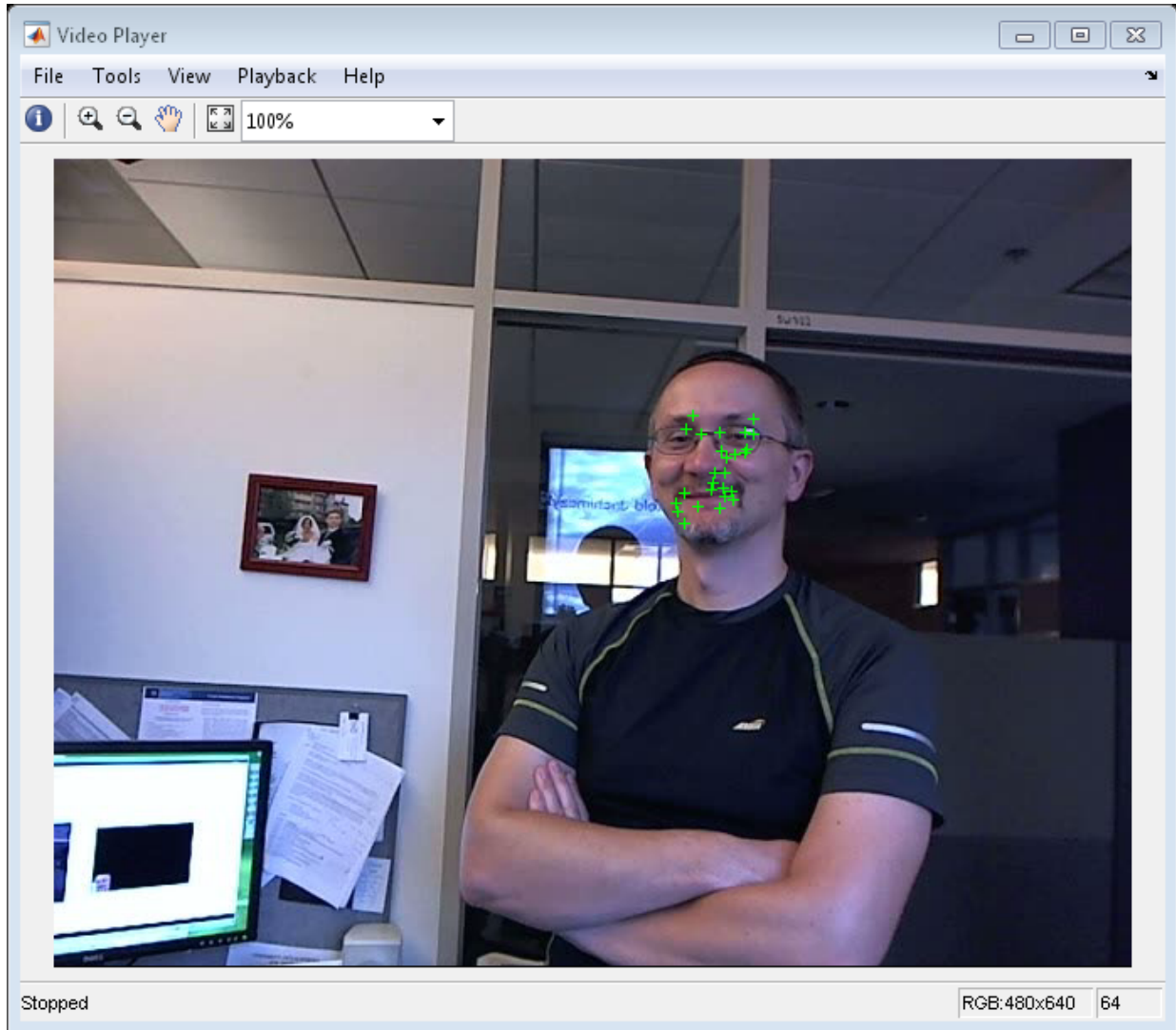
```
    step(videoPlayer, out);  
end
```



**Release the video reader and player.**

```
release(videoPlayer);
```

```
release(videoFileReader);
```



- “Face Detection and Tracking Using CAMShift”
- “Face Detection and Tracking Using the KLT Algorithm”
- “Face Detection and Tracking Using Live Video Acquisition”

## References

Lucas, Bruce D. and Takeo Kanade. “An Iterative Image Registration Technique with an Application to Stereo Vision,” *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, April, 1981, pp. 674–679.

Tomasi, Carlo and Takeo Kanade. *Detection and Tracking of Point Features*, Computer Science Department, Carnegie Mellon University, April, 1991.

Shi, Jianbo and Carlo Tomasi. “Good Features to Track,” *IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.

Kalal, Zdenek, Krystian Mikolajczyk, and Jiri Matas. “Forward-Backward Error: Automatic Detection of Tracking Failures,” *Proceedings of the 20th International Conference on Pattern Recognition*, 2010, pages 2756–2759, 2010.

## See Also

vision.HistogramBasedTracker | detectHarrisFeatures  
| detectMinEigenFeatures | detectSURFFeatures |  
estimateGeometricTransform | imrect | insertMarker

## External Websites

- Object Recognition and Tracking for Augmented Reality
- Detect and Track Multiple Faces in a Live Video Stream

**Introduced in R2012b**

# initialize

**System object:** vision.PointTracker

**Package:** vision

Initialize video frame and points to track

## Syntax

```
initialize(H,POINTS,I)
```

## Description

`initialize(H,POINTS,I)` initializes points to track and sets the initial video frame. The method sets the  $M$ -by-2 `POINTS` array of [x y] coordinates with the points to track, and sets the initial video frame, `I`. The input, `POINTS`, must be an  $M$ -by-2 array of [x y] coordinates. The input, `I`, must be a 2-D grayscale or RGB image, and must be the same size as the images passed to the `step` method.

## setPoints

**System object:** vision.PointTracker

**Package:** vision

Set points to track

### Syntax

```
setPoints(H,POINTS)
```

```
setPoints(H,POINTS,POINT_VALIDITY)
```

### Description

`setPoints(H,POINTS)` sets the points for tracking. The method sets the  $M$ -by-2 `POINTS` array of  $[x\ y]$  coordinates with the points to track. This method can be used if the points need to be re-detected because too many of them have been lost during tracking.

`setPoints(H,POINTS,POINT_VALIDITY)` additionally lets you mark points as either valid or invalid. The input logical vector `POINT_VALIDITY` of length  $M$ , contains the true or false value corresponding to the validity of the point to be tracked. The length  $M$  corresponds to the number of points. A false value indicates an invalid point, and it should not be tracked. You can use this method with the `vision.GeometricTransformEstimator` object to determine the transformation between the point locations in the previous and current frames, and then mark the outliers as invalid.

# step

**System object:** vision.PointTracker

**Package:** vision

Track points in video using Kanade-Lucas-Tomasi (KLT) algorithm

## Syntax

```
[POINTS,POINT_VALIDITY] = step(H,I)
```

```
[POINTS,POINT_VALIDITY,SCORES] = step(H,I)
```

## Description

[POINTS,POINT\_VALIDITY] = step(H,I) tracks the points in the input frame, I. The input POINTS contain an *M*-by-2 array of [x y] coordinates that correspond to the new locations of the points in the input frame, I. The input, POINT\_VALIDITY provides an *M*-by-1 logical array, indicating whether or not each point has been reliably tracked. The input frame, I must be the same size as the image passed to the initialize method.

[POINTS,POINT\_VALIDITY,SCORES] = step(H,I) additionally returns the confidence score for each point. The *M*-by-1 output array, SCORE, contains values between 0 and 1. These values are computed as a function of the sum of squared differences between the *BlockSize*-by-*BlockSize* neighborhood around the point in the previous frame, and the corresponding neighborhood in the current frame. The greatest tracking confidence corresponds to a perfect match score of 1.

---

**Note:** H specifies the System object on which to run this step method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---



# vision.PSNR System object

**Package:** vision

Compute peak signal-to-noise ratio (PSNR) between images

## Description

---

**Note:** The `vision.PSNR` System object will be removed in a future release. Use the `psnr` function with equivalent functionality instead.

---

The PSNR object computes the peak signal-to-noise ratio (PSNR) between images. This ratio is often used as a quality measurement between an original and a compressed image.

## Construction

`H = vision.PSNR` returns a System object, `H`, that computes the peak signal-to-noise ratio (PSNR) in decibels between two images.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Methods

<code>clone</code>	Create peak signal to noise ratio object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method

isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Compute peak signal-to-noise ratio

## Examples

Compute the PSNR between an original image and its reconstructed image.

```
hdct2d = vision.DCT;  
hidct2d = vision.IDCT;  
hpsnr = vision.PSNR;  
I = double(imread('cameraman.tif'));  
J = step(hdct2d, I);  
J(abs(J) < 10) = 0;  
It = step(hidct2d, J);  
psnr = step(hpsnr, I, It)  
imshow(I, [0 255]), title('Original image');  
figure, imshow(It, [0 255]), title('Reconstructed image');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the PSNR block reference page. The object properties correspond to the block parameters.

## See Also

[vision.DCT](#) | [vision.IDCT](#)

**Introduced in R2012a**

# clone

**System object:** vision.PSNR

**Package:** vision

Create peak signal to noise ratio object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.PSNR

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs**( $H$ ).

# getNumOutputs

**System object:** vision.PSNR

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.PSNR

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the PSNR System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.PSNR

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.PSNR

**Package:** vision

Compute peak signal-to-noise ratio

### Syntax

$Y = \text{step}(H, X1, X2)$

### Description

$Y = \text{step}(H, X1, X2)$  computes the peak signal-to-noise ratio,  $Y$ , between images  $X1$  and  $X2$ . The two images  $X1$  and  $X2$  must have the same size.

---

**Note:**  $H$  specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.Pyramid System object

**Package:** vision

Perform Gaussian pyramid decomposition

## Description

The `Pyramid` object computes Gaussian pyramid reduction or expansion. The image reduction step involves lowpass filtering and downsampling the image pixels. The image expansion step involves upsampling the image pixels and lowpass filtering.

## Construction

`gaussPyramid = vision.Pyramid` returns a System object, `gaussPyramid`, that computes a Gaussian pyramid reduction or expansion of an image.

`gaussPyramid = vision.Pyramid(Name, Value)` configures the System object properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## To compute a Gaussian pyramid:

- 1 Define and set up your pyramid object using the constructor.
- 2 Call the `step` method with the input image, `I` and the pyramid object, `gaussPyramid`. See the syntax below for using the `step` method.

`J = step(gaussPyramid, I)` computes `J`, the Gaussian pyramid decomposition of input `I`.

## Properties

### Operation

Reduce or expand the input image

Specify whether to reduce or expand the input image as **Reduce** or **Expand**. If this property is set to **Reduce**, the object applies a lowpass filter and then downsamples the input image. If this property is set to **Expand**, the object upsamples and then applies a lowpass filter to the input image.

Default: **Reduce**

### PyramidLevel

Level of decomposition

Specify the number of times the object upsamples or downsamples each dimension of the image by a factor of 2.

Default: 1

### SeparableFilter

How to specify the coefficients of low pass filter

Indicate how to specify the coefficients of the lowpass filter as **Default** or **Custom**.

Default: **Default**

### CoefficientA

Coefficient 'a' of default separable filter

Specify the coefficients in the default separable filter  $1/4 - a/2$   $1/4$   $a$   $1/4$   $1/4 - a/2$  as a scalar value. This property applies when you set the ??? property to **Default**.

Default: 0.375

### CustomSeparableFilter

Separable filter coefficients

Specify separable filter coefficients as a vector. This property applies when you set the ??? property to `Custom`.

Default: `[0.0625 0.25 0.375 0.25 0.0625]`

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`.

Default: `Floor`

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`.

Default: `Wrap`

#### **SeparableFilterDataType**

`CustomSeparableFilter` word and fraction lengths

Specify the coefficients fixed-point data type as `Same word length as input`, `Custom`.

Default: `Custom`

#### **CustomSeparableFilterDataType**

`CustomSeparableFilter` word and fraction lengths

Specify the coefficients fixed-point type as a signed `numericType` object with a `Signedness` of `Auto`. This property applies when you set the ??? property to `Custom`.

Default: `numericType([],16,14)`

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input`, or `Custom`.

Default: `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

Default: `numericType([],32,10)`

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Same as input`, or `Custom`.

Default: `Same as product`

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

Default: `numericType([],32,0)`

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input`, or `Custom`.

Default: `Custom`

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a signed, scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

Default: `numericType([],32,10)`

## Methods

<code>clone</code>	Create pyramid object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Compute Gaussian pyramid decomposition of input

## Examples

### Resize Image Using Gaussian Pyramid Decomposition

Set the level of decomposition

```
gaussPyramid = vision.Pyramid('PyramidLevel', 2);
```

Cast to single, otherwise the returned output J, will be a fixed point object.

```
I = im2single(imread('cameraman.tif'));
J = step(gaussPyramid, I);
```

Display results.

```
figure, imshow(I); title('Original Image');
figure, imshow(J); title('Reduced Image');
```

### See Also

`imresize` | `impyramid`

**Introduced in R2012a**

# clone

**System object:** vision.Pyramid

**Package:** vision

Create pyramid object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Pyramid

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .



# getNumOutputs

**System object:** vision.Pyramid

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Pyramid

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the Pyramid System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.Pyramid

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

### step

**System object:** vision.Pyramid

**Package:** vision

Compute Gaussian pyramid decomposition of input

### Syntax

```
J = step(gaussPyramid,I)
```

### Description

`J = step(gaussPyramid,I)` computes J, the Gaussian pyramid decomposition of input I.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.ShapeInserter System object

**Package:** vision

Draw rectangles, lines, polygons, or circles on an image

## Description

---

**Note:** The `vision.ShapeInserter` System object will be removed in a future release. Use the `insertShape` function instead.

---

The `ShapeInserter` object can draw multiple rectangles, lines, polygons, or circles in a 2-D grayscale or truecolor RGB image. The output image can then be displayed or saved to a file.

## Construction

`shapeInserter = vision.ShapeInserter` returns a System object, `shapeInserter`, that draws multiple rectangles, lines, polygons, or circles on images by overwriting pixel values.

`shapeInserter = vision.ShapeInserter(Name, Value)` returns a shape inserter System object, `shapeInserter`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”
“Code Generation Support, Usage Notes, and Limitations”.

## To insert a shape:

- 1 Define and set up your shape inserter using the constructor.

- 2 Call the `step` method with the input image, `I`, the shape inserter object, `shapeInserter`, points `PTS`, and any optional properties. See the syntax below for using the `step` method.

`J = step(shapeInserter, I, PTS)` draws the shape specified by the `???` property on input image `I`. The input `PTS` specify the coordinates for the location of the shape. The shapes are embedded on the output image `J`.

`J = step(shapeInserter, I, PTS, ROI)` draws a shape inside an area defined by the `ROI` input. This applies only when you set the `???` property to `true`. The `ROI` input defines a rectangular area as `[x y width height]`, where `[x y]` determine the upper-left corner location of the rectangle, and `width` and `height` specify the size.

`J = step(shapeInserter, I, PTS, . . . , CLR)` draws the shape with the border or fill color specified by the input `CLR`. This applies when you set the `???` property or the `???` property to 'Input port'.

## Properties

### Shape

Shape to draw

You can specify the type of shape as `Rectangles`, `Lines`, `Polygons`, or `Circles`. When you specify the type of shape to draw, you must also specify the location. The `PTS` input specifies the location of the points. The table shows the format for the points input for the different shapes. For a more detailed explanation on how to specify shapes and lines, see “Draw Shapes and Lines”.

Shape property	PTS input
'Rectangles'	$M$ -by-4 matrix, where $M$ specifies the number of rectangles. Each row specifies a rectangle in the format <code>[x y width height]</code> , where <code>[x y]</code> determine the upper-left corner location of the rectangle, and <code>width</code> and <code>height</code> specify the size.
'Lines'	$M$ -by- $2L$ matrix, where $M$ specifies the number of polylines. Each row specifies a polyline as a series of consecutive point locations, <code>[<math>x_1, y_1, x_2, y_2, \dots, x_L, y_L</math>]</code> .

Shape property	PTS input
'Polygons'	$M$ -by- $2L$ matrix, where $M$ specifies the number of polygons. Each row specifies a polygon as an array of consecutive point locations, $[x_1, y_1, x_2, y_2, \dots, x_L, y_L]$ . The point define the vertices of a polygon.
'Circles'	$M$ -by-3 matrix, where $M$ specifies the number of circles. Each row specifies a circle in the format $[x \ y \ radius]$ , where $[x \ y]$ define the coordinates of the center of the circle.

Default: Rectangles

### Fill

Enable filling shape

Set this property to **true** to fill the shape with an intensity value or a color. This property only applies for the 'Rectangles', 'Polygons', and 'Circles' shapes.

Default: false

### LineWidth

Line width

Specify the line width of the shape.

Default: 1

### BorderColorSource

Source of border color

Specify how the shape's border color is provided as `Input port` or `Property`. This property applies when you set the ??? property to `Lines` or, when you do not set the `Shape` property to `Lines` and you set the `Fill` property to `false`. When you set the ??? property to `Input port`, a border color vector must be provided as an input to the System object's `step` method.

Default: Property

### **BorderColor**

Border color of shape

Specify the appearance of the shape's border as **Black**, **White**, or **Custom**. When you set this property to **Custom**, you can use the ??? property to specify the value. This property applies when you set the ??? property to **Property**.

Default: Black

### **CustomBorderColor**

Intensity or color value for shape's border

Specify an intensity or color value for the shape's border. For an intensity image input, this property can be set to either a scalar intensity value for one shape, or an  $R$ -element vector where  $R$  specifies the number of shapes. For a color input image, you can set this property to:

A  $P$ -element vector where  $P$  is the number of color planes.

A  $P$ -by- $R$  matrix where  $P$  is the number of color planes and  $R$  is the number of shapes.

This property applies when you set the ??? property to **Custom**. This property is tunable when you set the ??? property to **false**.

Default: [200 255 100]

### **FillColorSource**

Source of fill color

Specify how to provide the shape's fill color as **Input port** or **Property**. This property applies when you set the ??? property to **true**, and you do not set the ??? property to **Lines**. When you set the ??? to **Input port**, you must provide a fill color vector as an input to the System object's **step** method.

Default: Property

### **FillColor**

Fill color of shape



Specify the intensity of the shading inside the shape as **Black**, **White**, or **Custom**. When you set this property to **Custom**, you can use the ??? property to specify the value. This property applies when set the ??? property to **Property**.

Default: **Black**

### **CustomFillColor**

Intensity or color value for shape's interior

Specify an intensity or color value for the shape's interior. For an intensity input image, you can set this property to a scalar intensity value for one shape, or an  $R$ -element vector where  $R$  specifies the number of shapes. For a color input image, you can set this property to:

A  $P$ -element vector where  $P$  is the number of color planes.

A  $P$ -by- $R$  matrix where  $P$  is the number of color planes and  $R$  is the number of shapes.

This property applies when you set the ??? property to **Custom**. This property is tunable when you set the ??? property to **false**.

Default: [200 255 100]

### **Opacity**

Opacity of the shading inside shapes

Specify the opacity of the shading inside the shape by a scalar value between 0 and 1, where 0 is transparent and 1 is opaque. This property applies when you set the ??? property to **true**.

Default: 0.6

### **ROIInputPort**

Enable definition of area for drawing shapes via input

Set this property to **true** to define the area in which to draw the shapes via an input to the **step** method. Specify the area as a four-element vector, [ $r$   $c$  *height* *width*], where  $r$  and  $c$  are the row and column coordinates of the upper-left corner of the area, and *height* and *width* represent the height (in rows) and width (in columns) of the area. When you set the property to **false**, the entire image will be used as the area in which to draw.

Default: **false**

### **Antialiasing**

Enable performing smoothing algorithm on shapes

Set this property to `true` to perform a smoothing algorithm on the line, polygon, or circle. This property applies when you set the ??? property to `Lines`, `Polygons`, or `Circles`.

Default: `false`

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies when you set the ??? property to `true` and/or the ??? property to `true`.

Default: `Floor`

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` , or `Saturate`. This property applies when you set the ??? property to `true` and/or the ??? property to `true`.

Default: `Wrap`

#### **OpacityDataType**

Opacity word length

Specify the opacity fixed-point data type as `Same word length as input`, or `Custom`. This property applies when you set the ??? property to `true`.

Default: `Custom`

#### **CustomOpacityDataType**

Opacity word length

Specify the opacity fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and the `OpacityDataType` property to `Custom`.

Default: `numericType([], 16)`

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input`, or `Custom`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`, and the `???` property to `Custom`.

Default: `numericType(true, 32, 14)`

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as product`, `Same as first input`, or `Custom`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`.

Default: `Same as product`

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `true` and/or the `???` property to `true`, and the `AccumulatorDataType` property to `Custom`.

Default: `numericType([], 32, 14)`

## Methods

<code>clone</code>	Create shape inserter object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Draw specified shape on image

## Examples

### Draw a Black Rectangle in a Grayscale Input Image

Create the shape inserter object.

```
shapeInserter = vision.ShapeInserter;
```

Read the input image.

```
I = imread('cameraman.tif');
```

Define the rectangle dimensions as [x y width height].

```
rectangle = int32([10 10 30 30]);
```

Draw the rectangle and display the result.

```
J = step(shapeInserter, I, rectangle);  
imshow(J);
```

### Draw Two Yellow Circles in a Grayscale Input Image.

Class of yellow must match class of the input, I.

```
yellow = uint8([255 255 0]); % [R G B]; class of yellow must match class of I
```

Create the shape inserter object.

```
shapeInserter = vision.ShapeInserter('Shape','Circles','BorderColor','Custom','CustomB
```

Read the input image.

```
I = imread('cameraman.tif');
```

Define the circle dimensions

```
circles = int32([30 30 20; 80 80 25]); % [x1 y1 radius1;x2 y2 radius2]
```

Convert I to an RGB image.

```
RGB = repmat(I,[1,1,3]); % convert I to an RGB image
```

Draw the circles and display the result.

```
J = step(shapeInserter, RGB, circles);  
imshow(J);
```

### **Draw a Red Triangle in a Color Image**

Create a shape inserter object and read the image file.

```
shapeInserter = vision.ShapeInserter('Shape','Polygons','BorderColor','Custom','Custom  
I = imread('autumn.tif');
```

Define vertices which will form a triangle: [x1 y1 x2 y2 x3 y3].

```
polygon = int32([50 60 100 60 75 30]);
```

Draw the triangles and display the result.

```
J = step(shapeInserter, I, polygon);  
imshow(J);
```

### **See Also**

[insertMarker](#) | [insertShape](#) | [insertText](#)

**Introduced in R2012a**

# clone

**System object:** vision.ShapeInserter

**Package:** vision

Create shape inserter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.ShapeInserter

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

## getNumOutputs

**System object:** vision.ShapeInserter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.



# isLocked

**System object:** vision.ShapeInserter

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the ShapeInserter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.ShapeInserter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## step

**System object:** vision.ShapeInserter

**Package:** vision

Draw specified shape on image

## Syntax

`J = step(shapeInserter, I, PTS)`

`J = step(shapeInserter, I, PTS, ROI)`

`J = step(shapeInserter, I, PTS, ..., CLR)`

## Description

`J = step(shapeInserter, I, PTS)` draws the shape specified by the `???` property on input image `I`. The input `PTS` specify the coordinates for the location of the shape. The shapes are embedded on the output image `J`.

`J = step(shapeInserter, I, PTS, ROI)` draws a shape inside an area defined by the `ROI` input. This applies only when you set the `???` property to `true`. The `ROI` input defines a rectangular area as `[x y width height]`, where `[x y]` determine the upper-left corner location of the rectangle, and `width` and `height` specify the size.

`J = step(shapeInserter, I, PTS, ..., CLR)` draws the shape with the border or fill color specified by the input `CLR`. This applies when you set the `???` property or the `???` property to 'Input port'.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## Input Arguments

### **shapeInserter**

Shape inserter object with shape and properties specified.

### **I**

Input  $M$ -by- $N$  matrix of  $M$  intensity values or an  $M$ -by- $N$ -by- $P$  matrix of  $M$  color values where  $P$  is the number of color planes.

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integer
- 8-, 16-, and 32-bit unsigned integer

### **PTS**

Input matrix coordinates describing location and dimension of shape. This property must be an integer value. If you enter non-integer value, the object rounds it to the nearest integer.

You can specify the type of shape as **Rectangles**, **Lines**, **Polygons**, or **Circles**. When you specify the type of shape to draw, you must also specify the location. The PTS input specifies the location of the points. The table shows the format for the points input for the different shapes. For a more detailed explanation on how to specify shapes and lines, see “Draw Shapes and Lines”.

Depending on the shape you choose by setting the ??? property, the input matrix PTS must be in one of the following formats:

<b>Shape</b>	<b>Format of PTS</b>
Lines	$M$ -by-4 matrix of $M$ number of lines.

Shape	Format of PTS
	$\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$ <p>Each row of the matrix corresponds to a different line, and of the same form as the vector for a single line. <math>L</math> is the number of vertices.</p>
Rectangles	<p><math>M</math>-by-4 matrix of <math>M</math> number of rectangles.</p> $\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$ <p>Each row of the matrix corresponds to a different rectangle and of the same form as the vector for a single rectangle. The <math>[x \ y]</math> coordinates correspond to the upper-left corner of the rectangle with respect to the image origin. The width and height must be greater than zero.</p>

Shape	Format of PTS
Polygons	<p><math>M</math>-by-<math>2L</math> matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>Each row of the matrix corresponds to a different polygon and of the same form as the vector for a single polygon. <math>L</math> is the number of vertices.</p>
Circles	<p><math>M</math>-by-3 matrix</p> $\begin{bmatrix} x_1 & y_1 & radius_1 \\ x_2 & y_2 & radius_2 \\ \vdots & \vdots & \vdots \\ x_M & y_M & radius_M \end{bmatrix}$ <p>Each row of the matrix corresponds to a different circle and of the same form as the vector for a single circle.</p>

The table below shows the data types required for the inputs,  $I$  and  $PTS$ .

Input image $I$	Input matrix $PTS$
built-in integer	built-in or fixed-point integer
fixed-point integer	built-in or fixed-point integer
double	double, single, or build-in integer
single	double, single, or build-in integer

**RGB**

Scalar, vector, or matrix describing one plane of the RGB input video stream.

**ROI**

Input 4-element vector of integers [x y width height], that define a rectangular area in which to draw the shapes. The first two elements represent the one-based coordinates of the upper-left corner of the area. The second two elements represent the width and height of the area.

- Double-precision floating point
- Single-precision floating point
- 8-, 16-, and 32-bit signed integer
- 8-, 16-, and 32-bit unsigned integer

**CLR**

This port can be used to dynamically specify shape color.

$P$ -element vector or an  $M$ -by- $P$  matrix, where  $M$  is the number of shapes, and  $P$ , the number of color planes. You can specify a color (RGB), for each shape, or specify one color for all shapes. The data type for the CLR input must be the same as the input image.

## Output Arguments

**J**

Output image. The shapes are embedded on the output image.

# vision.StandardDeviation System object

**Package:** vision

Find standard deviation of input or sequence of inputs

## Description

The `StandardDeviation` object finds standard deviation of input or sequence of inputs.

## Construction

`H = vision.StandardDeviation` returns a System object, `H`, that computes the standard deviation of the entire input.

`H = vision.StandardDeviation(Name, Value)` returns a standard deviation System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as *(Name1, Value1, ..., NameN, ValueN)*.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### RunningStandardDeviation

Enable calculation over successive calls to step method

Set this property to `true` to enable the calculation of standard deviation over successive calls to the `step` method. The default is `false`.



**ResetInputPort**

Reset in running standard deviation mode

Set this property to `true` to enable resetting the running standard deviation. When the property is set to `true`, a reset input must be specified to the `step` method to reset the running standard deviation. This property applies when you set the `???` property to `true`. The default is `false`.

**ResetCondition**

Reset condition for running standard deviation mode

Specify the event to reset the running standard deviation to `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. This property applies when you set the `???` property to `true`. The default is `Non-zero`.

**Dimension**

Numerical dimension to operate along

Specify how the standard deviation calculation is performed over the data as `All`, `Row`, `Column`, or `Custom`. The default is `All`.

**CustomDimension**

Numerical dimension to operate along

Specify the dimension (one-based value) of the input signal, over which the standard deviation is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when you set the `???` property to `Custom`. The default is 1.

**ROIProcessing**

Enable region of interest processing

Set this property to `true` to enable calculating the standard deviation within a particular region of each image. This property applies when you set the `???` property to `All` and the `???` property to `false`. The default is `false`.

**ROIForm**

Type of region of interest

Specify the type of region of interest to **Rectangles**, **Lines**, **Label matrix**, or **Binary mask**. This property applies when you set the ??? property to **true**. The default is **Rectangles**.

### **ROIPortion**

Calculate over entire ROI or just perimeter

Specify the region over which to calculate the standard deviation to **Entire ROI**, or **ROI perimeter**. This property applies when you set the ??? property to **Rectangles**. The default is **Entire ROI**.

### **ROIStatistics**

Statistics for each ROI, or one for all ROIs

Specify what statistics to calculate as **Individual statistics for each ROI**, or **Single statistic for all ROIs**. This property does not apply when you set the ??? property to **Binary mask**. The default is **Individual statistics for each ROI**

### **ValidityOutputPort**

Output flag indicating if any part of ROI is outside input image

Set this property to **true** to return the validity of the specified ROI as completely or partially inside of the image. This applies when you set the ??? property to **Lines** or **Rectangles**.

Set this property to **true** to return the validity of the specified label numbers. This applies when you set the ??? property to **Label matrix**.

The default is **false**.

## **Methods**

clone	Create standard deviation object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
reset	Reset running standard deviation state
step	Compute standard deviation of input

## Examples

Determine the standard deviation in a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hstd2d = vision.StandardDeviation;  
std = step(hstd2d,img);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the [Standard Deviation](#) block reference page. The object properties correspond to the block parameters.

**Introduced in R2012a**

# clone

**System object:** vision.StandardDeviation

**Package:** vision

Create standard deviation object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.StandardDeviation

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.StandardDeviation

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

## isLocked

**System object:** vision.StandardDeviation

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the StandardDeviation System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.StandardDeviation

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



## reset

**System object:** vision.StandardDeviation

**Package:** vision

Reset running standard deviation state

## Syntax

reset(H)

## Description

reset(H) resets the internal states of System object *H*, used for computing running standard deviation when the `RunningStandardDeviation` property is `true`.

# stereoParameters class

Object for storing stereo camera system parameters

## Syntax

```
stereoParams = stereoParameters(cameraParameters1, cameraParameters2,  
rotationOfCamera2, translationOfCamera2)  
stereoParams = stereoParameters(paramStruct)
```

## Construction

`stereoParams = stereoParameters(cameraParameters1, cameraParameters2, rotationOfCamera2, translationOfCamera2)` returns an object that contains the parameters of a stereo camera system.

`stereoParams = stereoParameters(paramStruct)` returns a `stereoParameters` object containing the parameters specified by `paramStruct` input. `paramStruct` is returned by the `toStruct` method.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

Use the `toStruct` method to pass a `stereoParameters` object into generated code. See the “Code Generation for Depth Estimation From Stereo Video” example.

## Input Arguments

### **cameraParameters1** — Parameters of camera 1

`cameraParameters` object

Parameters of camera 1, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of camera 1.

**cameraParameters2 — Parameters of camera 2**

cameraParameters object

Parameters of camera 2, specified as a cameraParameters object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of camera 2.

**rotationOfCamera2 — Rotation of camera 2**

3-by-3 matrix

Rotation of camera 2 relative to camera 1, specified as a 3-by-3 matrix.

**translationOfCamera2 — Translation of camera 2**

3-element vector

Translation of camera 2 relative to camera 1, specified as a 3-element vector.

## Properties

**Intrinsic and extrinsic parameters of the two cameras:**

**CameraParameters1 — Parameters of camera 1**

cameraParameters object

Parameters of camera 1, specified as a cameraParameters object. The object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

**CameraParameters2 — Parameters of camera 2**

cameraParameters object

Parameters of camera 2, specified as a cameraParameters object. The object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

**Geometric relationship between the two cameras**

**RotationOfCamera2 — Rotation of camera 2**

3-by-3 matrix

Rotation of camera 2 relative to camera 1, specified as a 3-by-3 matrix.

**TranslationOfCamera2 — Translation of camera 2**

3-element vector

Translation of camera 2 relative to camera 1, specified as a 3-element vector.

### **FundamentalMatrix** — Fundamental matrix

3-by-3 matrix

Fundamental matrix, stored as a 3-by-3 matrix. The fundamental matrix relates the two stereo cameras, such that the following equation must be true:

$$[P_2 \ 1] * \textit{FundamentalMatrix} * [P_1 \ 1]' = 0$$

$P_1$ , the point in image 1 in pixels, corresponds to the point,  $P_2$ , in image 2.

### **EssentialMatrix** — Essential matrix

3-by-3 matrix

Essential matrix, stored as a 3-by-3 matrix. The essential matrix relates the two stereo cameras, such that the following equation must be true:

$$[P_2 \ 1] * \textit{EssentialMatrix} * [P_1 \ 1]' = 0$$

$P_1$ , the point in image 1, corresponds to  $P_2$ , the point in image 2. Both points are expressed in normalized image coordinates, where the origin is at the camera's optical center. The  $x$  and  $y$  pixel coordinates are normalized by the focal length  $f_x$  and  $f_y$ .

### **Accuracy of estimated parameters:**

#### **MeanReprojectionError** — Average Euclidean distance

number of pixels

Average Euclidean distance between reprojected points and detected points over all image pairs, specified in pixels.

### **Accuracy of estimated parameters:**

#### **NumPatterns** — Number of calibrated patterns

integer

Number of calibration patterns that estimate the extrinsics of the two cameras, stored as an integer.

#### **WorldPoints** — World coordinates

$M$ -by-2 array

World coordinates of key points in the calibration pattern, specified as an  $M$ -by-2 array.  $M$  represents the number of key points in the pattern.

### **WorldUnits** — World points units

'mm' (default) | string

World points units, specified as a string. The string describes the units of measure.

## Methods

`toStruct` Convert a stereo parameters object into a struct

## Output Arguments

### **stereoParams** — Stereo parameters

stereoParameters object

Stereo parameters, returned as a stereoParameters object. The object contains the parameters of the stereo camera system.

## Examples

### **Stereo Camera Calibration**

Specify calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');
leftImages = imageSet(fullfile(imageDir,'left'));
rightImages = imageSet(fullfile(imageDir,'right'));
images1 = leftImages.ImageLocation;
images2 = rightImages.ImageLocation;
```

Detect the checkerboards.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images1,images2);
```

Specify the world coordinates of the checkerboard keypoints.

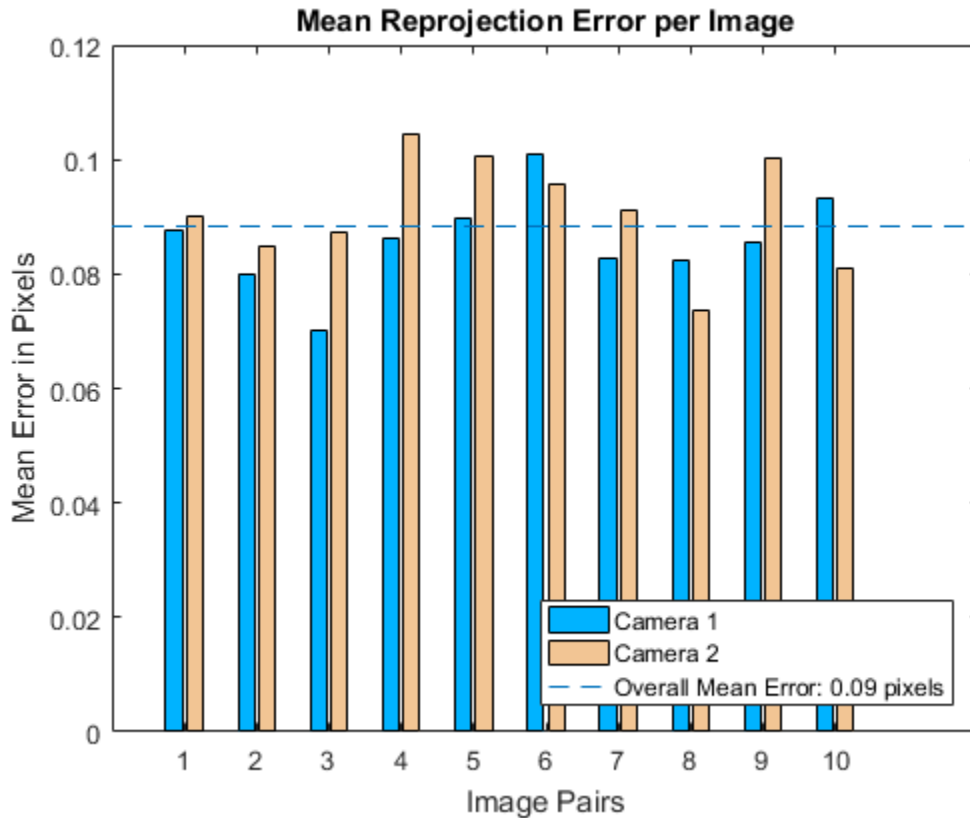
```
squareSizeInMM = 108;
worldPoints = generateCheckerboardPoints(boardSize,squareSizeInMM);
```

Calibrate the stereo camera system.

```
im = read(leftImages,1);  
params = estimateCameraParameters(imagePoints,worldPoints);
```

Visualize the calibration accuracy.

```
showReprojectionErrors(params);
```



## References

- [1] Zhang, Z. "A Flexible New Technique for Camera Calibration". *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

- [2] Heikkila, J, and O. Silven. “A Four-step Camera Calibration Procedure with Implicit Image Correction.” *IEEE International Conference on Computer Vision and Pattern Recognition*. 1997.

## See Also

cameraParameters | stereoCalibrationErrors | intrinsicsEstimationErrors | extrinsicsEstimationErrors | Camera Calibrator | detectCheckerboardPoints | estimateCameraParameters | estimateFundamentalMatrix | generateCheckerboardPoints | reconstructScene | rectifyStereoImages | showExtrinsics | showReprojectionErrors | Stereo Camera Calibrator | undistortImage | undistortPoints

## More About

- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Code Generation for Depth Estimation From Stereo Video”
- “Single Camera Calibration App”
- “Stereo Calibration App”

## Introduced in R2014a

# toStruct

**Class:** stereoParameters

Convert a stereo parameters object into a struct

## Syntax

```
paramStruct = toStruct(stereoParams)
```

## Description

`paramStruct = toStruct(stereoParams)` returns a struct containing the stereo parameters in the `stereoParams` input object. You can use the struct to create an identical `stereoParameters` object. Use the struct for C code generation. You can call `toStruct`, and then pass the resulting structure into the generated code, which re-creates the `stereoParameters` object.

## Input Arguments

**stereoParams** — Stereo parameters

`stereoParameters` object

Stereo parameters, specified as a `stereoParameters` object. The object contains the parameters of the stereo camera system.

## Output Arguments

**paramStruct** — Stereo parameters

struct

Stereo parameters, returned as a stereo parameters struct.

## Related Examples

- “Code Generation for Depth Estimation From Stereo Video”



**Introduced in R2015a**

# step

**System object:** vision.StandardDeviation

**Package:** vision

Compute standard deviation of input

## Syntax

```
Y = step(H,X)
Y = step(H,X,R)
Y = step(H,X,ROI)
Y = step(H,X,LABEL,LABELNUMBERS)
[Y, FLAG] = step(H,X,ROI)
[Y, FLAG] = step(H,X,LABEL,LABELNUMBERS)
```

## Description

$Y = \text{step}(H,X)$  computes the standard deviation of input elements  $X$ . When you set the `RunningStandardDeviation` property to `true`, the output  $Y$  corresponds to the standard deviation of the input elements over successive calls to the `step` method.

$Y = \text{step}(H,X,R)$  computes the standard deviation of the input elements over successive calls to the `step` method, and optionally resets its state based on the value of reset signal  $R$ , the `ResetInputPort` property and the `ResetCondition` property. This option applies when you set the `RunningStandardDeviation` property to `true` and the `ResetInputPort` property to `true`.

$Y = \text{step}(H,X,ROI)$  uses additional input  $ROI$  as the region of interest when you set the `ROIProcessing` property to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary` mask.

$Y = \text{step}(H,X,LABEL,LABELNUMBERS)$  computes the standard deviation of input image  $X$  for region labels contained in vector  $LABELNUMBERS$ , with matrix  $LABEL$  marking pixels of different regions. This option applies when you set the `ROIProcessing` property to `true` and the `ROIForm` property to `Label` matrix.

[Y, FLAG] = step(H,X,ROI) also returns output *FLAG*, which indicates whether the given region of interest is within the image bounds. This applies when you set the `ValidityOutputPort` property to `true`.

[Y, FLAG] = step(H,X,LABEL,LABELNUMBERS) also returns the output *FLAG* which indicates whether the input label numbers are valid when you set the `ValidityOutputPort` property to `true`.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## vision.TemplateMatcher System object

**Package:** vision

Locate template in image

### Description

The template matcher object locates a template in an image. Use the step syntax below with input image **I**, template **T**, template matcher object, **H**, and any optional properties.

**LOC = step(H,I,T)** computes the [x y] location coordinates, **LOC**, of the best template match between the image matrix, **I**, and the template matrix, **T**. The **step** method outputs the coordinates relative to the top left corner of the image. The **LOC** [x y] coordinates correspond to the center of the template. The object centers the location slightly different for templates with an odd or even number of pixels, (see the table below for details). The object computes the location by shifting the template in single-pixel increments throughout the interior of the image.

**METRIC = step(H,I,T)** computes the match metric values for input image, **I**, with **T** as the template. This applies when you set the **OutputValue** property to **Metric matrix**.

**LOC = step(H,I,T,ROI)** computes the location of the best template match, **LOC**, in the specified region of interest, **ROI**. This applies when you set the **OutputValue** property to **Best match location** and the **ROIInputPort** property to **true**. The input **ROI** must be a four element vector, [x y width height], where the first two elements represent the [x y] coordinates of the upper-left corner of the rectangular **ROI**.

**[LOC,ROIINVALID] = step(H,I,T,ROI)** computes the location of the best template match, **LOC**, in the specified region of interest, **ROI**. The **step** method also returns a logical flag **ROIINVALID** indicating if the specified **ROI** is outside the bounds of the input image **I**. This applies when you set the **OutputValue** property to **Best match location**, and both the **ROIInputPort** and **ROIValidityOutputPort** properties to **true**.

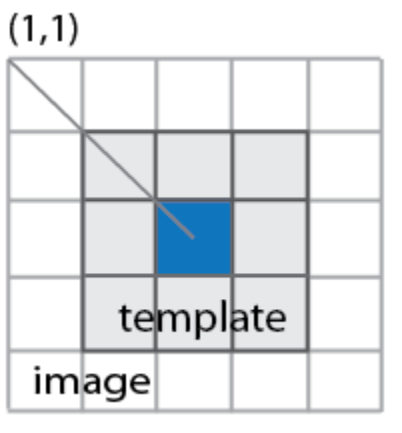
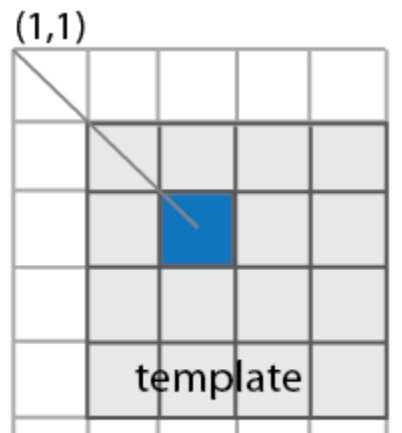
**[LOC,NVALS,NVALID] = step(H,I,T)** returns the location of the best template match **LOC**, the metric values around the best match **NVALS**, and a logical flag **NVALID**. A **false** value for **NVALID** indicates that the neighborhood around the best match extended outside the borders of the metric value matrix **NVALS**. This

applies when you set the `OutputValue` property to `Best match location` and the `BestMatchNeighborhoodOutputPort` property to `true`.

`[LOC,NVALS,NVALID,ROIVALID] = step(H,I,T,ROI)` returns the location of the best template match `LOC`, the metric values around the best match `NVALS`, and two logical flags, `NVALID` and `ROIVALID`. A `false` value for `NVALID` indicates that the neighborhood around the best match extended outside the borders of the metric value matrix `NVALS`. A `false` value for `ROIVALID` indicates that the specified `ROI` is outside the bounds of the input image `I`. This applies when you set the `OutputValue` property to `Best match location`, and the `BestMatchNeighborhoodOutputPort`, `ROIInputPort`, and `ROIValidityOutputPort` properties to `true`.

Typical use of the template matcher involves finding a small region within a larger image. The region is specified by the template image which can be as large as the input image, but which is typically smaller than the input image.

The object outputs the best match coordinates, relative to the top-left corner of the image. The `[x y]` coordinates of the location correspond to the center of the template. When you use a template with an odd number of pixels, the object uses the center of the template. When you use a template with an even number of pixels, the object uses the centered upper-left pixel for the location. The following table shows how the object outputs the location (`LOC`), of odd and even templates:

Odd number of pixels in template	Even number of pixels in template
	

## Construction

`H = vision.TemplateMatcher` returns a template matcher System object, `H`. This object performs template matching by shifting a template in single-pixel increments throughout the interior of an image.

`H = vision.TemplateMatcher(Name, Value)` returns a template matcher object, `H`, with each specified property set to the specified value.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### Metric

Metric used for template matching

Specify the metric to use for template matching as one of `Sum of absolute differences` | `Sum of squared differences` | `Maximum absolute difference`. The default is `Sum of absolute differences`.

### OutputValue

Type of output

Specify the output you want the object to return as one of `Metric matrix` | `Best match location`. The default is `Best match location`.

### SearchMethod

Specify search criteria to find minimum difference between two inputs

Specify how the object searches for the minimum difference between the two input matrices as `Exhaustive` or `Three-step`. If you set this property to `Exhaustive`, the object searches for the minimum difference pixel by pixel. If you set this property to `Three-step`, the object searches for the minimum difference using a steadily decreasing step size. The `Three-step` method is computationally less expensive than

the `Exhaustive` method, but sometimes does not find the optimal solution. This property applies when you set the `OutputValue` property to `Best match location`.

The default is `Exhaustive`.

### **BestMatchNeighborhoodOutputPort**

Enable metric values output

Set this property to `true` to return two outputs, *NMETRIC* and *NVALID*. The output *NMETRIC* denotes an  $N$ -by- $N$  matrix of metric values around the best match, where  $N$  is the value of the `???` property. The output *NVALID* is a logical indicating whether the object went beyond the metric matrix to construct output *NMETRIC*. This property applies when you set the `???` property to `Best match location`.

The default is `false`.

### **NeighborhoodSize**

Size of the metric values

Specify the size,  $N$ , of the  $N$ -by- $N$  matrix of metric values as an odd number. For example, if the matrix size is 3-by-3 set this property to `3`. This property applies when you set the `???` property to `Best match location` and the `???` property to `true`.

The default is `3`.

### **ROIInputPort**

Enable ROI specification through input

Set this property to `true` to define the Region of Interest (ROI) over which to perform the template matching. If you set this property to `true`, the ROI is specified using an input to the `step` method. Otherwise the entire input image is used.

The default is `false`.

### **ROIValidityOutputPort**

Enable output of a flag indicating if any part of ROI is outside input image

When you set this property to `true`, the object returns an ROI flag. The flag, when set to `false`, indicates a part of the ROI is outside of the input image. This property applies when you set the `???` property to `true`.

The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`.

#### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as first input`, `Custom`. This property applies when you set the `???` property to `Sum of squared differences`.

The default is `Custom`.

#### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Sum of squared differences`, and the `???` property to `Custom`.

The default is `numericType([ ], 32, 30)`.

#### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as first input` | `Custom`. The default is `Custom`.



### CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

The default is `numericType([],32,0)`.

### OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as first input` | `Custom`. The default is `Same as first input`. This property applies when you set the `???` property to `Metric matrix`. This property applies when you set the `???` property to `Best match location`, and the `???` property to `true`.

### CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `???` property to `Custom`.

The default is `numericType([],32,0)`.

## Methods

<code>clone</code>	Create template matcher object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and nontunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>step</code>	Finds the best template match within image

### Examples

Find the location of a particular chip on an image of an electronic board:

```
htm=vision.TemplateMatcher;
hmi = vision.MarkerInserter('Size', 10, ...
    'Fill', true, 'FillColor', 'White', 'Opacity', 0.75); I = imread('board.tif');

% Input image
I = I(1:200,1:200,:);

% Use grayscale data for the search
Igray = rgb2gray(I);

% Use a second similar chip as the template
T = Igray(20:75,90:135);

% Find the [x y] coordinates of the chip's center
Loc=step(htm,Igray,T);

% Mark the location on the image using white disc
J = step(hmi, I, Loc);

imshow(T); title('Template');
figure; imshow(J); title('Marked target');
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the [Template Matching](#) block reference page. The object properties correspond to the block parameters.

### See Also

[opticalFlowHS](#) | [opticalFlowLK](#) | [opticalFlowFarneback](#) | [vision.MarkerInserter](#) | [opticalFlowLKDoG](#)

**Introduced in R2012a**

# clone

**System object:** vision.TemplateMatcher

**Package:** vision

Create template matcher object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.TemplateMatcher

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of  $\text{getNumInputs}(H)$ .

# getNumOutputs

**System object:** vision.TemplateMatcher

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.TemplateMatcher

**Package:** vision

Locked status for input attributes and nontunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the TemplateMatcher System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

# release

**System object:** vision.TemplateMatcher

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# step

**System object:** vision.TemplateMatcher

**Package:** vision

Finds the best template match within image

## Syntax

LOC = step(H,I,T)

METRIC = step(H,I,T)

LOC = step(H,I,T,ROI)

[LOC,ROIINVALID] = step(H,I,T,ROI)

[LOC,NVALS,NVALID] = step(H,I,T)

[LOC,NVALS,NVALID,ROIINVALID] = step(H,I,T,ROI)

## Description

LOC = step(H,I,T) computes the [x y] location coordinates, LOC, of the best template match between the image matrix, I, and the template matrix, T. The step method outputs the coordinates relative to the top left corner of the image. The object computes the location by shifting the template in single-pixel increments throughout the interior of the image.

METRIC = step(H,I,T) computes the match metric values for input image, I, with T as the template. This applies when you set the **OutputValue** property to **Metric matrix**.

LOC = step(H,I,T,ROI) computes the location of the best template match, LOC, in the specified region of interest, ROI. This applies when you set the **OutputValue** property to **Best match location** and the **ROIInputPort** property to **true**. The input ROI must be a four element vector, [x y width height], where the first two elements represent the [x y] coordinates of the upper-left corner of the rectangular ROI.

[LOC,ROIINVALID] = step(H,I,T,ROI) computes the location of the best template match, LOC, in the specified region of interest, ROI. The step method also returns



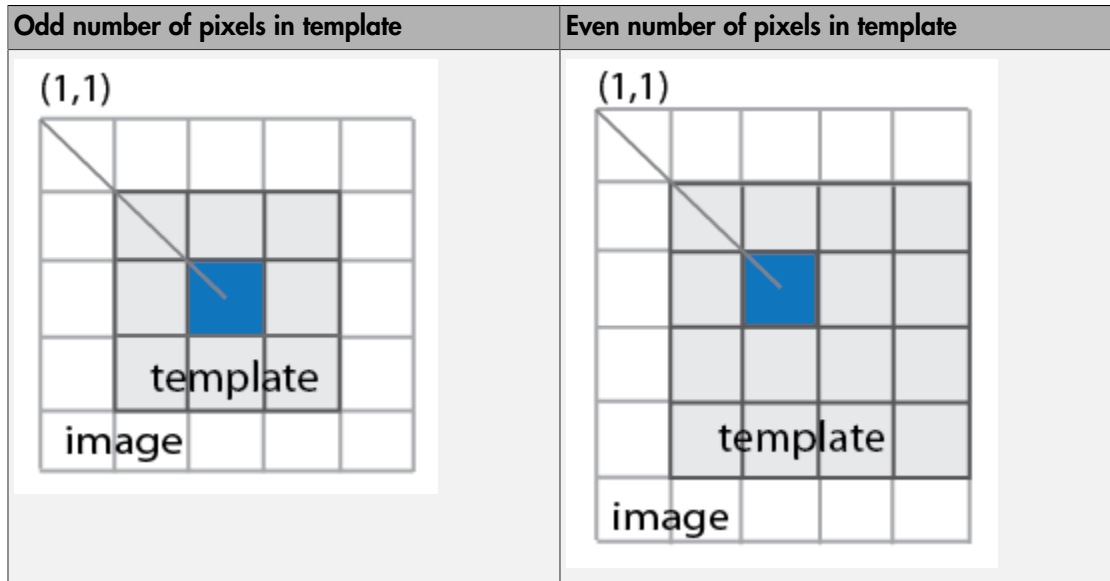
a logical flag `ROIVALID` indicating if the specified ROI is outside the bounds of the input image `I`. This applies when you set the `OutputValue` property to `Best match location`, and both the `ROIInputPort` and `ROIValidityOutputPort` properties to `true`.

`[LOC,NVALS,NVALID] = step(H,I,T)` returns the location of the best template match `LOC`, the metric values around the best match `NVALS`, and a logical flag `NVALID`. A `false` value for `NVALID` indicates that the neighborhood around the best match extended outside the borders of the metric value matrix `NVALS`. This applies when you set the `OutputValue` property to `Best match location` and the `BestMatchNeighborhoodOutputPort` property to `true`.

`[LOC,NVALS,NVALID,ROIVALID] = step(H,I,T,ROI)` returns the location of the best template match `LOC`, the metric values around the best match `NVALS`, and two logical flags, `NVALID` and `ROIVALID`. A `false` value for `NVALID` indicates that the neighborhood around the best match extended outside the borders of the metric value matrix `NVALS`. A `false` value for `ROIVALID` indicates that the specified ROI is outside the bounds of the input image `I`. This applies when you set the `OutputValue` property to `Best match location`, and the `BestMatchNeighborhoodOutputPort`, `ROIInputPort`, and `ROIValidityOutputPort` properties to `true`.

Typical use of the template matcher involves finding a small region within a larger image. The region is specified by the template image which can be as large as the input image, but which is typically smaller than the input image.

The object outputs the best match coordinates, relative to the top-left corner of the image. The `[x y]` coordinates of the location correspond to the center of the template. When you use a template with an odd number of pixels, the object uses the center of the template. When you use a template with an even number of pixels, the object uses the centered upper-left pixel for the location. The following table shows how the object outputs the location (`LOC`), of odd and even templates:



---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.TextInserter System object

**Package:** vision

Draw text on image or video stream

## Description

---

**Note:** The `vision.TextInserter` System object will be removed in a future release. Use the `insertText` function with equivalent functionality instead.

---

The text inserter System object draws text in an image. The output image can then be displayed or saved to a file. It can draw one or more arbitrary ASCII strings at multiple locations in an image. The object uses the FreeType 2.3.5 library, an open-source font engine, to produce stylized text bitmaps. To learn more about the FreeType Project, visit <http://www.freetype.org/>. The text inserter object does not support character sets other than ASCII.

## Construction

`txtInserter = vision.TextInserter('string')` returns a text inserter object, `txtInserter` with the input text provided in quotes added to the image. You can specify the location for the text, font style, and other text characteristics with the properties described below.

`txtInserter = vision.TextInserter(...,Name,Value)` returns a text inserter object, `txtInserter`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (*Name1*, *Value1*,...,*NameN*,*ValueN*).

### Code Generation Support

Supports MATLAB Function block: Yes

“System Objects in MATLAB Code Generation”

### Code Generation Support

“Code Generation Support, Usage Notes, and Limitations”.

### To insert text:

- 1 Define and set up your text inserter using the constructor.
- 2 Call the `step` method with the input image, `I`, the text inserter object, `txtInserter`, and any optional properties. See the syntax below for using the `step` method.

`J = step(txtInserter, I)` draws the text specified by the `???` property onto input image `I` and returns the modified image `J`. The input image `I`, can be a 2-D grayscale image or a truecolor RGB image.

`J = step(txtInserter, I, loc)` places the text at the `[x y]` coordinates specified by the `loc` input. This applies when you set the `???` property to `'Input port'`.

`J = step(txtInserter, I, vars)` uses the data in the input `vars` for variable substitution. This applies when the `Text` property contains ANSI C printf-style format specifications (`%d`, `%.2f`, etc.). The input `VARS` can be a scalar or a vector having length equal to the number of format specifiers in each element in the specified text string.

`J = step(txtInserter, I, cellidx)` draws the text string selected by the index `cellidx` input. This applies when you set the `???` property to a cell array of strings. For example, if you set the `Text` property to `{'str1','str2'}`, and the `cellidx` property to `1`, the object inserts the string `'str1'` in the image. Values of `cellidx` outside of a valid range result in no text drawn on the image.

`J = step(txtInserter, I, color)` sets the text intensity or color, specified by the `???` property. The `Color` property can be a scalar or 3-element vector. This property applies when you set the `???` property to `Input port`.

`J = step(txtInserter, I, opac)` sets the text opacity, specified by the `???` property. This property applies when you set the `???` property to `'Input port'`.

`J = step(txtInserter, I, cellidx, vars, color, loc, opac)` draws the text specified by the `???` property onto input image `I`, using the index `cellidx`. It also uses the text variable substitution data `vars`, and color value `color`. The object draws the text at location specified by `loc` with opacity `opac`. You can use any combination or all possible inputs. Properties must be set appropriately.

## Properties

### Text

Text string to draw on image or video stream

Specify the text string to be drawn on image or video stream as a single text string or a cell array of strings. The string(s) can include ANSI C printf-style format specifications, such as %d, %f, or %s.

### ColorSource

Source of intensity or color of text

Specify the intensity or color value of the text as `Property` or `Input port`. The default is `Property`.

### Color

Text color or intensity

Specify the intensity or color of the text as a scalar integer value or a 3-element vector respectively. Alternatively, if the `???` property is a cell array of  $N$  number of strings, specify a  $M$ -by-1 vector of intensity values or  $M$ -by-3 matrix of color values that correspond to each string. The default is `[0 0 0]`. This property applies when you set the `???` property to `Property`. This property is tunable.

### LocationSource

Source of text location

Specify the location of the text as `Property` or `Input port`. The default is `Property`.

### Location

Top-left corner of text bounding box

Specify the top-left corner of the text bounding box as a 2-element vector of integers, `[x y]`. The default is `[1 1]`. This property applies when you set the `???` property to `Property`. This property is tunable.

### **OpacitySource**

Source of opacity of text

Specify the opacity of the text as `Property` or `Input port`. The default is `Property`.

### **Opacity**

Opacity of text

Specify the opacity of the text as numeric scalar between 0 and 1. This property applies when you set the `???` property to `Property`. The default is 1. This property is tunable.

### **TransposedInput**

Specifies if input image data order is row major

Set this property to `true` to indicate that the input image data order is row major.

The default is `false`.

### **Font**

Font face of text

Specify the font of the text as one of the available truetype fonts installed on your system. To get a list of available fonts, use TAB completion from the command prompt:

```
txtInserter = vision.TextInserter;  
txtInserter.Font = '<Press TAB to get a list>
```

### **FontSize**

Font size in points

Specify the font size as any positive integer value. The default is 12.

### **Antialiasing**

Perform smoothing algorithm on text edges

Set this property to `true` to smooth the edges of the text. The default is `true`.

## Methods

clone	Create text inserter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Draws the specified text onto input image

## Examples

### Display a Text String in an Image

```
textColor    = [255, 255, 255]; % [red, green, blue]
textLocation = [100 315];      % [x y] coordinates
textInserter = vision.TextInserter('Peppers are good for you!', 'Color', textColor, 'F...
I = imread('peppers.png');
J = step(textInserter, I);
imshow(J);
```

Warning: The vision.TextInserter will be removed in a future release. Use the insertText function with equivalent functionality instead.



### Display Four Numeric Values on an Image at Different Locations

Set up the text using the text inserter object.

```
textInserter = vision.TextInserter('%d', 'LocationSource', 'Input port', 'Color', [255
```

Warning: The `vision.TextInserter` will be removed in a future release. Use the `insertText` function with equivalent functionality instead.

Read the image.

```
I = imread('peppers.png');
```



**Insert the text.**

```
J = step(textInserter, I, int32([1 2 3 4]), int32([1 1; 500 1; 1 350; 500 350]));
```

**Display the image containing the numeric values.**

```
imshow(J);
```



**Display Two Strings at Different Locations**

**Set up the text inserter object.**

```
textInserter = vision.TextInserter('%s', 'LocationSource', 'Input port', 'Color', [255
```

Warning: The `vision.TextInserter` will be removed in a future release. Use the `insertText` function with equivalent functionality instead.

**Read the image.**

```
I = imread('peppers.png');
```

**Create null separated strings.**

```
strings = uint8(['left' 0 'right']);
```

**Insert the text.**

```
J = step(textInserter, I, strings, int32([10 1; 450 1]));
```

**Display the image with numeric values.**

```
imshow(J);
```



### See Also

[vision.ShapeInserter](#) | [vision.AlphaBlender](#) | [insertText](#) | [vision.MarkerInserter](#)

**Introduced in R2012a**

# clone

**System object:** vision.TextInserter

**Package:** vision

Create text inserter object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.TextInserter

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.TextInserter

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs,  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.TextInserter

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the TextInserter System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## release

**System object:** vision.TextInserter

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---



## step

**System object:** vision.TextInserter

**Package:** vision

Draws the specified text onto input image

## Syntax

`TXTIMG = step(H, IMG)`

`TXTIMG = step(H, IMG)`

`TXTIMG = step(H, IMG, CELLIDX)`

`TXTIMG = step(H, IMG, VARS)`

`TXTIMG = step(H, IMG, COLOR)`

`TXTIMG = step(H, IMG, LOC)`

`TXTIMG = step(H, IMG, OPAC)`

`TXTIMG = step(H, IMG, CELLIDX, VARS, COLOR, LOC, OPAC)`

## Description

`TXTIMG = step(H, IMG)` draws the specified text onto input image *IMG* and returns the modified image *Y*. The image *IMG* can either be an *M*-by-*N* matrix of intensity values or an *M*-by-*N*-by-*P* array color video signal where *P* is the number of color planes.

`TXTIMG = step(H, IMG)` draws the specified text onto input image *IMG* and returns the modified image *TXTIMG*. The image *IMG* can either be an *M*-by-*N* matrix of intensity values or an *M*-by-*N*-by-*P* array color video signal where *P* is the number of color planes.

`TXTIMG = step(H, IMG, CELLIDX)` draws the text string selected by index, *CELLIDX*, when you set the **Text** property to a cell array of strings. For example, if you set the **Text** property to {'str1','str2'}, and the **CELLIDX** property to 1, the **step** method inserts the string 'str1' in the image. Values of *CELLIDX* outside of valid range result in no text drawn on the image.

`TXTIMG = step(H, IMG, VARS)` uses the data in *VARS* for variable substitution, when the **Text** property contains ANSI C printf-style format specifications (%d, %.2f, etc.). *VARS* is a scalar or a vector having length equal to the number of format specifiers in each element in the specified text string.

`TXTIMG = step(H, IMG, COLOR)` uses the given scalar or 3-element vector `COLOR` for the text intensity or color respectively, when you set the `ColorSource` property to `Input port`.

`TXTIMG = step(H, IMG, LOC)` places the text at the [x y] coordinates specified by `LOC`. This applies when you set the `LocationSource` property to `Input port`.

`TXTIMG = step(H, IMG, OPAC)` uses `OPAC` for the text opacity when you set the `OpacitySource` property to set to `Input port`.

`TXTIMG = step(H, IMG, CELLIDX, VARS, COLOR, LOC, OPAC)` draws the specified text onto image `IMG` using text string selection index `CELLIDX`, text variable substitution data `VARs`, intensity or color value `COLOR` at location `LOC` with opacity `OPAC`. You can use any combination or all possible inputs. Properties must be set appropriately.

---

**Note:** `H` specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# vision.Variance System object

**Package:** vision

Find variance values in an input or sequence of inputs

## Description

The `Variance` object finds variance values in an input or sequence of inputs.

## Construction

`H = vision.Variance` returns a System object, `H`, that computes the variance of an input or a sequence of inputs.

`H = vision.Variance(Name, Value)` returns a variance System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

Code Generation Support
Supports MATLAB Function block: Yes
“System Objects in MATLAB Code Generation”.
“Code Generation Support, Usage Notes, and Limitations”.

## Properties

### RunningVariance

Enable calculation over successive calls to `step` method

Set this property to `true` to enable the calculation of the variance over successive calls to the `step` method. The default is `false`.

### ResetInputPort

Enable resetting via an input in running variance mode

Set this property to `true` to enable resetting the running variance. When the property is set to `true`, a reset input must be specified to the `step` method to reset the running variance. This property applies when you set the `???` property to `true`. The default is `false`.

### **ResetCondition**

Reset condition for running variance mode

Specify the event to reset the running variance as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. This property applies when you set the `???` property to `true`. The default is `Non-zero`.

### **CustomDimension**

Numerical dimension to operate along

Specify the dimension (one-based value) of the input signal, over which the variance is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when you set the `Dimension` property to `Custom`. The default is `1`.

### **Dimension**

Numerical dimension to operate along

Specify how the variance calculation is performed over the data as `All`, `Row`, `Column`, or `Custom`. This property applies only when you set the `???` property to `false`. The default is `All`.

### **ROIForm**

Type of region of interest

Specify the type of region of interest as `Rectangles`, `Lines`, `Label matrix`, or `Binary mask`. This property applies when you set the `???` property to `true`. Full ROI processing support requires a Computer Vision System Toolbox license. If you only have the DSP System Toolbox license, the `ROIForm` property value options are limited to `Rectangles`. The default is `Rectangles`.

### **ROIPortion**

Calculate over entire ROI or just perimeter

Specify the region over which to calculate the variance as `Entire ROI`, or `ROI perimeter`. This property applies when you set the `???` property to `Rectangles`. The default is `Entire ROI`.

### **ROIProcessing**

Enable region of interest processing

Set this property to `true` to enable calculating the variance within a particular region of each image. This property applies when you set the `???` property to `All` and the `???` property to `false`. Full ROI processing support requires a Computer Vision System Toolbox license. If you only have the DSP System Toolbox license, the `???` property value options are limited to `Rectangles`. The default is `false`.

### **ROIStatistics**

Statistics for each ROI, or one for all ROIs

Specify what statistics to calculate as `Individual statistics for each ROI`, or `Single statistic for all ROIs`. This property does not apply when you set the `???` property to `Binary mask`. The default is `Individual statistics for each ROI`.

### **ValidityOutputPort**

Output flag indicating if any part of ROI is outside input image

Set this property to `true` to return the validity of the specified ROI as completely or partially inside of the image. This applies when you set the `???` property to `Lines` or `Rectangles`.

Set this property to `true` to return the validity of the specified label numbers. This applies when you set the `???` property to `Label matrix`.

The default is `false`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

### **InputSquaredProductDataType**

Input squared product and fraction lengths

Specify the input-squared product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **CustomInputSquaredProductDataType**

Input squared product word and fraction lengths

Specify the input-squared product fixed-point type as a scaled `numericType` object. This property applies when you set the `InputSquaredProductDataType` property to `Custom`. The default is `numericType(true, 32, 15)`.

### **InputSumSquaredProductDataType**

Input-sum-squared product and fraction lengths

Specify the input-sum-squared product fixed-point data type as `Same as input-squared product` or `Custom`. The default is `Same as input-squared product`.

### **CustomInputSumSquaredProductDataType**

Input sum-squared product and fraction lengths

Specify the input-sum-squared product fixed-point type as a scaled `numericType` object. This property applies when you set the `InputSumSquaredProductDataType` property to `Custom`. The default is `numericType(true, 32, 23)`.

### **AccumulatorDataType**

Data type of the accumulator

Specify the accumulator fixed-point data type as `Same as input`, or `Custom`. The default is `Same as input`.

### CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object. This property applies when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

### OutputDataType

Data type of output

Specify the output fixed-point data type as `Same as accumulator`, `Same as input`, or `Custom`. The default is `Same as accumulator`.

### CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object.

This property applies when you set the `???` property to `Custom`. The default is `numericType(true,32,30)`.

## Methods

<code>clone</code>	Create variance object with same property values
<code>getNumInputs</code>	Number of expected inputs to step method
<code>getNumOutputs</code>	Number of outputs from step method
<code>isLocked</code>	Locked status for input attributes and non-tunable properties
<code>release</code>	Allow property value and input characteristics changes
<code>reset</code>	Reset the internal states of the variance object
<code>step</code>	Compute variance of input

### Examples

Determine the variance in a grayscale image.

```
img = im2single(rgb2gray(imread('peppers.png')));  
hvar2d = vision.Variance;  
var2d = step(hvar2d,img);
```

**Introduced in R2012a**



# clone

**System object:** vision.Variance

**Package:** vision

Create variance object with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

## getNumInputs

**System object:** vision.Variance

**Package:** vision

Number of expected inputs to step method

### Syntax

$N = \text{getNumInputs}(H)$

### Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method.

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

# getNumOutputs

**System object:** vision.Variance

**Package:** vision

Number of outputs from step method

## Syntax

$N = \text{getNumOutputs}(H)$

## Description

$N = \text{getNumOutputs}(H)$  returns the number of outputs  $N$  for the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.Variance

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the **Variance** System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute **step**. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a **true** value.

# release

**System object:** vision.Variance

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## **reset**

**System object:** vision.Variance

**Package:** vision

Reset the internal states of the variance object

## **Syntax**

reset(H)

## **Description**

reset(H) resets the internal states of System object H to their initial values.

## step

**System object:** vision.Variance

**Package:** vision

Compute variance of input

## Syntax

```
Y = step(H,X)
Y = step(H,X,R)
Y = step(H,X,ROI)
Y = step(H,X,LABEL,LABELNUMBERS)
[Y,FLAG] = step(H,X,ROI)
[Y,FLAG] = step(H,X,LABEL,LABELNUMBERS)
```

## Description

`Y = step(H,X)` computes the variance of input *X*. When you set the `RunningVariance` property to `true`, the output *Y* corresponds to the standard deviation of the input elements over successive calls to the `step` method.

`Y = step(H,X,R)` computes the variance of the input elements *X* over successive calls to the `step` method, and optionally resets its state based on the value of the reset signal *R*, the `ResetInputPort` property and the `ResetCondition` property. This option applies when you set the `RunningVariance` property to `true` and the `ResetInputPort` to `true`.

`Y = step(H,X,ROI)` computes the variance of input image *X* within the given region of interest *ROI* when you set the `ROIProcessing` property to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

`Y = step(H,X,LABEL,LABELNUMBERS)` computes the variance of input image *X* for region labels contained in vector *LABELNUMBERS*, with matrix *LABEL* marking pixels of different regions. This option applies when you set the `ROIProcessing` property to `true` and the `ROIForm` property to `Label matrix`.

[Y,FLAG] = step(H,X,ROI) also returns the output *FLAG*, which indicates whether the given region of interest is within the image bounds. This applies when you set both the `ROIProcessing` and the `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

[Y,FLAG] = step(H,X,LABEL,LABELNUMBERS) also returns the *FLAG*, which indicates whether the input label numbers are valid. This applies when you set both the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Label matrix`.

---

**Note:** H specifies the System object on which to run this `step` method.

The object performs an 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. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# vision.VideoPlayer System object

**Package:** vision

Play video or display image

## Description

The `VideoPlayer` object can play a video or display image sequences.

---

**Note:** If you own the MATLAB Coder product, you can generate C or C++ code from MATLAB code in which an instance of this system object is created. When you do so, the scope system object is automatically declared as an *extrinsic* variable. In this manner, you are able to see the scope display in the same way that you would see a figure using the `plot` function, without directly generating code from it. For the full list of system objects supporting code generation, see “Code Generation Support, Usage Notes, and Limitations” in the MATLAB Coder documentation.

---

## Construction

`videoPlayer = vision.VideoPlayer` returns a video player object, `videoPlayer`, for displaying video frames. Each call to the `step` method displays the next video frame.

`videoPlayer = vision.VideoPlayer(Name, Value)` configures the video player properties, specified as one or more name-value pair arguments. Unspecified properties have default values.

### To display video frames:

- 1 Define and set up your video player object using the constructor.
- 2 Call the `step` method with the video player object, `VideoPlayer`, and any optional properties. See the syntax below for using the `step` method.

`step(videoPlayer, I)` displays one grayscale or truecolor RGB video frame, `I`, in the video player.

### Properties

#### Name

Caption display on video player window

Specify the caption to display on the video player window as a string.

Default: Video

#### Position

Size and position of the video player window in pixels

Specify the size and position of the video player window in pixels as a four-element vector of the form: [left bottom width height]. This property is tunable.

Default: Dependent on the screen resolution. Window positioned in the center of the screen with size of 410 pixels in width by 300 pixels in height.

### Methods

clone	Create video player with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and non-tunable properties
isOpen	Visible or hidden status for video player figure
release	Allow property value and input characteristics changes
reset	Reset displayed frame number to zero
step	Play video or image sequence
show	Turn figure visibility on
hide	Turn figure visibility off

## Examples

### Play a Video File

Read video from a file and set up player object.

```
videoFReader = vision.VideoFileReader('viplanedeparture.avi');  
videoPlayer = vision.VideoPlayer;
```

Play video. Every call to the `step` method reads another frame.

```
while ~isDone(videoFReader)  
    frame = step(videoFReader);  
    step(videoPlayer, frame);  
end
```

Close the file reader and video player.

```
release(videoFReader);  
release(videoPlayer);
```

- “Face Detection and Tracking Using CAMShift”
- “Face Detection and Tracking Using the KLT Algorithm”
- “Face Detection and Tracking Using Live Video Acquisition”
- “Video Display in a Custom User Interface”

### See Also

| [vision.DeployableVideoPlayer](#) | [vision.VideoFileReader](#) | [vision.VideoFileWriter](#) | [imshow](#) | [imshow](#)

**Introduced in R2012a**

# clone

**System object:** vision.VideoPlayer

**Package:** vision

Create video player with same property values

## Syntax

```
C = clone(sysObj)
```

## Description

`C = clone(sysObj)` creates another instance of the System object, `sysObj`, with the same property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

# getNumInputs

**System object:** vision.VideoPlayer

**Package:** vision

Number of expected inputs to step method

## Syntax

$N = \text{getNumInputs}(H)$

## Description

$N = \text{getNumInputs}(H)$  returns the number of expected inputs,  $N$  to the **step** method

The **getNumInputs** method returns a positive integer that is the number of expected inputs (not counting the object itself) to the **step** method. This value will change if you alter any properties that turn inputs on or off. You must call the **step** method with the number of input arguments equal to the result of **getNumInputs** ( $H$ ).

## getNumOutputs

**System object:** vision.VideoPlayer

**Package:** vision

Number of outputs from step method

### Syntax

$N = \text{getNumOutputs}(H)$

### Description

$N = \text{getNumOutputs}(H)$  returns the number of arguments  $N$  from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

# isLocked

**System object:** vision.VideoPlayer

**Package:** vision

Locked status for input attributes and non-tunable properties

## Syntax

TF = isLocked(H)

## Description

TF = isLocked(H) returns the locked status, TF of the VideoPlayer System object.

isLocked returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, isLocked returns a `true` value.

## isOpen

**System object:** vision.VideoPlayer

**Package:** vision

Visible or hidden status for video player figure

## Syntax

isOpen(h)

## Description

isOpen(h) returns the visible or hidden status, as a logical, for the video player window. This method is not supported in code generation.



# release

**System object:** vision.VideoPlayer

**Package:** vision

Allow property value and input characteristics changes

## Syntax

release(H)

## Description

release(H) releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

---

**Note:** You can use the **release** method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

## **reset**

**System object:** vision.VideoPlayer

**Package:** vision

Reset displayed frame number to zero

## **Syntax**

reset(H)

## **Description**

reset(H) resets the displayed frame number of the video player to zero.

## step

**System object:** vision.VideoPlayer

**Package:** vision

Play video or image sequence

## Syntax

```
step(videoPlayer,I)
```

## Description

`step(videoPlayer,I)` displays one grayscale or truecolor RGB video frame,`I`, in the video player.

## show

**System object:** vision.VideoPlayer

**Package:** vision

Turn figure visibility on

## Syntax

show(H)

## Description

show(H) turns video player figure visibility on.

# hide

**System object:** vision.VideoPlayer

**Package:** vision

Turn figure visibility off

## Syntax

hide(H)

## Description

hide(H) turns video player figure visibility off.

## matlab.System class

**Package:** matlab

Base class for System objects

### Description

`matlab.System` is the base class for System objects. In your class definition file, you must subclass your object from this base class (or from another class that derives from this base class). Subclassing allows you to use the implementation and service methods provided by this base class to build your object. Type this syntax as the first line of your class definition file to directly inherit from the `matlab.System` base class, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System
```

---

**Note:** You must set `Access = protected` for each `matlab.System` method you use in your code.

---

### Methods

<code>allowModelReferenceDiscreteSampleTimeInheritanceImpl</code>	Model reference sample time inheritance status for discrete sample times
<code>getDiscreteStateImpl</code>	Discrete state property values
<code>getInputNamesImpl</code>	Names of System block input ports
<code>getNumInputsImpl</code>	Number of inputs to step method
<code>getNumOutputsImpl</code>	Number of outputs from <code>step</code> method
<code>getOutputNamesImpl</code>	Names of System block output ports
<code>getPropertyGroupsImpl</code>	Property groups for System object display
<code>getSimulateUsingImpl</code>	Specify value for Simulate using parameter
<code>infoImpl</code>	Information about System object
<code>isInactivePropertyImpl</code>	Inactive property status

<code>isInputSizeLockedImpl</code>	Locked input size status
<code>loadObjectImpl</code>	Load System object from MAT file
<code>processTunedPropertiesImpl</code>	Action when tunable properties change
<code>releaseImpl</code>	Release resources
<code>resetImpl</code>	Reset System object states
<code>saveObjectImpl</code>	Save System object in MAT file
<code>setProperty</code>	Set property values using name-value pairs
<code>setupImpl</code>	Initialize System object
<code>showFiSettingsImpl</code>	Fixed point data type tab visibility for System objects
<code>showSimulateUsingImpl</code>	Simulate Using visibility
<code>stepImpl</code>	System output and state update equations
<code>supportsMultipleInstanceImpl</code>	Support System object in Simulink For Each subsystem
<code>validateInputsImpl</code>	Validate inputs to step method
<code>validatePropertiesImpl</code>	Validate property values

## Attributes

In addition to the attributes available for MATLAB objects, you can apply the following attributes to any property of a custom System object.

<b>Nontunable</b>	After an object is locked (after <code>step</code> or <code>setup</code> has been called), use <b>Nontunable</b> to prevent a user from changing that property value. By default, all properties are tunable. The <b>Nontunable</b> attribute is useful to lock a property that has side effects when changed. This attribute is also useful for locking a property value assumed to be constant during processing. You should always specify properties that affect the number of input or output ports as <b>Nontunable</b> .
<b>Logical</b>	Use <b>Logical</b> to limit the property value to a logical, scalar value. Any scalar value that can be converted to a logical is also valid, such as 0 or 1.

PositiveInteger	Use <code>PositiveInteger</code> to limit the property value to a positive integer value.
DiscreteState	Use <code>DiscreteState</code> to mark a property so it will display its state value when you use the <code>getDiscreteState</code> method.

To learn more about attributes, see “Property Attributes” in the MATLAB Object-Oriented Programming documentation.

## Examples

### Create a Basic System Object

Create a simple System object, `AddOne`, which subclasses from `matlab.System`. You place this code into a MATLAB file, `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access = protected)
        % stepImpl method is called by the step method.
        function y = stepImpl(~,x)
            y = x + 1;
        end
    end
end
```

Use this object by creating an instance of `AddOne`, providing an input, and using the `step` method.

```
hAdder = AddOne;
x = 1;
y = step(hAdder,x)
```

Assign the `Nontunable` attribute to the `InitialValue` property, which you define in your class definition file.

```
properties (Nontunable)
    InitialValue
end
```

### See Also

`matlab.system.StringSet` | `matlab.system.mixin.FiniteSource`



## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Method Attributes”
- “Define Basic System Objects”
- “Define Property Attributes”

# allowModelReferenceDiscreteSampleTimeInheritanceImpl

**Class:** matlab.System

**Package:** matlab

Model reference sample time inheritance status for discrete sample times

## Syntax

```
flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
```

## Description

```
flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
```

indicates whether a System object in a referenced model is allowed to inherit the sample time of the parent model. Use this method only for System objects that use discrete sample time and are intended for inclusion in Simulink via the MATLAB System block.

During model compilation, Simulink sets the model reference sample time inheritance before the System object `setupImpl` method is called.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

**flag**

Flag indicating whether model reference discrete sample time inheritance is allowed for the MATLAB System block containing the System object, returned as a logical value.

The default value for this argument depends on the number of inputs to the System object. To use the default value, you do not need to include the this method in your System object class definition file.

Number of System object Inputs	Default Value and Override Effects
No inputs	<b>Default: false</b> — Model reference discrete sample time inheritance is not allowed. If your System object uses discrete sample time in its algorithm, override the default by returning <b>true</b> from <code>allowModelReferenceDiscreteSampleTimeInheritanceImpl</code> .
One or more inputs	<b>Default: true</b> — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed. If your System object does not use sample time in its algorithm, override the default by returning <b>false</b> from <code>allowModelReferenceDiscreteSampleTimeInheritanceImpl</code> .

## Examples

### Set Sample Time Inheritance for System Object

For a System object that has one or more inputs, to disallow model reference discrete sample time inheritance for that object, set the sample time inheritance to **false**. Include this code in your class definition file for the object.

```
methods (Access = protected)
    function flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(~)
        flag = false;
    end
end
```

### See Also

matlab.System

### How To

- “Set Model Reference Discrete Sample Time Inheritance”
- “Overview of Model Referencing”

- “Inherit Sample Times”

# getDiscreteStateImpl

**Class:** matlab.System

**Package:** matlab

Discrete state property values

## Syntax

```
s = getDiscreteStateImpl(obj)
```

## Description

`s = getDiscreteStateImpl(obj)` returns a struct `s` of state values. The field names of the struct are the object's `DiscreteState` property names. To restrict or change the values returned by `getDiscreteState` method, you can override this `getDiscreteStateImpl` method.

`getDiscreteStatesImpl` is called by the `getDiscreteState` method, which is called by the `setup` method.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

**s**

State values, returned as a struct

# Examples

## Get Discrete State Values

Use the `getDiscreteStateImpl` method in your class definition file to get the discrete states of the object.

```
methods (Access = protected)
  function s = getDiscreteStateImpl(obj)
  end
end
```

## See Also

`setupImpl`

## How To

- “Define Property Attributes”

# getInputNamesImpl

**Class:** matlab.System

**Package:** matlab

Names of System block input ports

## Syntax

```
[name1,name2,...] = getInputNamesImpl(obj)
```

## Description

[name1,name2,...] = getInputNamesImpl(obj) returns the names of the input ports to System object, obj implemented in a MATLAB System block. The number of returned input names matches the number of inputs returned by the getNumInputs method. If you change a property value that changes the number of inputs, the names of those inputs also change.

getInputNamesImpl is called by the getInputNames method by the MATLAB System block.

---

**Note:** You must set Access = protected for this method.

---

## Input Arguments

**obj**

System object

## Output Arguments

**name1,name2,...**

Names of the inputs for the specified object, returned as strings

**Default:** empty string

## Examples

### Specify Input Port Name

Specify in your class definition file the names of two input ports as 'upper' and 'lower'.

```
methods (Access = protected)
    function varargout = getInputNamesImpl(obj)
        numInputs = getNumInputs(obj);
        varargout = cell(1,numInputs);
        varargout{1} = 'upper';
        if numInputs > 1
            varargout{2} = 'lower';
        end
    end
end
```

### See Also

[getNumInputsImpl](#) | [getOutputNamesImpl](#)

### How To

- “Validate Property and Input Values”



# getNumInputsImpl

**Class:** matlab.System

**Package:** matlab

Number of inputs to step method

## Syntax

```
num = getNumInputsImpl(obj)
```

## Description

`num = getNumInputsImpl(obj)` returns the number of inputs `num` expected by the `step` method. The `System` object input argument is not included in the count. For example, if your `step` method syntax is `step(h_obj,x1,x2,x3)`, `getNumInputs` returns 3.

If your `step` method has a variable number of inputs (uses `varargin`), implement the `getNumInputsImpl` method in your class definition file.

If the number of inputs expected by the `step` method is fixed (does not use `varargin`), the default `getNumInputsImpl` determines the required number of inputs directly from the `step` method. In this case, you do not need to include `getNumInputsImpl` in your class definition file.

`getNumInputsImpl` is called by the `getNumInputs` method and by the `setup` method if the number of inputs has not been determined already.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

---

### Input Arguments

**obj**

System object

### Output Arguments

**num**

Number of inputs expected by the `step` method for the specified object, returned as an integer.

**Default:** 1

### Examples

#### Set Number of Inputs

Specify the number of inputs (2, in this case) expected by the `step` method.

```
methods (Access = protected)
    function num = getNumInputsImpl(~)
        num = 2;
    end
end
```

#### Set Number of Inputs to Zero

Specify that the `step` method does not accept any inputs.

```
methods (Access = protected)
    function num = getNumInputsImpl(~)
        num = 0;
    end
end
```

### See Also

`setupImpl` | `stepImpl` | `getNumOutputsImpl`

## How To

- “Change Number of Step Inputs or Outputs”

# getNumOutputsImpl

**Class:** matlab.System

**Package:** matlab

Number of outputs from `step` method

## Syntax

```
num = getNumOutputsImpl (obj)
```

## Description

`num = getNumOutputsImpl (obj)` returns the number of outputs from the `step` method.

If your `step` method has a variable number of outputs (uses `varargout`), implement the `getNumOutputsImpl` method in your class definition file to determine the number of outputs. Use `nargout` in the `stepImpl` method to assign the expected number of outputs.

If the number of outputs expected by the `step` method is fixed (does not use `varargout`), the object determines the required number of outputs from the `step` method. In this case, you do not need to implement the `getNumOutputsImpl` method.

`getNumOutputsImpl` is called by the `getNumOutputs` method, if the number of outputs has not been determined already.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

---

## Input Arguments

**obj**

System object

## Output Arguments

**num**

Number of outputs from the `step` method for the specified object, returned as an integer.

## Examples

### Set Number of Outputs

Specify the number of outputs (2, in this case) returned from the `step` method.

```
methods (Access = protected)
    function num = getNumOutputsImpl(~)
        num = 2;
    end
end
```

### Set Number of Outputs to Zero

Specify that the `step` method does not return any outputs.

```
methods (Access = protected)
    function num = getNumOutputsImpl(~)
        num = 0;
    end
end
```

### Use `nargout` for Variable Number of Outputs

Use `nargout` in the `stepImpl` method when you have a variable number of outputs and will generate code.

```
methods (Access = protected)
```

```
function varargout = stepImpl(~,varargin)
    for i = 1:nargout
        varargout{i} = varargin{i}+1;
    end
end
end
```

### See Also

[stepImpl](#) | [getNumInputsImpl](#) | [setupImpl](#)

### How To

- “Change Number of Step Inputs or Outputs”

# getOutputNamesImpl

**Class:** matlab.System

**Package:** matlab

Names of System block output ports

## Syntax

```
[name1,name2,...] = getOutputNamesImpl(obj)
```

## Description

[name1,name2,...] = getOutputNamesImpl(obj) returns the names of the output ports from System object, obj implemented in a MATLAB System block. The number of returned output names matches the number of outputs returned by the getNumOutputs method. If you change a property value that affects the number of outputs, the names of those outputs also change.

getOutputNamesImpl is called by the getOutputNames method and by the MATLAB System block.

---

**Note:** You must set Access = protected for this method.

---

## Input Arguments

**obj**

System object

## Output Arguments

**name1,name2,...**

Names of the outputs for the specified object, returned as strings.

**Default:** empty string

## Examples

### Specify Output Port Name

Specify the name of an output port as 'count'.

```
methods (Access = protected)
    function outputName = getOutputNamesImpl(~)
        outputName = 'count';
    end
end
```

### See Also

[getNumOutputsImpl](#) | [getInputNamesImpl](#)

### How To

- “Validate Property and Input Values”



# getPropertyGroupsImpl

**Class:** matlab.System

**Package:** matlab

Property groups for System object display

## Syntax

```
group = getPropertyGroupsImpl
```

## Description

`group = getPropertyGroupsImpl` returns the groups of properties to display. You define property sections (`matlab.system.display.Section`) and section groups (`matlab.system.display.SectionGroup`) within this method. Sections arrange properties into groups. Section groups arrange sections and properties into groups. If a System object, included through the MATLAB System block, has a section, but that section is not in a section group, its properties appear above the block dialog tab panels.

If you do not include a `getPropertyGroupsImpl` method in your code, all public properties are included in the dialog box by default. If you include a `getPropertyGroupsImpl` method but do not list a property, that property does not appear in the dialog box.

When the System object is displayed at the MATLAB command line, the properties are grouped as defined in `getPropertyGroupsImpl`. If your `getPropertyGroupsImpl` defines multiple section groups, only properties from the first section group are displayed at the command line. To display properties in other sections, a link is provided at the end of a System object property display. Group titles are also displayed at the command line. To omit the "Main" title for the first group of properties, set `TitleSource` to 'Auto' in `matlab.system.display.SectionGroup`.

`getPropertyGroupsImpl` is called by the MATLAB System block and when displaying the object at the command line.

---

**Note:** You must set `Access = protected` and `Static` for this method.

---

## Output Arguments

### group

Property group or groups

## Examples

### Define Block Dialog Tabs

Define two block dialog tabs, each containing specific properties. For this example, you use the `getPropertyGroupsImpl`, `matlab.system.display.SectionGroup`, and `matlab.system.display.Section` methods in your class definition file.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title', 'Value parameters', ...
            'PropertyList', {'StartValue', 'EndValue'});

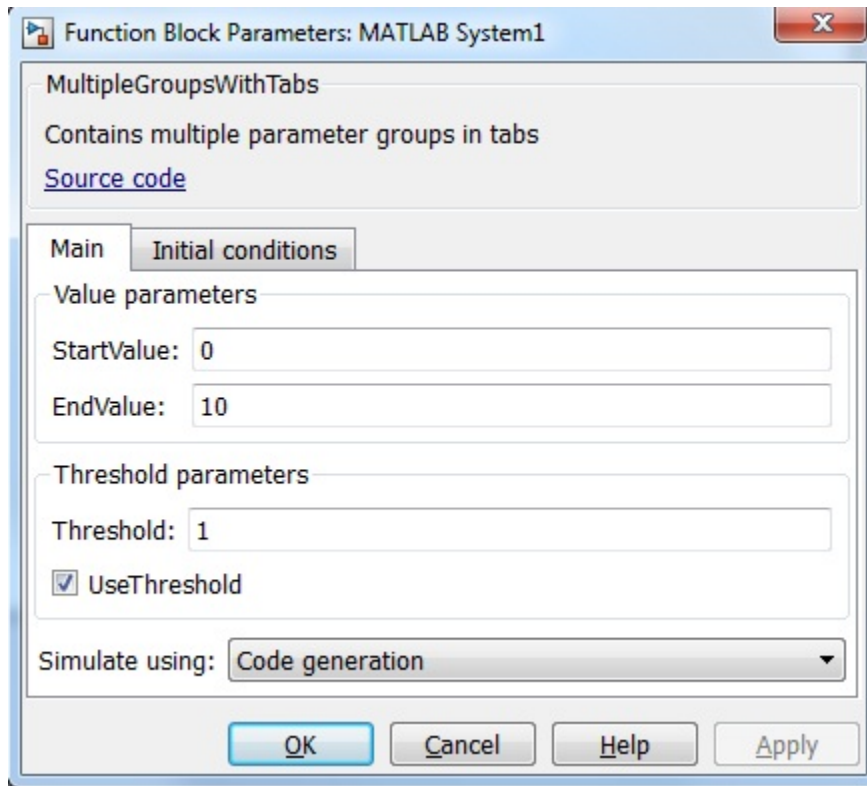
        thresholdGroup = matlab.system.display.Section(...
            'Title', 'Threshold parameters', ...
            'PropertyList', {'Threshold', 'UseThreshold'});

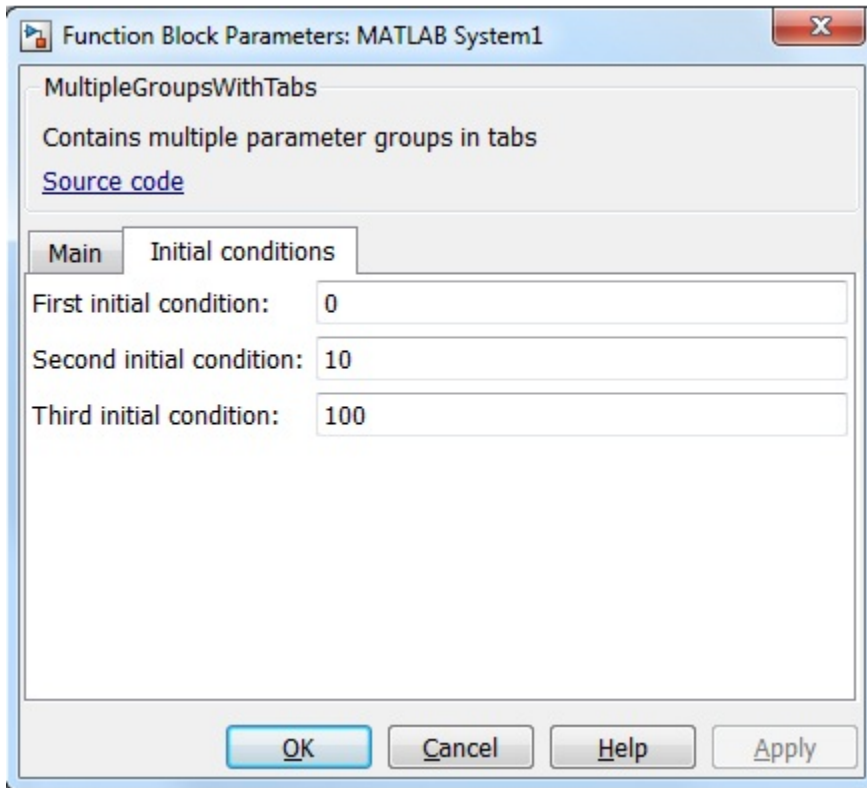
        mainGroup = matlab.system.display.SectionGroup(...
            'Title', 'Main', ...
            'Sections', [valueGroup, thresholdGroup]);

        initGroup = matlab.system.display.SectionGroup(...
            'Title', 'Initial conditions', ...
            'PropertyList', {'IC1', 'IC2', 'IC3'});

        groups = [mainGroup, initGroup];
    end
end
```

The resulting dialog box appears as follows.





### See Also

`matlab.system.display.Header` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`

### How To

- “Add Property Groups to System Object and MATLAB System Block”

# getSimulateUsingImpl

**Class:** matlab.System

**Package:** matlab

Specify value for Simulate using parameter

## Syntax

```
simmode = getSimulateUsingImpl
```

## Description

`simmode = getSimulateUsingImpl` returns a string, `simmode`, that specifies the Simulink simulation mode. The simulation mode restricts your System object to simulation using either code generation or interpreted execution. The associated `showSimulateUsingImpl` method controls whether the **Simulate using** option is displayed on the MATLAB System block dialog box.

`getSimulateUsingImpl` is called by the MATLAB System block.

---

**Note:** You must set `Access = protected` and `Static` for this method.

---

## Output Arguments

**simmode**

Simulation mode, returned as the string 'Code generation' or 'Interpreted execution'. If you do not include the `getSimulateUsingImpl` method in your class definition file, the simulation mode is unrestricted. Depending on the value returned by the associated `showSimulateUsingImpl` method, the simulation mode is displayed as either a dropdown list on the dialog box or not at all.

# Examples

## Specify the Simulation Mode

In the class definition file of your System object, define the simulation mode to display in the MATLAB System block. To prevent **Simulate using** from displaying, see `showSimulateUsingImpl`.

```
methods (Static, Access = protected)
    function simMode = getSimulateUsingImplj
        simMode = 'Interpreted execution';
    end
end
```

## See Also

`showSimulateUsingImpl`

## How To

- “Control Simulation Type in MATLAB System Block”

# infoImpl

**Class:** matlab.System

**Package:** matlab

Information about System object

## Syntax

```
s = infoImpl(obj,varargin)
```

## Description

`s = infoImpl(obj,varargin)` lets you set up information to return about the current configuration of a System object `obj`. This information is returned in a struct from the `info` method. The default `infoImpl` method, which is used if you do not include `infoImpl` in your class definition file, returns an empty struct.

`infoImpl` is called by the `info` method.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object

**varargin**

Optional. Allow variable number of inputs

## Examples

### Specify System object Information

Define the `infoImpl` method to return current count information.

```
methods (Access = protected)
    function s = infoImpl(obj)
        s = struct('Count',obj.pCount);
    end
end
```

### How To

- “Define System Object Information”



# isInactivePropertyImpl

**Class:** matlab.System

**Package:** matlab

Inactive property status

## Syntax

```
flag = isInactivePropertyImpl(obj,prop)
```

## Description

`flag = isInactivePropertyImpl(obj,prop)` specifies whether a public, non-state property is inactive for the current object configuration. An *inactive property* is a property that is not relevant to the object, given the values of other properties. Inactive properties are not shown if you use the `disp` method to display object properties. If you attempt to use public access to directly access or use `get` or `set` on an inactive property, a warning occurs.

`isInactiveProperty` is called by the `disp` method and by the `get` and `set` methods.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object handle

**prop**

Public, non-state property name

# Output Arguments

## flag

Inactive status Indicator of the input property `prop` for the current object configuration, returned as a logical scalar value

# Examples

## Specify When a Property Is Inactive

Display the `InitialValue` property only when the `UseRandomInitialValue` property value is `false`.

```
methods (Access = protected)
function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
    else
        flag = false;
    end
end
end
end
```

## See Also

`setProperties`

## How To

- “Hide Inactive Properties”

# isInputSizeLockedImpl

**Class:** matlab.System

**Package:** matlab

Locked input size status

## Syntax

```
flag = isInputSizeLockedImpl(obj,i)
```

## Description

`flag = isInputSizeLockedImpl(obj,i)` indicates whether the  $i^{\text{th}}$  input port to the `step` method has its size locked. If `flag` is `true`, the size is locked and inputs to the System object cannot change size while the object is locked. If `flag` is `false`, the input is variable size and is not locked. In the unlocked case, the size of inputs to the object can change while the object is running and locked.

`isInputSizeLockedImpl` executes once for each input during System object initialization.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object

**i**

step method input port number

## Output Arguments

### **flag**

Flag indicating whether the size of inputs to the specified port is locked, returned as a logical scalar value. If the value of `isInputSizeLockedImpl` is `true`, the size of the current input to that port is compared to the first input to that port. If the sizes do not match, an error occurs.

**Default:** `false`

## Examples

### Check If Input Size Is Locked

Specify in your class definition file to check whether the size of the System object input is locked.

```
methods (Access = protected)
    function flag = isInputSizeLockedImpl(~,index)
        flag = true;
    end
end
```

### See Also

`matlab.System`

### How To

- “Specify Locked Input Size”
- “What You Cannot Change While Your System Is Running”

# loadObjectImpl

**Class:** matlab.System

**Package:** matlab

Load System object from MAT file

## Syntax

```
loadObjectImpl(obj)
```

## Description

`loadObjectImpl(obj)` loads a saved System object, `obj`, from a MAT file. Your `loadObjectImpl` method should correspond to your `saveObjectImpl` method to ensure that all saved properties and data are loaded.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object

## Examples

### Load System object

Load a saved System object. In this example, the object contains a child object, protected and private properties, and a discrete state. It also saves states if the object is locked and calls the `loadObjectImpl` method from the `matlab.System` class.

```
methods (Access = protected)
```

```
function loadObjectImpl(obj,s,wasLocked)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if wasLocked
        obj.state = s.state;
    end

    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
```

### See Also

saveObjectImpl

### How To

- “Load System Object”
- “Save System Object”

# processTunedPropertiesImpl

**Class:** matlab.System

**Package:** matlab

Action when tunable properties change

## Syntax

```
processTunedPropertiesImpl(obj)
```

## Description

`processTunedPropertiesImpl(obj)` specifies the actions to perform when one or more tunable property values change. This method is called as part of the next call to the `step` method after a tunable property value changes. A property is tunable only if its `Nontunable` attribute is `false`, which is the default.

`processTunedPropertiesImpl` is called by the `step` method.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its `System` object will be used in the Simulink MATLAB `System` block.

---

## Tips

Use this method when a tunable property affects the value of a different property.

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty` within `processTunedPropertiesImpl`.

In MATLAB when multiple tunable properties are changed before running the `System` object, `processTunedPropertiesImpl` is called only once for all the changes. `isChangedProperty` returns `true` for all the changed properties.

In Simulink, when a parameter is changed in a MATLAB System block dialog, the next simulation step calls `processTunedPropertiesImpl` before calling `stepImpl`. All tunable parameters are considered changed and `processTunedPropertiesImpl` method is called for each of them. `isChangedProperty` returns `true` for all the dialog properties.

## Input Arguments

### **obj**

System object

## Examples

### Specify Action When Tunable Property Changes

Use `processTunedPropertiesImpl` to recalculate the lookup table if the value of either the `NumNotes` or `MiddleC` property changes before the next call to the `step` method. `propChange` indicates if either property has changed.

```
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj,obj.NumNotes) || ...
                    isChangedProperty(obj,obj.MiddleC)
        if propChange
            obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
        end
    end
end
```

### See Also

[validatePropertiesImpl](#) | [setProperties](#)

### How To

- “Validate Property and Input Values”
- “Define Property Attributes”



# releaseImpl

**Class:** matlab.System

**Package:** matlab

Release resources

## Syntax

```
releaseImpl(obj)
```

## Description

`releaseImpl(obj)` releases any resources used by the System object, such as file handles. This method also performs any necessary cleanup tasks. To release resources for a System object, you must use `releaseImpl` instead of a destructor.

`releaseImpl` is called by the `release` method. `releaseImpl` is also called when the object is deleted or cleared from memory, or when all references to the object have gone out of scope.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object

## Examples

### Close a File and Release Its Resources

Use the `releaseImpl` method to close a file opened by the System object.

```
methods (Access = protected)
  function releaseImpl(obj)
    fclose(obj.pFileID);
  end
end
```

### How To

- “Release System Object Resources”

# resetImpl

**Class:** matlab.System

**Package:** matlab

Reset System object states

## Syntax

```
resetImpl(obj)
```

## Description

`resetImpl(obj)` defines the state reset equations for a System object. Typically you reset the states to a set of initial values, which is useful for initialization at the start of simulation.

`resetImpl` is called by the `reset` method only if the object is locked. The object remains locked after it is reset. `resetImpl` is also called by the `setup` method, after the `setupImpl` method.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

---

## Input Arguments

**obj**

System object

### Examples

#### Reset Property Value

Use the `reset` method to reset the state of the counter stored in the `pCount` property to zero.

```
methods (Access = protected)
  function resetImpl(obj)
    obj.pCount = 0;
  end
end
```

#### See Also

`releaseImpl`

#### How To

- “Reset Algorithm State”

# saveObjectImpl

**Class:** matlab.System

**Package:** matlab

Save System object in MAT file

## Syntax

saveObjectImpl(obj)

## Description

saveObjectImpl(obj) defines the System object obj property and state values to be saved in a MAT file when a user calls **save** on that object. **save** calls **saveObject**, which then calls **saveObjectImpl**. To save a System object in generated code, the object must be unlocked and it cannot contain or be a child object.

If you do not define a **saveObjectImpl** method for your System object class, only public properties and properties with the **DiscreteState** attribute are saved.

To save any private or protected properties or state information, you must define a **saveObjectImpl** in your class definition file.

End users can use **load**, which calls **loadObjectImpl** to load a System object into their workspace.

---

**Tip** Save the state of an object only if the object is locked. When the user loads that saved object, it loads in that locked state.

To save child object information, use the associated **saveObject** method within the **saveObjectImpl** method.

---

---

**Note:** You must set **Access = protected** for this method.

---

# Input Arguments

## obj

System object

# Examples

## Define Property and State Values to Save

Define what is saved for the System object. Call the base class version of `saveObjectImpl` to save public properties. Then, save any child System objects and any protected and private properties. Finally, save the state if the object is locked.

```
methods (Access = protected)
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.child = matlab.System.saveObject(obj.child);
        s.protectedprop = obj.protectedprop;
        s.pdependentprop = obj.pdependentprop;
        if isLocked(obj)
            s.state = obj.state;
        end
    end
end
```

## See Also

`loadObjectImpl`

## How To

- “Save System Object”
- “Load System Object”

# setProperties

**Class:** matlab.System

**Package:** matlab

Set property values using name-value pairs

## Syntax

```
setProperties(obj,numargs,name1,value1,name2,value2,...)
```

```
setProperties(obj,numargs,arg1,...,argN,propvalname1,...propvalnameN)
```

## Description

`setProperties(obj,numargs,name1,value1,name2,value2,...)` provides the name-value pair inputs to the System object constructor. Use this syntax if every input must specify both name and value.

`setProperties(obj,numargs,arg1,...,argN,propvalname1,...propvalnameN)` provides the value-only inputs, which you can follow with the name-value pair inputs to the System object during object construction. Use this syntax if you want to allow users to specify one or more inputs by their values only.

## Input Arguments

### **obj**

System object

### **numargs**

Number of inputs passed in by the object constructor

### **name1,name2,...**

Name of property

**value1, value2, ...**

Value of the property

**arg1, ... argN**

Value of property (for value-only input to the object constructor)

**propvalname1, ... propvalnameN**

Name of the value-only property

## Examples

### Setup Value-Only Inputs

Set up an object so users can specify value-only inputs for VProp1, VProp2, and other property values via name-value pairs when constructing the object.

```
methods
    function obj = MyFile(varargin)
        setProperties(obj, nargin, varargin{:}, 'VProp1', 'VProp2');
    end
end
```

### How To

- “Set Property Values at Construction Time”



# setupImpl

**Class:** matlab.System

**Package:** matlab

Initialize System object

## Syntax

```
setupImpl(obj)  
setupImpl(obj, input1, input2, ...)
```

## Description

`setupImpl(obj)` sets up a System object and implements one-time tasks that do not depend on any inputs to its `stepImpl` method. You typically use `setupImpl` to set private properties so they do not need to be calculated each time `stepImpl` method is called. To acquire resources for a System object, you must use `setupImpl` instead of a constructor.

`setupImpl` executes the first time the `step` method is called on an object after that object has been created. It also executes the next time `step` is called after an object has been released.

`setupImpl(obj, input1, input2, ...)` sets up a System object using one or more of the `stepImpl` input specifications. The number and order of inputs must match the number and order of inputs defined in the `stepImpl` method. You pass the inputs into `setupImpl` to use the specifications, such as size and data types in the one-time calculations.

`setupImpl` is called by the `setup` method, which is done automatically as the first subtask of the `step` method on an unlocked System object.

---

**Note:** You can omit this method from your class definition file if your System object does not require any setup tasks.

You must set `Access = protected` for this method.

Do not use `setupImpl` to initialize or reset states. For states, use the `resetImpl` method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

---

## Tips

To validate properties or inputs use the `validatePropertiesImpl`, `validateInputsImpl`, or `setProperties` methods. Do not include validation in `setupImpl`.

Do not use the `setupImpl` method to set up input values.

## Input Arguments

### **obj**

System object handle

### **input1, input2, ...**

Inputs to the `stepImpl` method

## Examples

### Setup a File for Writing

This example shows how to open a file for writing using the `setupImpl` method in your class definition file.

```
methods (Access = protected)
    function setupImpl(obj)
        obj.pFileID = fopen(obj.FileName, 'wb');
        if obj.pFileID < 0
            error('Opening the file failed');
        end
    end
end
```

end

### Initialize Properties Based on Step Inputs

This example shows how to use `setupImpl` to specify that `step` initialize the properties of an input. In this case, calls to the object's `step` method, which include input `u`, initialize the object states in a matrix of size `u`.

```
methods (Access = protected)
    function setupImpl(obj, u)
        obj.State = zeros(size(u), 'like', u);
    end
end
```

### See Also

[validatePropertiesImpl](#) | [validateInputsImpl](#) | [setProperty](#)

### How To

- “Initialize Properties and Setup One-Time Calculations”
- “Set Property Values at Construction Time”

# showFiSettingsImpl

**Class:** matlab.System

**Package:** matlab

Fixed point data type tab visibility for System objects

## Syntax

```
flag = showFiSettingsImpl
```

## Description

`flag = showFiSettingsImpl` specifies whether the Data Types tab appears on the MATLAB System block dialog box. The Data Types tab includes parameters to control processing of fixed point data the MATLAB System block. You cannot specify which parameters appear on the tab. If you implement `showFiSettingsImpl`, the simulation mode is set code generation.

`showFiSettingsImpl` is called by the MATLAB System block.

The parameters that appear on the Data Types tab, which cannot be customized, are

- **Saturate on integer overflow** is a check box to control the action to take on integer overflow for built-in integer types. The default is that the box is checked, which indicates to saturate. This is also the default for when **Same as MATLAB** is selected as the **MATLAB System fimath** option.
- **Treat these inherited Simulink signal types as fi objects** is a pull down that indicates which inherited data types to treat as fi data types. Valid options are **Fixed point** and **Fixed point & integer**. The default value is **Fixed point**.
- **MATLAB System fimath** has two radio button options: **Same as MATLAB** and **Specify Other**. The default, **Same as MATLAB**, uses the current MATLAB fixed-point math settings. **Specify Other** enables the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see `fimath`, in the Fixed-Point Designer documentation.

**Note:** If you do not want to display the tab, you do not need to implement this method in your class definition file.

You must set `Access = protected` and `Static` for this method.

---

## Output Arguments

### **flag**

Flag indicating whether to display the Data Types tab on the MATLAB System block mask, returned as a logical scalar value. Returning a `true` value displays the tab. A `false` value does not display the tab.

**Default:** `false`

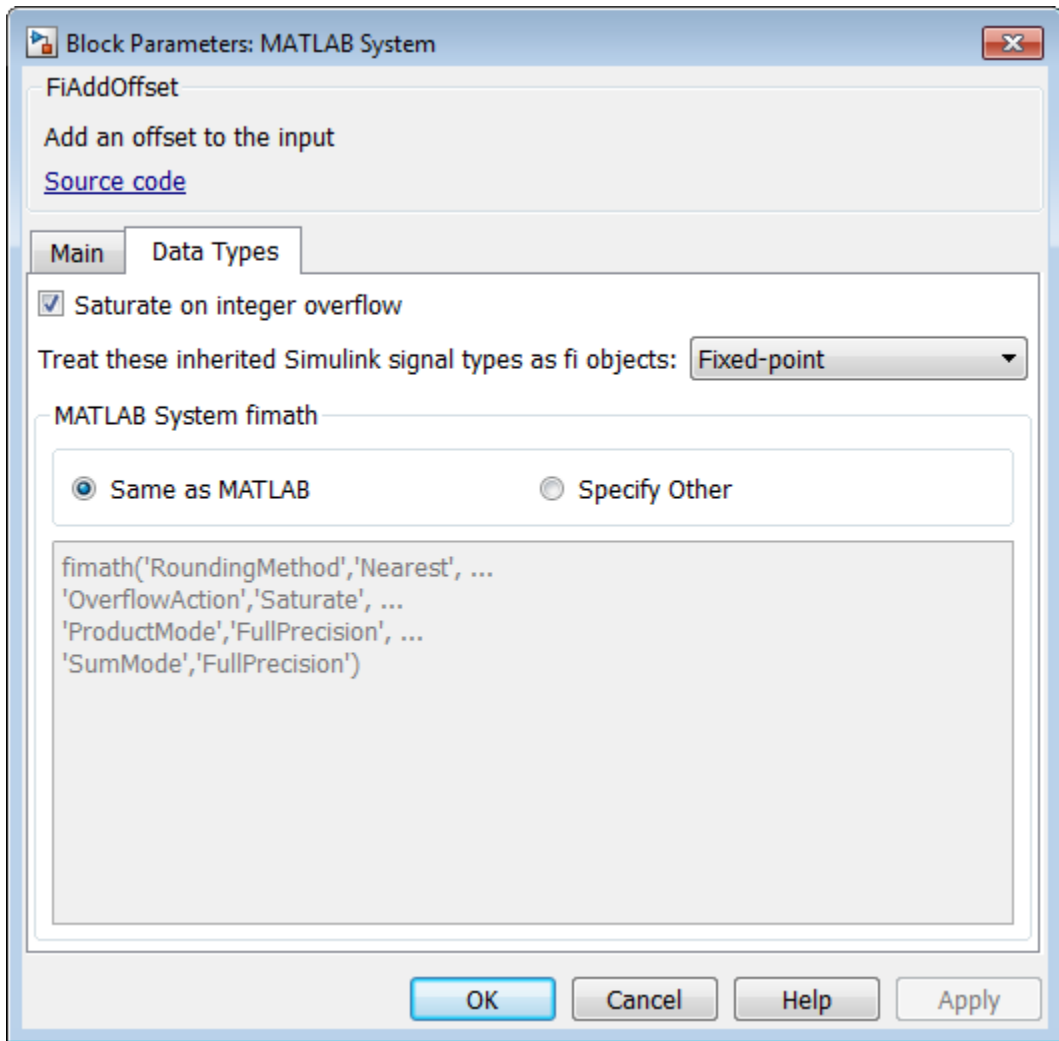
## Examples

### **Show the Data Types Tab**

Show the Data Types tab on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
    function isVisible = showFiSettingsImpl
        isVisible = true;
    end
end
```

If you set the flag, `isVisible`, to `true`, the tab appears as follows when you add the object to Simulink with the MATLAB System block.



## How To

- “Add Data Types Tab to MATLAB System Block”

# showSimulateUsingImpl

**Class:** matlab.System

**Package:** matlab

Simulate Using visibility

## Syntax

```
flag = showSimulateUsingImpl
```

## Description

`flag = showSimulateUsingImpl` specifies whether the **Simulate using** parameter and dropdown list appear on the MATLAB System block dialog box.

`showSimulateUsingImpl` is called by the MATLAB System block.

---

**Note:** You must set `Access = protected` and `Static` for this method.

---

## Output Arguments

**flag**

Flag indicating whether to display the **Simulate using** parameter and dropdown list on the MATLAB System block mask, returned as a logical scalar value. A `true` value displays the parameter and dropdown list. A `false` value hides the parameter and dropdown list.

**Default:** `true`

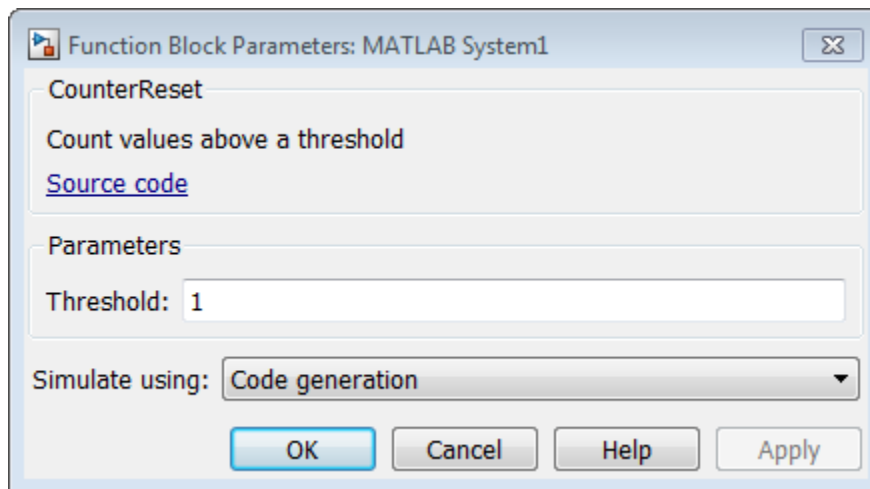
## Examples

### Hide the Simulate using Parameter

Hide the **Simulate using** parameter on the MATLAB System block dialog box.

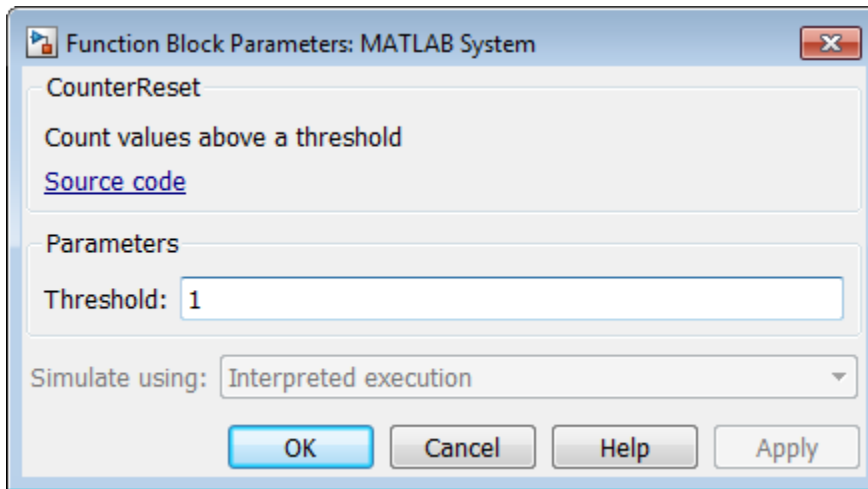
```
methods (Static, Access = protected)
    function flag = showSimulateUsingImpl
        flag = false;
    end
end
```

If you set the flag to `true` or omit the `showSimulateUsingImpl` method, which defaults to `true`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.



If you also specify a single value for `getSimulateUsingImpl`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.





## See Also

getSimulateUsingImpl

## How To

- “Control Simulation Type in MATLAB System Block”

# stepImpl

**Class:** matlab.System

**Package:** matlab

System output and state update equations

## Syntax

```
[output1,output2,...] = stepImpl(obj,input1,input2,...)
```

## Description

[output1,output2,...] = stepImpl(obj,input1,input2,...) defines the algorithm to execute when you call the `step` method on the specified object `obj`. The `step` method calculates the outputs and updates the object's state values using the inputs, properties, and state update equations.

`stepImpl` is called by the `step` method.

---

**Note:** You must set `Access = protected` for this method.

---

## Tips

The number of input arguments and output arguments must match the values returned by the `getNumInputsImpl` and `getNumOutputsImpl` methods, respectively

## Input Arguments

**obj**

System object handle

**input1,input2,...**

Inputs to the `step` method

## Output Arguments

### output

Output returned from the `step` method.

## Examples

### Specify System Object Algorithm

Use the `stepImpl` method to increment two numbers.

```
methods (Access = protected)
    function [y1,y2] = stepImpl(obj,x1,x2)
        y1 = x1 + 1;
        y2 = x2 + 1;
    end
end
```

### See Also

[getNumInputsImpl](#) | [getNumOutputsImpl](#) | [validateInputsImpl](#)

### How To

- “Define Basic System Objects”
- “Change Number of Step Inputs or Outputs”

# supportsMultipleInstanceImpl

**Class:** matlab.System

**Package:** matlab

Support System object in Simulink For Each subsystem

## Syntax

```
flag = supportsMultipleInstanceImpl(obj)
```

## Description

`flag = supportsMultipleInstanceImpl(obj)` indicates whether you can use the System object in a Simulink For Each subsystem via the MATLAB System block. To enable For Each support, you must include the `supportsMultipleInstanceImpl` in your class definition file and have it return `true`. Do not enable For Each support if your System object allocates exclusive resources that may conflict with other System objects, such as allocating file handles, memory by address, or hardware resources.

During Simulink model compilation and propagation, the MATLAB System block calls the `supportMultipleInstance` method, which then calls the `supportsMultipleInstanceImpl` method to determine For Each support.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

### flag

Boolean value indicating whether the System object can be used in a For Each subsystem. The default value, if you do not include the `supportMultipleInstance` method, is `false`.

## Examples

### Enable For-Each Support for System Object

Specify in your class definition file that the System object can be used in a Simulink For Each subsystem.

```
methods (Access = protected)
    function flag = supportsMultipleInstanceImpl(obj)
        flag = true;
    end
end
```

### See Also

`matlab.System`

### How To

- “Enable For Each Subsystem Support”

# validateInputsImpl

**Class:** matlab.System

**Package:** matlab

Validate inputs to step method

## Syntax

```
validateInputsImpl(obj,input1,input2,...)
```

## Description

`validateInputsImpl(obj,input1,input2,...)` validates inputs to the `step` method at the beginning of initialization. Validation includes checking data types, complexity, cross-input validation, and validity of inputs controlled by a property value.

`validateInputsImpl` is called by the `setup` method before `setupImpl`. `validateInputsImpl` executes only once.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.

---

## Input Arguments

**obj**

System object handle

**input1,input2,...**

Inputs to the `setup` method

## Examples

### Validate Input Type

Validate that the input is numeric.

```
methods (Access = protected)
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
            error('Input must be numeric');
        end
    end
end
end
```

### See Also

[validatePropertiesImpl](#) | [setupImpl](#)

### How To

- “Validate Property and Input Values”

# validatePropertiesImpl

**Class:** matlab.System

**Package:** matlab

Validate property values

## Syntax

```
validatePropertiesImpl(obj)
```

## Description

`validatePropertiesImpl(obj)` validates interdependent or interrelated property values at the beginning of object initialization, such as checking that the dependent or related inputs are the same size.

`validatePropertiesImpl` is the first method called by the `setup` method. `validatePropertiesImpl` also is called before the `processTunedPropertiesImpl` method.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.

---

## Tips

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty(obj,property)` within `validatePropertiesImpl`.

## Input Arguments

**obj**

System object handle



## Examples

### Validate a Property

Validate that the `useIncrement` property is `true` and that the value of the `increment` property is greater than zero.

```
methods (Access = protected)
  function validatePropertiesImpl(obj)
    if obj.useIncrement && obj.increment < 0
      error('The increment value must be positive');
    end
  end
end
```

### See Also

[processTunedPropertiesImpl](#) | [setupImpl](#) | [validateInputsImpl](#)

### How To

- “Validate Property and Input Values”

## **matlab.system.display.Action class**

**Package:** matlab.system.display

Custom button

### **Syntax**

```
matlab.system.display.Action(action)
matlab.system.display.Action(action,Name,Value)
```

### **Description**

`matlab.system.display.Action(action)` defines a button to display on the MATLAB System block. This button executes a function by launching a System object method or invoking any MATLAB function or code.

A typical button function launches a figure. The launched figure is decoupled from the block dialog box. Changes to the block are not synced to the displayed figure.

You define `matlab.system.display.Action` within the `getPropertyGroupsImpl` method in your class definition file. You can define multiple buttons using separate instances of `matlab.system.display.Action` in your class definition file.

`matlab.system.display.Action(action,Name,Value)` includes `Name,Value` pair arguments, which you can use to specify any properties.

### **Input Arguments**

#### **action**

Action taken when the user presses the specified button on the MATLAB System block dialog. The action is defined as a function handle or as a MATLAB command string. If you define the action as a function handle, the function definition must define two inputs. These inputs are a `matlab.system.display.ActionData` object and a System object instance, which can be used to invoke a method.

A `matlab.system.display.ActionData` object is the callback object for a display action. You use the `UserData` property of `matlab.system.display.ActionData` to store persistent data, such as a figure handle.

## 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`.

## Properties

You specify these properties as part of the input using `Name`, `Value` pair arguments. Optionally, you can define them using `object.property` syntax.

- `ActionCalledFcn` — Action to take when the button is pressed. You cannot specify this property using a Name-Value pair argument.
- `Label` — Text string to display on the button. The default value is an empty string.
- `Description` — Text string for the button tooltip. The default value is an empty string.
- `Placement` — Text string indicating where on a separate row in the property group to place the button. Valid values are 'first', 'last', or a property name. If you specify a property name, the button is placed above that property. The default value is 'last'.
- `Alignment` — Text string indicating how to align the button. Valid values are 'left' and 'right'. The default value is 'left'.

## Examples

### Define Button on MATLAB System Block

Define a **Visualize** button and its associated function to open a figure that plots a ramp using the parameter values in the block dialog.

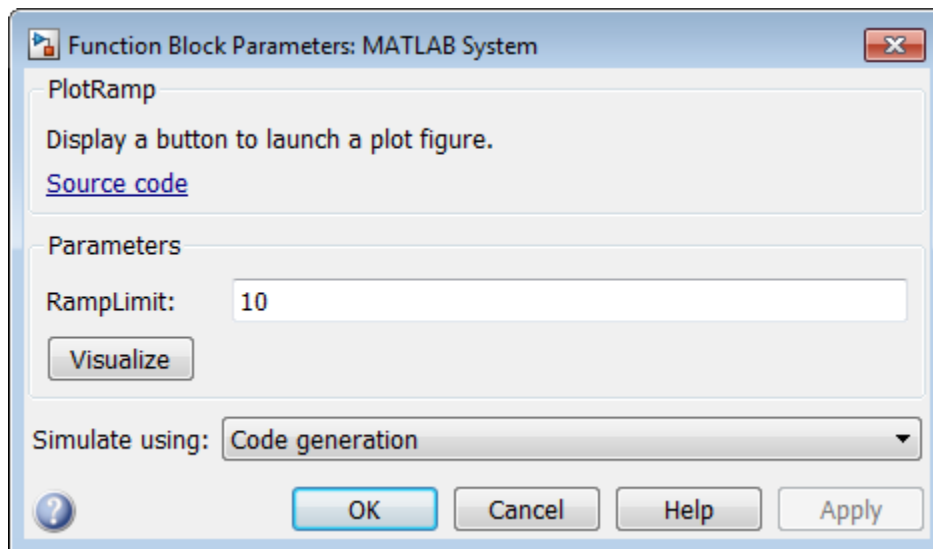
```
methods(Static, Access = protected)
```

```
function group = getPropertyGroupsImpl
group = matlab.system.display.Section(mfilename('class'));
group.Actions = matlab.system.display.Action(@(~,obj)...
    visualize(obj),'Label','Visualize');
end
end

methods
function obj = PlotRamp(varargin)
    setProperties(obj,nargin,varargin{:});
end

function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
end
end
```

When you specify the System object in the MATLAB System block, the resulting block dialog box appears as follows.



To open the same figure, rather than multiple figures, when the button is pressed more than once, use this code instead.

```
methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(actionData,obj)...
            visualize(obj,actionData),'Label','Visualize');
    end
end

methods
    function obj = ActionDemo(varargin)
        setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj,actionData)
        f = actionData.UserData;
        if isempty(f) || ~ishandle(f)
            f = figure;
            actionData.UserData = f;
        else
            figure(f); % Make figure current
        end

        d = 1:obj.RampLimit;
        plot(d);
    end
end
```

## See Also

[matlab.System.getPropertyGroupsImpl](#) | [matlab.system.display.Section](#) | [matlab.system.display.SectionGroup](#)

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Button to MATLAB System Block”

## matlab.system.display.Header class

**Package:** matlab.system.display

Header for System objects properties

### Syntax

```
matlab.system.display.Header(N1,V1,...Nn,Vn)  
matlab.system.display.Header(Obj,...)
```

### Description

`matlab.system.display.Header(N1,V1,...Nn,Vn)` defines a header for the System object, with the header properties defined in Name-Value (N,V) pairs. You use `matlab.system.display.Header` within the `getHeaderImpl` method. The available header properties are

- **Title** — Header title string. The default value is an empty string.
- **Text** — Header description text string. The default value is an empty string.
- **ShowSourceLink** — Show link to source code for the object.

`matlab.system.display.Header(Obj,...)` creates a header for the specified System object (`Obj`) and sets the following property values:

- **Title** — Set to the `Obj` class name.
- **Text** — Set to help summary for `Obj`.
- **ShowSourceLink** — Set to `true` if `Obj` is MATLAB code. In this case, the **Source Code** link is displayed. If `Obj` is P-coded and the source code is not available, set this property to `false`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

## Methods

getHeaderImpl

Header for System object display

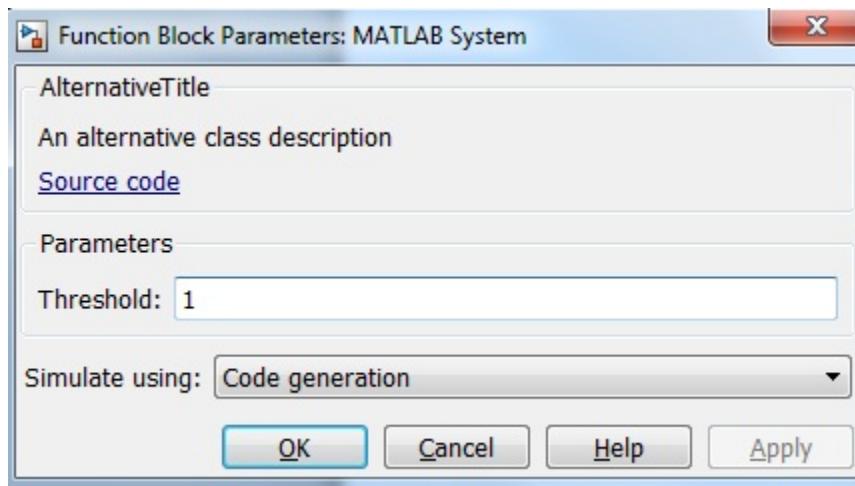
## Examples

### Define System Block Header

Define a header in your class definition file.

```
methods (Static, Access = protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header(mfilename('class'), ...
            'Title', 'AlternativeTitle', ...
            'Text', 'An alternative class description');
    end
end
```

The resulting output appears as follows. In this case, **Source code** appears because the ShowSourceLink property was set to true.



## See Also

matlab.system.display.Section | matlab.system.display.SectionGroup

### **How To**

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Header to MATLAB System Block”



# getHeaderImpl

**Class:** matlab.system.display.Header

**Package:** matlab.system.display

Header for System object display

## Syntax

```
header = getHeaderImpl
```

## Description

`header = getHeaderImpl` returns the header to display for the System object. If you do not specify the `getHeaderImpl` method, no title or text appears for the header in the block dialog box.

`getHeaderImpl` is called by the MATLAB System block

---

**Note:** You must set `Access = protected` and `Static` for this method.

---

## Output Arguments

**header**

Header text

## Examples

### Define Header for System Block Dialog Box

Define a header in your class definition file for the EnhancedCounter System object.

```
methods (Static, Access = protected)
```

```
function header = getHeaderImpl
    header = matlab.system.display.Header('EnhancedCounter',...
        'Title', 'Enhanced Counter');
end
end
```

### See Also

getPropertyGroupsImpl

### How To

- “Add Header to MATLAB System Block”

# matlab.system.display.Section class

**Package:** matlab.system.display

Property group section for System objects

## Syntax

```
matlab.system.display.Section(N1,V1,...Nn,Vn)
matlab.system.display.Section(Obj,...)
```

## Description

`matlab.system.display.Section(N1,V1,...Nn,Vn)` creates a property group section for displaying System object properties, which you define using property Name-Value pairs (N,V). You use `matlab.system.display.Section` to define property groups using the `getPropertyGroupsImpl` method. The available Section properties are

- **Title** — Section title string. The default value is an empty string.
- **TitleSource** — Source of section title string. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the string from the **Title** property. If the **Obj** name is given, the default value is **Auto**, which uses the **Obj** name.
- **Description** — Section description string. The default value is an empty string.
- **PropertyList** — Section property list as a cell array of property names. The default value is an empty array. If the **Obj** name is given, the default value is all eligible display properties.

---

**Note:** Certain properties are not eligible for display either in a dialog box or in the System object summary on the command-line. Property types that cannot be displayed are: hidden, abstract, private or protected access, discrete state, and continuous state. Dependent properties do not display in a dialog box, but do display in the command-line summary.

---

`matlab.system.display.Section(Obj,...)` creates a property group section for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to 'Auto', which uses the `Obj` name.
- `PropertyList` — Set to all publically-available properties in the `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

## Methods

## Examples

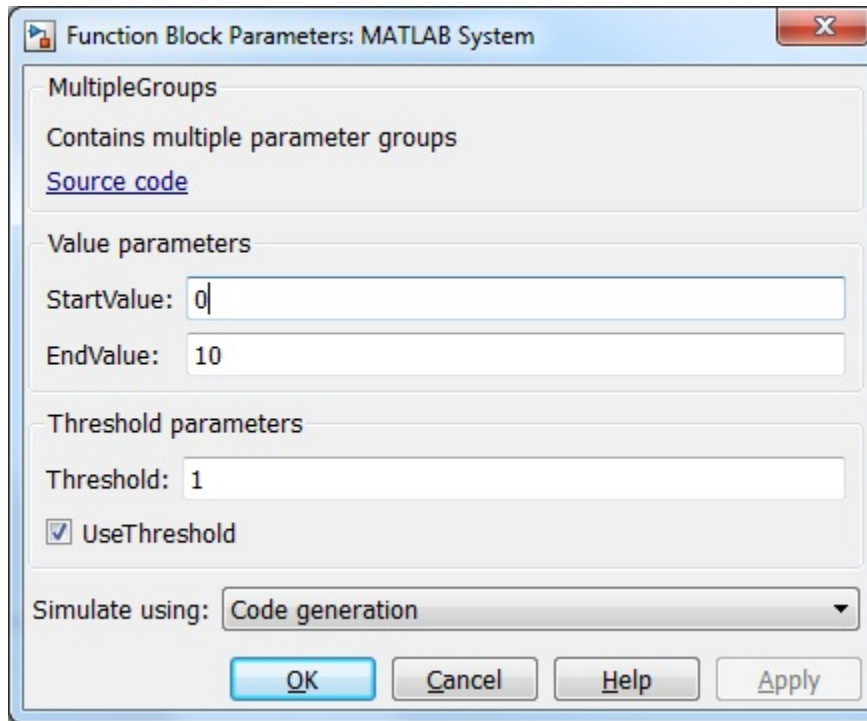
### Define Property Groups

Define two property groups in your class definition file by specifying their titles and property lists.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});
        groups = [valueGroup,thresholdGroup];
    end
end
```

When you specify the System object in the MATLAB System block, the resulting dialog box appears as follows.



## See Also

matlab.system.display.Header | matlab.system.display.SectionGroup |  
getPropertyGroupsImpl

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Property Groups to System Object and MATLAB System Block”

## matlab.system.display.SectionGroup class

**Package:** matlab.system.display

Section group for System objects

### Syntax

```
matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)  
matlab.system.display.SectionGroup(Obj,...)
```

### Description

`matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)` creates a group for displaying System object properties and display sections created with `matlab.system.display.Section`. You define such sections or properties using property Name-Value pairs (N,V). A section group can contain both properties and sections. You use `matlab.system.display.SectionGroup` to define section groups using the `getPropertyGroupsImpl` method. Section groups display as separate tabs in the MATLAB System block. The available Section properties are

- **Title** — Group title string. The default value is an empty string.
- **TitleSource** — Source of group title string. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the string from the **Title** property. If the **Obj** name is given, the default value is **Auto**, which uses the **Obj** name. In the System object property display at the MATLAB command line, you can omit the default "Main" title for the first group of properties by setting **TitleSource** to 'Auto'.
- **Description** — Group or tab description that appears above any properties or panels. The default value is an empty string.
- **PropertyList** — Group or tab property list as a cell array of property names. The default value is an empty array. If the **Obj** name is given, the default value is all eligible display properties.
- **Sections** — Group sections as an array of section objects. If the **Obj** name is given, the default value is the default section for the **Obj**.

`matlab.system.display.SectionGroup(Obj, ...)` creates a section group for the specified System object (Obj) and sets the following property values:

- `TitleSource` — Set to 'Auto'.
- `Sections` — Set to `matlab.system.display.Section` object for Obj.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

## Examples

### Define Block Dialog Tabs

Define in your class definition file two tabs, each containing specific properties. For this example, you use the `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` methods.

```
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

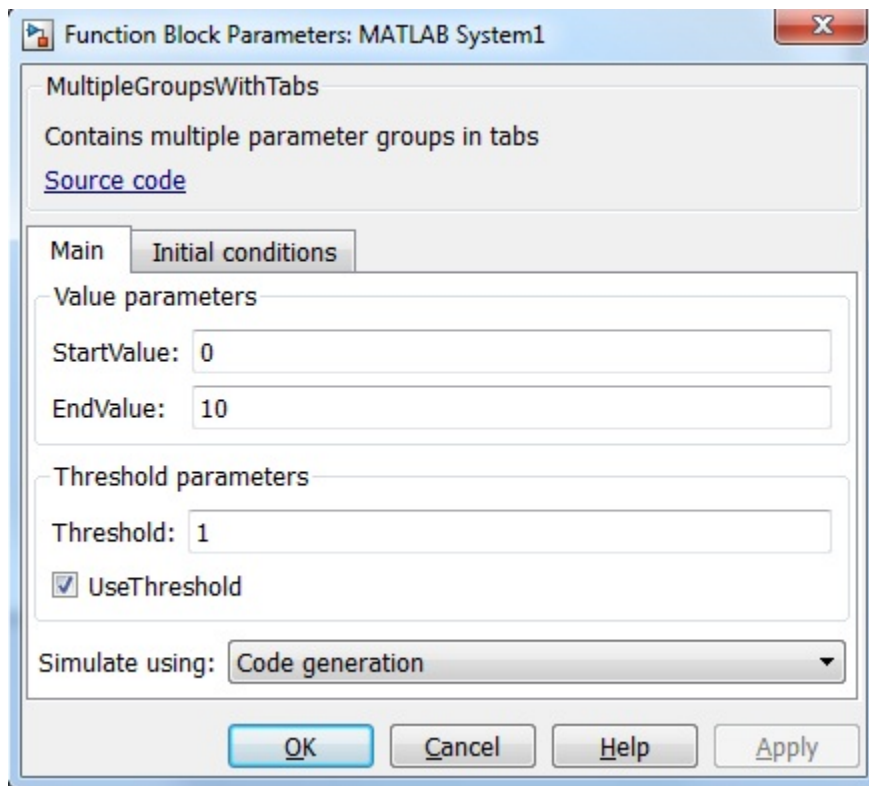
        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});

        mainGroup = matlab.system.display.SectionGroup(...
            'Title','Main', ...
            'Sections',[valueGroup,thresholdGroup]);

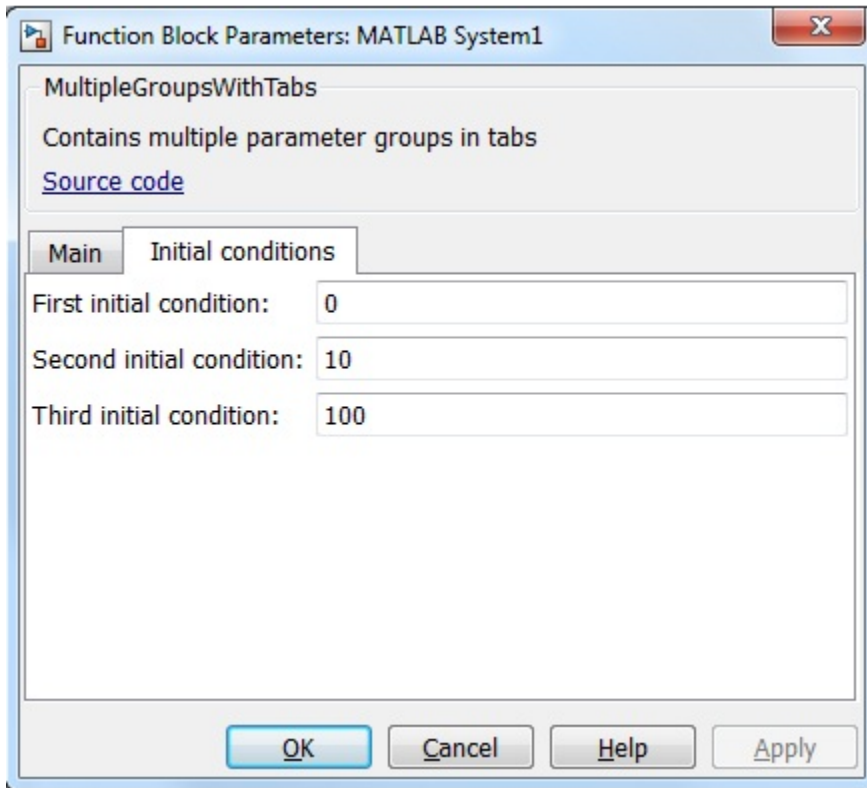
        initGroup = matlab.system.display.SectionGroup(...
            'Title','Initial conditions', ...
            'PropertyList',{'IC1','IC2','IC3'});

        groups = [mainGroup,initGroup];
    end
end
```

The resulting dialog appears as follows when you add the object to Simulink with the MATLAB System block.







## See Also

matlab.system.display.Header | matlab.system.display.Section |  
getPropertyGroupsImpl

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Add Property Groups to System Object and MATLAB System Block”

# matlab.system.mixin.CustomIcon class

**Package:** matlab.system.mixin

Custom icon mixin class

## Description

matlab.system.mixin.CustomIcon is a class that defines the `getIcon` method. This method customizes the name of the icon used for the System object implemented through a MATLAB System block.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...  
    matlab.system.mixin.CustomIcon
```

## Methods

`getIconImpl` Name to display as block icon

## See Also

`matlab.System`

## Tutorials

- “Define MATLAB System Block Icon”

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

## getIconImpl

**Class:** matlab.system.mixin.CustomIcon

**Package:** matlab.system.mixin

Name to display as block icon

### Syntax

```
icon = getIconImpl(obj)
```

### Description

`icon = getIconImpl(obj)` returns the string or cell array of strings to display on the block icon of the System object implemented through the **MATLAB System** block. If you do not specify the `getIconImpl` method, the block displays the class name of the System object as the block icon. For example, if you specify `pkg.MyObject` in the **MATLAB System** block, the default icon is labeled `My Object`

`getIconImpl` is called by the `getIcon` method, which is used by the **MATLAB System** block during Simulink model compilation.

---

**Note:** You must set `Access = protected` for this method.

---

### Input Arguments

**obj**

System object handle

### Output Arguments

**icon**

String or cell array of strings to display as the block icon. Each cell is displayed as a separate line.

### Examples

#### Add System Block Icon Name

Specify in your class definition file the name of the block icon as 'Enhanced Counter' using two lines.

```
methods (Access = protected)
    function icon = getIconImpl(~)
        icon = {'Enhanced', 'Counter'};
    end
end
```

#### See Also

matlab.system.mixin.CustomIcon

#### How To

- “Define MATLAB System Block Icon”

# matlab.system.mixin.FiniteSource class

**Package:** matlab.system.mixin

Finite source mixin class

## Description

`matlab.system.mixin.FiniteSource` is a class that defines the `isDone` method, which reports the state of a finite data source, such as an audio file.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System &...  
    matlab.system.mixin.FiniteSource
```

## Methods

<code>isDoneImpl</code>	End-of-data flag
-------------------------	------------------

## See Also

`matlab.System`

## Tutorials

- “Define Finite Source Objects”

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

# isDoneImpl

**Class:** matlab.system.mixin.FiniteSource

**Package:** matlab.system.mixin

End-of-data flag

## Syntax

```
status = isDoneImpl(obj)
```

## Description

`status = isDoneImpl(obj)` indicates if an end-of-data condition has occurred. The `isDone` method should return **false** when data from a finite source has been exhausted, typically by having read and output all data from the source. You should also define the result of future reads from an exhausted source in the `isDoneImpl` method.

`isDoneImpl` is called by the `isDone` method.

---

**Note:** You must set `Access = protected` for this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

**status**

Logical value, **true** or **false**, that indicates if an end-of-data condition has occurred or not, respectively.

## Examples

### Check for End-of-Data

Set up the `isDoneImpl` method in your class definition file so the `isDone` method checks whether the object has completed eight iterations.

```
methods (Access = protected)
    function bdone = isDoneImpl(obj)
        bdone = obj.NumIters==8;
    end
end
```

### See Also

`matlab.system.mixin.FiniteSource`

### How To

- “Define Finite Source Objects”

## **matlab.system.mixin.Nondirect class**

**Package:** matlab.system.mixin

Nondirect feedthrough mixin class

### **Description**

`matlab.system.mixin.Nondirect` is a class that uses the `output` and `update` methods to process nondirect feedthrough data through a `System` object.

For `System` objects that use direct feedthrough, the object's input is needed to generate the output at that time. For these direct feedthrough objects, the `step` method calculates the output and updates the state values. For nondirect feedthrough, however, the object's output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with `outputImpl` is separated from updating the state values with `updateImpl`. If you use the `matlab.system.mixin.Nondirect` mixin and include the `stepImpl` method in your class definition file, an error occurs. In this case, you must include the `updateImpl` and `outputImpl` methods instead.

The following cases describe when `System` objects in Simulink use direct or nondirect feedthrough.

- `System` object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the `System` object code.
- `System` object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- `System` object does not support code generation — Default `isInputDirectFeedthrough` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Use the `Nondirect` mixin to allow a `System` object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.



To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...  
    matlab.system.mixin.Nondirect
```

## Methods

<code>isInputDirectFeedthroughImpl</code>	Direct feedthrough status of input
<code>outputImpl</code>	Output calculation from input or internal state of <code>System</code> object
<code>updateImpl</code>	Update object states based on inputs

## See Also

`matlab.system`

## Tutorials

- “Use Update and Output for Nondirect Feedthrough”

## How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

# isInputDirectFeedthroughImpl

**Class:** matlab.system.mixin.Nondirect

**Package:** matlab.system.mixin

Direct feedthrough status of input

## Syntax

[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,u1,u2,...,uN)

## Description

[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,u1,u2,...,uN) indicates whether each input is a direct feedthrough input. If direct feedthrough is `true`, the output depends on the input at each time instant.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties or implement or access tunable properties in this method.

---

If you do not include the `isInputDirectFeedthroughImpl` method in your System object class definition file, all inputs are assumed to be direct feedthrough.

The following cases describe when System objects in Simulink code generation use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthrough` method returns false, indicating that direct

feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

`isInputDirectFeedthroughImpl` is called by the `isInputDirectFeedthrough` method.

## Input Arguments

**obj**

System object handle

**u1, u2, ..., uN**

Specifications of the inputs to the algorithm or `step` method.

## Output Arguments

**flag1, ..., flagN**

Logical value or either `true` or `false`. This value indicates whether the corresponding input is direct feedthrough or not, respectively. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

## Examples

### Specify Input as Nondirect Feedthrough

Use `isInputDirectFeedthroughImpl` in your class definition file to mark the inputs as nondirect feedthrough.

```
methods (Access = protected)
    function flag = isInputDirectFeedthroughImpl(~,~)
        flag = false;
    end
end
```

### See Also

`matlab.system.mixin.Nondirect`

**How To**

- “Use Update and Output for Nondirect Feedthrough”

# outputImpl

**Class:** matlab.system.mixin.Nondirect

**Package:** matlab.system.mixin

Output calculation from input or internal state of System object

## Syntax

$[y_1, y_2, \dots, y_N] = \text{outputImpl}(\text{obj}, u_1, u_2, \dots, u_N)$

## Description

$[y_1, y_2, \dots, y_N] = \text{outputImpl}(\text{obj}, u_1, u_2, \dots, u_N)$  implements the output equations for the System object. The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

outputImpl is called by the output method. It is also called before the updateImpl method in the step method. For sink objects, calling updateImpl before outputImpl locks the object. For all other types of objects, calling updateImpl before outputImpl causes an error.

---

**Note:** You must set Access = protected for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

---

## Input Arguments

### obj

System object handle

### u1, u2, . . . uN

Inputs from the algorithm or step method. The number of inputs must match the number of inputs returned by the getNumInputs method. Nondirect feedthrough inputs

are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

## Output Arguments

$y_1, y_2, \dots, y_N$

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

## Examples

### Set Up Output that Does Not Depend on Input

Specify in your class definition file that the output does not directly depend on the current input with the `outputImpl` method. `PreviousInput` is a property of the `obj`.

```
methods (Access = protected)
    function [y] = outputImpl(obj, ~)
        y = obj.PreviousInput(end);
    end
end
```

### See Also

`matlab.system.mixin.Nondirect`

### How To

- “Use Update and Output for Nondirect Feedthrough”

# updateImpl

**Class:** matlab.system.mixin.Nondirect

**Package:** matlab.system.mixin

Update object states based on inputs

## Syntax

updateImpl(obj, u1, u2, ..., uN)

## Description

updateImpl(obj, u1, u2, ..., uN) implements the state update equations for the system. You use this method when your algorithm outputs depend only on the object's internal state and internal properties. Do not use this method to update the outputs from the inputs.

updateImpl is called by the `update` method and after the `outputImpl` method in the `step` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink MATLAB System block.

---

## Input Arguments

**obj**

System object handle

**u1, u2, ..., uN**

Inputs to the algorithm or `step` method. The number of inputs must match the number of inputs returned by the `getNumInputs` method.

### Examples

#### Set Up Output that Does Not Depend on Current Input

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```
methods (Access = protected)
    function updateImpl(obj,u)
        obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
    end
end
```

#### See Also

`matlab.system.mixin.Nondirect`

#### How To

- “Use Update and Output for Nondirect Feedthrough”



# matlab.system.mixin.Propagates class

**Package:** matlab.system.mixin

Signal characteristics propagation mixin class

## Description

`matlab.system.mixin.Propagates` defines the output size, data type, and complexity of a System object. Use this mixin class and its methods when you will include your System object in Simulink via the `MATLAB System` block. This mixin is called by the `MATLAB System` block during Simulink model compilation.

Implement the methods of this class when Simulink cannot infer the output specifications directly from the inputs or when you want bus support. If you do not include this mixin, Simulink cannot propagate the output or bus data type, an error occurs.

To use this mixin, subclass from this `matlab.system.mixin.Propagates` in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file. `ObjectName` is the name of your System object.

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.Propagates
```

## Methods

<code>getDiscreteStateSpecificationImpl</code>	Discrete state size, data type, and complexity
<code>getOutputDataTypeImpl</code>	Data types of output ports
<code>getOutputSizeImpl</code>	Sizes of output ports
<code>isOutputComplexImpl</code>	Complexity of output ports
<code>isOutputFixedSizeImpl</code>	Fixed- or variable-size output ports
<code>propagatedInputComplexity</code>	Complexity of input during Simulink propagation

<code>propagatedInputDataType</code>	Data type of input during Simulink propagation
<code>propagatedInputFixedSize</code>	Fixed-size status of input during Simulink propagation
<code>propagatedInputSize</code>	Size of input during Simulink propagation

---

**Note:** If your `System` object has exactly one input and one output and no discrete property states, or if you do not need bus support, you do not have to implement any of these methods. The `matlab.system.mixin.Propagates` provides default values in these cases.

---

### See Also

`matlab.System`

### Tutorials

- “Set Output Data Type”
- “Set Output Size”
- “Set Output Complexity”
- “Specify Whether Output Is Fixed- or Variable-Size”
- “Specify Discrete State Output Specification”

### How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

# getDiscreteStateSpecificationImpl

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Discrete state size, data type, and complexity

## Syntax

```
[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,name)
```

## Description

`[sz,dt,cp] = getDiscreteStateSpecificationImpl(obj,name)` returns the size, data type, and complexity of the property, `name`. This property must be a discrete state property. You must define this method if your System object has discrete state properties and is used in the MATLAB System block. If you define this method for a property that is not discrete state, an error occurs during model compilation.

You always set the `getDiscreteStateSpecificationImpl` method access to `protected` because it is an internal method that users do not directly call or run.

`getDiscreteStateSpecificationImpl` is called by the MATLAB System block during Simulink model compilation.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object handle

### **name**

Name of discrete state property of the System object

## Output Arguments

### **sz**

Vector containing the length of each dimension of the property.

**Default:** [1 1]

### **dt**

Data type of the property. For built-in data types, **dt** is a string. For fixed-point data types, **dt** is a `numericType` object.

**Default:** `double`

### **cp**

Complexity of the property as a scalar, logical value, where `true` = complex and `false` = real.

**Default:** `false`

## Examples

### Specify Discrete State Property Size, Data Type, and Complexity

Specify in your class definition file the size, data type, and complexity of a discrete state property.

```
methods (Access = protected)
    function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
        sz = [1 1];
        dt = 'double';
        cp = false;
    end
```

end

## See Also

matlab.system.mixin.Propagates

## How To

- “Specify Discrete State Output Specification”

## getOutputDataTypeImpl

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Data types of output ports

### Syntax

```
[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)
```

### Description

[dt\_1,dt\_2,...,dt\_n] = getOutputDataTypeImpl(obj) returns the data types of each output port. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `step` method.

For System objects with one input and one output and where you want the input and output data types to be the same, you do not need to implement this method. In this case `getOutputDataTypeImpl` assumes the input and output data types are the same and returns the data type of the input.

If your System object has more than one input or output or you need the output and input data types to be different, you must implement the `getOutputDataTypeImpl` method to define the output data type. For bus output, you must also specify the name of the output bus. You use `propagatedInputDataType` within the `getOutputDataTypeImpl` method to obtain the input type, if necessary.

During Simulink model compilation and propagation, the MATLAB System block calls the `getOutputDataType` method, which then calls the `getOutputDataTypeImpl` method to determine the output data type.

---

**Note:** You must set `Access = protected` for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object

## Output Arguments

**dt\_1, dt\_2, ...**

Data type of the property. For built-in data types, **dt** is a string. For fixed-point data types, **dt** is a `numerictype` object.

## Examples

### Specify Output Data Type

Specify, in your class definition file, the data type of a System object with one output.

```
methods (Access = protected)
    function dt_1 = getOutputDataTypeImpl(~)
        dt_1 = 'double';
    end
end
```

### Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

```
properties(Nontunable)
    OutputBusName = 'myBus';
end

methods (Access = protected)
    function out = getOutputDataTypeImpl(obj)
        out = obj.OutputBusName;
    end
end
```

[end](#)

**See Also**

`matlab.system.mixin.Propagates` | `propagatedInputDataType`



# getOutputSizeImpl

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Sizes of output ports

## Syntax

```
[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)
```

## Description

[sz\_1,sz\_2,...,sz\_n] = getOutputSizeImpl(obj) returns the sizes of each output port. The number of outputs must match the value returned from the getNumOutputs method or the number of output arguments listed in the step method.

For System objects with one input and one output and where you want the input and output sizes to be the same, you do not need to implement this method. In this case getOutputSizeImpl assumes the input and output sizes are the same and returns the size of the input. For variable-size inputs in MATLAB, the size varies each time you run step on your object. For variable-size inputs in Simulink, the output size is the maximum input size.

If your System object has more than one input or output or you need the output and input sizes to be different, you must implement the getOutputSizeImpl method to define the output size. You also must use the propagatedInputSize method if the output size differs from the input size.

During Simulink model compilation and propagation, the MATLAB System block calls the getOutputSize method, which then calls the getOutputSizeImpl method to determine the output size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

---

**Note:** You must set Access = protected for this method.

You cannot modify any properties in this method.

---

### Input Arguments

**obj**

System object handle

### Output Arguments

**sz\_1, sz\_2, ...**

Vector containing the size of each output port.

### Examples

#### Specify Output Size

Specify in your class definition file the size of a System object output.

```
methods (Access = protected)
    function sz_1 = getOutputSizeImpl(obj)
        sz_1 = [1 1];
    end
end
```

#### Specify Multiple Output Ports

Specify in your class definition file the sizes of multiple System object outputs.

```
methods (Access = protected)
    function [sz_1, sz_2] = getOutputSizeImpl(obj)
        sz_1 = propagatedInputSize(obj, 1);
        sz_2 = [1 1];
    end
end
```

#### Specify Output When Using Propagated Input Size

Specify in your class definition file the size of System object output when it is dependant on the propagated input size.

```
methods (Access = protected)
    function varargout = getOutputSizeImpl(obj)
        varargout{1} = propagatedInputSize(obj,1);
        if obj.HasSecondOutput
            varargout{2} = [1 1];
        end
    end
end
end
```

## See Also

matlab.system.mixin.Propagates | propagatedInputSize

## How To

- “Set Output Size”

## isOutputComplexImpl

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Complexity of output ports

### Syntax

```
[cp_1,cp_2,...,cp_n] = isOutputComplexImpl(obj)
```

### Description

[cp\_1,cp\_2,...,cp\_n] = isOutputComplexImpl(obj) returns whether each output port has complex data. The number of outputs must match the value returned from the getNumOutputs method or the number of output arguments listed in the step method.

For System objects with one input and one output and where you want the input and output complexities to be the same, you do not need to implement this method. In this case isOutputComplexImpl assumes the input and output complexities are the same and returns the complexity of the input.

If your System object has more than one input or output or you need the output and input complexities to be different, you must implement the isOutputComplexImpl method to define the output complexity. You also must use the propagatedInputComplexity method if the output complexity differs from the input complexity.

During Simulink model compilation and propagation, the MATLAB System block calls the isOutputComplex method, which then calls the isOutputComplexImpl method to determine the output complexity.

---

**Note:** You must set Access = protected for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

**cp\_1, cp\_2, ...**

Logical, scalar value indicating whether the specific output port is complex (`true`) or real (`false`).

## Examples

### Specify Output as Real-Valued

Specify in your class definition file that the output from a System object is a real value.

```
methods (Access = protected)
    function c1 = isOutputComplexImpl(obj)
        c1 = false;
    end
end
```

### See Also

[matlab.system.mixin.Propagates](#) | [propagatedInputComplexity](#)

### How To

- “Set Output Complexity”

# isOutputFixedSizeImpl

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Fixed- or variable-size output ports

## Syntax

```
[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)
```

## Description

[flag\_1,flag\_2,...flag\_n] = isOutputFixedSizeImpl(obj) indicates whether each output port is fixed size. The number of outputs must match the value returned from the getNumOutputs method, which is the number of output arguments listed in the step method.

For System objects with one input and one output and where you want the input and output fixed sizes to be the same, you do not need to implement this method. In this case isOutputFixedSizeImpl assumes the input and output fixed sizes are the same and returns the fixed size of the input.

If your System object has more than one input or output or you need the output and input fixed sizes to be different, you must implement the isOutputFixedSizeImpl method to define the output fixed size. You also must use the propagatedInputFixedSize method if the output fixed size status differs from the input fixed size status.

During Simulink model compilation and propagation, the MATLAB System block calls the isOutputFixedSize method, which then calls the isOutputFixedSizeImpl method to determine the output fixed size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

---

**Note:** You must set Access = protected for this method.

You cannot modify any properties in this method.

---

## Input Arguments

**obj**

System object handle

## Output Arguments

**flag\_1, flag\_2, ...**

Logical, scalar value indicating whether the specific output port is fixed size (`true`) or variable size (`false`).

## Examples

### Specify Output as Fixed Size

Specify in your class definition file that the output from a System object is of fixed size.

```
methods (Access = protected)
    function c1 = isOutputFixedSizeImpl(obj)
        c1 = true;
    end
end
```

### See Also

[matlab.system.mixin.Propagates](#) | [propagatedInputFixedSize](#)

### How To

- “Specify Whether Output Is Fixed- or Variable-Size”

# propagatedInputComplexity

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Complexity of input during Simulink propagation

## Syntax

```
flag = propagatedInputComplexity(obj,index)
```

## Description

`flag = propagatedInputComplexity(obj,index)` returns `true` or `false` to indicate whether the input argument for the indicated System object is complex. `index` specifies the `step` method input for which to return the complexity flag.

You can use `propagatedInputComplexity` only from within the `isOutputComplexImpl` method in your class definition file. Use `isOutputComplexImpl` when:

- Your System object has more than one input or output.
- The input complexity determines the output complexity.
- The output complexity must differ from the input complexity.

## Input Arguments

### **obj**

System object

### **index**

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first `step` input is always `obj`.



## Output Arguments

### flag

Complexity of the specified input, returned as `true` or `false`

## Examples

### Match Input and Output Complexity

Get the complexity of the second input to the `step` method and set the output to match it. Assume that the first input of the `step` method has no impact on the output complexity.

```
methods (Access = protected)
    function outcomplx = isOutputComplexImpl(obj)
        outcomplx = propagatedInputComplexity(obj,2);
    end
end
```

### See Also

[matlab.system.mixin.Propagates](#) | [isOutputComplexImpl](#)

### How To

- “Set Output Complexity”

# propagatedInputDataType

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Data type of input during Simulink propagation

## Syntax

```
dt = propagatedInputDataType(obj,index)
```

## Description

`dt = propagatedInputDataType(obj,index)` returns the data type of an input argument for a System object. `index` specifies the `step` method input for which to return the data type.

You can use `propagatedInputDataType` only from within `getOutputDataTypeImpl`. Use `getOutputDataTypeImpl` when:

- Your System object has more than one input or output.
- The input data type status determines the output data type.
- The output data type must differ from the input data type.

## Input Arguments

### **obj**

System object

### **index**

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first step input is always `obj`.

## Output Arguments

**dt**

Data type of the specified `step` method input, returned as a string for floating-point input or as a numeric type for fixed-point input.

## Examples

### Match Input and Output Data Type

Get the data type of the second input to the `step` method. If the second input data type is `double`, then the output data type is `int32`. For all other cases, the output data type matches the second input data type. Assume that the first input to the `step` method has no impact on the output.

```
methods (Access = protected)
    function dt = getOutputDataTypeImpl(obj)
        if strcmpi(propagatedInputDataType(obj,2), 'double')
            dt = 'int32';
        else
            dt = propagatedInputDataType(obj,2);
        end
    end
end
```

### See Also

“Data Type Propagation” | [matlab.system.mixin.Propagates](#) | [getOutputDataTypeImpl](#)

### How To

- “Set Output Data Type”

# propagatedInputFixedSize

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Fixed-size status of input during Simulink propagation

## Syntax

```
flag = propagatedInputFixedSize(obj,index)
```

## Description

`flag = propagatedInputFixedSize(obj,index)` returns `true` or `false` to indicate whether an input argument of a `System` object is fixed size. `index` specifies the `step` method input for which to return the fixed-size flag.

You can use `propagatedInputFixedSize` only from within `isOutputFixedSizeImpl`. Use `isOutputFixedSizeImpl` when:

- Your `System` object has more than one input or output.
- The input fixed-size status determines the output fixed-size status.
- The output fixed-size status must differ from the input fixed-size status.

## Input Arguments

### **obj**

System object

### **index**

Index of the specified `step` method input. Do not count the `obj` in the `index`. The first step input is always `obj`.

## Output Arguments

### flag

Fixed-size status of the specified `step` method input, returned as `true` or `false`.

## Examples

### Match Fixed-Size Status of Input and Output

Get the fixed-size status of the third input to the `step` method and set the output to match it. Assume that the first and second inputs to the `step` method have no impact on the output.

```
methods (Access = protected)
    function outtype = isOutputFixedSizeImpl(obj)
        outtype = propagatedInputFixedSize(obj,3)
    end
end
```

### See Also

[matlab.system.mixin.Propagates](#) | [isOutputFixedSizeImpl](#)

### How To

- “Specify Whether Output Is Fixed- or Variable-Size”

## propagatedInputSize

**Class:** matlab.system.mixin.Propagates

**Package:** matlab.system.mixin

Size of input during Simulink propagation

### Syntax

```
sz = propagatedInputSize(obj,index)
```

### Description

`sz = propagatedInputSize(obj,index)` returns, as a vector, the input size of the specified System object. The `index` specifies the input to the `step` method for which to return the size information. (Do not count the `obj` in the `index`. The first input is always `obj`.)

You can use `propagatedInputSize` only from within the `getOutputSizeImpl` method in your class definition file. Use `getOutputSizeImpl` when:

- Your System object has more than one input or output.
- The input size determines the output size.
- The output size must differ from the input size.

---

**Note:** For variable-size inputs, the propagated input size from `propagatedInputSizeImpl` differs depending on the environment.

- MATLAB — When you first run `step` on an object, it uses the actual sizes of the inputs.
  - Simulink — The maximum of all the input sizes is set before the model runs and does not change during the run.
-

## Input Arguments

### **obj**

System object

### **index**

Index of the specified `step` method input

## Output Arguments

### **sz**

Size of the specified input, returned as a vector

## Examples

### Match Size of Input and Output

Get the size of the second input to the `step` method. If the first dimension of the second input to the `step` method has a size greater than 1, then set the output size to a 1 x 2 vector. For all other cases, the output is a 2 x 1 matrix. Assume that the first input to the `step` method has no impact on the output size.

```
methods (Access = protected)
    function outsz = getOutputSizeImpl(obj)
        sz = propagatedInputSize(obj,2);
        if sz(1) == 1
            outsz = [1,2];
        else
            outsz = [2,1];
        end
    end
end
```

### See Also

[matlab.system.mixin.Propagates](#) | [getOutputSizeImpl](#)

## **How To**

- “Set Output Size”



# matlab.system.StringSet class

**Package:** matlab.system

Set of valid string values

## Description

`matlab.system.StringSet` defines a list of valid string values for a property. This class validates the string in the property and enables tab completion for the property value. A *StringSet* allows only predefined or customized strings as values for the property.

A `StringSet` uses two linked properties, which you must define in the same class. One is a public property that contains the current string value. This public property is displayed to the user. The other property is a hidden property that contains the list of all possible string values. This hidden property should also have the transient attribute so its value is not saved to disk when you save the System object.

The following considerations apply when using StringSets:

- The string property that holds the current string can have any name.
- The property that holds the `StringSet` must use the same name as the string property with the suffix “Set” appended to it. The string set property is an instance of the `matlab.system.StringSet` class.
- Valid strings, defined in the `StringSet`, must be declared using a cell array. The cell array cannot be empty nor can it have any empty strings. Valid strings must be unique and are case-sensitive.
- The string property must be set to a valid `StringSet` value.

## Examples

### Set String Property Values

Set the string property, `Flavor`, and the `StringSet` property, `FlavorSet` in your class definition file.

```
properties
    Flavor = 'Chocolate';
end

properties (Hidden,Transient)
    FlavorSet = ...
        matlab.system.StringSet({'Vanilla', 'Chocolate'});
end
```

### See Also

matlab.System

### How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Limit Property Values to Finite String Set”

# Functions Alphabetical

---

alignColorToDepth  
assignDetectionsToTracks  
bbox2points  
bboxOverlapRatio  
bundleAdjustment  
OCR Trainer  
listTrueTypeFonts  
pfromkinect  
Camera Calibrator  
Stereo Camera Calibrator  
cameraMatrix  
cameraPose  
extractLBPFeatures  
configureKalmanFilter  
depthToPointCloud  
detectBRISKFeatures  
detectCheckerboardPoints  
detectFASTFeatures  
detectHarrisFeatures  
detectMinEigenFeatures  
detectMSERFeatures  
detectPeopleACF  
detectSURFFeatures  
disparity  
epipolarLine  
estimateCameraParameters  
estimateFundamentalMatrix  
estimateGeometricTransform  
estimateUncalibratedRectification  
evaluateImageRetrieval  
extractFeatures  
extractHOGFeatures

extrinsics  
generateCheckerboardPoints  
indexImages  
integralFilter  
integralImage  
insertMarker  
insertObjectAnnotation  
insertShape  
insertText  
isEpipoleInImage  
isfilterseparable  
lineToBorderPoints  
matchFeatures  
mplay  
ocr  
pcdenoise  
pcmerge  
pcdownsample  
pcread  
pcregrigid  
pcwrite  
pctransform  
pcnormals  
pcfitcylinder  
pcfitplane  
pcfitsphere  
plotCamera  
reconstructScene  
rectifyStereoImages  
retrieveImages  
rotationMatrixToVector  
rotationVectorToMatrix  
selectStrongestBbox  
showExtrinsics  
showMatchedFeatures  
showPointCloud  
pcshow  
pcshowpair  
showReprojectionErrors  
stereoAnaglyph

trainCascadeObjectDetector  
trainImageCategoryClassifier  
Training Image Labeler  
triangulate  
triangulateMultiview  
undistortImage  
undistortPoints  
vision.getCoordinateSystem  
vision.setCoordinateSystem  
visionlib  
visionSupportPackages  
ocvStructToKeyPoints  
ocvMxGpuArrayToGpuMat\_{DataType}  
ocvMxGpuArrayFromGpuMat\_{DataType}  
ocvMxArrayToSize  
ocvMxArrayToMat\_{DataType}  
ocvMxArrayToImage\_{DataType}  
ocvMxArrayToCvRect  
ocvMxArrayFromVector  
ocvMxArrayFromPoints2f  
ocvMxArrayFromMat\_{DataType}  
ocvMxArrayFromImage\_{DataType}  
ocvKeyPointsToStruct  
ocvCvRectToMxArray  
ocvCvRectToBoundingBox\_{DataType}  
ocvCvBox2DToMxArray  
ocvCheckFeaturePointsStruct

# alignColorToDepth

Align Kinect color image to depth image

## Syntax

```
[alignedFlippedImage,flippedDepthImage] = alignColorToDepth(  
depthImage,colorImage,depthDevice)
```

## Description

[alignedFlippedImage,flippedDepthImage] = alignColorToDepth(depthImage,colorImage,depthDevice) returns a truecolor image alignedFlippedImage, which is aligned with flippedDepthImage. The inputs are obtained from Kinect for Windows.

This function requires the Image Acquisition Toolbox™.

---

**Note:** The alignColorToDepth will be removed in a future release. Use the pcfFromKinect function with equivalent functionality instead.

---

## Examples

### Plot Point Cloud From Kinect for Windows.

Plot a color point cloud from Kinect images. This example requires the Image Acquisition Toolbox software and the Kinect camera and connection.

Create System objects for the Kinect device.

```
colorDevice = imaq.VideoDevice('kinect',1)  
depthDevice = imaq.VideoDevice('kinect',2)
```

Change the returned type of color image from single to uint8.

```
colorDevice.ReturnedDataType = 'uint8';
```

Warm up the cameras.

```
step(colorDevice);  
step(depthDevice);
```

Load one frame from each device. The initial frame executes slowly because the objects must wake up the devices.

```
colorImage = step(colorDevice);  
depthImage = step(depthDevice);
```

Convert the depth image to a point cloud.

```
xyzPoints = depthToPointCloud(depthImage,depthDevice);
```

Align the color image with the depth image.

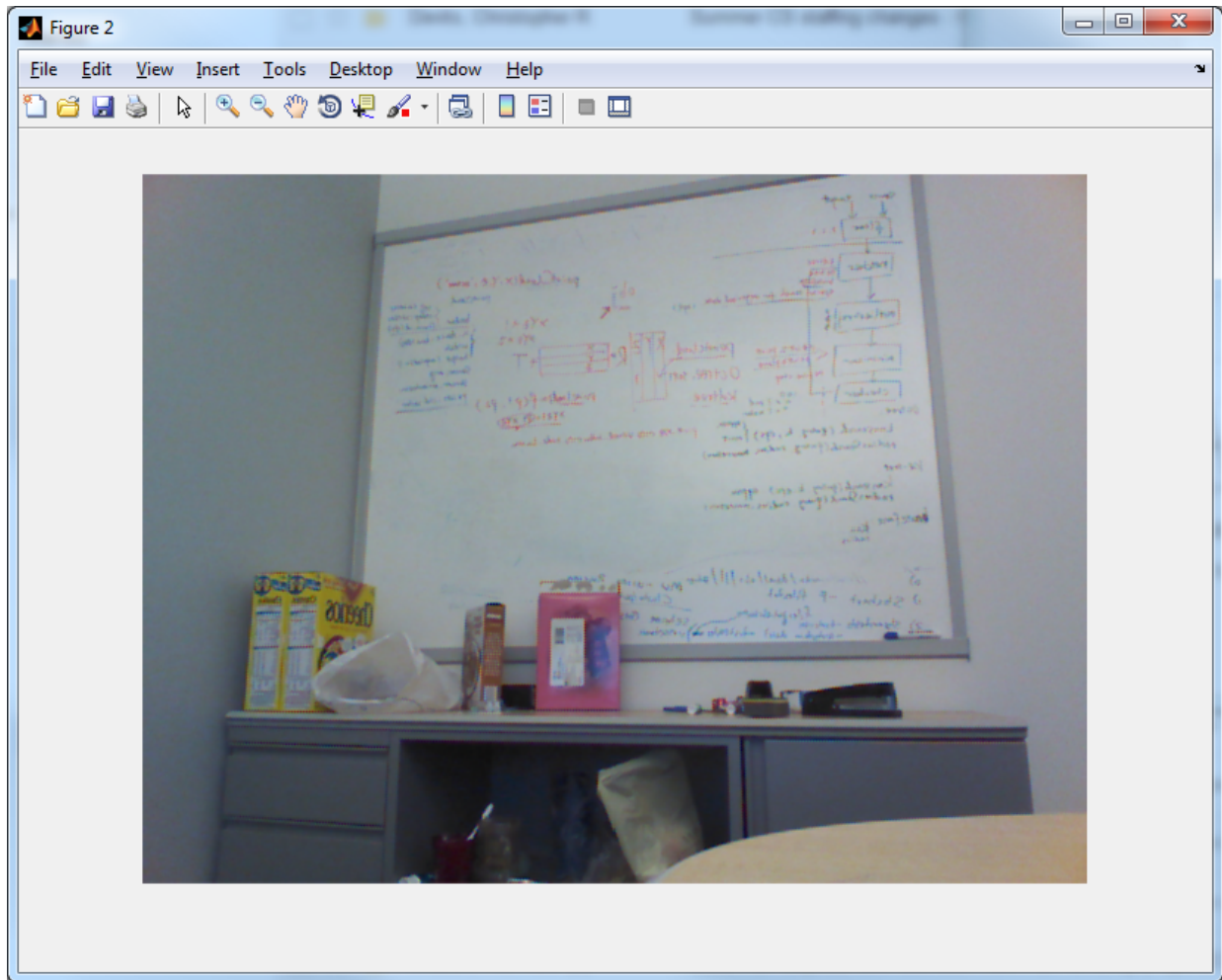
```
alignedColorImage = alignColorToDepth(depthImage,colorImage,depthDevice);
```

Render the point cloud with color. The axis is set to better visualize the point cloud.

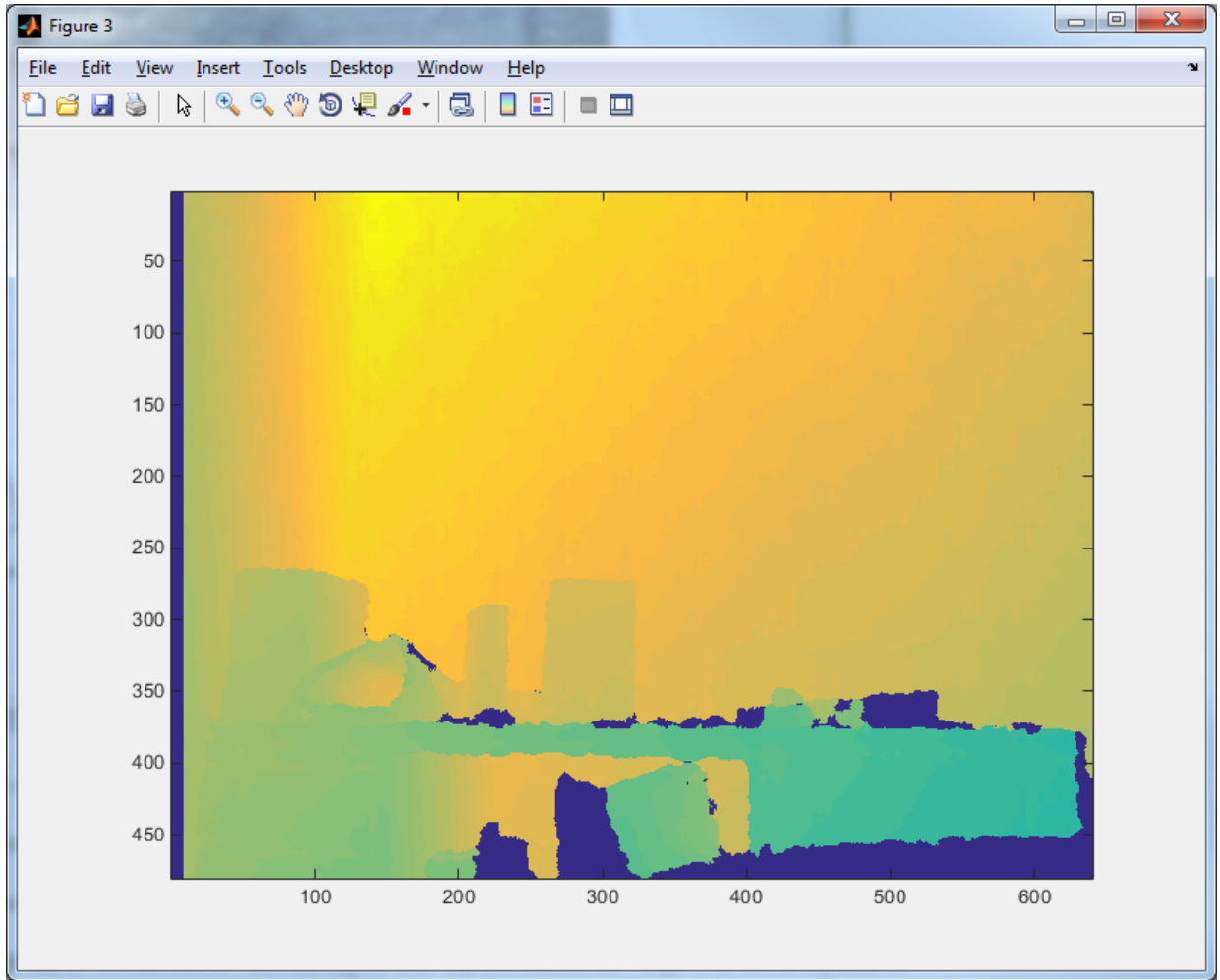
```
pcshow(xyzPoints,alignedColorImage,'VerticalAxis','y','VerticalAxisDir','down');  
xlabel('X (m)');  
ylabel('Y (m)');  
zlabel('Z (m)');
```

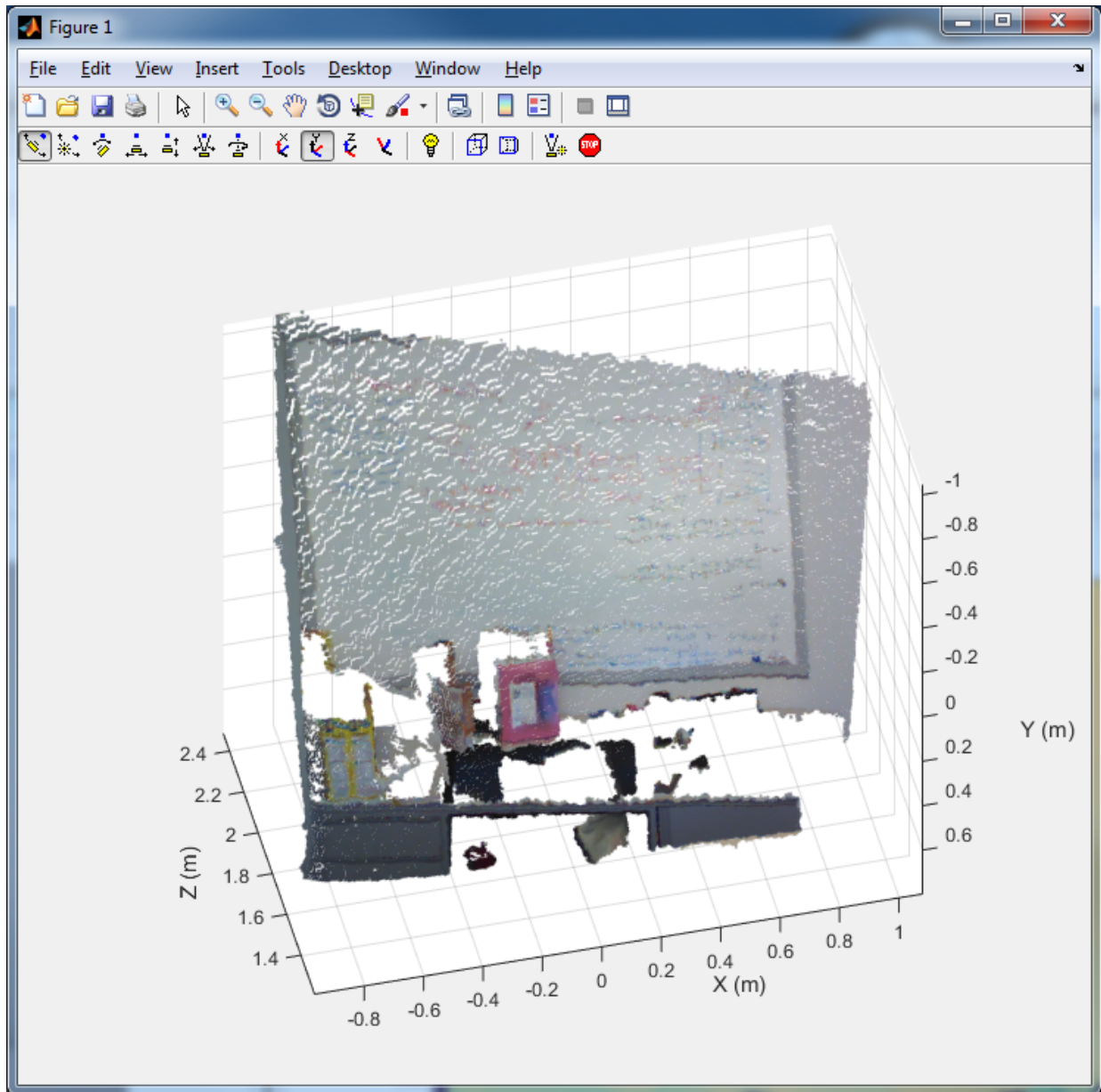
Release the System objects.

```
release(colorDevice);  
release(depthDevice);
```









- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **depthImage** — Depth image

*M*-by-*N* matrix

Depth image, specified as an *M*-by-*N* matrix. The original images, **depthImage** and **colorImage** from Kinect are mirror images of the scene.

Data Types: `uint16`

### **colorImage** — Color image

*M*-by-*N*-by-3 RGB truecolor image

Color image, specified as an *M*-by-*N*-by-3 RGB truecolor image returned by Kinect. The original images, **depthImage** and **colorImage** from Kinect are mirror images of the scene.

Data Types: `uint16`

### **depthDevice** — Video input object

`videoinput` object | `imaq.VideoDevice` object

Video input object, specified as either a `videoinput` object or an `imaq.VideoDevice` object configured for Kinect for Windows.

## Output Arguments

### **flippedDepthImage** — Depth image

*M*-by-*N* matrix

Depth image, returned as an *M*-by-*N* matrix. The Kinect system, designed for gaming applications, returns a mirror image of the scene. This function corrects the output depth image to match the actual scene. It returns the **flippedDepthImage** left-to-right version of the input **depthImage**.

You can obtain the same functionality using `fliplr(depthImage)`.

### **alignedFlippedImage** — Aligned flipped image

*M*-by-*N*-by-3 truecolor image

Aligned flipped image, returned as an *M*-by-*N*-by-3 truecolor image. The image is aligned with the **flippedDepthImage** output. The Kinect system, designed for gaming

applications, returns a mirror image of the scene. This function corrects the depth image and the aligned flipped image outputs to match the actual scene.

### **More About**

- “Coordinate Systems”

### **See Also**

`imaq.VideoDevice` | `pcfromkinect` | `pcshow` | `videoinput`

**Introduced in R2014b**

# assignDetectionsToTracks

Assign detections to tracks for multiobject tracking

## Syntax

```
[assignments,unassignedTracks,unassignedDetections] =  
assignDetectionsToTracks( costMatrix,costOfNonAssignment)  
[assignments,unassignedTracks,unassignedDetections] =  
assignDetectionsToTracks(costMatrix, unassignedTrackCost,  
unassignedDetectionCost)
```

## Description

[assignments,unassignedTracks,unassignedDetections] = assignDetectionsToTracks( costMatrix,costOfNonAssignment) assigns detections to tracks in the context of multiple object tracking using the James Munkres's variant of the Hungarian assignment algorithm. It also determines which tracks are missing and which detections should begin new tracks. It returns the indices of assigned and unassigned tracks, and unassigned detections. The `costMatrix` must be an  $M$ -by- $N$  matrix. In this matrix,  $M$  represents the number of tracks, and  $N$  is the number of detections. Each value represents the cost of assigning the  $N^{\text{th}}$  detection to the  $M^{\text{th}}$  track. The lower the cost, the more likely that a detection gets assigned to a track. The `costOfNonAssignment` scalar input represents the cost of a track or a detection remaining unassigned.

[assignments,unassignedTracks,unassignedDetections] = assignDetectionsToTracks(costMatrix, unassignedTrackCost, unassignedDetectionCost) specifies the cost of unassigned tracks and detections separately. The `unassignedTrackCost` must be a scalar value, or an  $M$ -element vector, where  $M$  represents the number of tracks. For the  $M$ -element vector, each element represents the cost of not assigning any detection to that track. The `unassignedDetectionCost` must be a scalar value or an  $N$ -element vector, where  $N$  represents the number of detections.

### Code Generation Support:

Compile-time constant input: No restriction

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Assign Detections to Tracks in a Single Video Frame

This example shows you how to assign a detection to a track for a single video frame.

Set the predicted locations of objects in the current frame. Obtain predictions using the Kalman filter System object.

```
predictions = [1,1;2,2];
```

Set the locations of the objects detected in the current frame. For this example, there are 2 tracks and 3 new detections. Thus, at least one of the detections is unmatched, which can indicate a new track.

```
detections = [1.1,1.1;2.1,2.1;1.5,3];
```

Preallocate a cost matrix.

```
cost = zeros(size(predictions,1),size(detections,1));
```

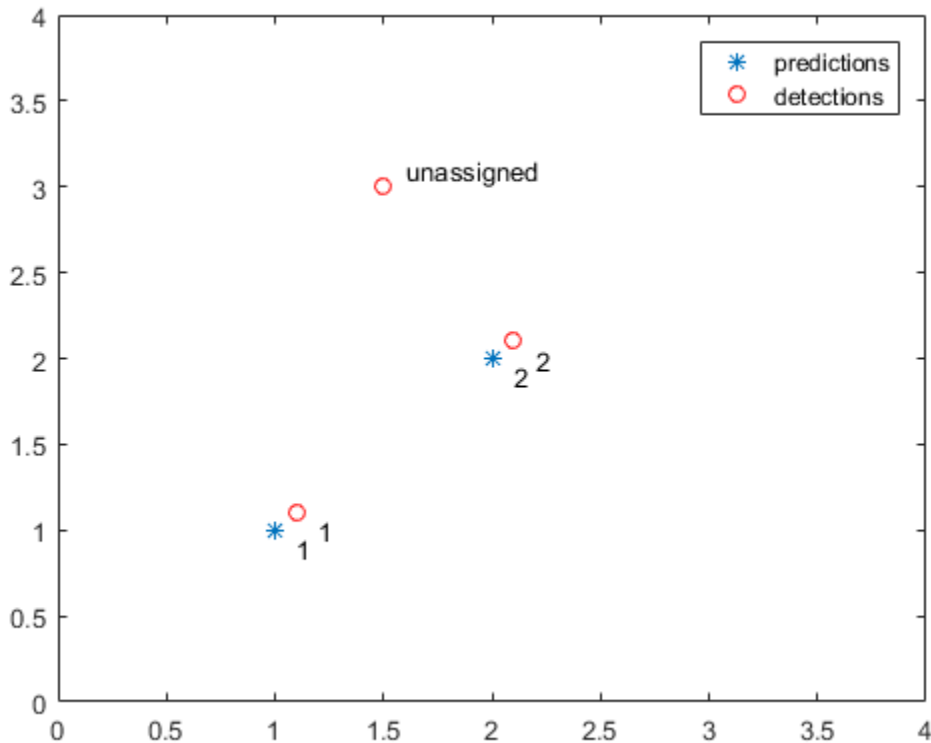
Compute the cost of each prediction matching a detection. The cost here, is defined as the Euclidean distance between the prediction and the detection.

```
for i = 1:size(predictions, 1)
    diff = detections - repmat(predictions(i,:),[size(detections,1),1]);
    cost(i, :) = sqrt(sum(diff.^2,2));
end
```

Associate detections with predictions. Detection 1 should match to track 1, and detection 2 should match to track 2. Detection 3 should be unmatched.

```
[assignment,unassignedTracks,unassignedDetections] = assignDetectionsToTracks(cost,0.2);
figure;
plot(predictions(:,1),predictions(:,2),'*',detections(:,1),detections(:,2),'ro');
hold on;
legend('predictions','detections');
for i = 1:size(assignment,1)
    text(predictions(assignment(i,1),1)+0.1,predictions(assignment(i,1),2)-0.1,num2str(i));
    text(detections(assignment(i,2),1)+0.1,detections(assignment(i,2),2)-0.1,num2str(i));
end
for i = 1:length(unassignedDetections)
    text(detections(unassignedDetections(i),1)+0.1,detections(unassignedDetections(i),2)-0.1,num2str(i));
end
```

```
end  
xlim([0,4]);  
ylim([0,4]);
```

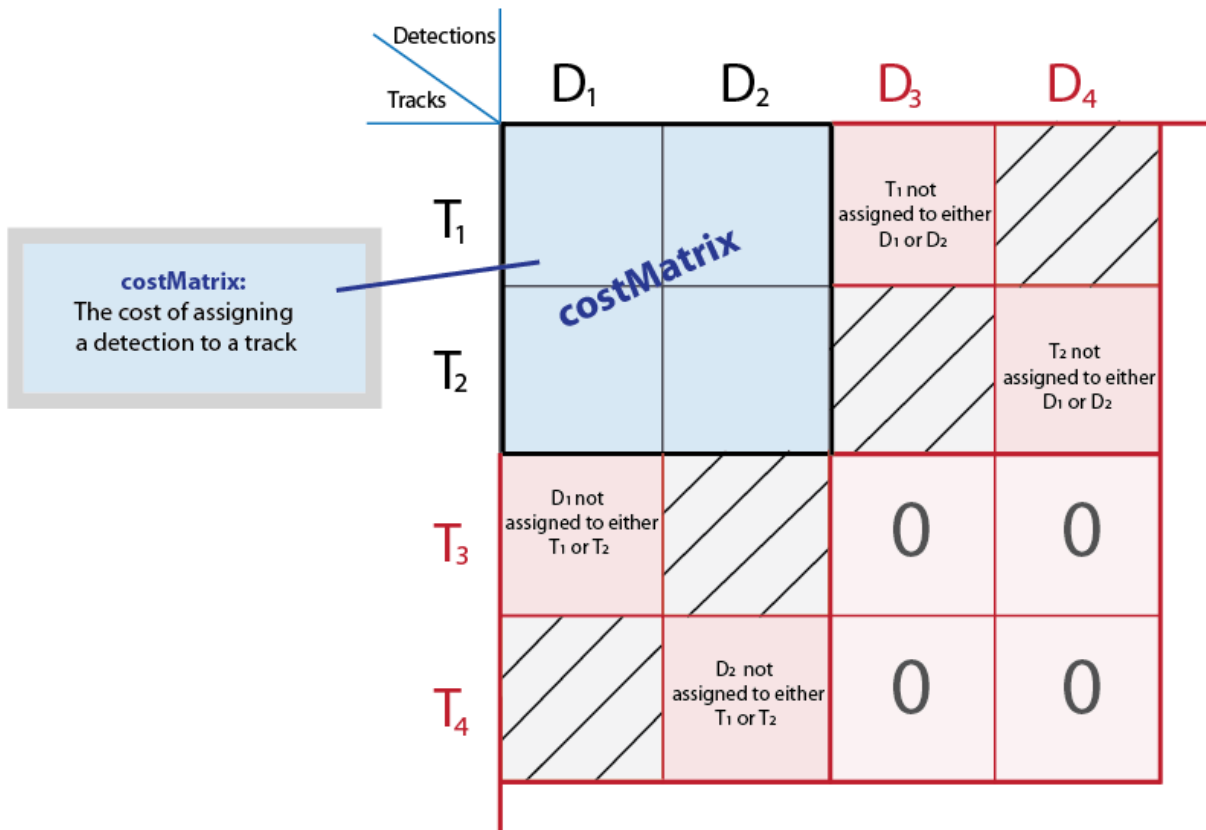


## Input Arguments

**costMatrix** — Cost of assigning detection to track  
*M*-by-*N* matrix

Cost of assigning a detection to a track, specified as an *M*-by-*N* matrix, where *M* represents the number of tracks, and *N* is the number of detections. The cost matrix value must be real and nonsparse. The lower the cost, the more likely that a detection

gets assigned to a track. Each value represents the cost of assigning the  $N^{\text{th}}$  detection to the  $M^{\text{th}}$  track. If there is no likelihood of an assignment between a detection and a track, the `costMatrix` input is set to `Inf`. Internally, this function pads the cost matrix with dummy rows and columns to account for the possibility of unassigned tracks and detections. The padded rows represent detections not assigned to any tracks. The padded columns represent tracks not associated with any detections. The function applies the Hungarian assignment algorithm to the padded matrix.



Data Types: `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `single` | `double`

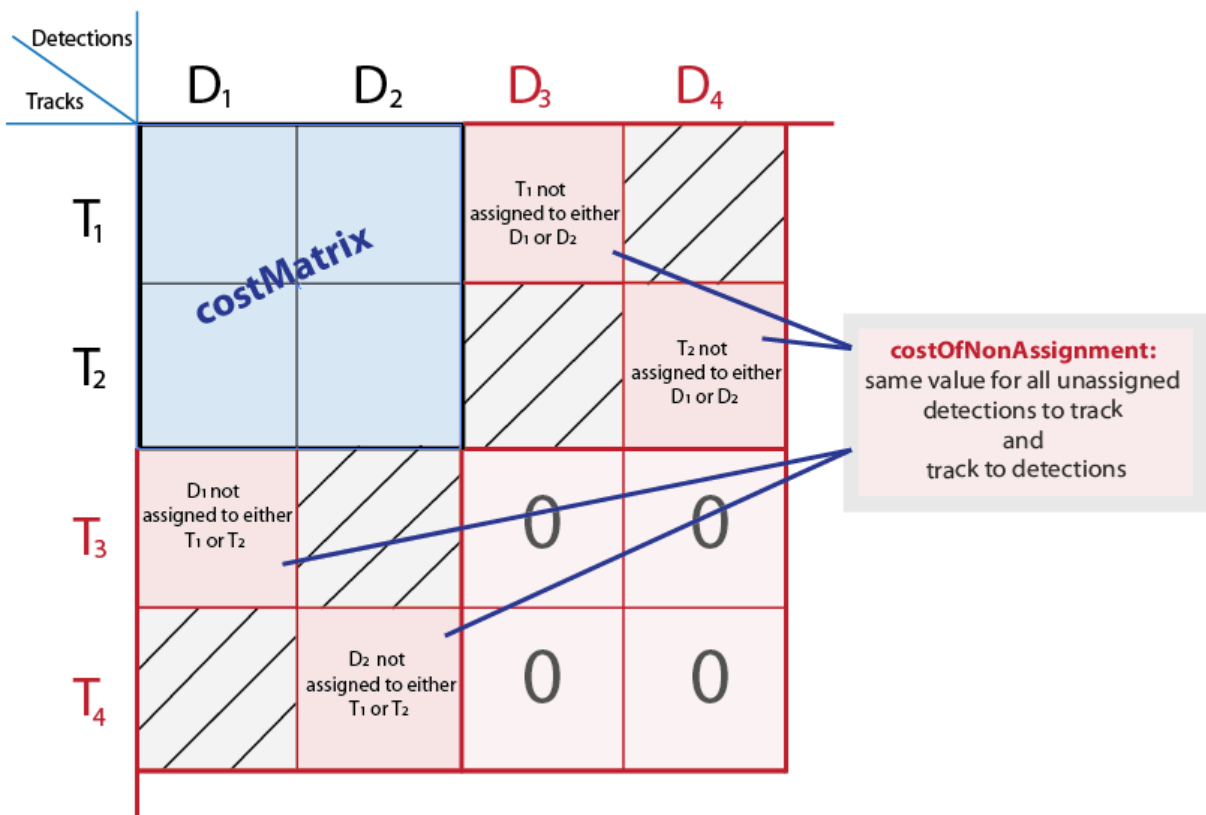
**costOfNonAssignment** — Cost of not assigning detection to any track or track to any detection

scalar | finite



Cost of not assigning detection to any track or track to detection. You can specify this value as a scalar value representing the cost of a track or a detection remaining unassigned. An unassigned detection may become the start of a new track. If a track is unassigned, the object does not appear. The higher the `costOfNonAssignment` value, the higher the likelihood that every track will be assigned a detection.

Internally, this function pads the cost matrix with dummy rows and columns to account for the possibility of unassigned tracks and detections. The padded rows represent detections not assigned to any tracks. The padded columns represent tracks not associated with any detections. To apply the same value to all elements in both the rows and columns, use the syntax with the `costOfNonAssignment` input. To vary the values for different detections or tracks, use the syntax with the `unassignedTrackCost` and `unassignedDetectionCost` inputs.



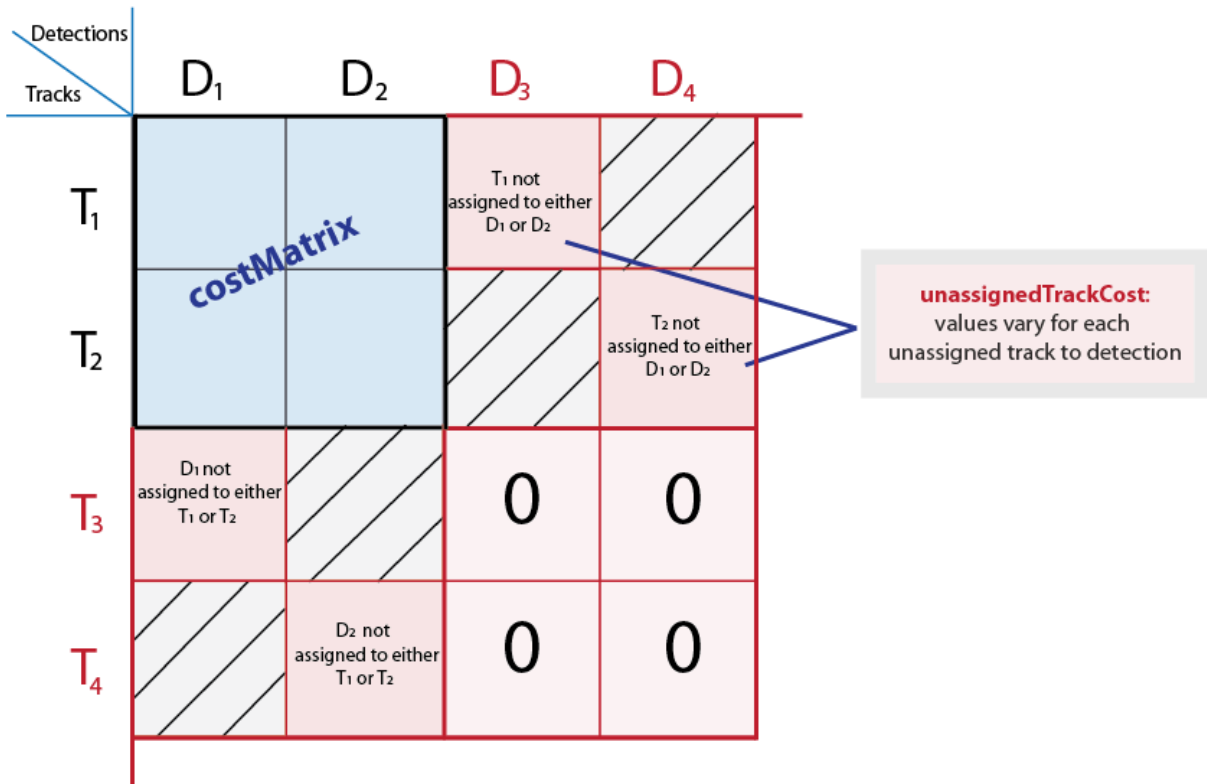
Data Types: `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `single` | `double`

#### **unassignedTrackCost** — Cost or likelihood of an unassigned track

*M*-element vector | scalar | finite

Cost or likelihood of an unassigned track. You can specify this value as a scalar value, or an *M*-element vector, where *M* represents the number of tracks. For the *M*-element vector, each element represents the cost of not assigning any detection to that track. A scalar input represents the same cost of being unassigned for all tracks. The cost may vary depending on what you know about each track and the scene. For example, if an object is about to leave the field of view, the cost of the corresponding track being unassigned should be low.

Internally, this function pads the cost matrix with dummy rows and columns to account for the possibility of unassigned tracks and detections. The padded rows represent detections not assigned to any tracks. The padded columns represent tracks not associated with any detections. To vary the values for different detections or tracks, use the syntax with the `unassignedTrackCost` and `unassignedDetectionCost` inputs. To apply the same value to all elements in both the rows and columns, use the syntax with the `costOfNonAssignment` input.



Data Types: int8 | uint8 | int16 | uint16 | int32 | uint32 | single | double

### unassignedDetectionCost — Cost of unassigned detection

*N*-element vector | scalar | finite

Cost of unassigned detection, specified as a scalar value or an *N*-element vector, where *N* represents the number of detections. For the *N*-element vector, each element represents the cost of starting a new track for that detection. A scalar input represents the same cost of being unassigned for all tracks. The cost may vary depending on what you know about each detection and the scene. For example, if a detection appears close to the edge of the image, it is more likely to be a new object.

Internally, this function pads the cost matrix with dummy rows and columns to account for the possibility of unassigned tracks and detections. The padded rows represent detections not assigned to any tracks. The padded columns represent tracks not

associated with any detections. To vary the values for different detections or tracks, use the syntax with the `unassignedTrackCost` and `unassignedDetectionCost` inputs. To apply the same value to all elements in both the rows and columns, use the syntax with the `costOfNonAssignment` input.

Detections Tracks		D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
		T <sub>1</sub>	<b>costMatrix</b>		T <sub>1</sub> not assigned to either D <sub>1</sub> or D <sub>2</sub>
T <sub>2</sub>	/	T <sub>2</sub> not assigned to either D <sub>1</sub> or D <sub>2</sub>			
T <sub>3</sub>	D <sub>1</sub> not assigned to either T <sub>1</sub> or T <sub>2</sub>	/	0	0	
T <sub>4</sub>	/	D <sub>2</sub> not assigned to either T <sub>1</sub> or T <sub>2</sub>	0	0	

**unassignedDetectionCost:**  
values vary for each  
unassigned detection to track

Data Types: `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `single` | `double`

## Output Arguments

### **assignments** — Index pairs of tracks and corresponding detections

*L*-by-2 matrix

Index pairs of tracks and corresponding detections. This value is returned as an *L*-by-2 matrix of index pairs, with *L* number of pairs. The first column represents the track index and the second column represents the detection index.

Data Types: `uint32`

### **unassignedTracks** — Unassigned tracks

*P*-element vector

Unassigned tracks, returned as a *P*-element vector. *P* represents the number of unassigned tracks. Each element represents a track to which no detections are assigned.

Data Types: `uint32`

### **unassignedDetections** — Unassigned detections

*Q*-element vector

Unassigned detections, returned as a *Q*-element vector, where *Q* represents the number of unassigned detections. Each element represents a detection that was not assigned to any tracks. These detections can begin new tracks.

Data Types: `uint32`

## More About

- “Multiple Object Tracking”
- Munkres' Assignment Algorithm Modified for Rectangular Matrices

## References

- [1] Miller, Matt L., Harold S. Stone, and Ingemar J. Cox, “Optimizing Murty's Ranked Assignment Method,” *IEEE Transactions on Aerospace and Electronic Systems*, 33(3), 1997.

[2] Munkres, James, "Algorithms for Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, Volume 5, Number 1, March, 1957.

### **See Also**

vision.KalmanFilter | configureKalmanFilter

**Introduced in R2012b**

## **bbox2points**

Convert rectangle to corner points list

### **Syntax**

```
points = bbox2points(rectangle)
```

### **Description**

`points = bbox2points(rectangle)` converts the input rectangle, specified as [*x y width height*] into a list of four [*x y*] corner points. The `rectangle` input must be either a single bounding box or a set of bounding boxes.

#### **Code Generation Support:**

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

### **Examples**

#### **Convert Bounding Box to List of Points and Apply Rotation**

**Define a bounding box.**

```
bbox = [10 20 50 60];
```

**Convert the bounding box to a list of four points.**

```
points = bbox2points(bbox);
```

**Define a rotation transformation.**

```
theta = 10;  
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1]);
```

**Apply the rotation.**

```
points2 = transformPointsForward(tform, points);
```

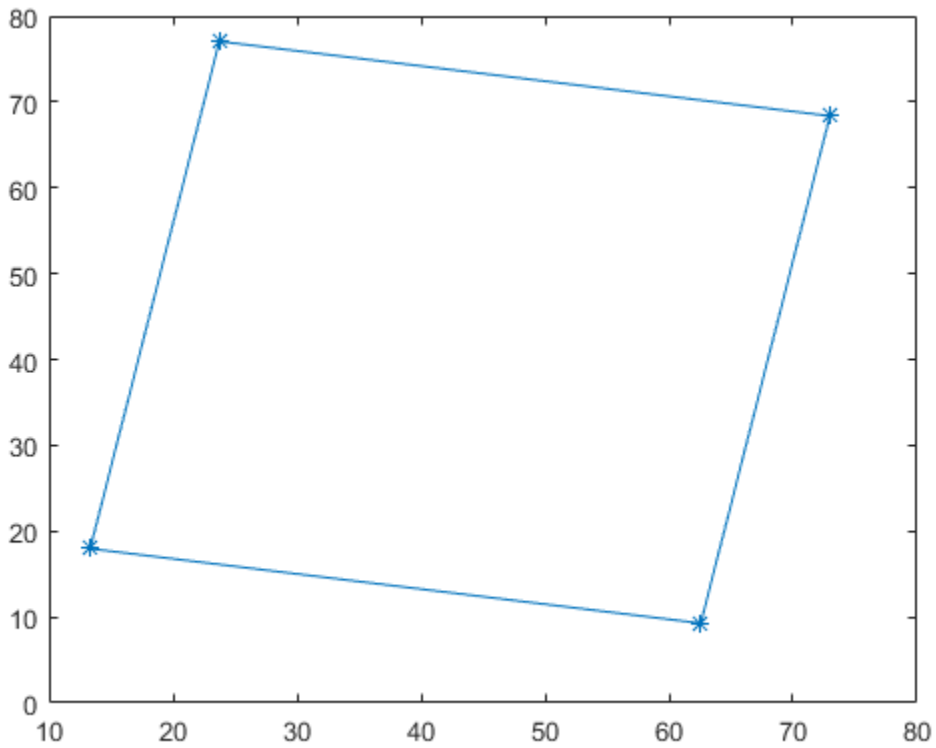


Close the polygon for display.

```
points2(end+1, :) = points2(1, :);
```

Plot the rotated box.

```
plot(points2(:, 1), points2(:, 2), '*-');
```



## Input Arguments

**rectangle** — Bounding box

4-element vector |  $M$ -by-4 matrix

Bounding box, specified as a 4-element vector, [ $x$   $y$  *width* *height*], or a set of bounding boxes, specified as an  $M$ -by-4 matrix.

Data Types: `single` | `double` | `int16` | `int32` | `uint16` | `uint32`

## Output Arguments

### **points** — Rectangle corner coordinates

4-by-2 matrix | 4-by-2- $M$  array

List of rectangle corners, returned as a 4-by-2 matrix of [ $x,y$ ] coordinates, or a 4-by-2-by- $M$  array of [ $x,y$ ] coordinates.

- For a single input bounding box, the function returns the 4-by-2 matrix.
- For multiple input bounding boxes, the function returns the 4-by-2- $M$  array for  $M$  bounding boxes.

Data Types: `single` | `double` | `int16` | `int32` | `uint16` | `uint32`

## See Also

`affine2d` | `projective2d`

**Introduced in R2014b**

# bboxOverlapRatio

Compute bounding box overlap ratio

## Syntax

```
overlapRatio = bboxOverlapRatio(bboxA, bboxB)
overlapRatio = bboxOverlapRatio(bboxA, bboxB, ratioType)
```

## Description

`overlapRatio = bboxOverlapRatio(bboxA, bboxB)` returns the overlap ratio between each pair of bounding boxes `bboxA` and `bboxB`. The function returns the `overlapRatio` value between 0 and 1, where 1 implies a perfect overlap.

`overlapRatio = bboxOverlapRatio(bboxA, bboxB, ratioType)` additionally lets you specify the method to use for computing the ratio. You must set the `ratioType` to either 'Union' or 'Min'.

### Code Generation Support:

Compile-time constant input: No restriction

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

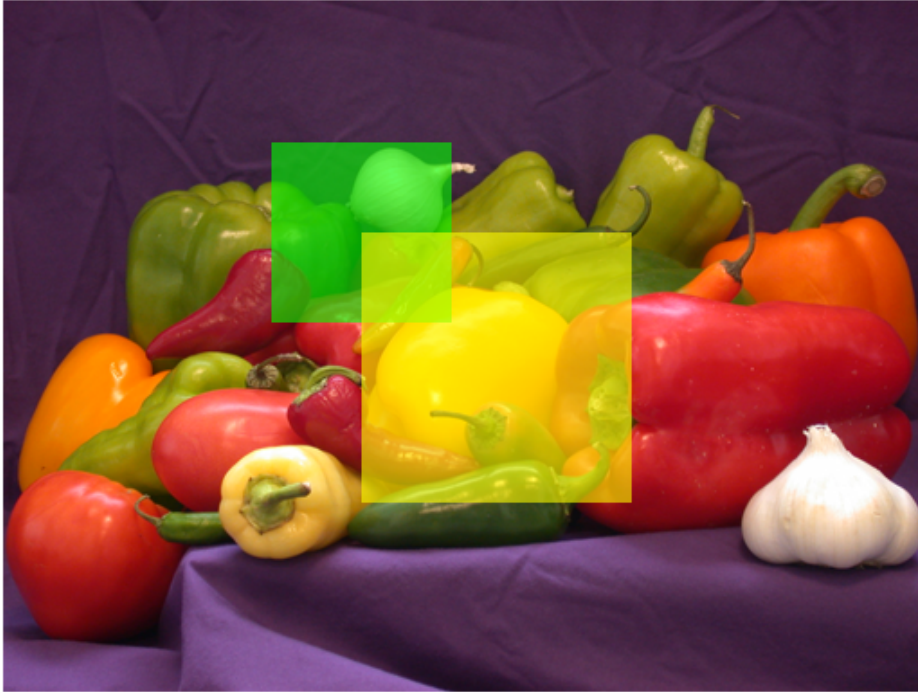
### Compute the Overlap Ratio Between Two Bounding Boxes

Define two bounding boxes in the format [x y width height].

```
bboxA = [150,80,100,100];
bboxB = bboxA + 50;
```

Display the bounding boxes on an image.

```
I = imread('peppers.png');
RGB = insertShape(I, 'FilledRectangle', bboxA, 'Color', 'green');
RGB = insertShape(RGB, 'FilledRectangle', bboxB, 'Color', 'yellow');
imshow(RGB)
```



Compute the overlap ratio between the two bounding boxes.

```
overlapRatio = bboxOverlapRatio(bboxA, bboxB)
```

```
overlapRatio =
```

```
0.0833
```

### **Compute Overlap Ratio Between Each Pair of Bounding Boxes**

Randomly generate two sets of bounding boxes.

```
bboxA = 10*rand(5,4);
bboxB = 10*rand(10,4);
```

Ensure that the width and height of the boxes are positive.

```
bboxA(:,3:4) = bboxA(:,3:4) + 10;
bboxB(:,3:4) = bboxB(:,3:4) + 10;
```

Compute the overlap ratio between each pair.

```
overlapRatio = bboxOverlapRatio(bboxA,bboxB)
```

```
overlapRatio =
```

```
Columns 1 through 7
```

0.2431	0.2329	0.3418	0.5117	0.7972	0.1567	0.1789
0.3420	0.1655	0.7375	0.5188	0.2786	0.3050	0.2969
0.4844	0.3290	0.3448	0.1500	0.1854	0.4976	0.5629
0.3681	0.0825	0.3499	0.0840	0.0658	0.5921	0.6498
0.3752	0.1114	0.3114	0.0696	0.0654	0.5408	0.6234

```
Columns 8 through 10
```

0.4339	0.0906	0.5766
0.4350	0.2477	0.2530
0.4430	0.5027	0.2685
0.1930	0.7433	0.0676
0.2046	0.7557	0.0717

## Input Arguments

### **bboxA** — Bounding box

*M*-by-4 matrix

Bounding box, specified as an *M*-by-4 matrix. Each row of **bboxA** contains a vector in the format [*x y width height*], where *x* and *y* correspond to the upper left corner of the bounding box. Bounding boxes inputs **bboxA** and **bboxB** must be real, finite, and nonparse.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**bboxB — Bounding box***M*-by-4 matrix

Bounding box, specified as an *M*-by-4 matrix. Each row of **bboxB** contains a vector in the format  $[x\ y\ width\ height]$ , where *x* and *y* correspond to the upper left corner of the bounding box. Bounding boxes inputs **bboxA** and **bboxB** must be real, finite, and nonsparse.

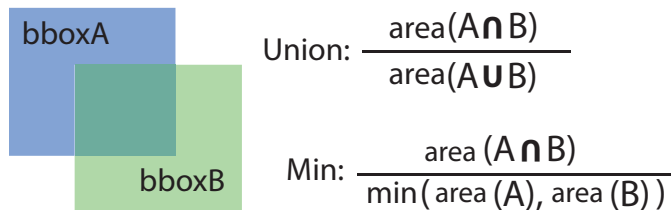
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**ratioType — Ratio type**

'Union' (default) | 'Min'

Ratio type, specified as the string 'Union' or 'Min'.

- Set the ratio type to 'Union' to compute the ratio as the area of intersection between **bboxA** and **bboxB**, divided by the area of the union of the two.
- Set the ratio type to 'Min' to compute the ratio as the area of intersection between **bboxA** and **bboxB**, divided by the minimum area of the two bounding boxes.

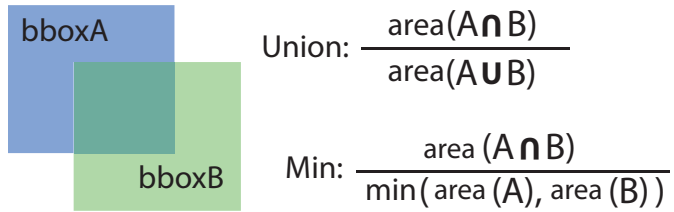


Data Types: char

**Output Arguments****overlapRatio — Overlap ratio between two bounding boxes***M*-by-*N* matrix

Overlap ratio between two bounding boxes, returned as an *M*-by-*N* matrix. Each (*I*, *J*) element in the output matrix corresponds to the overlap ratio between row *I* in **bboxA** and row *J* in **bboxB**. The function returns **overlapRatio** in the between 0 and 1, where 1 implies a perfect overlap. If either **bboxA** or **bboxB** is double, then the function returns **overlapRatio** as double. Otherwise, the function returns it as single.

The function computes the overlap ratio based on the ratio type. You can set `ratioType` to 'Union' or 'Min':



Data Types: single | double

## More About

- “Multiple Object Tracking”

## See Also

`selectStrongestBbox`

**Introduced in R2014b**

# bundleAdjustment

Refine camera poses and 3-D points

## Syntax

```
[xyzRefinedPoints,refinedPoses] = bundleAdjustment(xyzPoints,  
pointTracks,cameraPoses,cameraParameters)  
[ ____,reprojectionErrors] = bundleAdjustment( ____ )  
[ ____ ] = bundleAdjustment( ____,Name,Value)
```

## Description

[xyzRefinedPoints,refinedPoses] = bundleAdjustment(xyzPoints, pointTracks,cameraPoses,cameraParameters) returns the refined 3-D points and camera poses that minimize reprojection errors. The refinement procedure is a variant of the Levenberg-Marquardt algorithm.

[ \_\_\_\_,reprojectionErrors] = bundleAdjustment( \_\_\_\_ ) additionally returns reprojection errors for each 3-D world point using the arguments from the previous syntax.

[ \_\_\_\_ ] = bundleAdjustment( \_\_\_\_,Name,Value) uses additional options specified by one or more Name,Value pair arguments. Unspecified properties have default values.

### Code Generation Support:

Supports Code Generation: No

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Refine Camera Poses and 3-D Points

Load data for initialization.



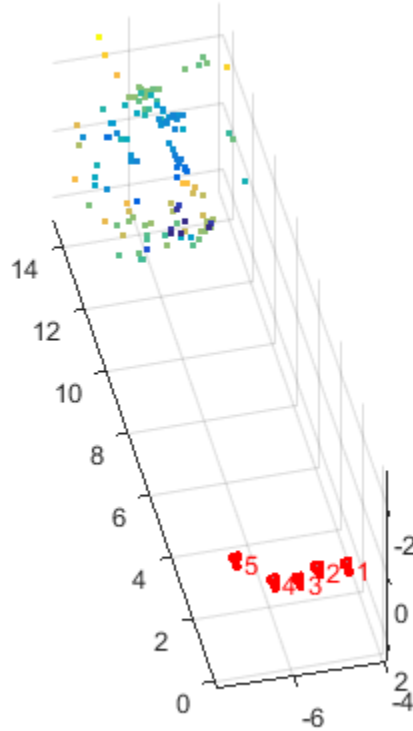
```
load('sfmGlobe');
```

Refine the camera poses and points.

```
[xyzRefinedPoints,refinedPoses] = bundleAdjustment(xyzPoints,pointTracks,cameraPoses,c
```

Display the refined camera poses and 3-D world points.

```
cameraSize = 0.1;
for j = 1:height(refinedPoses)
    id = refinedPoses.ViewId(j);
    loc = refinedPoses.Location{j};
    orient = refinedPoses.Orientation{j};
    plotCamera('Location',loc,'Orientation',orient,'Size',cameraSize,'Color','r','Label',id);
    hold on
end
pcshow(xyzRefinedPoints,'VerticalAxis','y','VerticalAxisDir','down','MarkerSize',45);
grid on
```



- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **xyzPoints** — Unrefined 3-D points

*M*-by-3 matrix

Unrefined 3-D points, specified as an *M*-by-3 matrix of  $[x,y,z]$  locations.

**pointTracks** — Matching points across multiple images*N*-element array of `pointTrack` objects

Matching points across multiple images, specified as an *N*-element array of `pointTrack` objects. Each element contains two or more matching points across multiple images.

**cameraPoses** — Camera pose information

three-column table

Camera pose `ViewId`, `Orientation`, and `Location` information, specified as a three-column table. The view IDs relate to the IDs in the `pointTracks` object. The orientations are specified as 3-by-3 rotation matrices. The locations are specified as a three-element vectors.

**cameraParameters** — Camera parameters`cameraParameters` object

Camera parameters, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

**Name-Value Pair Arguments**

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

Example: `'MaxIterations', '50'`

**'MaxIterations'** — Maximum number of iterations

50 (default) | positive integer

Maximum number of iterations before the Levenberg-Marquardt algorithm stops, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer.

**'AbsoluteTolerance'** — Absolute termination tolerance

1.0 (default) | positive scalar

Absolute termination tolerance of the mean squared reprojection error in pixels, specified as the comma-separated pair consisting of `'AbsoluteTolerance'` and a positive scalar.

**'RelativeTolerance' — Relative termination tolerance**

1e-5 (default) | positive scalar

Relative termination tolerance of the reduction in reprojection error between iterations, specified as the comma-separated pair consisting of 'RelativeTolerance' and a positive scalar.

**'PointsUndistorted' — Flag to indicate lens distortion**

false (default) | true

Flag to indicate lens distortion, specified as the comma-separated pair consisting of 'PointsUndistorted' and either false or true. When you set PointsUndistorted to false, the 2-D points in pointTracks must be from images with lens distortion. To use undistorted points, use the undistortImage function first, then set PointsUndistorted to true.

**'FixedViewID' — View ID for fixed camera pose**

[] (default) | nonnegative integer

View ID for fixed camera pose, specified as the comma-separated pair consisting of 'FixedViewID' and a nonnegative integer. The ID corresponds to the ViewID of the fixed camera pose in cameraPoses. An empty value for FixedViewID means that all camera poses are optimized.

**'Verbose' — Display progress information**

False (default) | true

Display progress information, specified as the comma-separated pair consisting of 'Verbose' and either false or true.

## Output Arguments

**xyzRefinedPoints — 3-D locations of refined points world points**

$M$ -by-3 matrix

3-D locations of refined world points, returned as an  $M$ -by-3 matrix of  $[x,y,z]$  locations.

Data Types: single | double

**refinedPoses — Refined camera poses**

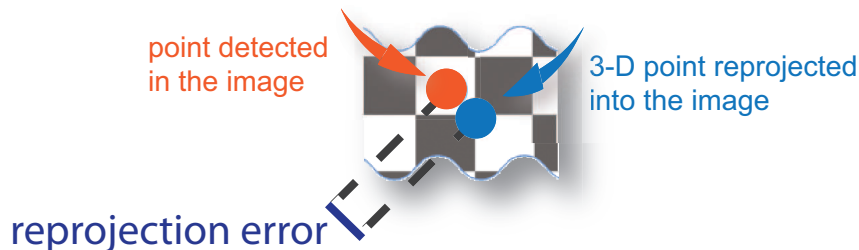
three-column table

Refined camera poses, returned as a table. The table contains three columns for 'ViewId', 'Orientation', and 'Location'.

### reprojectionErrors — Reprojection errors

$M$ -by-1 vector

Reprojection errors, returned as an  $M$ -by-1 vector. The function projects each world point back into each camera. Then in each image, the function calculates the reprojection error as the distance between the detected and the reprojected point. The reprojectionErrors vector contains the average reprojection error for each world point.



## More About

- “Structure from Motion”

## References

- [1] Lourakis, M.I.A., and A.A. Argyros. "SBA: A Software Package for Generic Sparse Bundle Adjustment." *ACM Transactions on Mathematical Software*. 2009.
- [2] Hartley, R., and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.
- [3] Triggs, B., P. McLauchlan, R. Hartley, and A. Fitzgibbon. "Bundle Adjustment: A Modern Synthesis." *Proceedings of the International Workshop on Vision Algorithms*. Springer-Verlag. 1999, pp. 298-372.

## See Also

viewSet | pointTrack | cameraParameters | cameraMatrix | cameraPose | triangulateMultiview | undistortImage | undistortPoints

**Introduced in R2016a**

# OCR Trainer

Train an optical character recognition model to recognize a specific set of characters

## Description

The **OCR Trainer** app allows you to label character data for OCR training interactively and to generate an OCR language data file for use with the `ocr` function.

## Open the OCR Trainer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `ocrTrainer`.

## Programmatic Use

`ocrTrainer` opens the OCR Trainer app.

`ocrTrainer(sessionFile)` opens the app and loads a saved OCR training session. `sessionFile` is the path to the MAT file containing the saved session.

`ocrTrainer CLOSE` closes all open apps.

## More About

- “Train Optical Character Recognition for Custom Fonts”

## See Also

`ocr`

Introduced in R2016a

## listTrueTypeFonts

List available TrueType fonts

### Syntax

```
fontNames = listTrueTypeFonts
```

### Description

`fontNames = listTrueTypeFonts` returns a cell array of sorted TrueType font names installed on the system.

### Examples

#### List Available TrueType Fonts

```
listTrueTypeFonts
```

```
ans =
```

```
'Agency FB'  
'Agency FB Bold'  
'Aharoni Bold'  
'Algerian'  
'Andalus'  
'Angsana New'  
'Angsana New Bold'  
'Angsana New Bold Italic'  
'Angsana New Italic'  
'AngsanaUPC'  
'AngsanaUPC Bold'  
'AngsanaUPC Bold Italic'  
'AngsanaUPC Italic'  
'Aparajita'  
'Aparajita Bold'  
'Aparajita Italic'  
'Arabic Typesetting'  
'Arial'
```



'Arial Black'  
'Arial Bold'  
'Arial Bold Italic'  
'Arial Italic'  
'Arial Narrow'  
'Arial Narrow Bold'  
'Arial Narrow Bold Italic'  
'Arial Narrow Italic'  
'Arial Rounded MT Bold'  
'Arial Unicode MS'  
'Baskerville Old Face'  
'Batang'  
'BatangChe'  
'Bauhaus 93'  
'Bell MT'  
'Bell MT Bold'  
'Bell MT Italic'  
'Berlin Sans FB'  
'Berlin Sans FB Bold'  
'Berlin Sans FB Demi Bold'  
'Bernard MT Condensed'  
'Blackadder ITC'  
'Bodoni MT'  
'Bodoni MT Black'  
'Bodoni MT Black Italic'  
'Bodoni MT Bold'  
'Bodoni MT Bold Italic'  
'Bodoni MT Condensed'  
'Bodoni MT Condensed Bold'  
'Bodoni MT Condensed Bold Italic'  
'Bodoni MT Condensed Italic'  
'Bodoni MT Italic'  
'Bodoni MT Poster Compressed'  
'Book Antiqua'  
'Book Antiqua Bold'  
'Book Antiqua Bold Italic'  
'Book Antiqua Italic'  
'Bookman Old Style'  
'Bookman Old Style Bold'  
'Bookman Old Style Bold Italic'  
'Bookman Old Style Italic'  
'Bookshelf Symbol 7'  
'Bradley Hand ITC'  
'Britannic Bold'

'Broadway'  
'Browallia New'  
'Browallia New Bold'  
'Browallia New Bold Italic'  
'Browallia New Italic'  
'BrowalliaUPC'  
'BrowalliaUPC Bold'  
'BrowalliaUPC Bold Italic'  
'BrowalliaUPC Italic'  
'Brush Script MT Italic'  
'Buxton Sketch'  
'Calibri'  
'Calibri Bold'  
'Calibri Bold Italic'  
'Calibri Italic'  
'Calibri Light'  
'Calibri Light Italic'  
'Californian FB'  
'Californian FB Bold'  
'Californian FB Italic'  
'Calisto MT'  
'Calisto MT Bold'  
'Calisto MT Bold Italic'  
'Calisto MT Italic'  
'Cambria'  
'Cambria Bold'  
'Cambria Bold Italic'  
'Cambria Italic'  
'Cambria Math'  
'Candara'  
'Candara Bold'  
'Candara Bold Italic'  
'Candara Italic'  
'Castellar'  
'Centaur'  
'Century'  
'Century Gothic'  
'Century Gothic Bold'  
'Century Gothic Bold Italic'  
'Century Gothic Italic'  
'Century Schoolbook'  
'Century Schoolbook Bold'  
'Century Schoolbook Bold Italic'  
'Century Schoolbook Italic'

'Chiller'  
'Colonna MT'  
'Comic Sans MS'  
'Comic Sans MS Bold'  
'Consolas'  
'Consolas Bold'  
'Consolas Bold Italic'  
'Consolas Italic'  
'Constantia'  
'Constantia Bold'  
'Constantia Bold Italic'  
'Constantia Italic'  
'Cooper Black'  
'Copperplate Gothic Bold'  
'Copperplate Gothic Light'  
'Corbel'  
'Corbel Bold'  
'Corbel Bold Italic'  
'Corbel Italic'  
'Cordia New'  
'Cordia New Bold'  
'Cordia New Bold Italic'  
'Cordia New Italic'  
'CordiaUPC'  
'CordiaUPC Bold'  
'CordiaUPC Bold Italic'  
'CordiaUPC Italic'  
'Courier New'  
'Courier New Bold'  
'Courier New Bold Italic'  
'Courier New Italic'  
'Curlz MT'  
'DFKai-SB'  
'DaunPenh'  
'David'  
'David Bold'  
'DilleniaUPC'  
'DilleniaUPC Bold'  
'DilleniaUPC Bold Italic'  
'DilleniaUPC Italic'  
'DokChampa'  
'Dotum'  
'DotumChe'  
'Ebrima'

'Ebrima Bold'  
'Edwardian Script ITC'  
'Elephant'  
'Elephant Italic'  
'Engravers MT'  
'Eras Bold ITC'  
'Eras Demi ITC'  
'Eras Light ITC'  
'Eras Medium ITC'  
'Estrangelo Edessa'  
'EucrosiaUPC'  
'EucrosiaUPC Bold'  
'EucrosiaUPC Bold Italic'  
'EucrosiaUPC Italic'  
'Euphemia'  
'FZDengXian Regular'  
'FangSong'  
'Felix Titling'  
'Footlight MT Light'  
'Forte'  
'FrankRuehl'  
'Franklin Gothic Book'  
'Franklin Gothic Book Italic'  
'Franklin Gothic Demi'  
'Franklin Gothic Demi Cond'  
'Franklin Gothic Demi Italic'  
'Franklin Gothic Heavy'  
'Franklin Gothic Heavy Italic'  
'Franklin Gothic Medium'  
'Franklin Gothic Medium Cond'  
'Franklin Gothic Medium Italic'  
'FreesiaUPC'  
'FreesiaUPC Bold'  
'FreesiaUPC Bold Italic'  
'FreesiaUPC Italic'  
'Freestyle Script'  
'French Script MT'  
'Gabriola'  
'Gadugi'  
'Gadugi Bold'  
'Garamond'  
'Garamond Bold'  
'Garamond Italic'  
'Gautami'

'Gautami Bold'  
'Georgia'  
'Georgia Bold'  
'Georgia Bold Italic'  
'Georgia Italic'  
'Gigi'  
'Gill Sans MT'  
'Gill Sans MT Bold'  
'Gill Sans MT Bold Italic'  
'Gill Sans MT Condensed'  
'Gill Sans MT Ext Condensed Bold'  
'Gill Sans MT Italic'  
'Gill Sans Ultra Bold'  
'Gill Sans Ultra Bold Condensed'  
'Gisha'  
'Gisha Bold'  
'Gloucester MT Extra Condensed'  
'Goudy Old Style'  
'Goudy Old Style Bold'  
'Goudy Old Style Italic'  
'Goudy Stout'  
'Gulim'  
'GulimChe'  
'Gungsuh'  
'GungsuhChe'  
'Haettenschweiler'  
'Harlow Solid Italic'  
'Harrington'  
'High Tower Text'  
'High Tower Text Italic'  
'Impact'  
'Imprint MT Shadow'  
'Informal Roman'  
'IrisUPC'  
'IrisUPC Bold'  
'IrisUPC Bold Italic'  
'IrisUPC Italic'  
'Iskoola Pota'  
'Iskoola Pota Bold'  
'JasmineUPC'  
'JasmineUPC Bold'  
'JasmineUPC Bold Italic'  
'JasmineUPC Italic'  
'Jokerman'

'Juice ITC'  
'KaiTi'  
'Kalinga'  
'Kalinga Bold'  
'Kartika'  
'Kartika Bold'  
'Khmer UI'  
'Khmer UI Bold'  
'KodchiangUPC'  
'KodchiangUPC Bold'  
'KodchiangUPC Bold Italic'  
'KodchiangUPC Italic'  
'Kokila'  
'Kokila Bold'  
'Kokila Italic'  
'Kristen ITC'  
'Kunstler Script'  
'Lao UI'  
'Lao UI Bold'  
'Latha'  
'Latha Bold'  
'Leelawadee'  
'Leelawadee Bold'  
'Levenim MT'  
'Levenim MT Bold'  
'LilyUPC'  
'LilyUPC Bold'  
'LilyUPC Bold Italic'  
'LilyUPC Italic'  
'Lucida Bright'  
'Lucida Bright Demibold'  
'Lucida Bright Demibold Italic'  
'Lucida Bright Italic'  
'Lucida Calligraphy Italic'  
'Lucida Console'  
'Lucida Fax Demibold'  
'Lucida Fax Demibold Italic'  
'Lucida Fax Italic'  
'Lucida Fax Regular'  
'Lucida Handwriting Italic'  
'Lucida Sans Demibold Italic'  
'Lucida Sans Demibold Roman'  
'Lucida Sans Italic'  
'Lucida Sans Regular'

---

'Lucida Sans Typewriter Bold'  
'Lucida Sans Typewriter Bold Oblique'  
'Lucida Sans Typewriter Oblique'  
'Lucida Sans Typewriter Regular'  
'Lucida Sans Unicode'  
'LucidaBrightDemiBold'  
'LucidaBrightDemiItalic'  
'LucidaBrightItalic'  
'LucidaBrightRegular'  
'LucidaSansDemiBold'  
'LucidaSansRegular'  
'LucidaTypewriterBold'  
'LucidaTypewriterRegular'  
'MS Gothic'  
'MS Mincho'  
'MS Outlook'  
'MS PGothic'  
'MS PMincho'  
'MS Reference Sans Serif'  
'MS Reference Specialty'  
'MS UI Gothic'  
'MT Extra'  
'MV Boli'  
'Magneto Bold'  
'Maiandra GD'  
'Malgun Gothic'  
'Malgun Gothic Bold'  
'Mangal'  
'Mangal Bold'  
'Matura MT Script Capitals'  
'Meiryo'  
'Meiryo Bold'  
'Meiryo Bold Italic'  
'Meiryo Italic'  
'Meiryo UI'  
'Meiryo UI Bold'  
'Meiryo UI Bold Italic'  
'Meiryo UI Italic'  
'Microsoft Himalaya'  
'Microsoft JhengHei'  
'Microsoft JhengHei Bold'  
'Microsoft JhengHei UI'  
'Microsoft JhengHei UI Bold'  
'Microsoft MHei'

'Microsoft MHei Bold'  
'Microsoft NeoGothic'  
'Microsoft NeoGothic Bold'  
'Microsoft New Tai Lue'  
'Microsoft New Tai Lue Bold'  
'Microsoft PhagsPa'  
'Microsoft PhagsPa Bold'  
'Microsoft Sans Serif'  
'Microsoft Tai Le'  
'Microsoft Tai Le Bold'  
'Microsoft Uighur'  
'Microsoft Uighur Bold'  
'Microsoft YaHei'  
'Microsoft YaHei Bold'  
'Microsoft YaHei UI'  
'Microsoft YaHei UI Bold'  
'Microsoft Yi Baiti'  
'MingLiU'  
'MingLiU-ExtB'  
'MingLiU\_HKSCS'  
'MingLiU\_HKSCS-ExtB'  
'Miriam'  
'Miriam Fixed'  
'Mistral'  
'Modern No. 20'  
'Mongolian Baiti'  
'Monotype Corsiva'  
'MoolBoran'  
'NSimSun'  
'Narkisim'  
'Niagara Engraved'  
'Niagara Solid'  
'Nirmala UI'  
'Nirmala UI Bold'  
'Nyala'  
'OCR A Extended'  
'Old English Text MT'  
'Onyx'  
'PMingLiU'  
'PMingLiU-ExtB'  
'Palace Script MT'  
'Palatino Linotype'  
'Palatino Linotype Bold'  
'Palatino Linotype Bold Italic'



'Palatino Linotype Italic'  
'Papyrus'  
'Parchment'  
'Perpetua'  
'Perpetua Bold'  
'Perpetua Bold Italic'  
'Perpetua Italic'  
'Perpetua Titling MT Bold'  
'Perpetua Titling MT Light'  
'Plantagenet Cherokee'  
'Playbill'  
'Poor Richard'  
'Pristina'  
'Raavi'  
'Raavi Bold'  
'Rage Italic'  
'Ravie'  
'Rockwell'  
'Rockwell Bold'  
'Rockwell Bold Italic'  
'Rockwell Condensed'  
'Rockwell Condensed Bold'  
'Rockwell Extra Bold'  
'Rockwell Italic'  
'Rod'  
'Sakkal Majalla'  
'Sakkal Majalla Bold'  
'Script MT Bold'  
'Segoe Marker'  
'Segoe Print'  
'Segoe Print Bold'  
'Segoe Script'  
'Segoe Script Bold'  
'Segoe UI'  
'Segoe UI Bold'  
'Segoe UI Bold Italic'  
'Segoe UI Italic'  
'Segoe UI Light'  
'Segoe UI Semibold'  
'Segoe UI Semilight'  
'Segoe UI Symbol'  
'Segoe WP'  
'Segoe WP Black'  
'Segoe WP Bold'

'Segoe WP Light'  
'Segoe WP SemiLight'  
'Segoe WP Semibold'  
'Shonar Bangla'  
'Shonar Bangla Bold'  
'Showcard Gothic'  
'Shruti'  
'Shruti Bold'  
'SimHei'  
'SimSun'  
'SimSun-ExtB'  
'Simplified Arabic'  
'Simplified Arabic Bold'  
'Simplified Arabic Fixed'  
'SketchFlow Print'  
'Snap ITC'  
'Stencil'  
'Sylfaen'  
'Symbol'  
'Tahoma'  
'Tahoma Bold'  
'Tempus Sans ITC'  
'Times New Roman'  
'Times New Roman Bold'  
'Times New Roman Bold Italic'  
'Times New Roman Italic'  
'Traditional Arabic'  
'Traditional Arabic Bold'  
'Trebuchet MS'  
'Trebuchet MS Bold'  
'Trebuchet MS Bold Italic'  
'Trebuchet MS Italic'  
'Tunga'  
'Tunga Bold'  
'Tw Cen MT'  
'Tw Cen MT Bold'  
'Tw Cen MT Bold Italic'  
'Tw Cen MT Condensed'  
'Tw Cen MT Condensed Bold'  
'Tw Cen MT Condensed Extra Bold'  
'Tw Cen MT Italic'  
'Utsaah'  
'Utsaah Bold'  
'Utsaah Italic'

```
'Vani'
'Vani Bold'
'Verdana'
'Verdana Bold'
'Verdana Bold Italic'
'Verdana Italic'
'Vijaya'
'Vijaya Bold'
'Viner Hand ITC'
'Vivaldi Italic'
'Vladimir Script'
'Vrinda'
'Vrinda Bold'
'Webdings'
'Wide Latin'
'Wingdings'
'Wingdings 2'
'Wingdings 3'
'YuGothic'
'YuGothic Bold'
```

### List All TrueType 'Lucida' Fonts

```
fontNames = listTrueTypeFonts;
LucidaFonts = fontNames(~cellfun(@isempty, regexp(fontNames, '^Lucida')))
```

```
LucidaFonts =
```

```
'Lucida Bright'
'Lucida Bright Demibold'
'Lucida Bright Demibold Italic'
'Lucida Bright Italic'
'Lucida Calligraphy Italic'
'Lucida Console'
'Lucida Fax Demibold'
'Lucida Fax Demibold Italic'
'Lucida Fax Italic'
'Lucida Fax Regular'
'Lucida Handwriting Italic'
'Lucida Sans Demibold Italic'
'Lucida Sans Demibold Roman'
'Lucida Sans Italic'
'Lucida Sans Regular'
```

```
'Lucida Sans Typewriter Bold'  
'Lucida Sans Typewriter Bold Oblique'  
'Lucida Sans Typewriter Oblique'  
'Lucida Sans Typewriter Regular'  
'Lucida Sans Unicode'  
'LucidaBrightDemiBold'  
'LucidaBrightDemiItalic'  
'LucidaBrightItalic'  
'LucidaBrightRegular'  
'LucidaSansDemiBold'  
'LucidaSansRegular'  
'LucidaTypewriterBold'  
'LucidaTypewriterRegular'
```

## Output Arguments

**fontNames** — Available TrueType fonts on system

cell array

Available TrueType fonts on system, returned as a cell array of sorted TrueType font names.

### See Also

`insertObjectAnnotation` | `insertText` | `listfonts`

**Introduced in R2015b**

# pcfromkinect

Point cloud from Kinect for Windows

## Syntax

```
ptCloud = pcfromkinect(depthImage,depthDevice)
ptCloud = pcfromkinect(depthImage,depthDevice,colorImage)
ptCloud = pcfromkinect(depthImage,depthDevice,colorImage,alignment)
```

## Description

`ptCloud = pcfromkinect(depthImage,depthDevice)` returns a point cloud from a Kinect depth image. The `depthDevice` input can be either a `videoinput` object or an `imaq.VideoDevice` object configured for Kinect (Versions 1 and 2) for Windows.

This function requires the Image Acquisition Toolbox software, which supports Kinect for Windows.

`ptCloud = pcfromkinect(depthImage,depthDevice,colorImage)` adds color to the returned point cloud, specified by the `colorImage` input.

The Kinect for Windows system, designed for gaming, produces `depthImage` and `colorImage` as mirror images of the scene. The returned point cloud is corrected to match the actual scene.

`ptCloud = pcfromkinect(depthImage,depthDevice,colorImage,alignment)` additionally returns the color point cloud with the origin specified at the center of the depth camera.

## Examples

### Plot Color Point Cloud from Kinect for Windows

Plot a color point cloud from Kinect images. This example requires the Image Acquisition Toolbox software and the Kinect camera and a connection to the camera.

Create a System object for the color device.

```
colorDevice = imaq.VideoDevice('kinect',1)
```

Create a System object for the depth device.

```
depthDevice = imaq.VideoDevice('kinect',2)
```

Initialize the camera.

```
step(colorDevice);  
step(depthDevice);
```

Load one frame from the device.

```
colorImage = step(colorDevice);  
depthImage = step(depthDevice);
```

Extract the point cloud.

```
ptCloud = pcfmfromkinect(depthDevice,depthImage,colorImage);
```

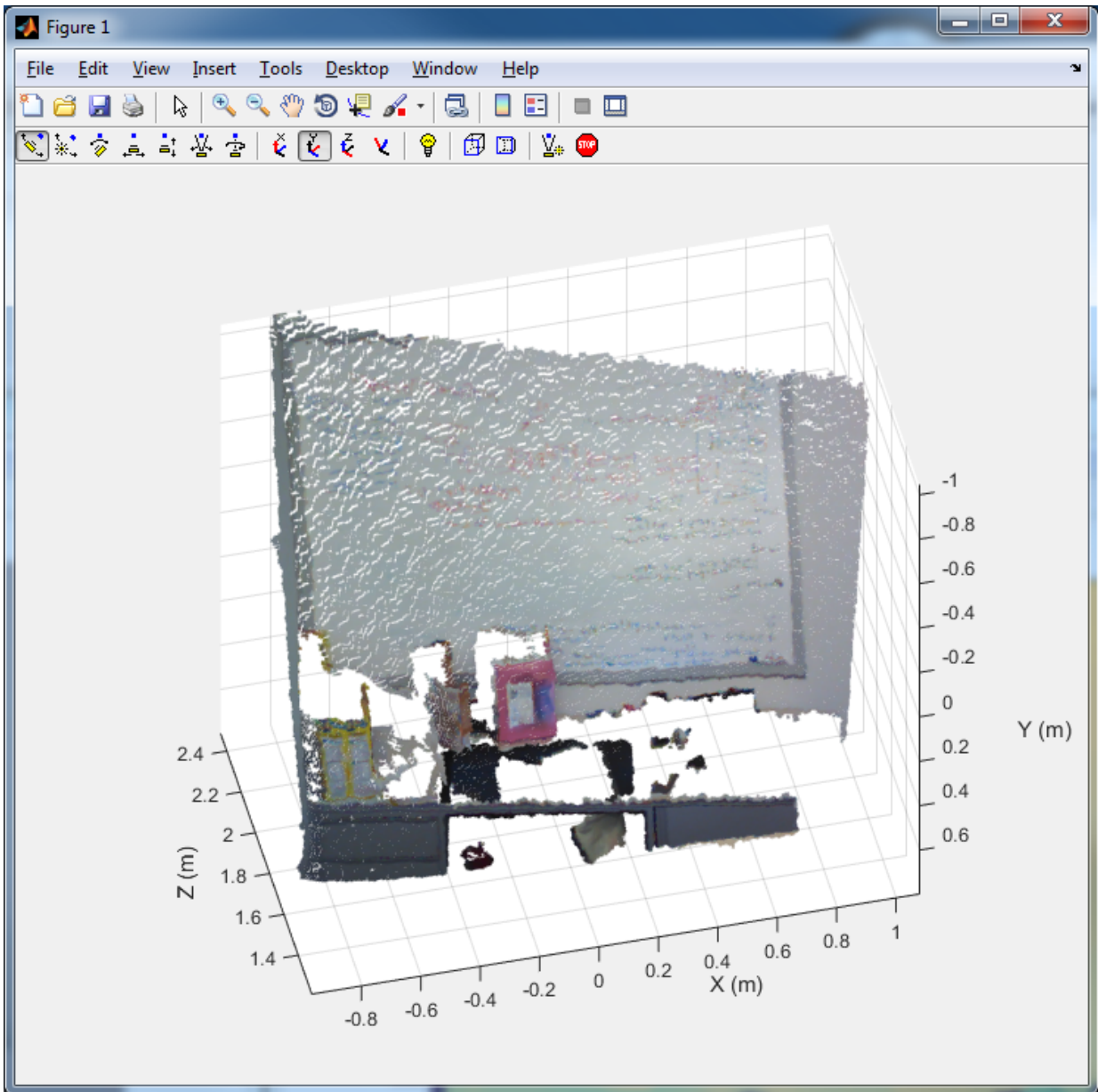
Initialize a point cloud player to visualize 3-D point cloud data. The axis is set appropriately to visualize the point cloud from Kinect.

```
player = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits,...  
    'VerticalAxis','y','VerticalAxisDir','down');
```

```
xlabel(player.Axes,'X (m)');  
ylabel(player.Axes,'Y (m)');  
zlabel(player.Axes,'Z (m)');
```

Acquire and view 500 frames of live Kinect point cloud data.

```
for i = 1:500  
    colorImage = step(colorDevice);  
    depthImage = step(depthDevice);  
  
    ptCloud = pcfmfromkinect(depthDevice,depthImage,colorImage);  
  
    view(player,ptCloud);  
end
```



Release the objects.

```
release(colorDevice);  
release(depthDevice);
```

- “Structure From Motion From Two Views”
- “Depth Estimation From Stereo Video”

## Input Arguments

### **depthImage** — Depth image

*M*-by-*N* matrix

Depth image, specified as an *M*-by-*N* pixel matrix. The original images, `depthImage` and `colorImage`, from Kinect are mirror images of the scene.

The Kinect depth camera has limited range. The limited range of the Kinect depth camera can cause pixel values in the depth image to not have corresponding 3-D coordinates. These missing pixel values are set to NaN in the `Location` property of the returned point cloud.

Data Types: `uint16`

### **depthDevice** — Input video object

`videoinput` object | `imaq.VideoDevice` object

Input video object, specified as either a `videoinput` object or an `imaq.VideoDevice` object configured for Kinect for Windows.

### **colorImage** — Color image

*M*-by-*N*-by-3 RGB truecolor image

Color image, specified as an *M*-by-*N*-by-3 RGB truecolor image that the Kinect returns. The original images, `depthImage` and `colorImage`, from Kinect are mirror images of the scene.

Data Types: `uint8`

### **alignment** — Direction of the image coordinate system

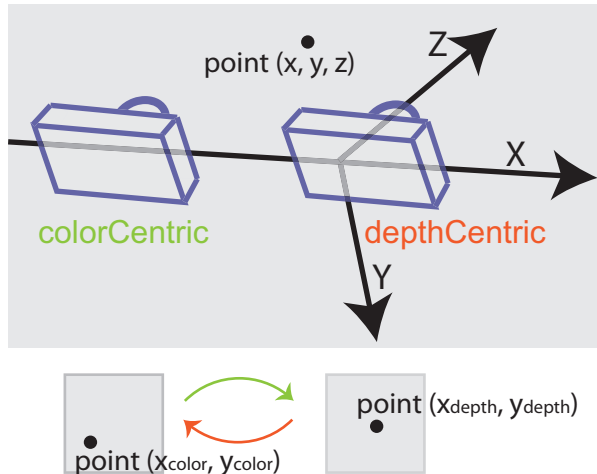
'`colorCentric`' (default) | '`depthCentric`'

Direction of the image coordinate system, specified as the string '`colorCentric`' or '`depthCentric`'. Set this value to '`colorCentric`' to align `depthImage`



with `colorImage`. Set `alignment` to `'depthCentric'` to align `colorImage` with `depthImage`.

The origin of a right-handed world coordinate system is at the center of the depth camera. The  $x$ -axis of the coordinate system points to the right, the  $y$ -axis points downward, and the  $z$ -axis points from the camera.



## Output Arguments

### **ptCloud** — Point cloud

`pointCloud` object

Point cloud, returned as a `pointCloud` object. The origin of the coordinate system of the returned point cloud is at the center of the depth camera.

## More About

- “Coordinate Systems”

## See Also

`pointCloud` | `pcplayer` | `planeModel` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcwrite` | `plot3` | `reconstructScene` | `scatter3` | `triangulate`

**Introduced in R2015b**

# Camera Calibrator

Estimate geometric parameters of a single camera

## Description

The **Camera Calibrator** app allows you to estimate camera intrinsics, extrinsics, and lens distortion parameters. You can use these camera parameters for various computer vision applications. These applications include removing the effects of lens distortion from an image, measuring planar objects, or reconstructing 3-D scenes from multiple cameras.

## Open the Camera Calibrator App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `cameraCalibrator`.

## Examples

### Open Camera Calibrator App

This example shows you the two ways to open the Camera Calibrator app.

Type `cameraCalibrator` on the MATLAB command line or select it from the MATLAB desktop **Apps** tab.

- “Evaluating the Accuracy of Single Camera Calibration”
- “Measuring Planar Objects with a Calibrated Camera”
- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Depth Estimation From Stereo Video”

### Programmatic Use

`cameraCalibrator` opens the Camera Calibrator app, which enables you to compute parameters needed to remove the effects of lens distortion from an image.

`cameraCalibrator(imageFolder, squareSize)` invokes the camera calibration app and loads calibration images from the `imageFolder`. The `squareSize` input must be a scalar in millimeters that specifies the size of the checkerboard square in the calibration pattern.

`cameraCalibrator(sessionFile)` invokes the app and loads a saved camera calibration session. Set the `sessionFile` to the name of the saved session file. The name must include the path to the MAT file containing the saved session.

`cameraCalibrator CLOSE` closes all open apps.

### More About

- “What Is Camera Calibration?”
- “Single Camera Calibration App”
- “Stereo Calibration App”

### See Also

`cameraParameters` | `stereoParameters` | `detectCheckerboardPoints` | `estimateCameraParameters` | `extrinsics` | `generateCheckerboardPoints` | `rectifyStereoImages` | `showExtrinsics` | `showReprojectionErrors` | `Stereo Camera Calibrator` | `triangulate` | `undistortImage`

**Introduced in R2013b**

# Stereo Camera Calibrator

Estimate geometric parameters of a stereo camera

## Description

The **Stereo Camera Calibrator** app allows you to estimate the intrinsic and extrinsic parameters of each camera in a stereo pair. You can also use the app to estimate the translation and rotation between the two cameras.

## Open the Stereo Camera Calibrator App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `stereoCameraCalibrator`.

## Examples

### Open Stereo Camera Calibrator App

This example shows you the two ways to open the Stereo Camera Calibrator app.

Type `stereocameraCalibrator` on the MATLAB command line or select it from the MATLAB desktop **Apps** tab.

- “Evaluating the Accuracy of Single Camera Calibration”
- “Measuring Planar Objects with a Calibrated Camera”
- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Depth Estimation From Stereo Video”
- “Structure From Motion From Two Views”

### Programmatic Use

`stereoCameraCalibrator` opens the Stereo Camera Calibrator app. You can use this app to estimate the intrinsic and extrinsic parameters of each camera in a stereo

pair. You can also use the app to estimate the translation and rotation between the two cameras.

`stereoCameraCalibrator(folder1, folder2, squareSize)` opens the Stereo Camera Calibrator app and loads the stereo calibration images. The app uses the checkerboard square size specified by the `squareSize` input. It also uses `folder1` images for camera 1 and `folder2` for camera 2.

`stereoCameraCalibrator(folder1, folder2, squareSize, squareSizeUnits)` additionally specifies the units of the square size. If you do not specify units, the app sets `squareSizeUnits` to `'mm'`. Units can be `'mm'`, `'cm'`, or `'in'`.

`stereoCameraCalibrator(sessionFile)` opens the app and loads a saved stereo calibration session. Set the `sessionFile` to the name of the saved session MAT-file as a string.

`stereoCameraCalibrator close` closes all open instances of the Stereo Camera Calibrator app.

## More About

- “What Is Camera Calibration?”
- “Single Camera Calibration App”
- “Stereo Calibration App”

## See Also

`cameraParameters` | `stereoParameters` | `Camera Calibrator` |  
`detectCheckerboardPoints` | `estimateCameraParameters` |  
`generateCheckerboardPoints` | `showExtrinsics` | `showReprojectionErrors` |  
`undistortImage`

**Introduced in R2014b**

# cameraMatrix

Camera projection matrix

## Syntax

```
camMatrix = cameraMatrix(cameraParams, rotationMatrix,  
translationVector)
```

## Description

`camMatrix = cameraMatrix(cameraParams, rotationMatrix, translationVector)` returns a 4-by-3 camera projection matrix. You can use this matrix to project 3-D world points in homogeneous coordinates into an image.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Compute Camera Matrix

```
% Create a set of calibration images.  
images = imageSet(fullfile(toolboxdir('vision'), 'visiondata', ...  
    'calibration', 'slr'));  
  
% Detect the checkerboard corners in the images.  
[imagePoints, boardSize] = detectCheckerboardPoints(images.ImageLocation);  
  
% Generate the world coordinates of the checkerboard corners in the  
% pattern-centric coordinate system, with the upper-left corner at (0,0).  
squareSize = 29; % in millimeters  
worldPoints = generateCheckerboardPoints(boardSize, squareSize);
```

```
% Calibrate the camera.
cameraParams = estimateCameraParameters(imagePoints, worldPoints);

% Load image at new location.
imOrig = imread(fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', ...
    'calibration', 'slr', 'image9.jpg'));
figure; imshow(imOrig);
title('Input Image');

% Undistort image.
im = undistortImage(imOrig, cameraParams);

% Find reference object in new image.
[imagePoints, boardSize] = detectCheckerboardPoints(im);

% Compute new extrinsics.
[rotationMatrix, translationVector] = extrinsics(...
    imagePoints, worldPoints, cameraParams);

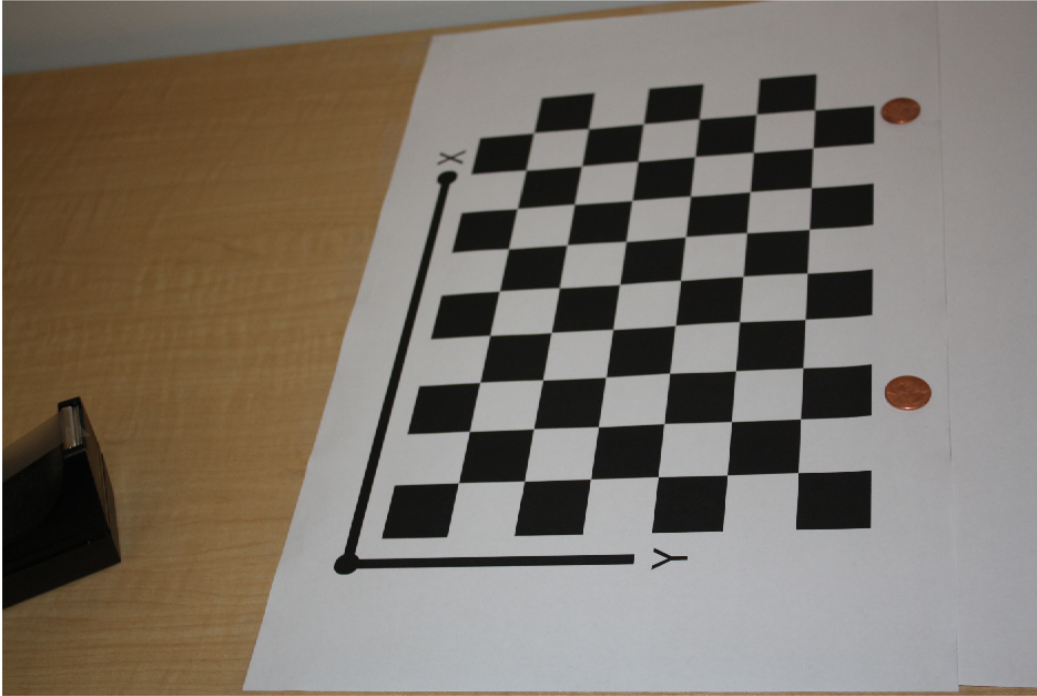
% Calculate camera matrix
P = cameraMatrix(cameraParams, rotationMatrix, translationVector)

P =

    1.0e+05 *
    0.0158   -0.0272    0.0000
    0.0405   -0.0046   -0.0000
    0.0199    0.0387    0.0000
    8.9741    9.4757    0.0073
```



Input Image



- “Evaluating the Accuracy of Single Camera Calibration”
- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Depth Estimation From Stereo Video”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

**cameraParams** — Camera parameters  
cameraParameters object

Camera parameters, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

#### **rotationMatrix** — Rotation of camera

3-by-3 matrix

Rotation of camera, specified as a 3-by-3 matrix. You can obtain this matrix using the `extrinsics` function. You can also obtain the matrix using the `cameraPose` function by transposing its `orientation` output. The `rotationMatrix` and `translationVector` inputs must be real, nonsparse, and of the same class.

#### **translationVector** — Translation of camera

1-by-3 vector

Translation of camera, specified as a 1-by-3 vector. The translation vector describes the transformation from the world coordinates to the camera coordinates. You can obtain this vector using the `extrinsics` function. You can also obtain the vector using the `location` and `orientation` outputs of the `cameraPose` function:  
`translationVector = -location * orientation'`  
. The `rotationMatrix`

The `translationVector` inputs must be real, nonsparse, and of the same class.

## Output Arguments

#### **camMatrix** — Camera projection matrix

4-by-3 matrix

Camera projection matrix, returned as a 4-by-3 matrix. The matrix contains the 3-D world points in homogenous coordinates that are projected into the image. When you set `rotationMatrix` and `translationVector` to `double`, the function returns `camMatrix` as `double`. Otherwise it returns `camMatrix` as `single`.

The function computes `camMatrix` as follows:

`camMatrix = [rotationMatrix; translationVector] × K.`

*K*: the intrinsic matrix

Then, using the camera matrix and homogeneous coordinates, you can project a world point onto the image.

$w \times [x,y,1] = [X,Y,Z,1] \times \text{camMatrix}.$

$(X, Y, Z)$ : world coordinates of a point  
 $(x, y)$ : coordinates of the corresponding image point  
 $w$ : arbitrary scale factor

Data Types: `single` | `double`

## More About

- “What Is Camera Calibration?”
- “Single Camera Calibration App”
- “Stereo Calibration App”

## See Also

`Camera Calibrator` | `cameraPose` | `estimateCameraParameters` | `extrinsics` | `triangulate`

**Introduced in R2014b**

## cameraPose

Compute relative rotation and translation between camera poses

### Syntax

```
[orientation,location] = cameraPose(F,cameraParams,inlierPoints1,  
inlierPoints2)  
[orientation,location] = cameraPose(F,cameraParams1,cameraParams2,  
inlierPoints1,inlierPoints2)
```

### Description

[orientation,location] = cameraPose(F,cameraParams,inlierPoints1,inlierPoints2) returns the orientation and the location of a calibrated camera relative to its previous pose. The two poses are related by the fundamental matrix,  $F$ . The function computes the camera pose up to scale and returns location as a unit vector.

[orientation,location] = cameraPose(F,cameraParams1,cameraParams2,inlierPoints1,inlierPoints2) returns the orientation and location of the second camera relative to the first one.

#### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Structure from Motion from Two Views

Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D images. This example shows you how to estimate the poses of a calibrated camera from two images, reconstruct the 3-D structure of the scene up to an

unknown scale factor, and then recover the actual scale factor by detecting an object of a known size.

## Overview

This example shows how to reconstruct a 3-D scene from a pair 2-D images taken with a camera calibrated using the Camera Calibrator app. The algorithm consists of the following steps:

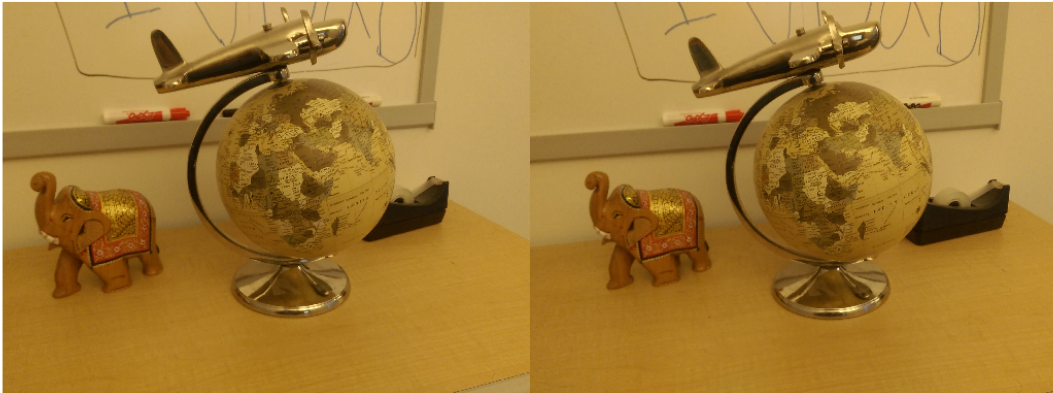
- 1 Match a sparse set of points between the two images. There are multiple ways of finding point correspondences between two images. This example detects corners in the first image using the `detectMinEigenFeatures` function, and tracks them into the second image using `vision.PointTracker`. Alternatively you can use `extractFeatures` followed by `matchFeatures`.
- 2 Estimate the fundamental matrix using `estimateFundamentalMatrix`.
- 3 Compute the motion of the camera using the `cameraPose` function.
- 4 Match a dense set of points between the two images. Re-detect the point using `detectMinEigenFeatures` with a reduced 'MinQuality' to get more points. Then track the dense points into the second image using `vision.PointTracker`.
- 5 Determine the 3-D locations of the matched points using `triangulate`.
- 6 Detect an object of a known size. In this scene there is a globe, whose radius is known to be 10cm. Use `pcfitsphere` to find the globe in the point cloud.
- 7 Recover the actual scale, resulting in a metric reconstruction.

## Read a Pair of Images

Load a pair of images into the workspace.

```
imageDir = fullfile(toolboxdir('vision'), 'visiondata', 'upToScaleReconstructionImages');
images = imageSet(imageDir);
I1 = read(images, 1);
I2 = read(images, 2);
figure
imshowpair(I1, I2, 'montage');
title('Original Images');
```

Original Images



#### Load Camera Parameters

This example uses the camera parameters calculated by the cameraCalibrator app. The parameters are stored in the `cameraParams` object, and include the camera intrinsics and lens distortion coefficients.

```
% Load precomputed camera parameters  
load upToScaleReconstructionCameraParameters.mat
```

#### Remove Lens Distortion

Lens distortion can affect the accuracy of the final reconstruction. You can remove the distortion from each of the images using the `undistortImage` function. This process straightens the lines that are bent by the radial distortion of the lens.

```
I1 = undistortImage(I1, cameraParams);  
I2 = undistortImage(I2, cameraParams);  
figure  
imshowpair(I1, I2, 'montage');  
title('Undistorted Images');
```

Undistorted Images



### Find Point Correspondences Between The Images

Detect good features to track. You can reduce `'MinQuality'` to detect fewer points, which are more uniformly distributed throughout the image. Tracking with the KLT algorithm is a good way to establish point correspondences when there is less camera motion.

```
% Detect feature points
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.1);

% Visualize detected points
figure
imshow(I1, 'InitialMagnification', 50);
title('150 Strongest Corners from the First Image');
hold on
plot(selectStrongest(imagePoints1, 150));

% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 5);

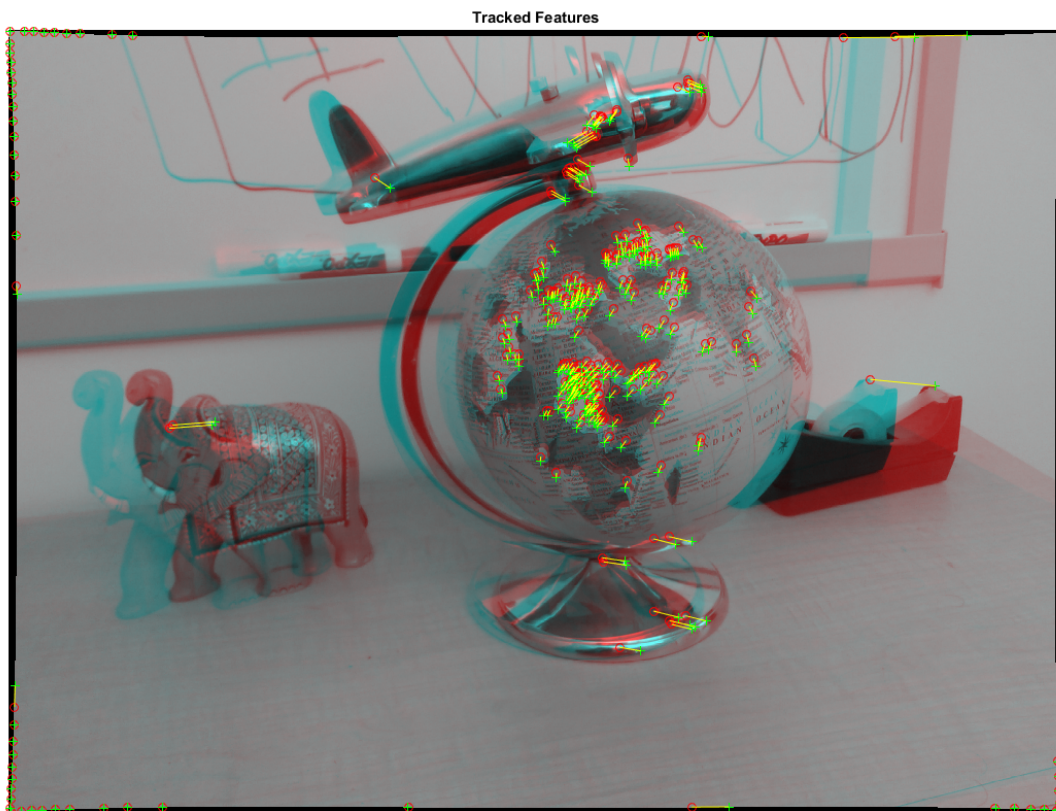
% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
```

```
matchedPoints1 = imagePoints1(validIdx, :);  
matchedPoints2 = imagePoints2(validIdx, :);  
  
% Visualize correspondences  
figure  
showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2);  
title('Tracked Features');
```







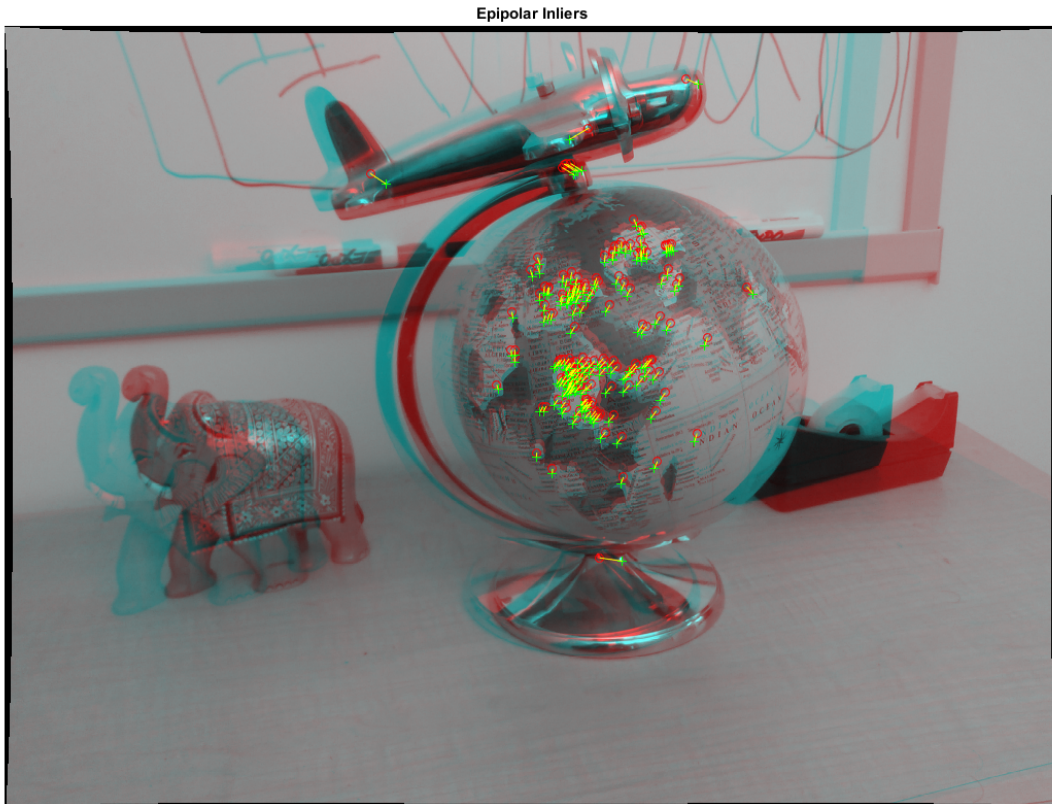
### Estimate the Fundamental Matrix

Use the `estimateFundamentalMatrix` function to compute the fundamental matrix and find the inlier points that meet the epipolar constraint.

```
% Estimate the fundamental matrix
[fMatrix, epipolarInliers] = estimateFundamentalMatrix(...
    matchedPoints1, matchedPoints2, 'Method', 'MSAC', 'NumTrials', 10000);
```

```
% Find epipolar inliers
inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);
```

```
% Display inlier matches
figure
showMatchedFeatures(I1, I2, inlierPoints1, inlierPoints2);
title('Epipolar Inliers');
```



#### Compute the Camera Pose

Compute the rotation and translation between the camera poses corresponding to the two images. Note that  $\mathbf{t}$  is a unit vector, because translation can only be computed up to scale.

```
[R, t] = cameraPose(fMatrix, cameraParams, inlierPoints1, inlierPoints2);
```

## Reconstruct the 3-D Locations of Matched Points

Re-detect points in the first image using lower 'MinQuality' to get more points. Track the new points into the second image. Estimate the 3-D locations corresponding to the matched points using the `triangulate` function, which implements the Direct Linear Transformation (DLT) algorithm [1]. Place the origin at the optical center of the camera corresponding to the first image.

```
% Detect dense feature points
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.001);

% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 5);

% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);

% Compute the camera matrices for each position of the camera
% The first camera is at the origin looking along the X-axis. Thus, its
% rotation matrix is identity, and its translation vector is 0.
camMatrix1 = cameraMatrix(cameraParams, eye(3), [0 0 0]);
camMatrix2 = cameraMatrix(cameraParams, R', -t*R');

% Compute the 3-D points
points3D = triangulate(matchedPoints1, matchedPoints2, camMatrix1, camMatrix2);

% Get the color of each reconstructed point
numPixels = size(I1, 1) * size(I1, 2);
allColors = reshape(I1, [numPixels, 3]);
colorIdx = sub2ind([size(I1, 1), size(I1, 2)], round(matchedPoints1(:,2)), ...
    round(matchedPoints1(:, 1)));
color = allColors(colorIdx, :);

% Create the point cloud
ptCloud = pointCloud(points3D, 'Color', color);
```

#### Display the 3-D Point Cloud

Use the `plotCamera` function to visualize the locations and orientations of the camera, and the `pcshow` function to visualize the point cloud.

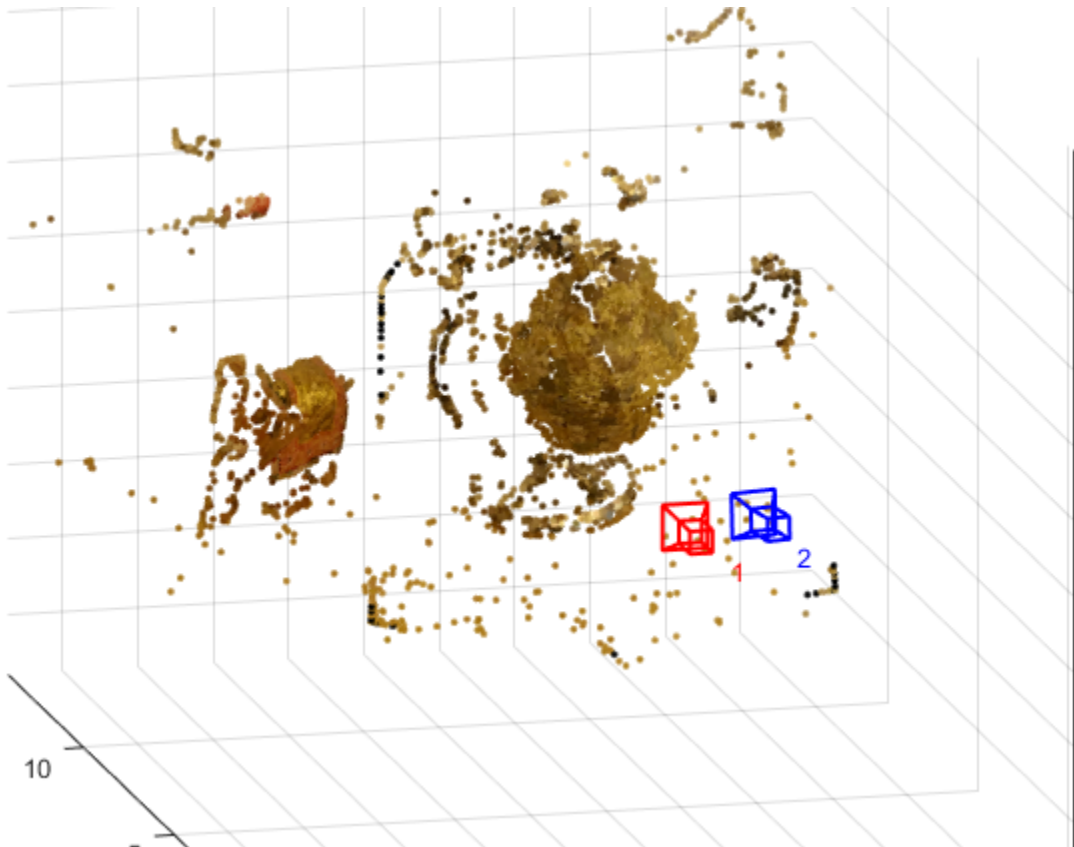
```
% Visualize the camera locations and orientations
cameraSize = 0.3;
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
           'Color', 'b', 'Label', '2', 'Opacity', 0);

% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
       'MarkerSize', 45);

% Rotate and zoom the plot
camorbit(0, -30);
camzoom(1.5);

% Label the axes
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis')

title('Up to Scale Reconstruction of the Scene');
```

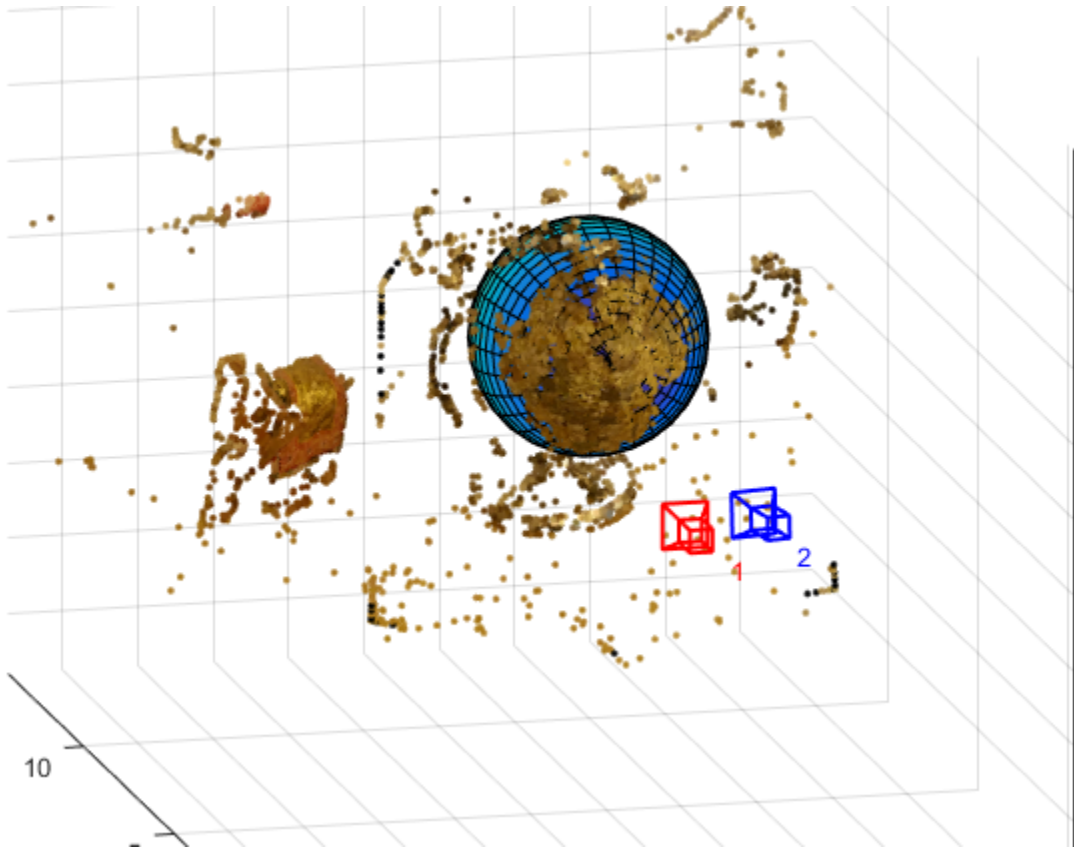


### Fit a Sphere to the Point Cloud to Find the Globe

Find the globe in the point cloud by fitting a sphere to the 3-D points using the `pcfitsphere` function.

```
% Detect the globe
globe = pcfitsphere(ptCloud, 0.1);

% Display the surface of the globe
plot(globe);
title('Estimated Location and Size of the Globe');
hold off
```



#### Metric Reconstruction of the Scene

The actual radius of the globe is 10cm. You can now determine the coordinates of the 3-D points in centimeters.

```
% Determine the scale factor
scaleFactor = 10 / globe.Radius;

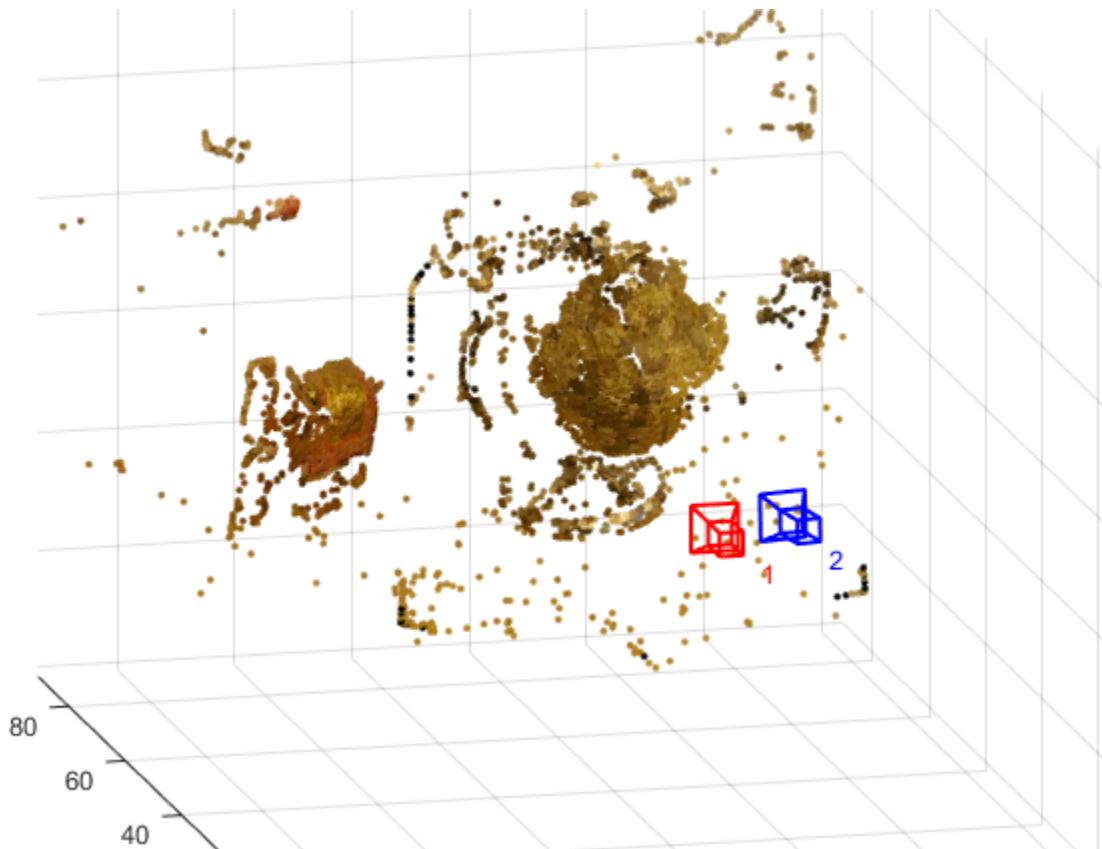
% Scale the point cloud
ptCloud = pointCloud(points3D * scaleFactor, 'Color', color);
t = t * scaleFactor;

% Visualize the point cloud in centimeters
cameraSize = 2;
```

```
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
          'Color', 'b', 'Label', '2', 'Opacity', 0);

% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
       'MarkerSize', 45);
camorbit(0, -30);
camzoom(1.5);

% Label the axes
xlabel('x-axis (cm)');
ylabel('y-axis (cm)');
zlabel('z-axis (cm)')
title('Metric Reconstruction of the Scene');
```



#### Summary

This example showed how to recover camera motion and reconstruct the 3-D structure of a scene from two images taken with a calibrated camera.



## References

[1] Hartley, Richard, and Andrew Zisserman. Multiple View Geometry in Computer Vision. Second Edition. Cambridge, 2000.

## Input Arguments

### **F** — Fundamental matrix

3-by-3 matrix

Fundamental matrix, specified as a 3-by-3 matrix. You can obtain this matrix using the `estimateFundamentalMatrix` function.

Data Types: `single` | `double`

### **cameraParams** — Camera parameters

`cameraParameters` object

Camera parameters, specified as a `cameraParameters` object. You can obtain this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

### **cameraParams1** — Camera parameters for camera 1

`cameraParameters` object

Camera parameters for camera 1, specified as a `cameraParameters` object. You can obtain this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

### **cameraParams2** — Camera parameters for camera 2

`cameraParameters` object

Camera parameters for camera 2, specified as a `cameraParameters` object. You can obtain this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

### **inlierPoints1** — Coordinates of corresponding points in view 1

`SURFPoints` | `cornerPoints` | `MSERRegions` |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Coordinates of corresponding points in view 1, specified as an  $M$ -by-2 matrix of  $M$  number of  $[x,y]$  coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object. You can obtain these points using the estimateFundamentalMatrix function.

### **inlierPoints2** — Coordinates of corresponding points in view 2

SURFPoints | cornerPoints | MSERRegions |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Coordinates of corresponding points in view 2, specified as an  $M$ -by-2 matrix of  $M$  number of  $[x,y]$  coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object. You can obtain these points using the estimateFundamentalMatrix function.

## Output Arguments

### **orientation** — Orientation of camera

3-by-3 matrix

Orientation of camera, returned as a 3-by-3 matrix. If you use only one camera, the matrix describes the orientation of the second camera pose relative to the first camera pose. If you use two cameras, the matrix describes the orientation of camera 2 relative to camera 1.

Data Types: single | double

### **location** — Location of camera

1-by-3 vector

Location of camera, returned as a 1-by-3 unit vector. If you use only one camera, the vector describes the location of the second camera pose relative to the first camera pose. If you use two cameras, the vector describes the location of camera 2 relative to camera 1.

Data Types: single | double

## More About

### Tips

You can compute the camera extrinsics, rotationMatrix and translationVector, corresponding to the camera pose, from orientation and location:

- rotationMatrix = orientation'

- `translationVector = -location * rotationMatrix`

The orientation of the previous camera pose is the identity matrix, `eye(3)`, and its location is, `[0,0,0]`.

You can then use `rotationMatrix` and `translationVector` as inputs to the `cameraMatrix` function.

You can compute four possible combinations of orientation and location from the input fundamental matrix. Three of the combinations are not physically realizable, because they project 3-D points behind one or both cameras. The `cameraPose` function uses `inlierPoints1` and `inlierPoints2` to determine the realizable combination.

- “Point Feature Types”
- “Single Camera Calibration App”
- “Stereo Calibration App”
- “Structure from Motion”

## See Also

Camera Calibrator | `cameraMatrix` | `estimateCameraParameters` | `estimateFundamentalMatrix` | `plotCamera` | `triangulate`

**Introduced in R2015b**

## extractLBPFeatures

Extract local binary pattern (LBP) features

### Syntax

```
features = extractLBPFeatures(I)  
features = extractLBPFeatures(I,Name,Value)
```

### Description

`features = extractLBPFeatures(I)` returns extracted uniform local binary pattern (LBP) from a grayscale image. The LBP features encode local texture information.

`features = extractLBPFeatures(I,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

#### Code Generation Support:

Supports code generation: Yes

Generates platform-dependent library: No

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Using LBP Features to Differentiate Images by Texture

Read images that contain different textures.

```
brickWall = imread('bricks.jpg');  
rotatedBrickWall = imread('bricksRotated.jpg');  
carpet = imread('carpet.jpg');
```

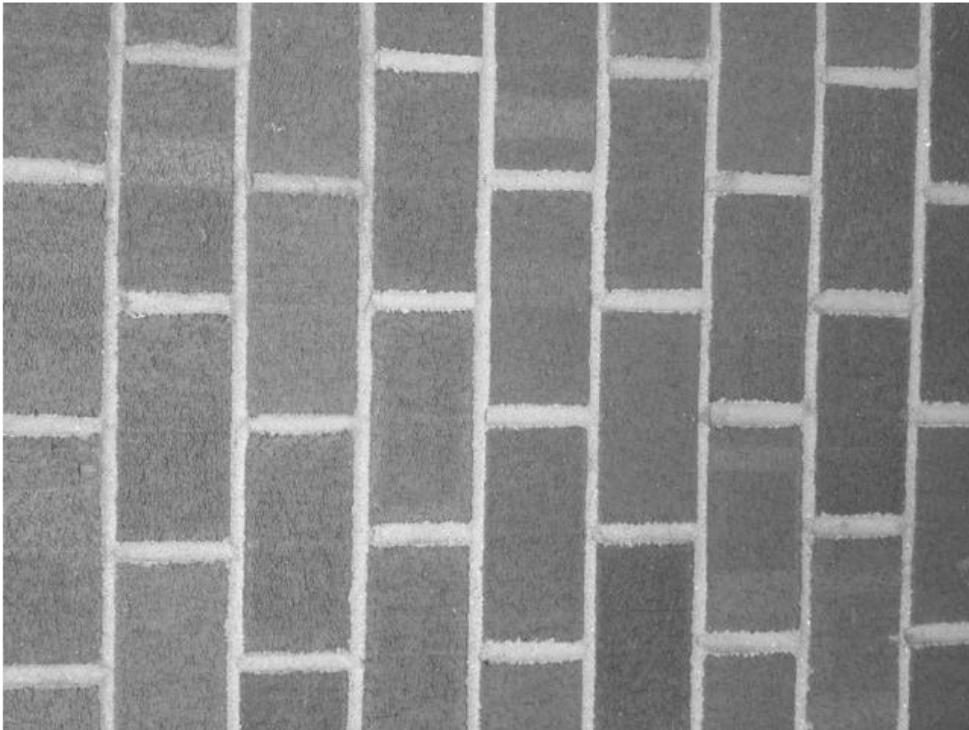
Display the images.

```
figure
```

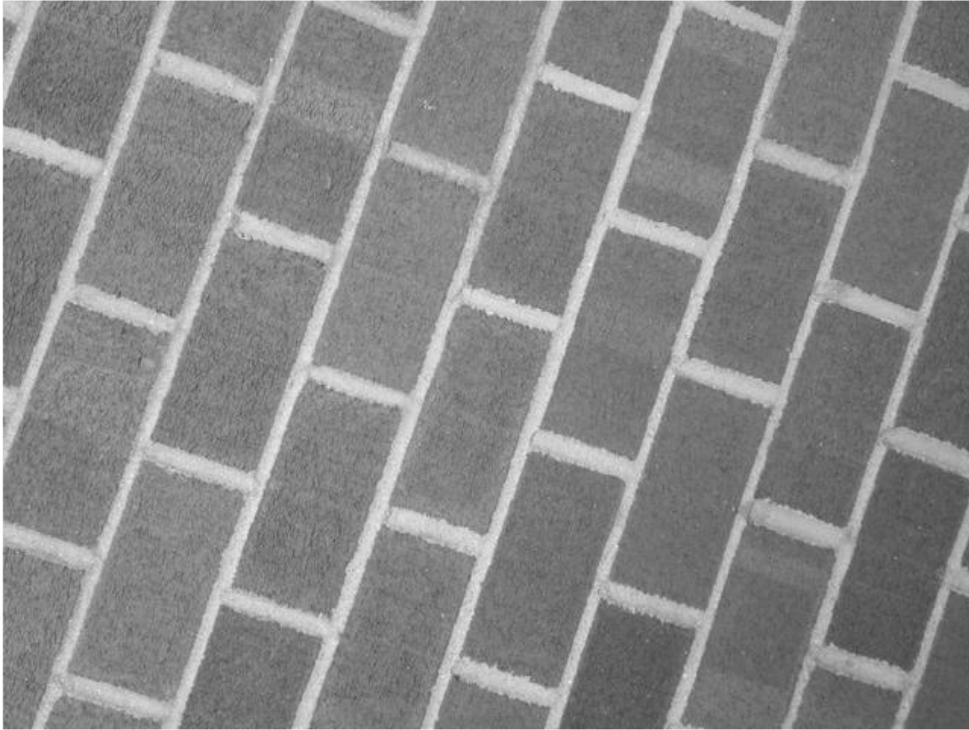
```
imshow(brickWall)
title('Bricks')

figure
imshow(rotatedBrickWall)
title('Rotated Bricks')

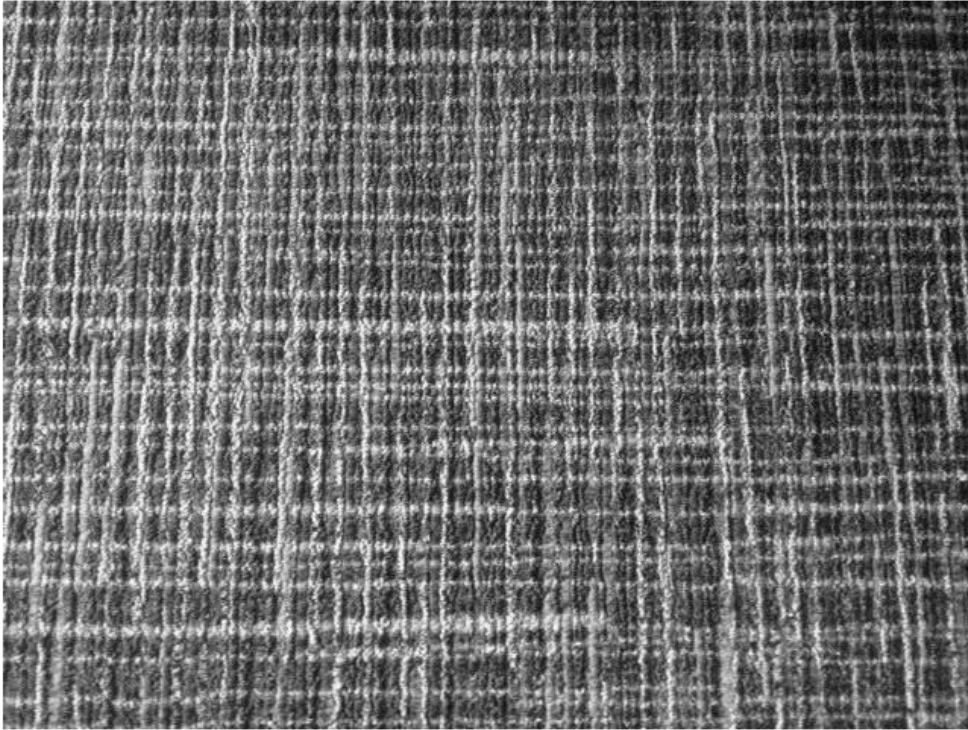
figure
imshow(carpet)
title('Carpet')
```

**Bricks**

**Rotated Bricks**



Carpet



Extract LBP features from the images to encode their texture information.

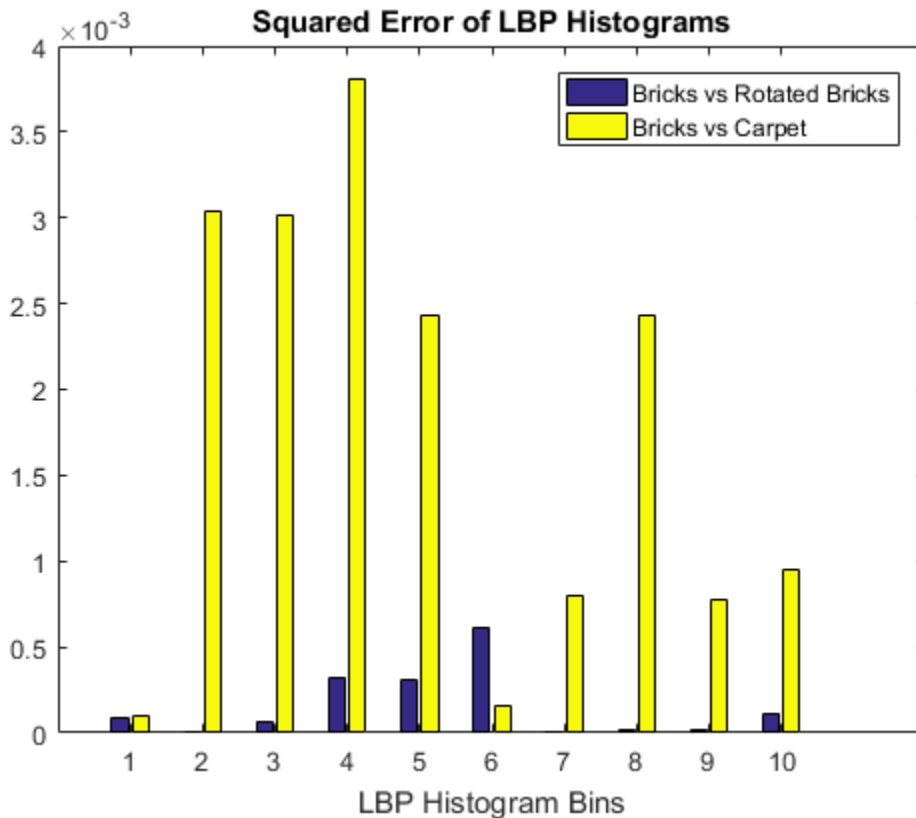
```
lbpBricks1 = extractLBPFeatures(brickWall, 'Upright', false);  
lbpBricks2 = extractLBPFeatures(rotatedBrickWall, 'Upright', false);  
lbpCarpet = extractLBPFeatures(carpet, 'Upright', false);
```

Gauge the similarity between the LBP features by computing the squared error between them.

```
brickVsBrick = (lbpBricks1 - lbpBricks2).^2;  
brickVsCarpet = (lbpBricks1 - lbpCarpet).^2;
```

Visualize the squared error to compare bricks versus bricks and bricks versus carpet. The squared error is smaller when images have similar texture.

```
figure
bar([brickVsBrick; brickVsCarpet]','grouped')
title('Squared Error of LBP Histograms')
xlabel('LBP Histogram Bins')
legend('Bricks vs Rotated Bricks','Bricks vs Carpet')
```



#### Apply L1 Normalization to LBP Features

Read in a sample image and convert it to grayscale.

```
I = imread('gantrycrane.png');
```



```
I = rgb2gray(I);
```

Extract unnormalized LBP features so that you can apply a custom normalization.

```
lbpFeatures = extractLBPFeatures(I, 'CellSize', [32 32], 'Normalization', 'None');
```

Reshape the LBP features into a *number of neighbors* -by- *number of cells* array to access histograms for each individual cell.

```
numNeighbors = 8;
numBins = numNeighbors*(numNeighbors-1)+3;
lbpCellHists = reshape(lbpFeatures,numBins,[]);
```

Normalize each LBP cell histogram using L1 norm.

```
lbpCellHists = bsxfun(@rdivide,lbpCellHists,sum(lbpCellHists));
```

Reshape the LBP features vector back to 1-by- *N* feature vector.

```
lbpFeatures = reshape(lbpCellHists,1,[]);
```

## Input Arguments

### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified as an *M*-by-*N* 2-D grayscale image that is real, and non-sparse.

Data Types: `logical` | `single` | `double` | `int16` | `uint8` | `uint16`

## Name-Value Pair Arguments

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

Example: `'NumNeighbors',8`

### **Algorithm Parameters**

The LBP algorithm parameters control how local binary patterns are computed for each pixel in the input image.

#### 'NumNeighbors' — Number of neighbors

8 (default) | positive integer

Number of neighbors used to compute the LBP for each pixel in the input image, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer. The set of neighbors is selected from a circularly symmetric pattern around each pixel. Increase the number of neighbors to encode greater detail around each pixel. Typical values range from 4 to 24.

#### 'Radius' — Radius of circular pattern to select neighbors

1 (default) | positive integer

Radius of circular pattern used to select neighbors for each pixel in the input image, specified as the comma-separated pair consisting of 'Radius' and a positive integer. To capture detail over a larger spatial scale, increase the radius. Typical values range from 1 to 5.

#### 'Upright' — Rotation invariance flag

true | logical scalar

Rotation invariance flag, specified as the comma-separated pair consisting of 'Upright' and a logical scalar. When you set this property to true, the LBP features do not encode rotation information. Set 'Upright' to false when rotationally invariant features are required.

#### 'Interpolation' — Interpolation method

'Linear' (default) | 'Nearest'

Interpolation method used to compute pixel neighbors, specified as the comma-separated pair consisting of 'Interpolation' and the string 'Linear' or 'Nearest'. Use 'Nearest' for faster computation, but with less accuracy.

#### Histogram Parameters

The histogram parameters determine how the distribution of binary patterns is aggregated over the image to produce the output features.

#### 'CellSize' — Cell size

size(I) (default) | 2-element vector

Cell size, specified as the comma-separated pair consisting of 'CellSize' and a 2-element vector. The number of cells is calculated as  $\text{floor}(\text{size}(I)/\text{CellSize})$ .

**'Normalization' — Type of normalization**

'L2' (default) | 'None'

Type of normalization applied to each LBP cell histogram, specified as the comma-separated pair consisting of 'Normalization' and the string 'L2' or 'None'. To apply a custom normalization method as a post-processing step, set this value to 'None'.

## Output Arguments

**features — LBP feature vector**1-by- $N$  vector

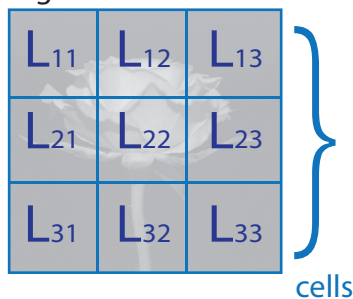
LBP feature vector, returned as a 1-by- $N$  vector of length  $N$  representing the number of features. LBP features encode local texture information, which you can use for tasks such as classification, detection, and recognition. The function partitions the input image into non-overlapping cells. To collect information over larger regions, select larger cell sizes. However, when you increase the cell size, you lose local detail.  $N$ , depends on the number of cells in the image,  $numCells$ , the number of neighbors,  $P$ , and the Upright parameter.

The number of cells is calculated as:

$$numCells = \text{prod}(\text{floor}(\text{size}(I)/\text{CellSize}))$$

The figure shows an image with nine cell histograms. Each histogram describes an LBP feature.

Image



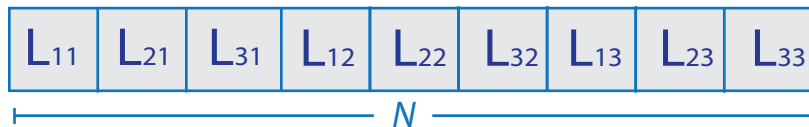
The size of the histogram in each cell is  $[1, B]$ , where  $B$  is the number of bins in the histogram. The number of bins depends on the Upright property and the number of neighbors,  $P$ .

Upright	Number of Bins
true	$numCells \times (P \times P - 1) + 3$
false	$numCells \times (P + 2)$

The overall LBP feature length,  $N$ , depends on the number of cells and the number of bins,  $B$ :

$$N = numCells \times B$$

LBP features = size(Lyx) = [1, B]



## More About

- “Local Feature Detection and Extraction”

## References

- [1] Ojala, T., M. Pietikainen, and T. Maenpaa. “Multiresolution Gray Scale and Rotation Invariant Texture Classification With Local Binary Patterns.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 24, Issue 7, July 2002, pp. 971-987.

## See Also

SURFPoints | MSERRegions | detectBRISKFeatures | detectFASTFeatures | detectHarrisFeatures | detectMinEigenFeatures | detectMSERFeatures | detectSURFFeatures | extractFeatures | extractHOGFeatures | matchFeatures

**Introduced in R2015b**

# configureKalmanFilter

Create Kalman filter for object tracking

## Syntax

```
kalmanFilter = configureKalmanFilter(MotionModel,InitialLocation,  
InitialEstimateError,MotionNoise,MeasurementNoise)
```

## Description

`kalmanFilter = configureKalmanFilter(MotionModel,InitialLocation,InitialEstimateError,MotionNoise,MeasurementNoise)` returns a `vision.KalmanFilter` object configured to track a physical object. This object moves with constant velocity or constant acceleration in an  $M$ -dimensional Cartesian space. The function determines the number of dimensions,  $M$ , from the length of the `InitialLocation` vector.

This function provides a simple approach for configuring the `vision.KalmanFilter` object for tracking a physical object in a Cartesian coordinate system. The tracked object may move with either constant velocity or constant acceleration. The statistics are the same along all dimensions. If you need to configure a Kalman filter with different assumptions, use the `vision.KalmanFilter` object directly.

## Examples

### Track an Occluded Object

Detect and track a ball using Kalman filtering, foreground detection, and blob analysis.

Create System objects to read the video frames, detect foreground physical objects, and display results.

```
videoReader = vision.VideoFileReader('singleball.mp4');  
videoPlayer = vision.VideoPlayer('Position',[100,100,500,400]);
```

```
foregroundDetector = vision.ForegroundDetector('NumTrainingFrames',10,'InitialVariance',1000000);
blobAnalyzer = vision.BlobAnalysis('AreaOutputPort',false,'MinimumBlobArea',70);
```

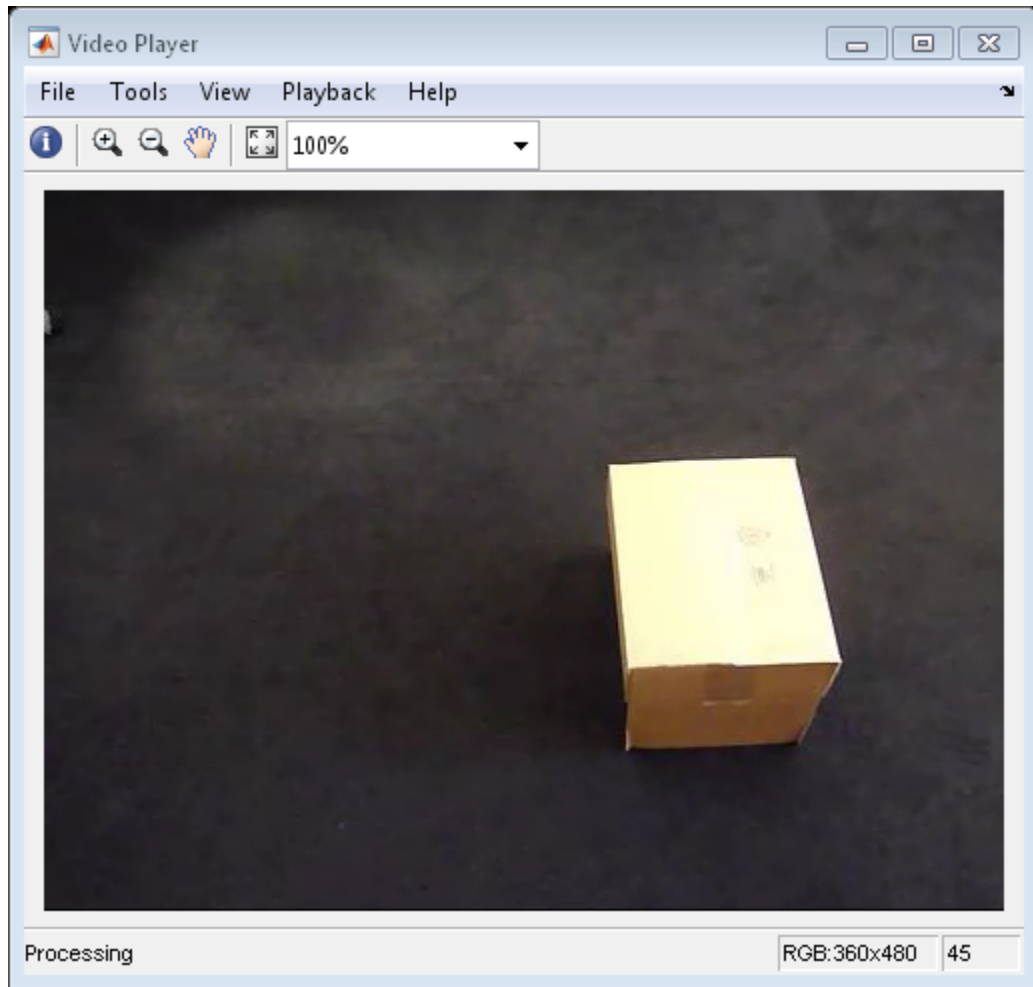
Process each video frame to detect and track the ball. After reading the current video frame, the example searches for the ball by using background subtraction and blob analysis. When the ball is first detected, the example creates a Kalman filter. The Kalman filter determines the ball's location, whether it is detected or not. If the ball is detected, the Kalman filter first predicts its state at the current video frame. The filter then uses the newly detected location to correct the state, producing a filtered location. If the ball is missing, the Kalman filter solely relies on its previous state to predict the ball's current location.

```
kalmanFilter = []; isTrackInitialized = false;
while ~isDone(videoReader)
    colorImage = step(videoReader);

    foregroundMask = step(foregroundDetector, rgb2gray(colorImage));
    detectedLocation = step(blobAnalyzer,foregroundMask);
    isObjectDetected = size(detectedLocation, 1) > 0;

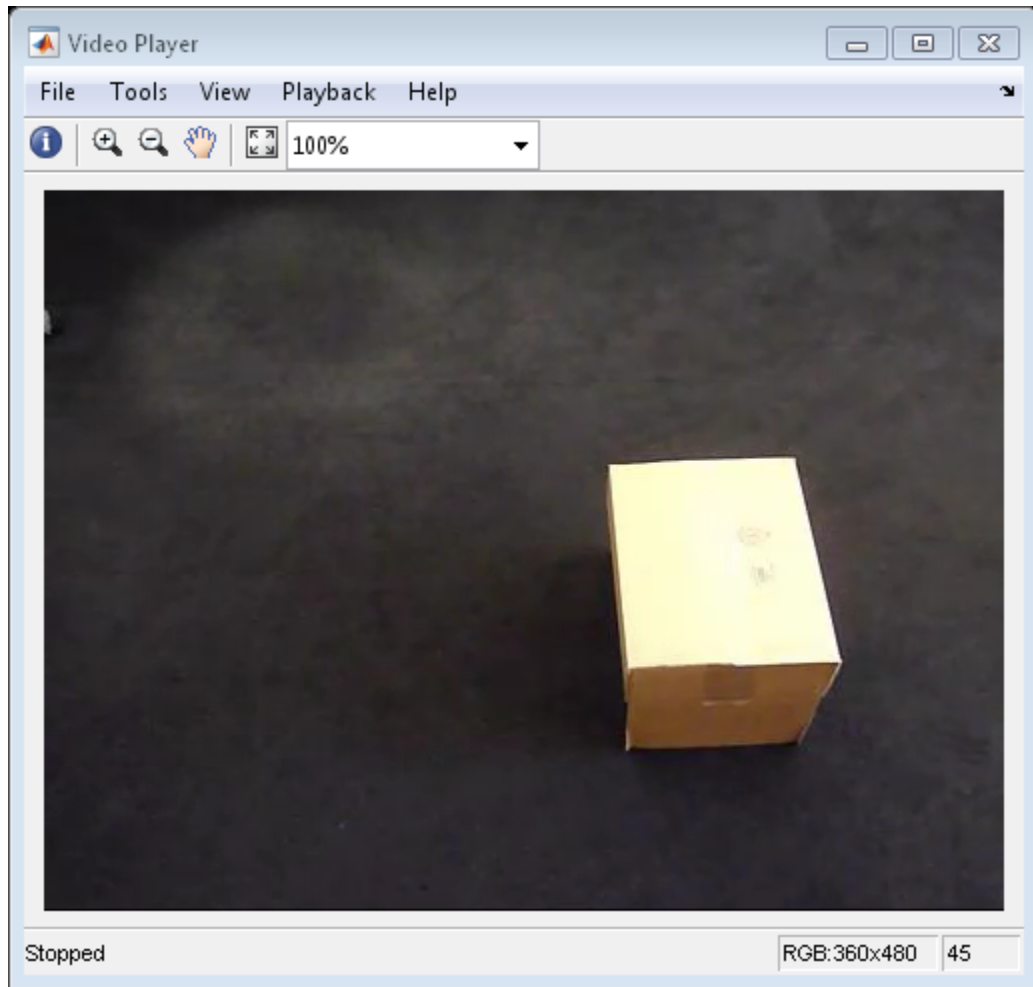
    if ~isTrackInitialized
        if isObjectDetected
            kalmanFilter = configureKalmanFilter('ConstantAcceleration',detectedLocation(1,:),
                isTrackInitialized = true;
        end
        label = ''; circle = zeros(0,3);
    else
        if isObjectDetected
            predict(kalmanFilter);
            trackedLocation = correct(kalmanFilter, detectedLocation(1,:));
            label = 'Corrected';
        else
            trackedLocation = predict(kalmanFilter);
            label = 'Predicted';
        end
        circle = [trackedLocation, 5];
    end

    colorImage = insertObjectAnnotation(colorImage,'circle',circle,label,'Color','red');
    step(videoPlayer,colorImage);
end
```



Release resources.

```
release(videoPlayer);  
release(videoReader);
```



- “Using Kalman Filter for Object Tracking”
- “Motion-Based Multiple Object Tracking”

## Input Arguments

### **MotionModel** — Motion model

'ConstantVelocity' | 'ConstantAcceleration'



Motion model, specified as the string 'ConstantVelocity' or 'ConstantAcceleration'. The motion model you select applies to all dimensions. For example, for the 2-D Cartesian coordinate system. This mode applies to both *X* and *Y* directions.

Data Types: char

### **InitialLocation** – Initial location of object

vector

Initial location of object, specified as a numeric vector. This argument also determines the number of dimensions for the coordinate system. For example, if you specify the initial location as a two-element vector,  $[x_0, y_0]$ , then a 2-D coordinate system is assumed.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **InitialEstimateError** – Initial estimate uncertainty variance

2-element vector | 3-element vector

Initial estimate uncertainty variance, specified as a two- or three-element vector. The initial estimate error specifies the variance of the initial estimates of location, velocity, and acceleration of the tracked object. The function assumes a zero initial velocity and acceleration for the object, at the location you set with the **InitialLocation** property. You can set the **InitialEstimateError** to an approximated value:

*(assumed values – actual values)<sup>2</sup> + the variance of the values*

The value of this property affects the Kalman filter for the first few detections. Later, the estimate error is determined by the noise and input data. A larger value for the initial estimate error helps the Kalman filter to adapt to the detection results faster. However, a larger value also prevents the Kalman filter from removing noise from the first few detections.

Specify the initial estimate error as a two-element vector for constant velocity or a three-element vector for constant acceleration:

<b>MotionModel</b>	<b>InitialEstimateError</b>
ConstantVelocity	$[LocationVariance, VelocityVariance]$
ConstantAcceleration	$[LocationVariance, VelocityVariance, AccelerationVariance]$

Data Types: `double` | `single`

#### **MotionNoise** — Deviation of selected and actual model

2-element vector | 3-element vector

Deviation of selected and actual model, specified as a two- or three-element vector. The motion noise specifies the tolerance of the Kalman filter for the deviation from the chosen model. This tolerance compensates for the difference between the object's actual motion and that of the model you choose. Increasing this value may cause the Kalman filter to change its state to fit the detections. Such an increase may prevent the Kalman filter from removing enough noise from the detections. The values of this property stay constant and therefore may affect the long-term performance of the Kalman filter.

<b>MotionModel</b>	<b>InitialEstimateError</b>
<code>ConstantVelocity</code>	<i>[LocationVariance, VelocityVariance]</i>
<code>ConstantAcceleration</code>	<i>[LocationVariance, VelocityVariance, AccelerationVariance]</i>

Data Types: `double` | `single`

#### **MeasurementNoise** — Variance inaccuracy of detected location

scalar

Variance inaccuracy of detected location, specified as a scalar. It is directly related to the technique used to detect the physical objects. Increasing the `MeasurementNoise` value enables the Kalman filter to remove more noise from the detections. However, it may also cause the Kalman filter to adhere too closely to the motion model you chose, putting less emphasis on the detections. The values of this property stay constant, and therefore may affect the long-term performance of the Kalman filter.

Data Types: `double` | `single`

## Output Arguments

#### **kalmanFilter** — Configured Kalman filter tracking

object

Configured Kalman filter, returned as a `vision.KalmanFilter` object for tracking.

## More About

### Algorithms

This function provides a simple approach for configuring the `vision.KalmanFilter` object for tracking. The Kalman filter implements a discrete time, linear State-Space System. The `configureKalmanFilter` function sets the `vision.KalmanFilter` object properties.

The <code>InitialLocation</code> property corresponds to the measurement vector used in the Kalman filter state-space model. This table relates the measurement vector, $M$ , to the state-space model for the Kalman filter.		
<b>State transition model, <math>A</math>, and Measurement model, <math>H</math></b>		
The state transition model, $A$ , and the measurement model, $H$ of the state-space model, are set to block diagonal matrices made from $M$ identical submatrices $A_s$ and $H_s$ , respectively:		
$A = \text{blkdiag}(A_s\_1, A_s\_2, \dots, A_s\_M)$		
$H = \text{blkdiag}(H_s\_1, H_s\_2, \dots, H_s\_M)$		
The submatrices $A_s$ and $H_s$ are described below:		
<b>MotionModel</b>	<b><math>A_s</math></b>	<b><math>H_s</math></b>
'ConstantVelocity'	[1 1; 0 1]	[1 0]
'ConstantAcceleration'	[1 1 0; 0 1 1; 0 0 1]	[1 0 0]
<b>The Initial State, <math>x</math>:</b>		
<b>MotionModel</b>	<b>Initial state, <math>x</math></b>	
'ConstantVelocity'	[InitialLocation(1), 0, ..., InitialLocation( $M$ ), 0]	
'ConstantAcceleration'	[InitialLocation(1), 0, 0, ..., InitialLocation( $M$ ), 0, 0]	
<b>The initial state estimation error covariance matrix, <math>P</math>:</b>		
$P = \text{diag}(\text{repmat}(\text{InitialError}, [1, M]))$		

<b>The process noise covariance, <math>Q</math>:</b>
--

$Q = \text{diag}(\text{ repmat}(\text{MotionNoise}, [1, M]))$
---

<b>The measurement noise covariance, <math>R</math>:</b>
--

$R = \text{diag}(\text{ repmat}(\text{MeasurementNoise}, [1, M]))$ .
--

- “Multiple Object Tracking”

### See Also

[vision.BlobAnalysis](#) | [vision.ForegroundDetector](#) | [vision.KalmanFilter](#)

**Introduced in R2012b**

# depthToPointCloud

Convert Kinect depth image to a 3-D point cloud

## Syntax

```
[xyzPoints,flippedDepthImage] = depthToPointCloud(depthImage,  
depthDevice)
```

## Description

[xyzPoints,flippedDepthImage] = depthToPointCloud(depthImage, depthDevice) returns an xyzPoints matrix of 3-D points and the flipped depth image, flippedDepthImage, from the input depthImage and the depthDevice object configured for Kinect for Windows.

This function requires the Image Acquisition Toolbox.

---

**Note:** The depthToPointCloud will be removed in a future release. Use the pcfromkinect function with equivalent functionality instead.

---

## Examples

### Plot Point Cloud from Kinect for Windows.

Plot a point cloud from Kinect images. This example requires the Image Acquisition Toolbox software and the Kinect camera and connection.

Create a System object for the Kinect device.

```
depthDevice = imaq.VideoDevice('kinect',2)
```

Warm up the camera.

```
step(depthDevice);
```

Load one frame from the device. The initial frame executes slowly because the object must wake up the device.

```
depthImage = step(depthDevice);
```

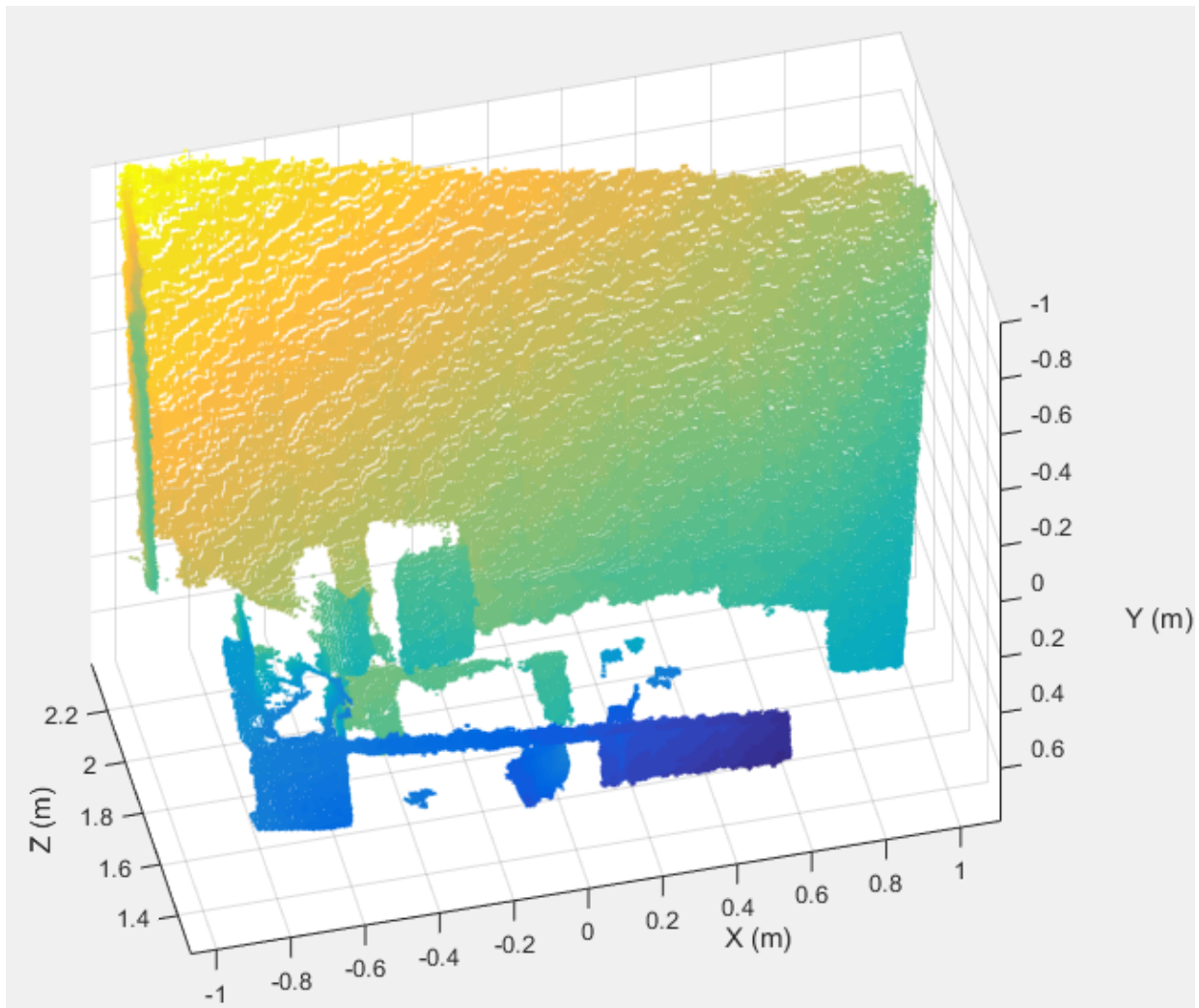
Convert the depth image to the point cloud.

```
xyzPoints = depthToPointCloud(depthImage,depthDevice);
```

Render the point cloud with false color. The axis is set to better visualize the point cloud.

```
pcshow(xyzPoints, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down');  
xlabel('X (m)');  
ylabel('Y (m)');  
zlabel('Z (m)');
```

Display the result.



Release the System object,

```
release(depthDevice);
```

- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **depthImage** — Depth image

*M*-by-*N* matrix

Depth image, specified as an *M*-by-*N* matrix returned by Kinect.

Data Types: `uint16`

### **depthDevice** — Video input object

`videoinput` object | `imaq.VideoDevice` object

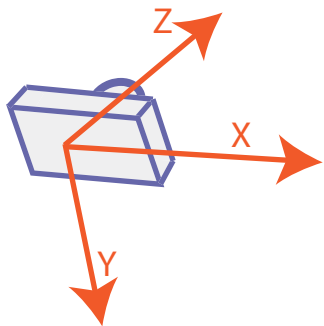
Video input object, specified as either a `videoinput` object or an `imaq.VideoDevice` object configured for Kinect for Windows.

## Output Arguments

### **xyzPoints** — 3-D point cloud

*M*-by-*N*-by-3 matrix

3-D point cloud, returned as an *M*-by-*N*-by-3 matrix. The function returns the point cloud units in meters. The origin of a right-handed world coordinate system is at the center of the depth camera. The *x*-axis of the coordinate system points to the right, the *y*-axis points down, and the *z*-axis points away from the camera.



The limited range of the Kinect depth camera might cause some pixels in the depth image to not have corresponding 3-D coordinates. The function sets the values for those pixels to NaN in `xyzPoints`.



Data Types: `single`

### **flippedDepthImage** — Depth image

*M*-by-*N* matrix

Depth image, returned as an *M*-by-*N* matrix. The Kinect system, designed for gaming applications, returns a mirror image of the scene. This function corrects the output depth image to match the actual scene. It returns the `flippedDepthImage` left-to-right version of the input `depthImage`.

You can obtain the same functionality using `fliplr(depthImage)`.

## **More About**

- “Coordinate Systems”

## **See Also**

`imag.VideoDevice` | `pcfromkinect` | `pcshow` | `videoinput`

**Introduced in R2014b**

## detectBRISKFeatures

Detect BRISK features and return `BRISKPoints` object

### Syntax

```
points = detectBRISKFeatures(I)
points = detectBRISKFeatures(I,Name,Value)
```

### Description

`points = detectBRISKFeatures(I)` returns a `BRISKPoints` object, `points`. The object contains information about BRISK features detected in a 2-D grayscale input image, `I`. The `detectBRISKFeatures` function uses a Binary Robust Invariant Scalable Keypoints (BRISK) algorithm to detect multiscale corner features.

`points = detectBRISKFeatures(I,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

#### Code Generation Support:

Supports MATLAB Function block: No

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Detect BRISK Points in an Image and Mark Their Locations

Read the image.

```
I = imread('cameraman.tif');
```

Find the BRISK points.

```
points = detectBRISKFeatures(I);
```

Display the results.

```
imshow(I); hold on;
```

```
plot(points.selectStrongest(20));
```



## Input Arguments

### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified in 2-D grayscale. The input image must be real and nonsparse.

Example:

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: `'MinQuality',0.1,'ROI',[50,150,100,200]` specifies that the detector must use a 10% minimum accepted quality of corners within the designated region of interest. This region of interest is located at  $x=50$ ,  $y=150$ . The ROI has a width of 100 pixels and a height of 200 pixels.

#### **'MinContrast' — Minimum intensity difference**

0.2 (default) | scalar

Minimum intensity difference between a corner and its surrounding region, specified as the comma-separated pair consisting of `'MinContrast'` and a scalar in the range (0 1). The minimum contrast value represents a fraction of the maximum value of the image class. Increase this value to reduce the number of detected corners.

#### **'MinQuality' — Minimum accepted quality of corners**

0.1 (default) | scalar

Minimum accepted quality of corners, specified as the comma-separated pair consisting of `'MinQuality'` and a scalar value in the range [0,1]. The minimum accepted quality of corners represents a fraction of the maximum corner metric value in the image. Increase this value to remove erroneous corners.

#### **'NumOctaves' — Number of octaves**

4 (default) | scalar

Number of octaves to implement, specified as a comma-separated pair consisting of `'NumOctaves'` and an integer scalar, greater than or equal to 0. Increase this value to detect larger blobs. Recommended values are between 1 and 4. When you set `NumOctaves` to 0, the function disables multiscale detection. It performs the detection at the scale of the input image, `I`.

#### **'ROI' — Rectangular region**

[1 1 size(I,2) size(I,1)] (default) | vector

Rectangular region for corner detection, specified as a comma-separated pair consisting of `'ROI'` and a vector of the format `[x y width height]`. The first two integer values `[x y]` represent the location of the upper-left corner of the region of interest. The last two integer values represent the width and height.

## Output Arguments

### **points** — Brisk points

BRISKPoints object

Brisk points, returned as a BRISKPoints object. The object contains information about the feature points detected in the 2-D grayscale input image.

## More About

- “Point Feature Types”

## References

[1] Leutenegger, S., M. Chli and R. Siegwart. “BRISK: Binary Robust Invariant Scalable Keypoints”, *Proceedings of the IEEE International Conference, ICCV*, 2011.

## See Also

MSERRegions | SURFPoints | BRISKPoints | cornerPoints | binaryFeatures | detectFASTFeatures | detectHarrisFeatures | detectMinEigenFeatures | detectMSERFeatures | detectSURFFeatures | extractFeatures | extractHOGFeatures | matchFeatures

**Introduced in R2014a**

## detectCheckerboardPoints

Detect checkerboard pattern in image

### Syntax

```
[imagePoints,boardSize] = detectCheckerboardPoints(I)
```

```
[imagePoints,boardSize,imagesUsed] = detectCheckerboardPoints(  
imageFileNames)
```

```
[imagePoints,boardSize,imagesUsed] = detectCheckerboardPoints(  
images)
```

```
[imagePoints,boardSize,pairsUsed] = detectCheckerboardPoints(  
imageFileNames1,imageFileNames2)
```

```
[imagePoints,boardSize,pairsUsed] = detectCheckerboardPoints(  
images1, images2)
```

### Description

`[imagePoints,boardSize] = detectCheckerboardPoints(I)` detects a black and white checkerboard of size greater than 4-by-4 squares in a 2-D truecolor or grayscale image. The function returns the detected points and dimensions of the checkerboard.

`[imagePoints,boardSize,imagesUsed] = detectCheckerboardPoints(imageFileNames)` detects a checkerboard pattern in a set of input images, provided as an array of file names.

`[imagePoints,boardSize,imagesUsed] = detectCheckerboardPoints(images)` detects a checkerboard pattern in a set of input images, provided as an array of grayscale or truecolor images.

`[imagePoints,boardSize,pairsUsed] = detectCheckerboardPoints(imageFileNames1,imageFileNames2)` detects a checkerboard pattern in stereo pairs of images, provided as cell arrays of file names.

`[imagePoints,boardSize,pairsUsed] = detectCheckerboardPoints(images1, images2)` detects a checkerboard pattern in stereo pairs of images, provided as arrays of grayscale or truecolor images.

**Code Generation Support:**

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

Code generation will not support specifying images as file names or cell arrays of file names. It supports only checkerboard detection in a single image or stereo pair of images. For example, these syntaxes are supported:

- `detectCheckerboardPoints(I1)`
- `detectCheckerboardPoints(I1,I2)`

I1 and I2 are single grayscale or RGB images.

## Examples

### Detect Checkerboard in a Set of Image Files

Create a cell array of file names of calibration images.

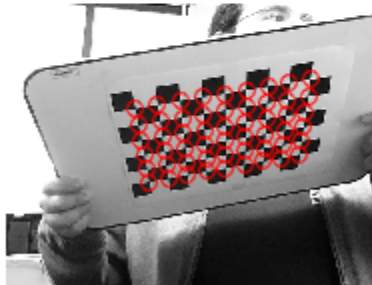
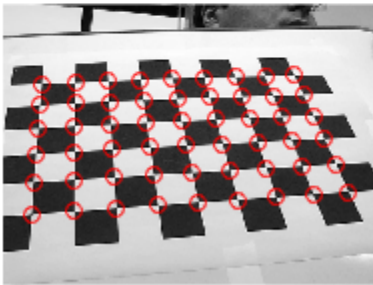
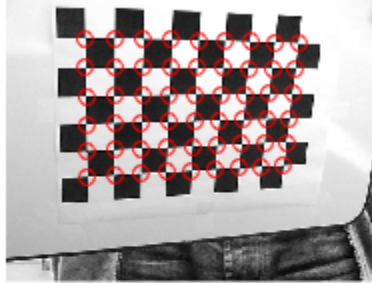
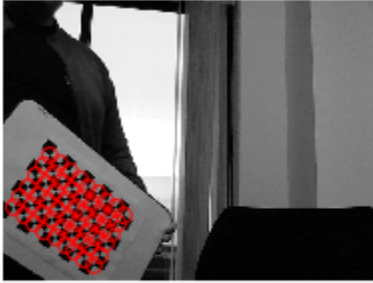
```
for i = 1:5
    imageFileName = sprintf('image%d.tif', i);
    imageFileNames{i} = fullfile(matlabroot, 'toolbox', 'vision', ...
        'visiondata', 'calibration', 'webcam', imageFileName);
end
```

Detect calibration pattern in the images.

```
[imagePoints, boardSize, imagesUsed] = detectCheckerboardPoints(imageFileNames);
```

Display the detected points.

```
imageFileNames = imageFileNames(imagesUsed);
for i = 1:numel(imageFileNames)
    I = imread(imageFileNames{i});
    subplot(2, 2, i);
    imshow(I); hold on; plot(imagePoints(:,1,i), imagePoints(:,2,i), 'ro');
end
```



## Detect Checkerboard in Stereo Images

Read in stereo images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');  
leftImages = imageSet(fullfile(imageDir,'left'));  
rightImages = imageSet(fullfile(imageDir,'right'));  
images1 = leftImages.ImageLocation;  
images2 = rightImages.ImageLocation;
```

Detect the checkerboards in the images.

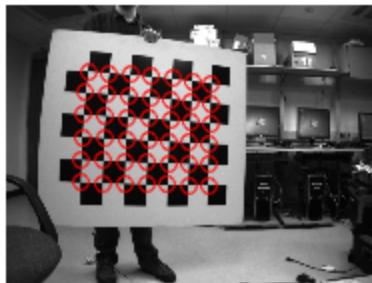
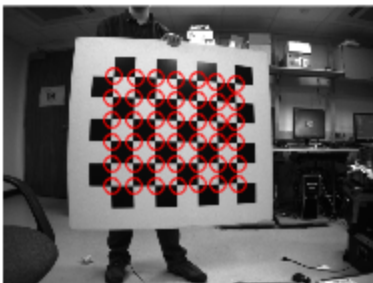
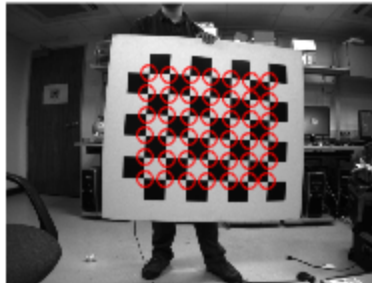
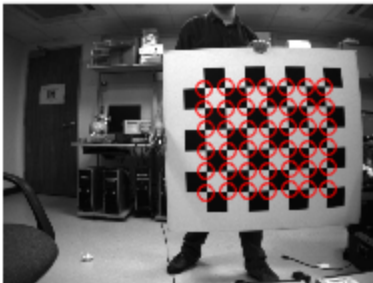
```
[imagePoints,boardSize,pairsUsed] = detectCheckerboardPoints(images1,images2);
```



Display points from images1.

```
images1 = images1(pairsUsed);
figure;
for i = 1:4
    I = imread(images1{i});
    subplot(2, 2, i);
    imshow(I);
    hold on
    plot(imagePoints(:,1,i,1), imagePoints(:,2,i,1), 'ro');
end
annotation('textbox', [0 0.9 1 0.1], 'String', 'Camera 1', 'EdgeColor', 'none', 'Horizontal');
```

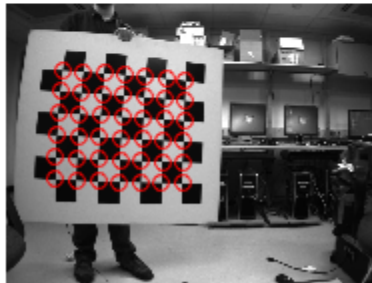
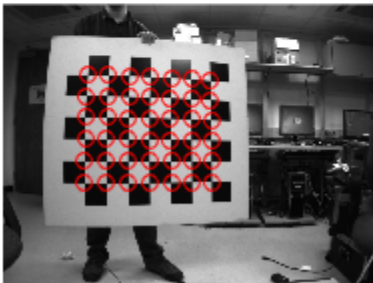
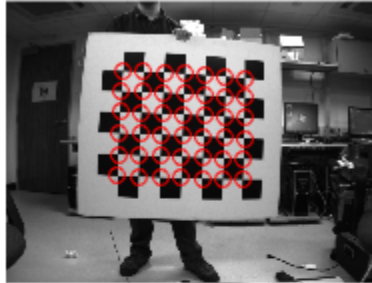
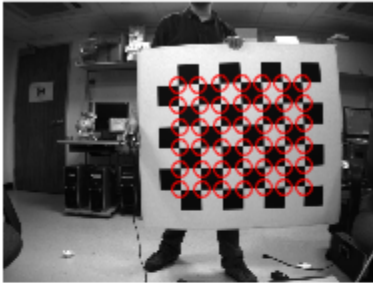
Camera 1



Display points from images2.

```
images2 = images2(pairsUsed);
figure;
for i = 1:4
    I = imread(images2{i});
    subplot(2, 2, i);
    imshow(I);
    hold on
    plot(imagePoints(:,1,i,2), imagePoints(:,2,i,2), 'ro');
end
annotation('textbox', [0 0.9 1 0.1], 'String', 'Camera 2', 'EdgeColor', 'none', 'Horizontal');
```

Camera 2



## Input Arguments

### **I** — Input image

$M$ -by- $N$ -by-3 truecolor image |  $M$ -by- $N$  2-D grayscale image

Input image, specified in either an  $M$ -by- $N$ -by-3 truecolor or  $M$ -by- $N$  2-D grayscale. The input image must be real and nonsparse. The function can detect checkerboards with a minimum size of 4-by-4 squares.

Data Types: single | double | int16 | uint8 | uint16 | logical

#### **imageFileNames** — Image file names

*N*-element cell array

Image file names, specified as an *N*-element cell array of *N* file names.

#### **imageFileNames1** — File names for camera 1 images

*N*-element cell array

File names for camera 1 images, specified as an *N*-element cell array of *N* file names. The images contained in this array must be in the same order as images contained in `imageFileNames2`, forming stereo pairs.

#### **imageFileNames2** — File names for camera 2 images

*N*-element cell array

File names for camera 2 images, specified as an *N*-element cell array of *N* file names. The images contained in this array must be in the same order as images contained in `imageFileNames1`, forming stereo pairs.

#### **images** — Images

*height-by-width-by-color channel-by-number of frames* array

Images, specified as an *H*-by-*W*-by-*B*-by-*F* array containing a set of grayscale or truecolor images. The input dimensions are:

*H* represents the image height.

*W* represents the image width.

*B* represents the color channel. A value of 1 indicates a grayscale image, and a value of 3 indicates a truecolor image.

*F* represents the number of image frames.

#### **images1** — Stereo pair images 1

*height-by-width-by-color channel-by-number of frames* array

Images, specified as an *H*-by-*W*-by-*B*-by-*F* array containing a set of grayscale or truecolor images. The input dimensions are:

*H* represents the image height.

*W* represents the image width.

*B* represents the color channel. A value of 1 indicates a grayscale image, and a value of 3 indicates a truecolor image.

*F* represents the number of image frames.

#### **images2** — Stereo pair images 2

*height-by-width-by-color channel-by-number of frames* array

Images, specified as an  $H$ -by- $W$ -by- $B$ -by- $F$  array containing a set of grayscale or truecolor images. The input dimensions are:

$H$  represents the image height.

$W$  represents the image width.

$B$  represents the color channel. A value of 1 indicates a grayscale image, and a value of 3 indicates a truecolor image.

$F$  represents the number of image frames.

## Output Arguments

### **imagePoints** — Detected checkerboard corner coordinates

$M$ -by-2 matrix |  $M$ -by-2-by- *number of images* array |  $M$ -by-2-by-*number of pairs of images*-by-*number of cameras* array

Detected checkerboard corner coordinates, returned as an  $M$ -by-2 matrix for one image. For multiple images, points are returned as an  $M$ -by-2-by-*number of images* array, and for stereo pairs of images, the function returns points as an  $M$ -by-2-by-*number of pairs*-by-*number of cameras* array.

For stereo pairs, `imagePoints(:, :, 1)` are the points from the first set of images, and `imagePoints(:, :, 2)` are the points from the second set of images. The output contains  $M$  number of  $[x\ y]$  coordinates. Each coordinate represents a point where square corners are detected on the checkerboard. The number of points the function returns depends on the value of `boardSize`, which indicates the number of squares detected. The function detects the points with sub-pixel accuracy.

The function calculates the number of points,  $M$ , as follows:

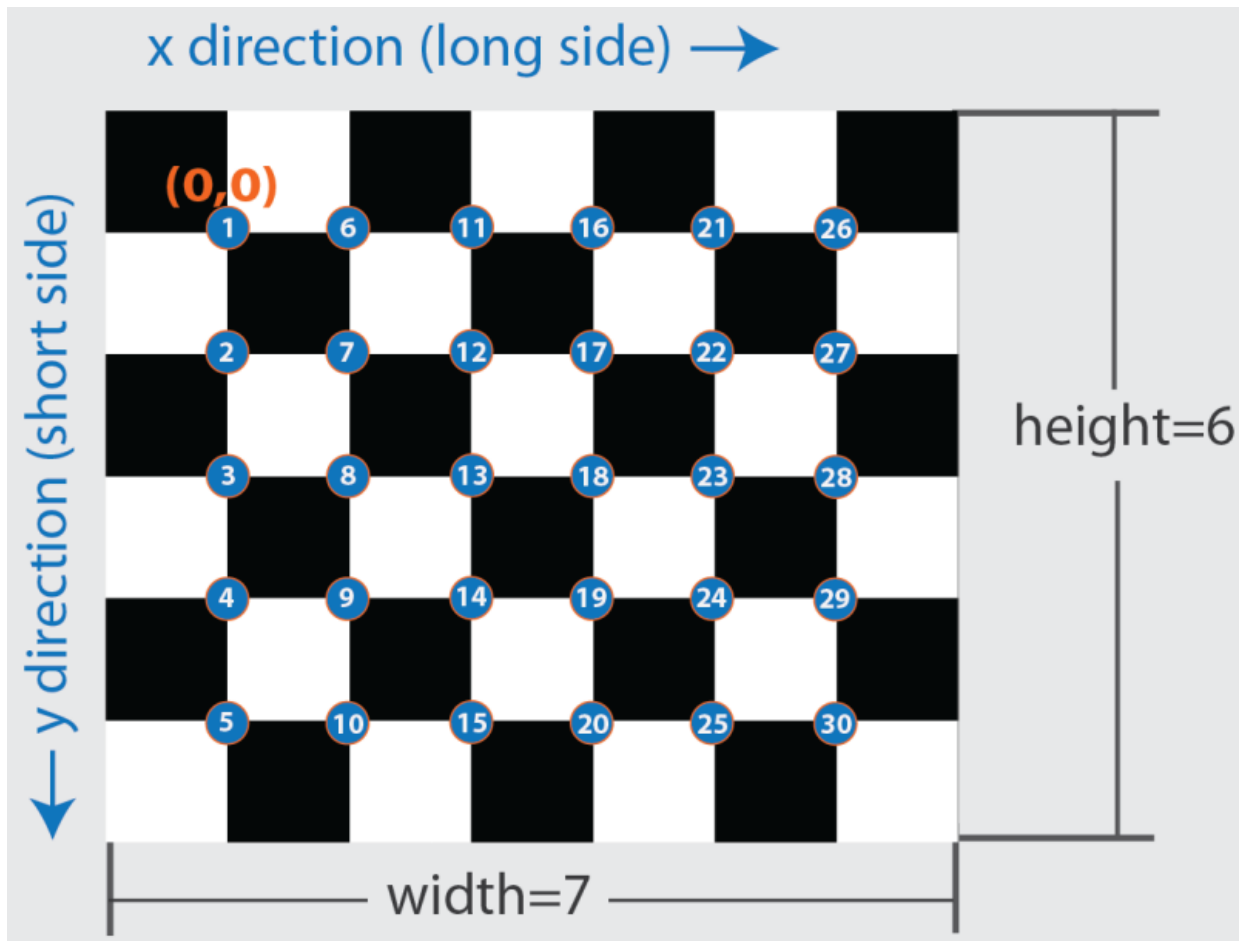
$M = \text{prod}(\text{boardSize}-1)$ .

If the checkerboard cannot be detected:

`imagePoints = []`

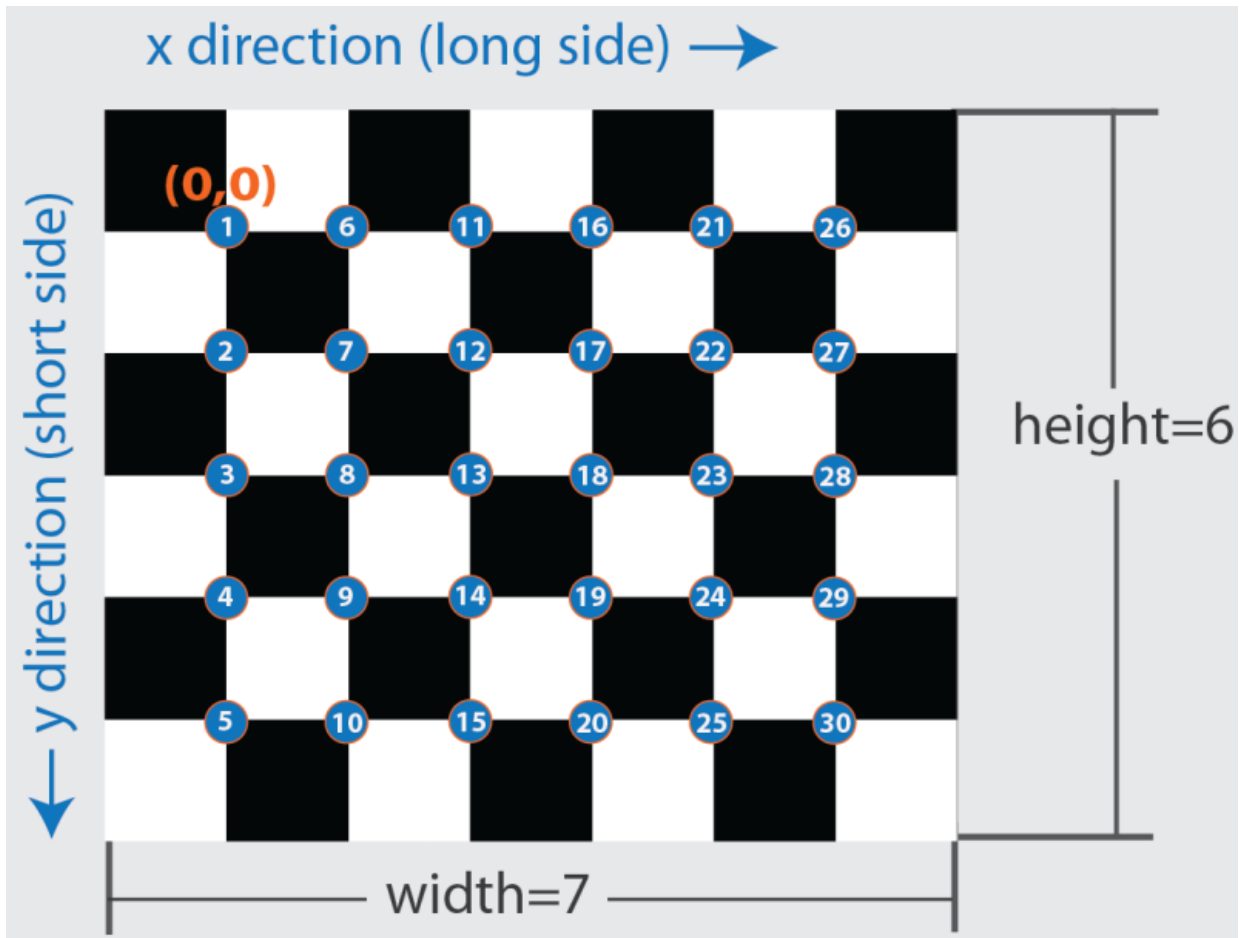
`boardSize = [0,0]`

When you specify the `imageFileNames` input, the function can return `imagePoints` as an  $M$ -by-2-by- $N$  array. In this array,  $N$  represents the number of images in which a checkerboard is detected.

**boardSize** — Checkerboard dimensions

2-element  $[height, width]$  vector

Checkerboard dimensions, returned as a 2-element  $[height, width]$  vector. The dimensions of the checkerboard are expressed in terms of the number of squares.



**imagesUsed** — Pattern detection flag

*N*-by-1 logical vector

Pattern detection flag, returned as an *N*-by-1 logical vector of *N* logicals. The function outputs the same number of logicals as there are input images. A **true** value indicates that the pattern was detected in the corresponding image. A **false** value indicates that the function did not detect a pattern.

**pairsUsed** — Stereo pair pattern detection flag

*N*-by-1 logical vector

Stereo pair pattern detection flag, returned as an  $N$ -by-1 logical vector of  $N$  logicals. The function outputs the same number of logicals as there are input images. A `true` value indicates that the pattern is detected in the corresponding stereo image pair. A `false` value indicates that the function does not detect a pattern.

### More About

- “Single Camera Calibration App”

### References

- [1] Geiger, A., F. Moosmann, O. Car, and B. Schuster. "Automatic Camera and Range Sensor Calibration using a Single Shot," *International Conference on Robotics and Automation (ICRA)*, St. Paul, USA, May 2012.

### See Also

`cameraParameters` | `stereoParameters` | `Camera Calibrator` |  
`estimateCameraParameters` | `generateCheckerboardPoints`

**Introduced in R2014a**



# detectFASTFeatures

Detect corners using FAST algorithm and return `cornerPoints` object

## Syntax

```
points = detectFASTFeatures(I)
points = detectFASTFeatures(I,Name,Value)
PointsGPU = detectFASTFeatures(gpuI, ___)
```

## Description

`points = detectFASTFeatures(I)` returns a `cornerPoints` object, `points`. The object contains information about the feature points detected in a 2-D grayscale input image, `I`. The `detectFASTFeatures` function uses the Features from Accelerated Segment Test (FAST) algorithm to find feature points.

`points = detectFASTFeatures(I,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Supports MATLAB Function block: No

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries

“Code Generation Support, Usage Notes, and Limitations”

`PointsGPU = detectFASTFeatures(gpuI, ___)` perform operation on a graphics processing unit (GPU), where `gpuI` is a `gpuArray` object that contains a 2-D grayscale input image. The output is a `cornerPoints` object. This syntax requires the Parallel Computing Toolbox.

## Examples

### Find Corner Points in an Image Using the FAST Algorithm

Read the image.

```
I = imread('cameraman.tif');
```

### Find the corners.

```
corners = detectFASTFeatures(I);
```

### Display the results.

```
imshow(I); hold on;  
plot(corners.selectStrongest(50));
```



- “Find Corner Points Using the Eigenvalue Algorithm” on page 3-129
- “Find Corner Points Using the Harris-Stephens Algorithm” on page 3-124

## Input Arguments

### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified in 2-D grayscale. The input image must be real and nonsparse.

Example:

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

### **gpuI — Input image**

*M*-by-*N* 2-D grayscale image

Input image stored on the GPU, specified in 2-D grayscale. The input image must be real and nonsparse.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## **Name-Value Pair Arguments**

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

Example: `'MinQuality', '0.01', 'ROI', [50, 150, 100, 200]` specifies that the detector must use a 1% minimum accepted quality of corners within the designated region of interest. This region of interest is located at  $x=50$ ,  $y=150$ . The ROI has a width of 100 pixels, and a height of 200 pixels.

### **'MinQuality' — Minimum accepted quality of corners**

0.1 (default)

Minimum accepted quality of corners, specified as the comma-separated pair consisting of `'MinQuality'` and a scalar value in the range [0,1].

The minimum accepted quality of corners represents a fraction of the maximum corner metric value in the image. Larger values can be used to remove erroneous corners.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **'MinContrast' — Minimum intensity**

0.2 (default)

Minimum intensity difference between corner and surrounding region, specified as the comma-separated pair consisting of `'MinContrast'` and a scalar value in the range (0,1).

The minimum intensity represents a fraction of the maximum value of the image class. Increasing the value reduces the number of detected corners.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 'ROI' — Rectangular region

[1 1 `size(I,2)` `size(I,1)`] (default) | vector

Rectangular region for corner detection, specified as a comma-separated pair consisting of 'ROI' and a vector of the format [`x y width height`]. The first two integer values [`x y`] represent the location of the upper-left corner of the region of interest. The last two integer values represent the width and height.

Example: 'ROI', [50,150,100,200]

## Output Arguments

### points — Corner points

cornerPoints object

Corner points object, returned as a cornerPoints object. The object contains information about the feature points detected in the 2-D grayscale input image.

### PointsGPU — Corner points

cornerPoints object

Corner points object, returned as a cornerPoints object. The object contains information about the feature points detected in the 2-D grayscale input image. The underlying feature point data of this object is stored on the GPU. Use the gather method of the cornerPoints object and the PointsGPU output to copy data back to the CPU.

```
pointsCPU = gather(pointsGPU);
```

## More About

- “Point Feature Types”

## References

- [1] Rosten, E., and T. Drummond. "Fusing Points and Lines for High Performance Tracking," *Proceedings of the IEEE International Conference on Computer Vision*, Vol. 2 (October 2005): pp. 1508–1511.

## See Also

MSERRegions | SURFPoints | BRISKPoints | cornerPoints | binaryFeatures |  
detectBRISKFeatures | detectHarrisFeatures | detectMinEigenFeatures  
| detectMSERFeatures | detectSURFFeatures | extractFeatures |  
extractHOGFeatures | matchFeatures

**Introduced in R2013a**

## detectHarrisFeatures

Detect corners using Harris–Stephens algorithm and return `cornerPoints` object

### Syntax

```
points = detectHarrisFeatures(I)
points = detectHarrisFeatures(I,Name,Value)
PointsGPU = detectHarrisFeatures(gpuI, ___)
```

### Description

`points = detectHarrisFeatures(I)` returns a `cornerPoints` object, `points`. The object contains information about the feature points detected in a 2-D grayscale input image, `I`. The `detectHarrisFeatures` function uses the Harris–Stephens algorithm to find these feature points.

`points = detectHarrisFeatures(I,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

#### Code Generation Support:

Compile-time constant input: 'FilterSize'

Supports MATLAB Function block: No

Generated code for this function uses a precompiled platform-specific shared library.

“Code Generation Support, Usage Notes, and Limitations”

`PointsGPU = detectHarrisFeatures(gpuI, ___)` perform operation on a graphics processing unit (GPU), where `gpuI` is a `gpuArray` object that contains a 2-D grayscale input image, and the output is a `cornerPoints` object. This syntax requires the Parallel Computing Toolbox.

### Examples

#### Find Corner Points Using the Harris–Stephens Algorithm

Read the image.

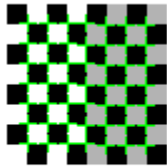
```
I = checkerboard;
```

**Find the corners.**

```
corners = detectHarrisFeatures(I);
```

**Display the results.**

```
imshow(I); hold on;  
plot(corners.selectStrongest(50));
```



- “Find Corner Points Using the Eigenvalue Algorithm” on page 3-129
- “Find Corner Points in an Image Using the FAST Algorithm” on page 3-119

## Input Arguments

**I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified in 2-D grayscale. The input image must be real and nonsparse.

Example:

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

**gpuI** — Input image

*M*-by-*N* 2-D grayscale image

Input image stored on the GPU, specified in 2-D grayscale. The input image must be real and nonsparse.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## Name-Value Pair Arguments

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

Example: `'MinQuality', '0.01', 'ROI', [50, 150, 100, 200]` specifies that the detector must use a 1% minimum accepted quality of corners within the designated region of interest. This region of interest is located at  $x=50$ ,  $y=150$ . The ROI has a width of 100 pixels and a height of 200 pixels.

### 'MinQuality' — Minimum accepted quality of corners

0.01 (default)

Minimum accepted quality of corners, specified as the comma-separated pair consisting of `'MinQuality'` and a scalar value in the range [0,1].

The minimum accepted quality of corners represents a fraction of the maximum corner metric value in the image. Larger values can be used to remove erroneous corners.

Example: `'MinQuality', 0.01`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### 'FilterSize' — Gaussian filter dimension

5 (default)

Gaussian filter dimension, specified as the comma-separated pair consisting of `'FilterSize'` and an odd integer value in the range [3, `min(size(I))`].

The Gaussian filter smooths the gradient of the input image.

The function uses the `FilterSize` value to calculate the filter's dimensions, `FilterSize-by-FilterSize`. It also defines the standard deviation of the Gaussian filter as `FilterSize/3`.

Example: `'FilterSize', 5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`



**'ROI' — Rectangular region**

[1 1 size(I,2) size(I,1)] (default) | vector

Rectangular region for corner detection, specified as a comma-separated pair consisting of 'ROI' and a vector of the format `[x y width height]`. The first two integer values `[x y]` represent the location of the upper-left corner of the region of interest. The last two integer values represent the width and height.

Example: 'ROI', [50,150,100,200]

## Output Arguments

**points — Corner points**

cornerPoints object

Corner points object, returned as a cornerPoints object. The object contains information about the feature points detected in the 2-D grayscale input image.

**PointsGPU — Corner points**

cornerPoints object

Corner points object, returned as a cornerPoints object. The object contains information about the feature points detected in the 2-D grayscale input image. The underlying feature point data of this object is stored on the GPU. Use the gather method and PointsGPU output to copy data back to the CPU.

```
pointsCPU = gather(pointsGPU);
```

## More About

- “Point Feature Types”

## References

- [1] Harris, C., and M. Stephens, "A Combined Corner and Edge Detector," *Proceedings of the 4th Alvey Vision Conference*, August 1988, pp. 147-151.

## See Also

MSERRegions | SURFPoints | BRISKPoints | cornerPoints | binaryFeatures | detectBRISKFeatures | detectFASTFeatures | detectMinEigenFeatures

| detectMSERFeatures | detectSURFFeatures | extractFeatures |  
extractHOGFeatures | matchFeatures

**Introduced in R2013a**

# detectMinEigenFeatures

Detect corners using minimum eigenvalue algorithm and return `cornerPoints` object

## Syntax

```
points = detectMinEigenFeatures(I)
points = detectMinEigenFeatures(I,Name,Value)
```

## Description

`points = detectMinEigenFeatures(I)` returns a `cornerPoints` object, `points`. The object contains information about the feature points detected in a 2-D grayscale input image, `I`. The `detectMinEigenFeatures` function uses the minimum eigenvalue algorithm developed by Shi and Tomasi to find feature points.

`points = detectMinEigenFeatures(I,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Supports Code Generation: Yes

Compile-time constant input: 'FilterSize'

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Find Corner Points Using the Eigenvalue Algorithm

**Read the image.**

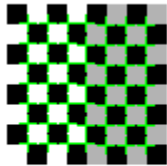
```
I = checkerboard;
```

**Find the corners.**

```
corners = detectMinEigenFeatures(I);
```

#### Display the results.

```
imshow(I); hold on;  
plot(corners.selectStrongest(50));
```



- “Find Corner Points Using the Harris-Stephens Algorithm” on page 3-124
- “Find Corner Points in an Image Using the FAST Algorithm” on page 3-119

## Input Arguments

### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified in 2-D grayscale. The input image must be real and nonsparse.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## Name-Value Pair Arguments

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

Example: `'MinQuality', '0.01', 'ROI', [50, 150, 100, 200]` specifies that the detector must use a 1% minimum accepted quality of corners within the designated region of interest. This region of interest is located at  $x=50$ ,  $y=150$ . The ROI has a width of 100 pixels, and a height of 200 pixels.

**'MinQuality' — Minimum accepted quality of corners**

0.01 (default)

Minimum accepted quality of corners, specified as the comma-separated pair consisting of 'MinQuality' and a scalar value in the range [0,1].

The minimum accepted quality of corners represents a fraction of the maximum corner metric value in the image. Larger values can be used to remove erroneous corners.

Example: 'MinQuality', 0.01

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**'FilterSize' — Gaussian filter dimension**

5 (default)

Gaussian filter dimension, specified as the comma-separated pair consisting of 'FilterSize' and an odd integer value in the range [3, inf).

The Gaussian filter smooths the gradient of the input image.

The function uses the `FilterSize` value to calculate the filter's dimensions, `FilterSize-by-FilterSize`. It also defines the standard deviation as `FilterSize/3`.

Example: 'FilterSize', 5

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**'ROI' — Rectangular region**

[1 1 size(I,2) size(I,1)] (default) | vector

Rectangular region for corner detection, specified as a comma-separated pair consisting of 'ROI' and a vector of the format `[x y width height]`. The first two integer values `[x y]` represent the location of the upper-left corner of the region of interest. The last two integer values represent the width and height.

Example: 'ROI', [50, 150, 100, 200]

## Output Arguments

**points — Corner points**

cornerPoints object

Corner points, returned as a `cornerPoints` object. The object contains information about the feature points detected in the 2-D grayscale input image.

### More About

- “Point Feature Types”

### References

[1] Shi, J., and C. Tomasi, "Good Features to Track," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 1994, pp. 593–600.

### See Also

`MSERRegions` | `SURFPoints` | `BRISKPoints` | `cornerPoints` | `binaryFeatures` | `detectBRISKFeatures` | `detectFASTFeatures` | `detectHarrisFeatures` | `detectMSERFeatures` | `detectSURFFeatures` | `extractFeatures` | `extractHOGFeatures` | `matchFeatures`

**Introduced in R2013a**

# detectMSERFeatures

Detect MSER features and return MSERRegions object

## Syntax

```
regions = detectMSERFeatures(I)
[regions,cc] = detectMSERFeatures(I)
[ ___ ] = detectMSERFeatures(I,Name,Value)
```

## Description

`regions = detectMSERFeatures(I)` returns an MSERRegions object, `regions`, containing information about MSER features detected in the 2-D grayscale input image, `I`. This object uses Maximally Stable Extremal Regions (MSER) algorithm to find regions.

`[regions,cc] = detectMSERFeatures(I)` optionally returns MSER regions in a connected component structure.

`[ ___ ] = detectMSERFeatures(I,Name,Value)` sets additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

For code generation, the function outputs `regions.PixelList` as an array. The region sizes are defined in `regions.Lengths`.

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries

“Code Generation Support, Usage Notes, and Limitations”

## Examples

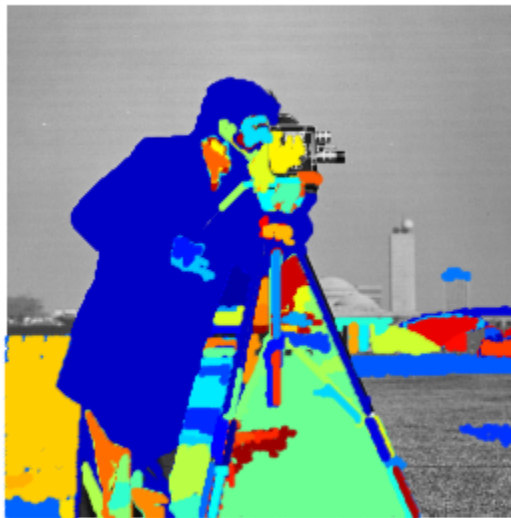
### Find MSER Regions in an Image

Read image and detect MSER regions.

```
I = imread('cameraman.tif');  
regions = detectMSERFeatures(I);
```

Visualize MSER regions which are described by pixel lists stored inside the returned 'regions' object.

```
figure; imshow(I); hold on;  
plot(regions, 'showPixelList', true, 'showEllipses', false);
```



Display ellipses and centroids fit into the regions.

```
figure; imshow(I); hold on;
```



```
plot(regions); % by default, plot displays ellipses and centroids
```



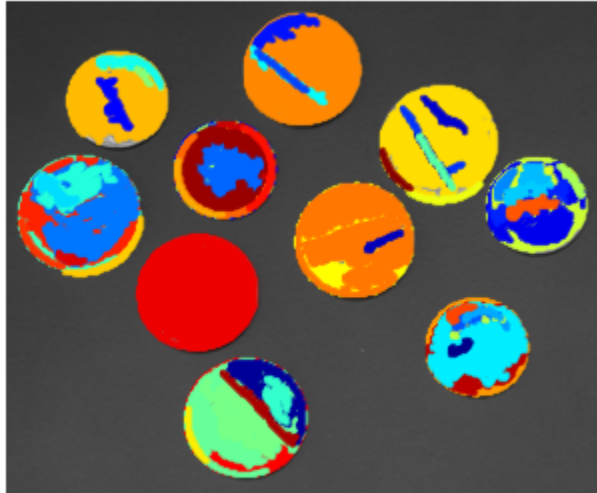
### Find circular MSER regions

Detect MSER regions.

```
I = imread('coins.png');  
[regions,mserCC] = detectMSERFeatures(I);
```

Show all detected MSER Regions.

```
figure  
imshow(I)  
hold on  
plot(regions,'showPixelList',true,'showEllipses',false)
```



Measure the MSER region eccentricity to gauge region circularity.

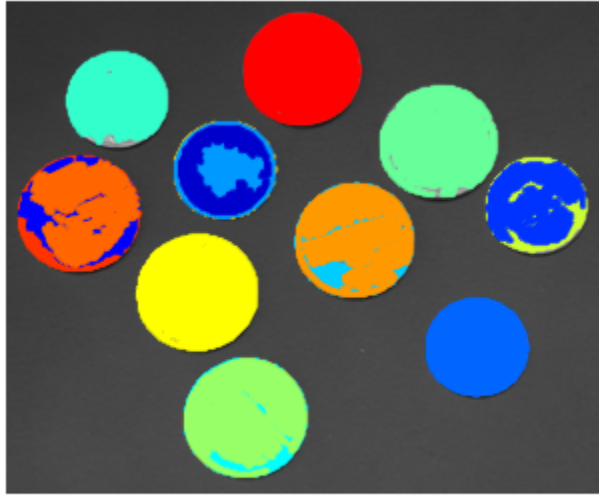
```
stats = regionprops('table',mserCC,'Eccentricity');
```

Threshold eccentricity values to only keep the circular regions. (Circular regions have low eccentricity.)

```
eccentricityIdx = stats.Eccentricity < 0.55;  
circularRegions = regions(eccentricityIdx);
```

Show the circular regions.

```
figure  
imshow(I)  
hold on  
plot(circularRegions,'showPixelList',true,'showEllipses',false)
```



- “Automatically Detect and Recognize Text in Natural Images”

## Input Arguments

### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified in grayscale. It must be real and nonsparse.

Data Types: `uint8` | `int16` | `uint16` | `single` | `double`

## Name-Value Pair Arguments

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

Example: 'RegionAreaRange',[30 14000], specifies the size of the region in pixels.

#### 'ThresholdDelta' — Step size between intensity threshold levels

2 (default) | percent numeric value

Step size between intensity threshold levels, specified as the comma-separated pair consisting of 'ThresholdDelta' and a numeric value in the range (0,100]. This value is expressed as a percentage of the input data type range used in selecting extremal regions while testing for their stability. Decrease this value to return more regions. Typical values range from 0.8 to 4.

#### 'RegionAreaRange' — Size of the region

[30 14000] (default) | two-element vector

Size of the region in pixels, specified as the comma-separated pair consisting of 'RegionAreaRange' and a two-element vector. The vector, [*minArea* *maxArea*], allows the selection of regions containing pixels to be between *minArea* and *maxArea*, inclusive.

#### 'MaxAreaVariation' — Maximum area variation between extremal regions

0.25 (default) | positive scalar

Maximum area variation between extremal regions at varying intensity thresholds, specified as the comma-separated pair consisting of 'MaxAreaVariation' and a positive scalar value. Increasing this value returns a greater number of regions, but they may be less stable. Stable regions are very similar in size over varying intensity thresholds. Typical values range from 0.1 to 1.0.

#### 'ROI' — Rectangular region of interest

[1 1 size(I,2) size(I,1)] (default) | vector

Rectangular region of interest, specified as a vector. The vector must be in the format [*x* *y* *width* *height*]. When you specify an ROI, the function detects corners within the area located at [*x* *y*] of size specified by [*width* *height*]. The [*x* *y*] elements specify the upper left corner of the region.

## Output Arguments

#### regions — MSER regions object

MSERegions object (default)

MSER regions object, returned as a MSERRegions object. The object contains information about MSER features detected in the grayscale input image.

### **cc** — Connected component structure

Connected component structure, returned as a structure with four fields. The connected component structure is useful for measuring region properties using the `regionprops` function. The four fields:

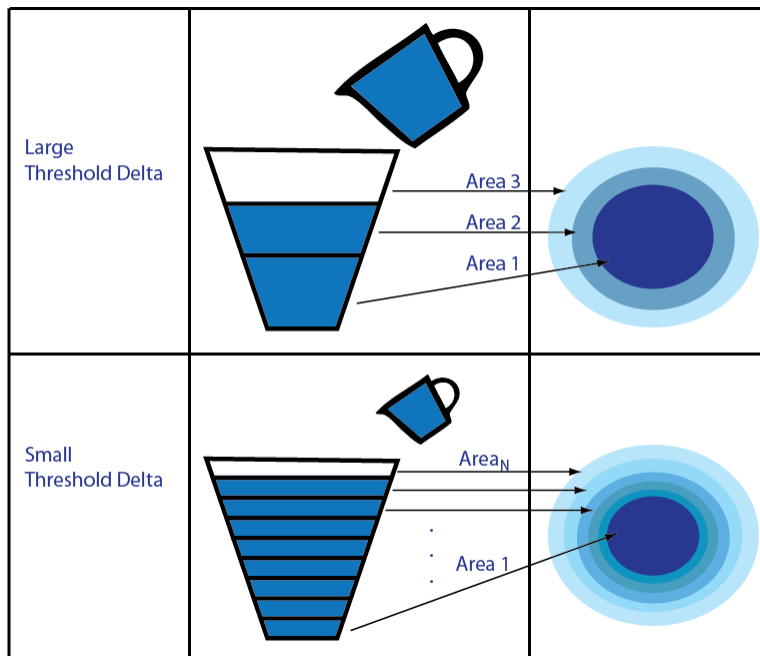
Field	Description
<code>Connectivity</code>	Connectivity of the MSER regions. Default: 8
<code>ImageSize</code>	Size of I.
<code>NumObjects</code>	Number of MSER regions in I.
<code>PixelIdxList</code>	1-by- <code>NumObjects</code> cell array containing <code>NumObjects</code> vectors. Each vector represents the linear indices of the pixels in the that element's corresponding MSER region.

## More About

### Algorithms

### Intensity Threshold Levels

The MSER detector incrementally steps through the intensity range of the input image to detect stable regions. The `???` parameter determines the number of increments the detector tests for stability. You can think of the threshold delta value as the size of a cup to fill a bucket with water. The smaller the cup, the more number of increments it takes to fill up the bucket. The bucket can be thought of as the intensity profile of the region.



The MSER object checks the variation of the region area size between different intensity thresholds. The variation must be less than the value of the  $\delta$  parameter to be considered stable.

At a high level, MSER can be explained, by thinking of the intensity profile of an image representing a series of buckets. Imagine the tops of the buckets flush with the ground, and a hose turned on at one of the buckets. As the water fills into the bucket, it overflows and the next bucket starts filling up. Smaller regions of water join and become bigger bodies of water, and finally the whole area gets filled. As water is filling up into a bucket, it is checked against the MSER stability criterion. Regions appear, grow and merge at different intensity thresholds.

- “Point Feature Types”

## References

- [1] Nister, D., and H. Stewenius, "Linear Time Maximally Stable Extremal Regions", *Lecture Notes in Computer Science*. 10th European Conference on Computer Vision, Marseille, France: 2008, no. 5303, pp. 183–196.

- [2] Matas, J., O. Chum, M. Urba, and T. Pajdla. "Robust wide baseline stereo from maximally stable extremal regions." *Proceedings of British Machine Vision Conference*, pages 384-396, 2002.
- [3] Obdrzalek D., S. Basovnik, L. Mach, and A. Mikulik. "Detecting Scene Elements Using Maximally Stable Colour Regions," *Communications in Computer and Information Science*, La Ferte-Bernard, France; 2009, vol. 82 CCIS (2010 12 01), pp 107–115.
- [4] Mikolajczyk, K., T. Tuytelaars, C. Schmid, A. Zisserman, T. Kadir, and L. Van Gool, "A Comparison of Affine Region Detectors"; *International Journal of Computer Vision*, Volume 65, Numbers 1–2 / November, 2005, pp 43–72 .

## See Also

MSERRegions | SURFPoints | BRISKPoints | cornerPoints | binaryFeatures | detectBRISKFeatures | detectFASTFeatures | detectHarrisFeatures | detectMinEigenFeatures | detectSURFFeatures | extractFeatures | extractHOGFeatures | matchFeatures

**Introduced in R2012a**

## detectPeopleACF

Detect people using aggregate channel features (ACF)

### Syntax

```
bboxes = detectPeopleACF(I)
[bboxes,scores] = detectPeopleACF(I)
[ ___ ] = detectPeopleACF(I,roi)
[ ___ ] = detectPeopleACF(Name,Value)
```

### Description

`bboxes = detectPeopleACF(I)` returns a matrix, `bboxes`, that contains the locations of detected upright people in the input image, `I`. The locations are represented as bounding boxes. The function uses the aggregate channel features (ACF) algorithm.

`[bboxes,scores] = detectPeopleACF(I)` also returns the detection scores for each bounding box.

`[ ___ ] = detectPeopleACF(I,roi)` also detects people within the rectangular search region specified by `roi`, using either of the previous syntaxes.

`[ ___ ] = detectPeopleACF(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. Unspecified properties have default values.

#### **Code Generation Support:**

Supports Code Generation: No

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### **Detect People Using Aggregated Channel Features**

Read an image.

```
I = imread('visionteam1.jpg');
```



Detect people in the image and store results as bounding boxes and score.

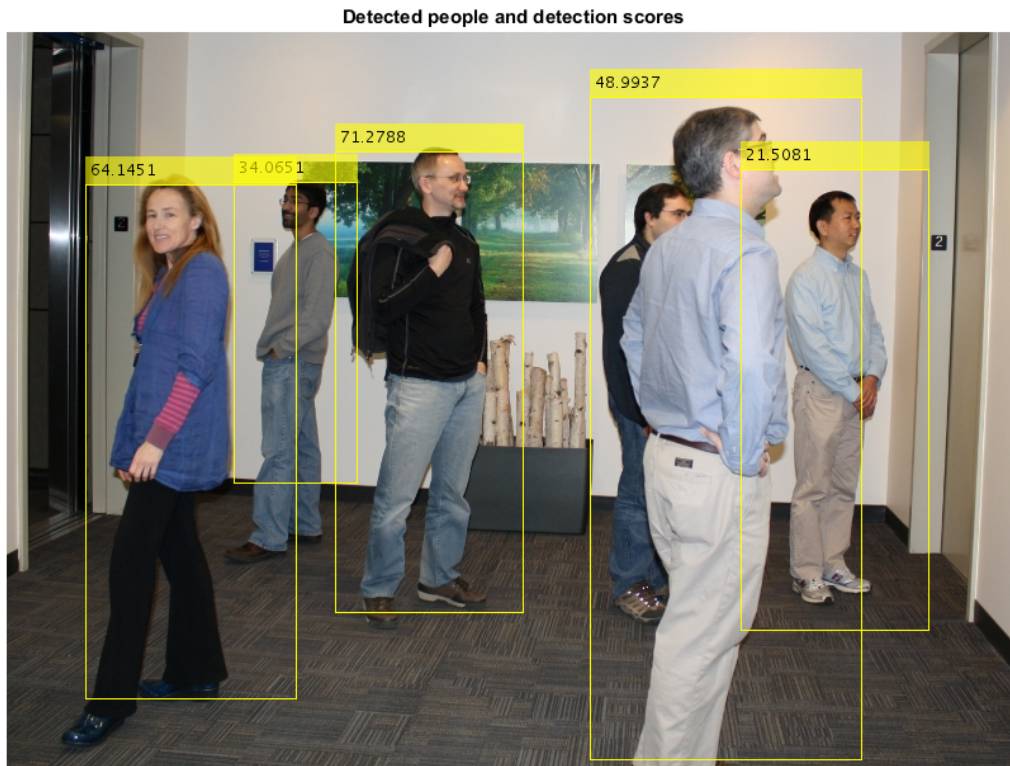
```
[bboxes,scores] = detectPeopleACF(I);
```

Annotate the detected upright people in the image.

```
I = insertObjectAnnotation(I, 'rectangle',bboxes,scores);
```

Display the results with annotation.

```
figure  
imshow(I)  
title('Detected people and detection scores')
```



- “Tracking Pedestrians from a Moving Car”

## Input Arguments

### **I** — Input image

truecolor image

Input image, specified as a truecolor image. The image must be real and nonsparse.

Data Types: `uint8` | `uint16` | `int16` | `double` | `single`

### **roi** — Rectangular search region

four-element vector

Rectangular search region, specified as a four-element vector,  $[x,y,width,height]$ . The `roi` must be fully contained in `I`.

## Name-Value Pair Arguments

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

Example: `'Threshold',-1`

### **'Model'** — ACF classification model

`'inria-100x41'` (default) | `'caltech-50x21'`

ACF classification model, specified as the comma-separated pair consisting of `'Model'` and the string `'inria-100x41'` or `'caltech-50x21'`. The `'inria-100x41'` model was trained using the INRIA Person dataset. The `'caltech-50x21'` model was trained using the Caltech Pedestrian dataset.

### **'Threshold'** — Classification threshold

`-1` (default) | numeric value

Classification threshold, specified as the comma-separated pair consisting of `'Threshold'` and a numerical value. Typical values are in the range  $[-1,2]$ . During multiscale object detection, the threshold value controls the person or nonperson classification in individual image subregions. Increase this threshold where many false detections can occur.

**'NumScaleLevels' — Number of scale levels per octave**

8 (default) | integer

Number of scale levels per octave, specified as the comma-separated pair consisting of 'NumScaleLevels', and an integer. Each octave is a power-of-two downscaling of the image. Increase this number to detect people at finer scale increments. Recommended values are in the range [4,8].

**'WindowStride' — Window stride for sliding window**

4 (default) | integer

Window stride for sliding window, specified as the comma-separated pair consisting of 'WindowStride', and an integer. Set this value to the amount you want to move the window, in the  $x$  and  $y$  directions. The sliding window scans the images for object detection. The function uses the same stride for the  $x$  and  $y$  directions.

**'SelectStrongest' — Select strongest bounding box**

true (default) | false

Select strongest bounding box, specified as the comma-separated pair consisting of 'SelectStrongest' and either `true` or `false`. The process, often referred to as nonmaximum suppression, eliminates overlapping bounding boxes based on their scores. Set this property to `true` to use the `selectStrongestBbox` function to select the strongest bounding box. Set this property to `false`, to perform a custom selection operation. Setting this property to `false` returns detected bounding boxes.

**'MinSize' — Minimum region size**two-element vector [*height width*] | [50 21] | [100 41]

Minimum region size in pixels, specified as the comma-separated pair consisting of 'MinSize', and a two-element vector [*height width*]. You can set this property to [50 21] for the 'caltech-50x21' model or [100 41] for the 'inria-100x41' model. You can reduce computation time by setting this value to the known minimum region size for detecting a person. By default, `MinSize` is set to the smallest region size possible to detect an upright person for the classification model selected.

**'MaxSize' — Maximum region size**size(I) (default) | two-element vector [*height width*]

Maximum region size in pixels, specified as the comma-separated pair consisting of 'MaxSize', and a two-element vector, [*height width*]. You can reduce computation time

by setting this value to the known region size for detecting a person. If you do not set this value, by default the function determines the height and width of the image using the size of `I`.

## Output Arguments

### **bboxes** — Locations of detected people

*M*-by-4 matrix

Locations of people detected using the aggregate channel features (ACF) algorithm, returned as an *M*-by-4 matrix. The locations are represented as bounding boxes. Each row in **bboxes** contains a four-element vector,  $[x,y,width,height]$ . This vector specifies the upper-left corner and size of a bounding box, in pixels, for a detected person.

### **scores** — Confidence value

*M*-by-1 vector

Confidence value for the detections, returned as an *M*-by-1 vector. The vector contains a positive value for each bounding box in **bboxes**. The score for each detection is the output of a soft-cascade classifier. The range of score values is  $[-\text{inf} \text{ } -\text{inf}]$ . Greater scores indicate a higher confidence in the detection.

## More About

- “Point Feature Types”

## References

- [1] Dollar, P., R. Appel, S. Belongie, and P. Perona. "Fast feature pyramids for object detection." *Pattern Analysis and Machine Intelligence, IEEE Transactions*. Vol. 36, Issue 8, 2014, pp. 1532–1545.
- [2] Dollar, C. Wojek, B. Shiele, and P. Perona. "Pedestrian detection: An evaluation of the state of the art." *Pattern Analysis and Machine Intelligence, IEEE Transactions*. Vol. 34, Issue 4, 2012, pp. 743–761.
- [3] Dollar, C., Wojek, B. Shiele, and P. Perona. "Pedestrian detection: A benchmark." *IEEE Conference on Computer Vision and Pattern Recognition*. 2009.

## **See Also**

`vision.PeopleDetector` | `vision.CascadeObjectDetector` | `selectStrongestBbox`

**Introduced in R2016a**

## detectSURFFeatures

Detect SURF features and return SURFPoints object

### Syntax

```
points = detectSURFFeatures(I)  
points = detectSURFFeatures(I,Name,Value)
```

### Description

`points = detectSURFFeatures(I)` returns a SURFPoints object, `points`, containing information about SURF features detected in the 2-D grayscale input image `I`. The `detectSURFFeatures` function implements the Speeded-Up Robust Features (SURF) algorithm to find blob features.

`points = detectSURFFeatures(I,Name,Value)` Additional control for the algorithm requires specification of parameters and corresponding values. An additional option is specified by one or more `Name,Value` pair arguments.

#### Code Generation Support:

Supports MATLAB Function block: No

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries.

“Code Generation Support, Usage Notes, and Limitations”

### Input Arguments

#### **I** — Input image

*M*-by-*N* 2-D grayscale image

Input image, specified as an *M*-by-*N* 2-D grayscale. The input image must be a real nonsparse value.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## 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`.

### 'MetricThreshold' — Strongest feature threshold

1000.0 (default) | non-negative scalar

Strongest feature threshold, specified as the comma-separated pair consisting of 'MetricThreshold' and a non-negative scalar. To return more blobs, decrease the value of this threshold.

### 'NumOctaves' — Number of octaves

3 (default) | scalar, greater than or equal to 1

Number of octaves to implement, specified as the comma-separated pair consisting of 'NumOctaves' and an integer scalar, greater than or equal to 1. Increase this value to detect larger blobs. Recommended values are between 1 and 4.

Each octave spans a number of scales that are analyzed using varying size filters:

Octave	Filter sizes
1	9-by-9, 15-by-15, 21-by-21, 27-by-27, ...
2	15-by-15, 27-by-27, 39-by-39, 51-by-51, ...
3	27-by-27, 51-by-51, 75-by-75, 99-by-99, ...
4	....

Higher octaves use larger filters and subsample the image data. Larger number of octaves will result in finding larger size blobs. Set the `NumOctaves` parameter appropriately for the image size. For example, a 50-by-50 image should not require you to set the `NumOctaves` parameter, greater than 2. The `NumScaleLevels` parameter controls the number of filters used per octave. At least three levels are required to analyze the data in a single octave.

### 'NumScaleLevels' — Number of scale levels per octave

4 (default) | integer scalar, greater than or equal to 3

Number of scale levels per octave to compute, specified as the comma-separated pair consisting of 'NumScaleLevels' and an integer scalar, greater than or equal to 3. Increase this number to detect more blobs at finer scale increments. Recommended values are between 3 and 6.

### 'ROI' — Rectangular region of interest

[1 1 size(I,2) size(I,1)] (default) | vector

Rectangular region of interest, specified as a vector. The vector must be in the format [*x y width height*]. When you specify an ROI, the function detects corners within the area located at [*x y*] of size specified by [*width height*]. The [*x y*] elements specify the upper left corner of the region.

## Output Arguments

### points — SURF features

SURFPoints object

SURF features, returned as a SURFPoints object. This object contains information about SURF features detected in a grayscale image.

## Examples

### Detect SURF Interest Points in a Grayscale Image

Read image and detect interest points.

```
I = imread('cameraman.tif');  
points = detectSURFFeatures(I);
```

Display locations of interest in image.

```
imshow(I); hold on;  
plot(points.selectStrongest(10));
```





- “Detect MSER Features in an Image” on page 2-151
- “Combine MSER Region Detector with SURF Descriptors” on page 2-153

## More About

- “Point Feature Types”

## References

- [1] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. “SURF:Speeded Up Robust Features.” *Computer Vision and Image Understanding (CVIU)*.Vol. 110, No. 3, pp. 346–359, 2008.

### **See Also**

MSERRegions | SURFPoints | BRISKPoints | cornerPoints | binaryFeatures | detectBRISKFeatures | detectFASTFeatures | detectHarrisFeatures | detectMinEigenFeatures | detectMSERFeatures | extractFeatures | extractHOGFeatures | matchFeatures

**Introduced in R2011b**

# disparity

Disparity map between stereo images

## Syntax

```
disparityMap = disparity(I1,I2)
d = disparity(I1,I2,Name,Value)
```

## Description

`disparityMap = disparity(I1,I2)` returns the disparity map, `disparityMap`, for a pair of stereo images, `I1` and `I2`.

`d = disparity(I1,I2,Name,Value)` Additional control for the disparity algorithm requires specification of parameters and corresponding values. One or more `Name, Value` pair arguments specifies an additional option.

### Code Generation Support:

Compile-time constant input: Method

Supports MATLAB Function block: No

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Compute the Disparity Map for a Pair of Stereo Images

Load the images and convert them to grayscale.

```
I1 = imread('scene_left.png');
I2 = imread('scene_right.png');
```

Show stereo anaglyph. Use red-cyan stereo glasses to view image in 3-D.

```
figure
imshow(stereoAnaglyph(I1,I2));
title('Red-cyan composite view of the stereo images');
```

**Red-cyan composite view of the stereo images**

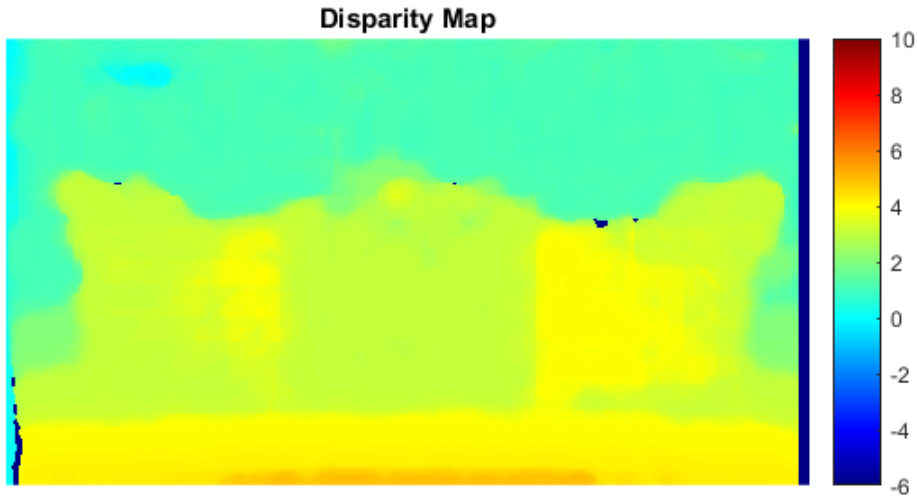


Compute the disparity map.

```
disparityRange = [-6 10];
disparityMap = disparity(rgb2gray(I1),rgb2gray(I2), 'BlockSize',15, 'DisparityRange',disparityRange);
```

**Display the disparity map. For better visualization, use the disparity range as the display range for imshow.**

```
figure
imshow(disparityMap, disparityRange);
title('Disparity Map');
colormap jet
colorbar
```



- “Depth Estimation From Stereo Video”
- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Uncalibrated Stereo Image Rectification”

## Input Arguments

### **I1** — Input image 1

*M*-by-*N* 2-D grayscale image

Input image referenced as **I1** corresponding to camera 1, specified in 2-D grayscale. The stereo images, **I1** and **I2**, must be rectified such that the corresponding points are located on the same rows. You can perform this rectification with the `rectifyStereoImages` function.

You can improve the speed of the function by setting the class of **I1** and **I2** to `uint8`, and the number of columns to be divisible by 4. Input images **I1** and **I2** must be real, finite, and nonsparse. They must be the same class.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

#### **I2** — Input image 2

*M*-by-*N* 2-D grayscale image

Input image referenced as **I2** corresponding to camera 2, specified in 2-D grayscale. The input images must be rectified such that the corresponding points are located on the same rows. You can improve the speed of the function by setting the class of **I1** and **I2** to `uint8`, and the number of columns to be divisible by 4. Input images **I1** and **I2** must be real, finite, and nonsparse. They must be the same class.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

## Name-Value Pair Arguments

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

Example: `'Method','BlockMatching'`, specifies the `'Method'` property be set to `'BlockMatching'`.

#### **'Method'** — Disparity estimation algorithm

`'SemiGlobal'` (default) | `'BlockMatching'`

Disparity estimation algorithm, specified as the comma-separated pair consisting of `'Method'` and the string `'BlockMatching'` or `'SemiGlobal'`. The disparity function implements the basic Block Matching[1] and the Semi-Global Block Matching[3] algorithms. In the `'BlockMatching'` method, the function computes disparity by comparing the sum of absolute differences (SAD) of each block of pixels in the image. In the `'SemiGlobal'` matching method, the function additionally forces similar disparity on neighboring blocks. This additional constraint results in a more complete disparity estimate than in the `'BlockMatching'` method.

The algorithms perform these steps:

- 1 Compute a measure of contrast of the image by using the Sobel filter.

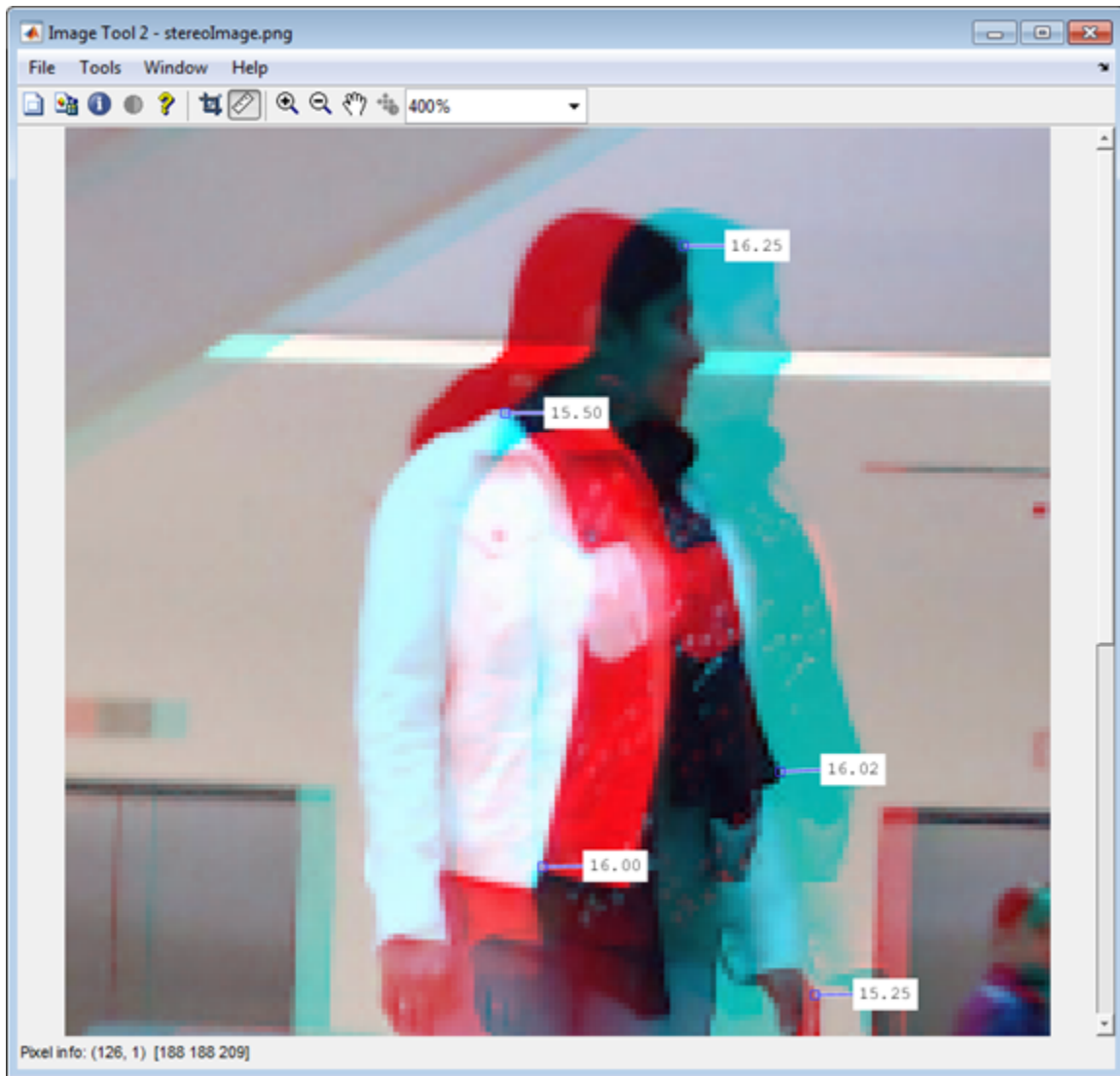
- 2 Compute the disparity for each pixel in `I1`.
- 3 Mark elements of the disparity map, `disparityMap`, that were not computed reliably. The function uses `-realmax('single')` to mark these elements.

### 'DisparityRange' — Range of disparity

[0 64] (default) | two-element vector

Range of disparity, specified as the comma-separated pair consisting of 'DisparityRange' and a two-element vector. The two-element vector must be in the format [*MinDisparity*, *MaxDisparity*]. Both elements must be an integer. The difference between *MaxDisparity* and *MinDisparity* must be divisible by 16. **DisparityRange** must be real, finite, and nonsparse. If the camera used to take `I1` was to the right of the camera used to take `I2`, then *MinDisparity* must be negative.

The disparity range depends on the distance between the two cameras and the distance between the cameras and the object of interest. Increase the **DisparityRange** when the cameras are far apart or the objects are close to the cameras. To determine a reasonable disparity for your configuration, display the stereo anaglyph of the input images in `imtool` and use the Distance tool to measure distances between pairs of corresponding points. Modify the *MaxDisparity* to correspond to the measurement.



**'BlockSize' — Square block size**  
15 (default) | odd integer



Square block size, specified as the comma-separated pair consisting of 'BlockSize' and an odd integer in the range [5,255]. This value sets the width for the square block size. The function uses the square block of pixels for comparisons between I1 and I2. BlockSize must be real, finite, and nonsparse.

**'ContrastThreshold' — Contrast threshold range**

0.5 (default) | scalar value

Contrast threshold range, specified as the comma-separated pair consisting of 'ContrastThreshold' and a scalar value in the range (0,1]. The contrast threshold defines an acceptable range of contrast values. Increasing this parameter results in fewer pixels being marked as unreliable. ContrastThreshold must be real, finite, and nonsparse.

**'UniquenessThreshold' — Minimum value of uniqueness**

15 (default) | non-negative integer

Minimum value of uniqueness, specified as the comma-separated pair consisting of 'UniquenessThreshold' and a nonnegative integer. Increasing this parameter results in the function marking more pixels unreliable. When the uniqueness value for a pixel is low, the disparity computed for it is less reliable. Setting the threshold to 0 disables uniqueness thresholding. UniquenessThreshold must be real, finite, and nonsparse.

The function defines uniqueness as a ratio of the optimal disparity estimation and the less optimal disparity estimation. For example:

Let  $K$  be the best estimated disparity, and let  $V$  be the corresponding SAD (Sum of Absolute Difference) value.

Consider  $V$  as the smallest SAD value over the whole disparity range, and  $v$  as the smallest SAD value over the whole disparity range, excluding  $K$ ,  $K-1$ , and  $K+1$ .

If  $v < V * (1 + 0.01 * \text{UniquenessThreshold})$ , then the function marks the disparity for the pixel as unreliable.

**'DistanceThreshold' — Maximum distance for left-to-right image checking**

[ ] (disabled) (default) | non-negative integer

Maximum distance for left-to-right image checking between two points, specified as the comma-separated pair consisting of 'DistanceThreshold' and a nonnegative integer. Increasing this parameter results in fewer pixels being marked as unreliable. Conversely, when you decrease the value of the distance threshold, you increase the reliability of the disparity map. You can set this parameter to an empty matrix [ ] to disable it. DistanceThreshold must be real, finite, and nonsparse.

The distance threshold specifies the maximum distance between a point in  $I_1$  and the same point found from  $I_2$ . The function finds the distance and marks the pixel in the following way:

Let  $p_1$  be a point in image  $I_1$ .

Step 1: The function searches for point  $p_1$ 's best match in image  $I_2$  (left-to-right check) and finds point  $p_2$ .

Step 2: The function searches for  $p_2$ 's best match in image  $I_1$  (right-to-left check) and finds point  $p_3$ .

If the search returns a distance between  $p_1$  and  $p_3$  greater than `DistanceThreshold`, the function marks the disparity for the point  $p_1$  as unreliable.

#### 'TextureThreshold' — Minimum texture threshold

0.0002 (default) | scalar value

Minimum texture threshold, specified as the comma-separated pair consisting of 'TextureThreshold' and a scalar value in the range [0, 1]. The texture threshold defines the minimum texture value for a pixel to be reliable. The lower the texture for a block of pixels, the less reliable the computed disparity is for the pixels. Increasing this parameter results in more pixels being marked as unreliable. You can set this parameter to 0 to disable it. This parameter applies only when you set `Method` to 'BlockMatching'.

The texture of a pixel is defined as the sum of the saturated contrast computed over the `BlockSize`-by-`BlockSize` window around the pixel. The function considers the disparity computed for the pixel unreliable and marks it, when the texture falls below the value defined by:

$$\text{Texture} < X * \text{TextureThreshold} * \text{BlockSize}^2$$

$X$  represents the maximum value supported by the class of the input images,  $I_1$  and  $I_2$ .

`TextureThreshold` must be real, finite, and nonsparse.

## Output Arguments

#### `disparityMap` — Disparity map

$M$ -by- $N$  2-D grayscale image

Disparity map for a pair of stereo images, returned as an  $M$ -by- $N$  2-D grayscale image. The function returns the disparity map with the same size as the input images,  $I_1$  and

I2. Each element of the output specifies the disparity for the corresponding pixel in the image references as I1. The returned disparity values are rounded to  $\frac{1}{16}$ th pixel.

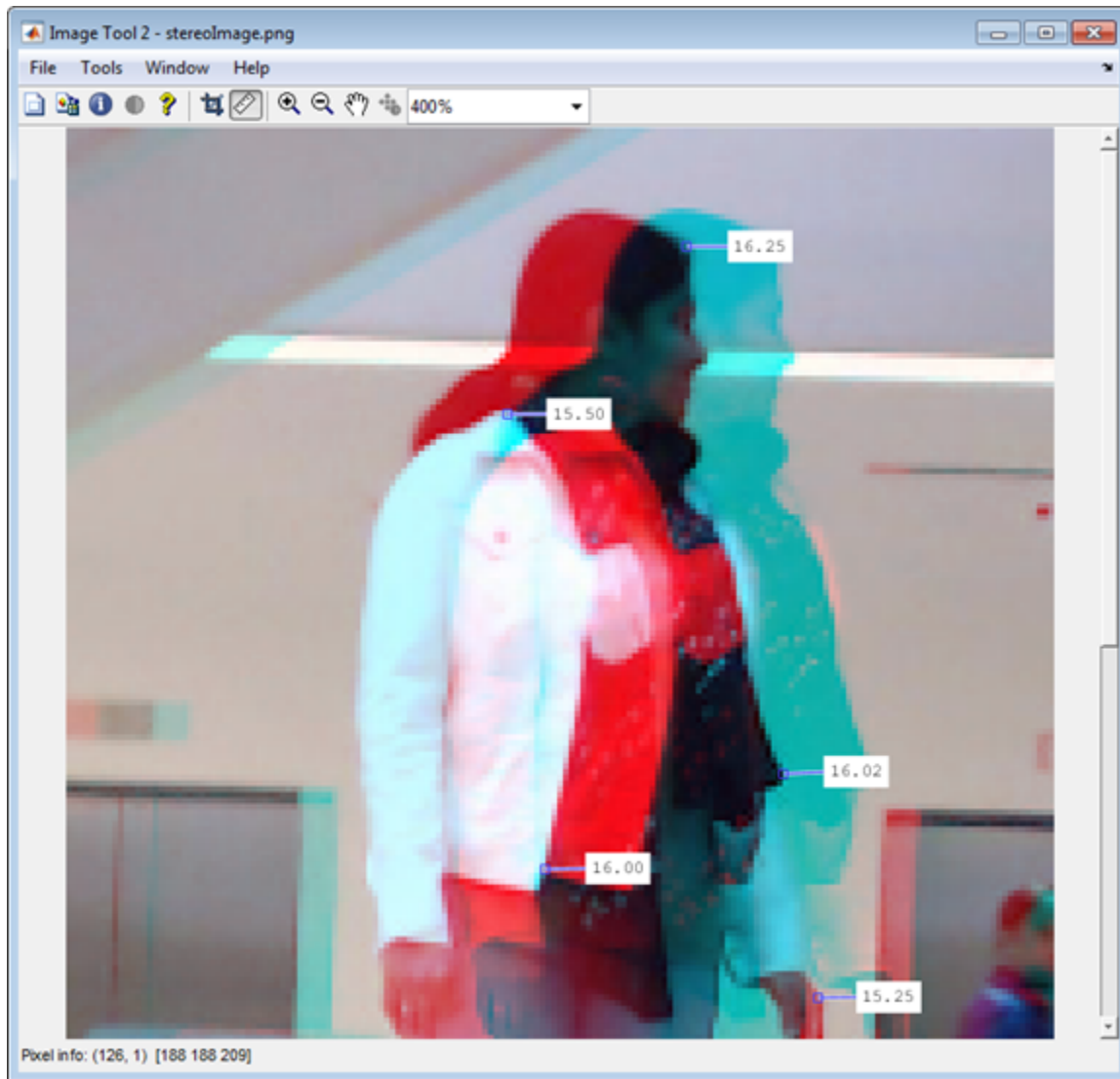
The function computes the disparity map in three steps:

- 1 Compute a measure of contrast of the image by using the Sobel filter.
- 2 Compute the disparity for each of the pixels by using block matching and the sum of absolute differences (SAD).
- 3 Optionally, mark the pixels which contain unreliable disparity values. The function sets the pixel to the value returned by `-realmax('single')`.

## More About

### Tips

If your resulting disparity map looks noisy, try modifying the `DisparityRange`. The disparity range depends on the distance between the two cameras and the distance between the cameras and the object of interest. Increase the `DisparityRange` when the cameras are far apart or the objects are close to the cameras. To determine a reasonable disparity for your configuration, display the stereo anaglyph of the input images in `imtool` and use the Distance tool to measure distances between pairs of corresponding points. Modify the *MaxDisparity* to correspond to the measurement.



## References

- [1] Konolige, K., *Small Vision Systems: Hardware and Implementation*, Proceedings of the 8th International Symposium in Robotic Research, pages 203-212, 1997.

- [2] Bradski, G. and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly, Sebastopol, CA, 2008.
- [3] Hirschmuller, H., *Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information*, International Conference on Computer Vision and Pattern Recognition, 2005.

### **See Also**

[estimateCameraParameters](#) | [estimateUncalibratedRectification](#) | [reconstructScene](#) | [rectifyStereoImages](#) | [Stereo Camera Calibrator](#)

**Introduced in R2011b**

## epipolarLine

Compute epipolar lines for stereo images

### Syntax

```
lines = epipolarLine(F,points)  
lines = epipolarLine(F',points)
```

### Description

`lines = epipolarLine(F,points)` returns an  $M$ -by-3 matrix, `lines`. The matrix represents the computed epipolar lines in the second image corresponding to the points, `points`, in the first image. The input `F` represents the fundamental matrix that maps points in the first image to the epipolar lines in the second image.

`lines = epipolarLine(F',points)` returns an  $M$ -by-3 matrix `lines`. The matrix represents the computed epipolar lines of the first image corresponding to the points, `points`, in the second image.

#### Code Generation Support:

Compile-time constant input: No restrictions

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

### Input Arguments

#### **F**

A 3-by-3 fundamental matrix. `F` must be double or single. If  $P1$  represents a point in the first image  $I1$  that corresponds to  $P2$ , a point in the second image  $I2$ , then:

$$[P_2,1] * F * [P_1,1]' = 0$$

In computer vision, the fundamental matrix is a 3-by-3 matrix which relates corresponding points in stereo images. When two cameras view a 3-D scene from two

distinct positions, there are a number of geometric relations between the 3-D points and their projections onto the 2-D images that lead to constraints between the image points. Two images of the same scene are related by epipolar geometry.

### points

An  $M$ -by-2 matrix, where each row contains the x and y coordinates of a point in the image.  $M$  represents the number of points.

points must be a double, single, or integer value.

## Output Arguments

### lines

An  $M$ -by-3 matrix, where each row must be in the format,  $[A,B,C]$ . This corresponds to the definition of the line:

$$A * x + B * y + C = 0.$$

$M$  represents the number of lines.

## Compute Fundamental Matrix

This example shows you how to compute the fundamental matrix. It uses the least median of squares method to find the inliers.

The points, `matched_points1` and `matched_points2`, have been putatively matched.

```
load stereoPointPairs
[fLMedS, inliers] = estimateFundamentalMatrix(matchedPoints1, matchedPoints2, 'Num'
```

Show the inliers in the first image.

```
I1 = imread('virectification_deskLeft.png');
figure;
subplot(121); imshow(I1);
title('Inliers and Epipolar Lines in First Image'); hold on;
plot(matchedPoints1(inliers,1), matchedPoints1(inliers,2), 'go')
```

#### Inliers and Epipolar Lines in First Image



**Compute the epipolar lines in the first image.**

```
epiLines = epipolarLine(fLMedS', matchedPoints2(inliers, :));
```

**Compute the intersection points of the lines and the image border.**

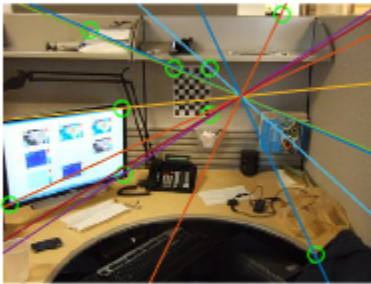
```
points = lineToBorderPoints(epiLines, size(I1));
```

**Show the epipolar lines in the first image**

```
line(points(:, [1,3])', points(:, [2,4])');
```



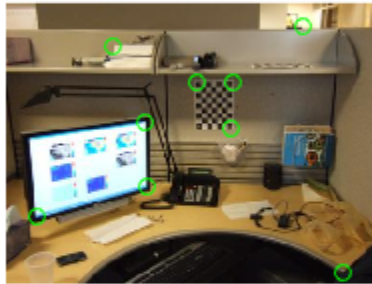
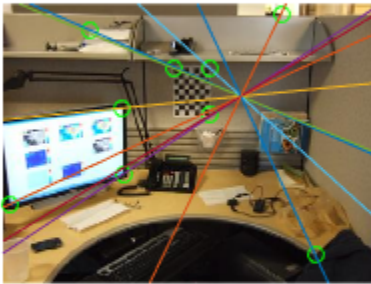
### Inliers and Epipolar Lines in First Image



Show the inliers in the second image.

```
I2 = imread('virectification_deskRight.png');  
subplot(122); imshow(I2);  
title('Inliers and Epipolar Lines in Second Image'); hold on;  
plot(matchedPoints2(inliers,1), matchedPoints2(inliers,2), 'go')
```

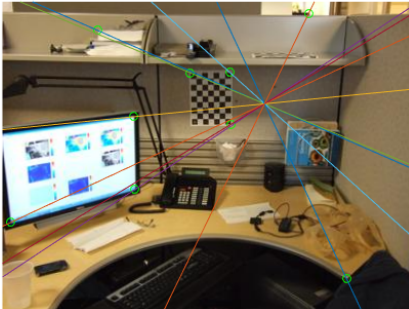
#### Inliers and Epipolar Lines in First Image and Epipolar Lines in Second Image



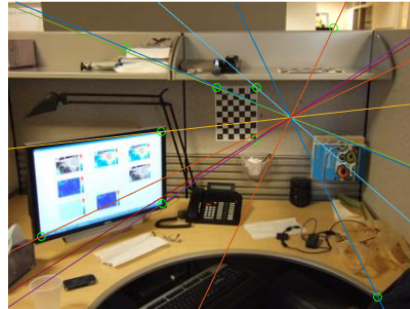
**Compute and show the epipolar lines in the second image.**

```
epiLines = epipolarLine(fLMedS, matchedPoints1(inliers, :));  
points = lineToBorderPoints(epiLines, size(I2));  
line(points(:, [1,3])', points(:, [2,4])');  
  
truesize;
```

Inliers and Epipolar Lines in First Image



Inliers and Epipolar Lines in Second Image



## See Also

`vision.ShapeInserter` | `size` | `line` | `isEpipoleInImage` | `lineToBorderPoints` | `estimateFundamentalMatrix`

Introduced in R2011a

## estimateCameraParameters

Calibrate a single or stereo camera

### Syntax

```
[cameraParams, imagesUsed, estimationErrors] =  
estimateCameraParameters(imagePoints, worldPoints)
```

```
[stereoParams, pairsUsed, estimationErrors] =  
estimateCameraParameters(imagePoints, worldPoints)
```

```
cameraParams = estimateCameraParameters( ____, Name, Value)
```

### Description

[cameraParams, imagesUsed, estimationErrors] = estimateCameraParameters(imagePoints, worldPoints) returns cameraParams, a cameraParameters object containing estimates for the intrinsic and extrinsic parameters and the distortion coefficients of a single camera. The function also returns the images you used to estimate the camera parameters and the standard estimation errors for the single camera calibration.

[stereoParams, pairsUsed, estimationErrors] = estimateCameraParameters(imagePoints, worldPoints) returns stereoParams, a stereoParameters object containing the parameters of the stereo camera. The function also returns the images you used to estimate the stereo parameters and the standard estimation errors for the stereo camera calibration.

cameraParams = estimateCameraParameters( \_\_\_\_, Name, Value) configures the cameraParams object properties specified by one or more Name, Value pair arguments, using any of the preceding syntaxes. Unspecified properties have their default values.

### Examples

#### Single Camera Calibration

Create a set of calibration images.

```
images = imageSet(fullfile(toolboxdir('vision'),'visiondata','calibration','fishEye'));  
imageFileNames = images.ImageLocation;
```

Detect the calibration pattern.

```
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate the world coordinates of the corners of the squares.

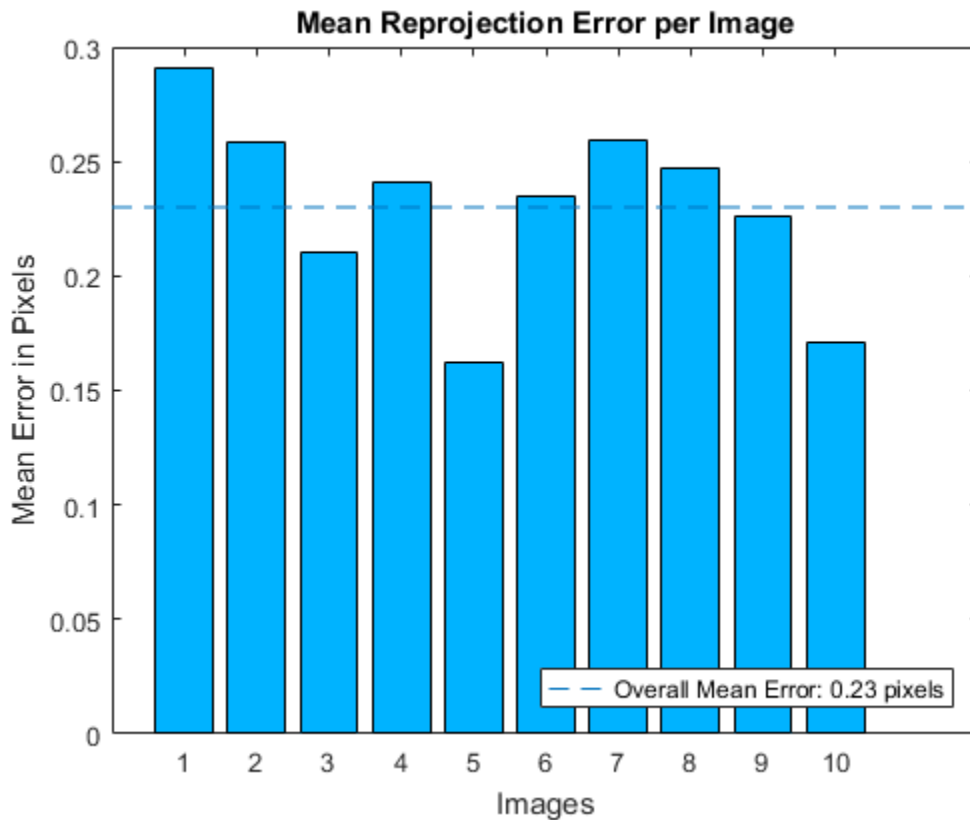
```
squareSizeInMM = 29;  
worldPoints = generateCheckerboardPoints(boardSize,squareSizeInMM);
```

Calibrate the camera.

```
params = estimateCameraParameters(imagePoints,worldPoints);
```

Visualize the calibration accuracy.

```
showReprojectionErrors(params);
```



Plot detected and reprojected points.

```
figure;  
imshow(imageFileNames{1});  
hold on;  
plot(imagePoints(:,1,1), imagePoints(:,2,1), 'go');  
plot(params.ReprojectedPoints(:,1,1), params.ReprojectedPoints(:,2,1), 'r+');  
legend('Detected Points', 'ReprojectedPoints');  
hold off;
```



## Stereo Camera Calibration

Specify calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');  
leftImages = imageSet(fullfile(imageDir,'left'));  
rightImages = imageSet(fullfile(imageDir,'right'));  
images1 = leftImages.ImageLocation;  
images2 = rightImages.ImageLocation;
```

Detect the checkerboards.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images1,images2);
```

Specify the world coordinates of the checkerboard keypoints.

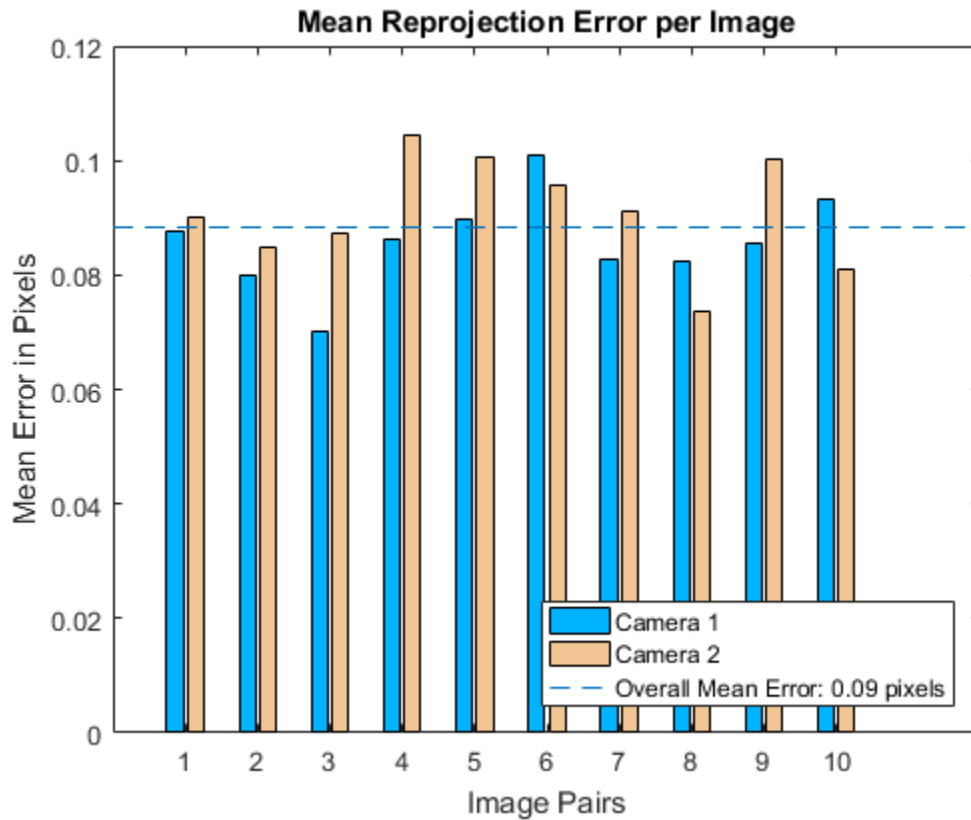
```
squareSizeInMM = 108;  
worldPoints = generateCheckerboardPoints(boardSize,squareSizeInMM);
```

Calibrate the stereo camera system.

```
im = read(leftImages,1);  
params = estimateCameraParameters(imagePoints,worldPoints);
```

Visualize the calibration accuracy.

```
showReprojectionErrors(params);
```



- “Evaluating the Accuracy of Single Camera Calibration”



## Input Arguments

### **imagePoints** — Key points of calibration pattern

*M*-by-2-by-*numImages* | *M*-by-2-by-*numPairs*-by-*numCameras* array

Key points of calibration pattern, specified as an array of  $[x,y]$  intrinsic image coordinates.

Calibration	Input Array of $[x,y]$ Key Points
Single Camera	<p><i>M</i>-by-2-by-<i>numImages</i> array of <math>[x,y]</math> points.</p> <ul style="list-style-type: none"> <li>• The number of images, <i>numImages</i>, must be greater than 2.</li> <li>• The number of keypoint coordinates in each pattern, <i>M</i>, must be greater than 3.</li> </ul>
Stereo Camera	<p><i>M</i>-by-2-by-<i>numPairs</i>-by-<i>numCameras</i> array of <math>[x,y]</math> points.</p> <ul style="list-style-type: none"> <li>• <i>numPairs</i> is the number of stereo image pairs containing the calibration pattern.</li> <li>• The number of keypoint coordinates in each pattern, <i>M</i>, must be greater than 3.</li> <li>• <code>imagePoints(:, :, :, 1)</code> are the points from camera 1.</li> <li>• <code>imagePoints(:, :, :, 2)</code> are the points from camera 2.</li> </ul>

Data Types: `single` | `double`

### **worldPoints** — Key points of calibration pattern in world coordinates

*M*-by-2 array

Key points of calibration pattern in world coordinates, specified as an *M*-by-2 array of *M* number of  $[x,y]$  world coordinates. The pattern must be planar; therefore, *z*-coordinates are zero.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: 'WorldUnits', 'mm' sets the world point units to millimeters.

#### 'WorldUnits' — World points units

'mm' (default) | string

World points units, specified as the comma-separated pair consisting of 'WorldUnits' and a string representing units.

#### 'EstimateSkew' — Estimate skew

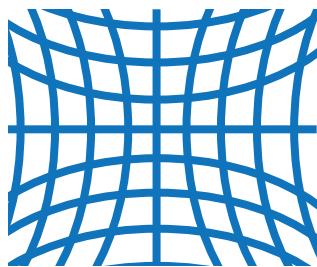
false (default) | logical scalar

Estimate skew, specified as the comma-separated pair consisting of 'EstimateSkew' and a logical scalar. When you set this property to true, the function estimates the image axes skew. When set to false, the image axes are exactly perpendicular and the function sets the skew to zero.

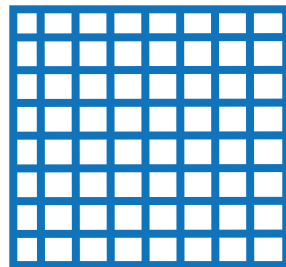
#### 'NumRadialDistortionCoefficients' — Number of radial distortion coefficients

2 (default) | 3

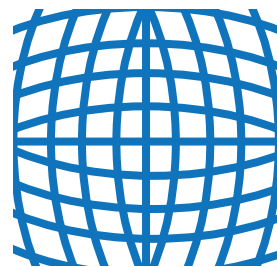
Number of radial distortion coefficients to estimate, specified as the comma-separated pair consisting of 'NumRadialDistortionCoefficients' and the value 2 or 3. Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.



Negative radial distortion  
"pincushion"



No distortion



Positive radial distortion  
"barrel"

The radial distortion coefficients model this type of distortion. The distorted points are denoted as  $(x_{\text{distorted}}, y_{\text{distorted}})$ :

$$x_{\text{distorted}} = x(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

- $x, y$  — Undistorted pixel locations.  $x$  and  $y$  are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus,  $x$  and  $y$  are dimensionless.
- $k_1, k_2,$  and  $k_3$  — Radial distortion coefficients of the lens.
- $r^2: x^2 + y^2$

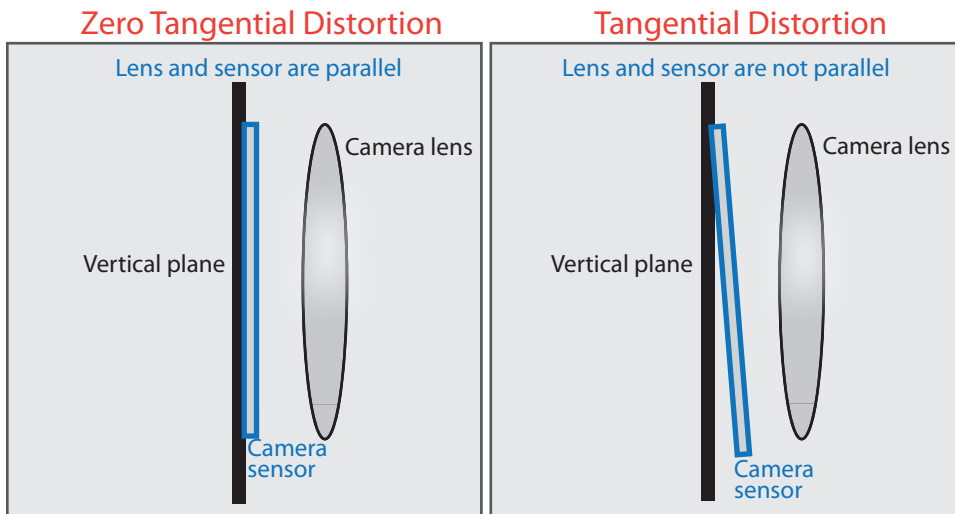
Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select 3 coefficients to include  $k_3$ .

### 'EstimateTangentialDistortion' — Tangential distortion flag

false (default) | logical scalar

Tangential distortion flag, specified as the comma-separated pair consisting of 'EstimateTangentialDistortion' and a logical scalar. When you set this property to **true**, the function estimates the tangential distortion. When you set it to **false**, the tangential distortion is negligible.

Tangential distortion occurs when the lens and the image plane are not parallel. The tangential distortion coefficients model this type of distortion.



The distorted points are denoted as  $(x_{\text{distorted}}, y_{\text{distorted}})$ :

$$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y]$$

- $x, y$  — Undistorted pixel locations.  $x$  and  $y$  are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus,  $x$  and  $y$  are dimensionless.
- $p_1$  and  $p_2$  — Tangential distortion coefficients of the lens.
- $r^2 = x^2 + y^2$

#### 'InitialIntrinsicMatrix' — Initial guess for camera intrinsics

[ ] (default) | 3-by-3 matrix

Initial guess for camera intrinsics, specified as the comma-separated pair consisting of 'InitialIntrinsicMatrix' and a 3-by-3 matrix. If you do not provide an initial value, the function computes the initial intrinsic matrix using linear least squares.

#### 'InitialRadialDistortion' — Initial guess for radial distortion coefficients

[ ] (default) | 2-element vector | 3-element vector

Initial guess for radial distortion coefficients, specified as the comma-separated pair consisting of 'InitialRadialDistortion' and a 2- or 3-element vector. If you do not provide an initial value, the function uses 0 as the initial value for all the coefficients.

## Output Arguments

#### cameraParams — Camera parameters

cameraParameters object

Camera parameters, returned as a cameraParameters object.

#### imagesUsed — Images used to estimate camera parameters

$P$ -by-1 logical array

Images you use to estimate camera parameters, returned as a  $P$ -by-1 logical array.  $P$  corresponds to the number of images. The array indicates which images you used to estimate the camera parameters. A logical true value in the array indicates which images you used to estimate the camera parameters.

The function computes a homography between the world points and the points detected in each image. If the homography computation fails for an image, the function issues a warning. The points for that image are not used for estimating the camera parameters. The function also sets the corresponding element of `imagesUsed` to `false`.

### **estimationErrors** — Standard errors of estimated parameters

`cameraCalibrationErrors` object | `stereoCalibrationErrors` object

Standard errors of estimated parameters, returned as a `cameraCalibrationErrors` object or a `stereoCalibrationErrors` object.

### **stereoParams** — Camera parameters for stereo system

`stereoParameters` object

Camera parameters for stereo system, returned as a `stereoParameters` object. The object contains the intrinsic, extrinsic, and lens distortion parameters of the stereo camera system.

### **pairsUsed** — Image pairs used to estimate camera parameters

$P$ -by-1 logical array

Image pairs used to estimate camera parameters, returned as a  $P$ -by-1 logical array.  $P$  corresponds to the number of image pairs. A logical `true` value in the array indicates which image pairs you used to estimate the camera parameters.

## More About

### Algorithms

### Calibration Algorithm

You can use the Camera Calibrator app with cameras up to a field of view (FOV) of 95 degrees.

The calibration algorithm assumes a pinhole camera model:

$$w \begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \begin{bmatrix} R \\ t \end{bmatrix} K$$

$(X, Y, Z)$ : world coordinates of a point

$(x,y)$ : coordinates of the corresponding image point

$w$ : arbitrary scale factor

$K$ : camera intrinsic matrix

$R$ : matrix representing the 3-D rotation of the camera

$t$ : translation of the camera relative to the world coordinate system

Camera calibration estimates the values of the intrinsic parameters, the extrinsic parameters, and the distortion coefficients. There are two steps involved in camera calibration:

- 1 Solve for the intrinsics and extrinsics in closed form, assuming that lens distortion is zero. [1]
- 2 Estimate all parameters simultaneously including the distortion coefficients using nonlinear least-squares minimization (Levenberg–Marquardt algorithm). Use the closed form solution from the preceding step as the initial estimate of the intrinsics and extrinsics. Then set the initial estimate of the distortion coefficients to zero. [1][2]
  - “What Is Camera Calibration?”
  - “Single Camera Calibration App”
  - “Coordinate Systems”

## References

- [1] Zhang, Z. “A Flexible New Technique for Camera Calibration”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.
- [2] Heikkila, J, and O. Silven. “A Four-step Camera Calibration Procedure with Implicit Image Correction”, *IEEE International Conference on Computer Vision and Pattern Recognition*, 1997.
- [3] Bouguet, J.Y. “Camera Calibration Toolbox for Matlab”, Computational Vision at the California Institute of Technology. Camera Calibration Toolbox for MATLAB.
- [4] Bradski, G., and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA: O’Reilly, 2008.

## See Also

cameraParameters | stereoParameters | Camera Calibrator |  
detectCheckerboardPoints | disparity | estimateFundamentalMatrix

| estimateUncalibratedRectification | generateCheckerboardPoints  
| reconstructScene | rectifyStereoImages | showExtrinsics |  
showReprojectionErrors | Stereo Camera Calibrator | undistortImage |  
undistortPoints

**Introduced in R2014b**

## estimateFundamentalMatrix

Estimate fundamental matrix from corresponding points in stereo images

### Syntax

```
estimateFundamentalMatrix  
F = estimateFundamentalMatrix(matchedPoints1,matchedPoints2)  
[F,inliersIndex] = estimateFundamentalMatrix(matchedPoints1,  
matchedPoints2)  
[F,inliersIndex,status] = estimateFundamentalMatrix(matchedPoints1,  
matchedPoints2)  
[F,inliersIndex,status] = estimateFundamentalMatrix(matchedPoints1,  
matchedPoints2,Name,Value)
```

### Description

`estimateFundamentalMatrix` estimates the fundamental matrix from corresponding points in stereo images. This function can be configured to use all corresponding points or to exclude outliers. You can exclude outliers by using a robust estimation technique such as random-sample consensus (RANSAC). When you use robust estimation, results may not be identical between runs because of the randomized nature of the algorithm.

`F = estimateFundamentalMatrix(matchedPoints1,matchedPoints2)` returns the 3-by-3 fundamental matrix, `F`, using the least median of squares (LMedS) method. The input points can be  $M$ -by-2 matrices of  $M$  number of [x y] coordinates, or `SURFPoints`, `MSERRegions`, or `cornerPoints` object.

`[F,inliersIndex] = estimateFundamentalMatrix(matchedPoints1,matchedPoints2)` additionally returns logical indices, `inliersIndex`, for the inliers used to compute the fundamental matrix. The `inliersIndex` output is an  $M$ -by-1 vector. The function sets the elements of the vector to `true` when the corresponding point was used to compute the fundamental matrix. The elements are set to `false` if they are not used.

`[F,inliersIndex,status] = estimateFundamentalMatrix(matchedPoints1,matchedPoints2)` additionally returns a status code.



`[F,inliersIndex,status] = estimateFundamentalMatrix(matchedPoints1,matchedPoints2,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

**Code Generation Support:**

Compile-time constant input: `Method`, `OutputClass`, `DistanceType`, and `ReportRuntimeError`.

Supports MATLAB Function block: Yes.

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Compute Fundamental Matrix

The RANSAC method requires that the input points are already putatively matched. We can, for example, use the `matchFeatures` function for this. Using the RANSAC algorithm eliminates any outliers which may still be contained within putatively matched points.

Load stereo points.

```
load stereoPointPairs
```

Estimate the fundamental matrix.

```
fRANSAC = estimateFundamentalMatrix(matchedPoints1,matchedPoints2,'Method','RANSAC','N
```

```
fRANSAC =
```

```
    0.0000   -0.0004    0.0348
    0.0004    0.0000   -0.0937
   -0.0426    0.0993    0.9892
```

### Use the Least Median of Squares Method to Find Inliers

Load the putatively matched points.

```
load stereoPointPairs
```

```
[fLMedS, inliers] = estimateFundamentalMatrix(matchedPoints1,matchedPoints2,'NumTrials
```

```
fLMedS =
```

```
    0.0000    -0.0004    0.0348
    0.0004     0.0000   -0.0937
   -0.0425     0.0993    0.9892
```

```
inliers =
```

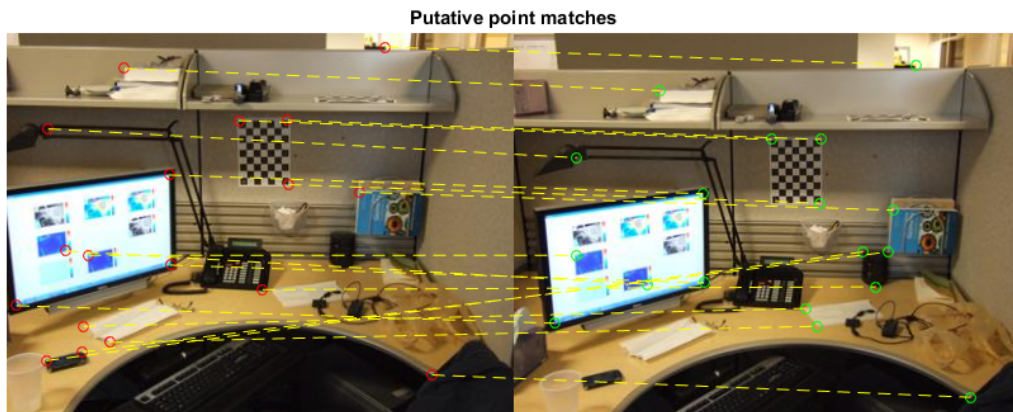
```
    1
    1
    1
    1
    1
    1
    1
    1
    1
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    1
```

Load the stereo images.

```
I1 = imread('viprectification_deskLeft.png');
I2 = imread('viprectification_deskRight.png');
```

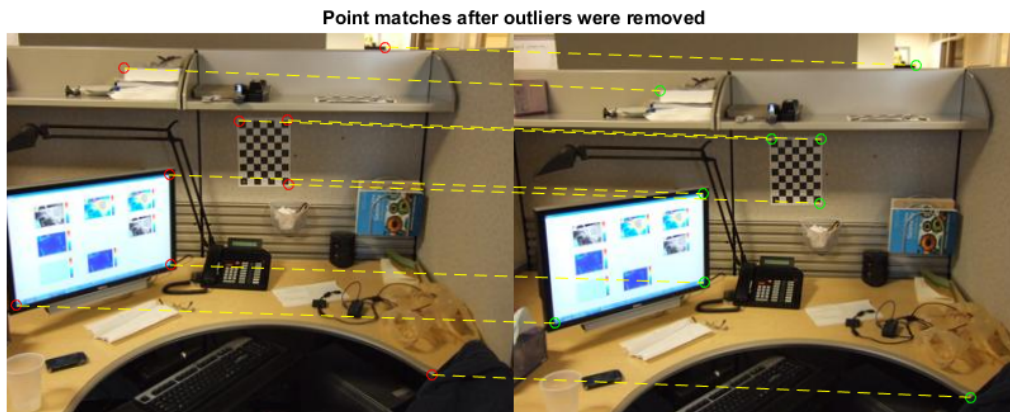
Show the putatively matched points.

```
figure;
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2,'montage','PlotOptions',{ 'ro' },
title('Putative point matches');
```



Show the inlier points.

```
figure;  
showMatchedFeatures(I1, I2, matchedPoints1(inliers,:), matchedPoints2(inliers,:), 'monta  
title('Point matches after outliers were removed');
```



#### Use the Normalized Eight-Point Algorithm to Compute the Fundamental Matrix

Load the stereo point pairs.

```
load stereoPointPairs
```

Compute the fundamental matrix for input points which do not contain any outliers.

```
inlierPts1 = matchedPoints1(knownInliers,:);  
inlierPts2 = matchedPoints2(knownInliers,:);  
fNorm8Point = estimateFundamentalMatrix(inlierPts1,inlierPts2,'Method','Norm8Point')
```

```
fNorm8Point =
```

```
    0.0000   -0.0004    0.0348  
    0.0004    0.0000   -0.0937  
   -0.0426    0.0993    0.9892
```

- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”

## Input Arguments

### **matchedPoints1 — Coordinates of corresponding points**

SURFPoints | cornerPoints | MSERRegions |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Coordinates of corresponding points in image one, specified as an  $M$ -by-2 matrix of  $M$  number of  $[x y]$  coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object. The `matchedPoints1` input must contain points which do not lie on a single planar surface, (e.g., a wall, table, or book) and are putatively matched by using a function such as `matchFeatures`.

### **matchedPoints2 — Coordinates of corresponding points**

SURFPoints | cornerPoints | MSERRegions |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Coordinates of corresponding points in image one, specified as an  $M$ -by-2 matrix of  $M$  number of  $[x y]$  coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object. The `matchedPoints2` input must contain points which do not lie on a single planar surface, (e.g., a wall, table, or book) and are putatively matched by using a function such as `matchFeatures`.

## Name-Value Pair Arguments

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

Example: `'Method', 'RANSAC'` specifies RANSAC as the method to compute the fundamental matrix.

### **'Method' — Method used to compute the fundamental matrix**

LMedS (default) | Norm8Point | RANSAC | MSAC | LTS

Method used to compute the fundamental matrix, specified as the comma-separated pair consisting of `'Method'` and one of five strings. You can set this parameter to one of the following methods:

**Norm8Point** Normalized eight-point algorithm. To produce reliable results, the inputs, `matchedPoints1` and `matchedPoints2` must match precisely.

- LMedS** Least Median of Squares. Select this method if you know that at least 50% of the points in `matchedPoints1` and `matchedPoints2` are inliers.
- RANSAC** RANdom SAMple Consensus. Select this method if you would like to set the distance threshold for the inliers.
- MSAC** M-estimator SAMple Consensus. Select the M-estimator SAMple Consensus method if you would like to set the distance threshold for the inliers. Generally, the MSAC method converges more quickly than the RANSAC method.
- LTS** Least Trimmed Squares. Select the Least Trimmed Squares method if you know a minimum percentage of inliers in `matchedPoints1` and `matchedPoints2`. Generally, the LTS method converges more quickly than the LMedS method.

To produce reliable results using the `Norm8Point` algorithm, the inputs, `matchedPoints1` and `matchedPoints2`, must match precisely. The other methods can tolerate outliers and therefore only require putatively matched input points. You can obtain putatively matched points by using the `matchFeatures` function.

**'OutputClass' — Fundamental matrix class**

'double' (default) | 'single'

Fundamental matrix class, specified as the comma-separated pair consisting of 'OutputClass' and either the string 'double' or 'single'. This specifies the class for the fundamental matrix and the function's internal computations.

**'NumTrials' — Number of random trials for finding the outliers**

500 (default) | integer

Number of random trials for finding the outliers, specified as the comma-separated pair consisting of 'NumTrials' and an integer value. This parameter applies when you set the `Method` parameter to `LMedS`, `RANSAC`, `MSAC`, or `LTS`.

When you set the `Method` parameter to either `LMedS` or `LTS`, the function uses the actual number of trials as the parameter value.

When you set the `Method` parameter to either `RANSAC` or `MSAC`, the function uses the maximum number of trials as the parameter value. The actual number of trials depends on `matchedPoints1`, `matchedPoints2`, and the value of the `Confidence` parameter.

Select the number of random trials to optimize speed and accuracy.

**'DistanceType' — Algebraic or Sampson distance type**

'Sampson' (default) | 'Algebraic'

Algebraic or Sampson distance type, specified as the comma-separated pair consisting of 'DistanceType' and either the Algebraic or Sampson string. The distance type determines whether a pair of points is an inlier or outlier. This parameter applies when you set the Method parameter to LMedS, RANSAC, MSAC, or LTS.

---

**Note:** For faster computations, set this parameter to Algebraic. For a geometric distance, set this parameter to Sampson.

---

Data Types: char

**'DistanceThreshold' — Distance threshold for finding outliers**

0.01 (default)

Distance threshold for finding outliers, specified as the comma-separated pair consisting of 'DistanceThreshold' and a positive value. This parameter applies when you set the Method parameter to RANSAC or MSAC.

**'Confidence' — Desired confidence for finding maximum number of inliers**

99 (default) | scalar

Desired confidence for finding maximum number of inliers, specified as the comma-separated pair consisting of 'Confidence' and a percentage scalar value in the range (0 100). This parameter applies when you set the Method parameter to RANSAC or MSAC.

**'InlierPercentage' — Minimum percentage of inliers in input points**

50 (default) | scalar

Minimum percentage of inliers in input points, specified as the comma-separated pair consisting of 'InlierPercentage' and percentage scalar value in the range (0 100). Specify the minimum percentage of inliers in matchedPoints1 and matchedPoints2. This parameter applies when you set the Method parameter to LTS.

**'ReportRuntimeError' — Report runtime error**

true (default) | false

Report runtime error, specified as the comma-separated pair consisting of 'ReportRuntimeError' and a logical value. Set this parameter to `true` to report run-time errors when the function cannot compute the fundamental matrix from `matchedPoints1` and `matchedPoints2`. When you set this parameter to `false`, you can check the `status` output to verify validity of the fundamental matrix.

## Output Arguments

### **F** — Fundamental matrix

3-by-3 matrix

Fundamental matrix, returns as a 3-by-3 matrix that is computed from the points in the inputs `matchedPoints1` and `matchedPoints2`.

$$[P_2 \ 1] * \textit{FundamentalMatrix} * [P_1 \ 1]' = 0$$

$P_1$ , the point in `matchedPoints1` of image 1 in pixels, corresponds to the point,  $P_2$ , the point in `matchedPoints2` in image 2.

In computer vision, the fundamental matrix is a 3-by-3 matrix which relates corresponding points in stereo images. When two cameras view a 3-D scene from two distinct positions, there are a number of geometric relations between the 3-D points and their projections onto the 2-D images that lead to constraints between the image points. Two images of the same scene are related by epipolar geometry.

### **inliersIndex** — Inliers index

$M$ -by-1 logical vector

Inliers index, returned as an  $M$ -by-1 logical index vector. An element set to `true` means that the corresponding indexed matched points in `matchedPoints1` and `matchedPoints2` were used to compute the fundamental matrix. An element set to `false` means the indexed points were not used for the computation.

Data Types: `logical`

### **status** — Status code

0 | 1 | 2

Status code, returned as one of the following possible values:



status	Value
0:	No error.
1:	matchedPoints1 and matchedPoints2 do not contain enough points. Norm8Point, RANSAC, and MSAC require at least 8 points, LMedS 16 points, and LTS requires $\text{ceil}(800/\text{InlierPercentage})$ .
2:	Not enough inliers found.

Data Types: int32

## More About

### Algorithms

### Computing the Fundamental Matrix

This function computes the fundamental matrix using the normalized eight-point algorithm [1]

When you choose the `Norm8Point` method, the function uses all points in `matchedPoints1` and `matchedPoints2` to compute the fundamental matrix.

When you choose any other method, the function uses the following algorithm to exclude outliers and compute the fundamental matrix from inliers:

- 1 Initialize the fundamental matrix,  $F$ , to a 3-by-3 matrix of zeros.
- 2 Set the loop counter  $n$ , to zero, and the number of loops  $N$ , to the number of random trials specified.
- 3 Loop through the following steps while  $n < N$ :
  - a Randomly select 8 pairs of points from `matchedPoints1` and `matchedPoints2`.
  - b Use the selected 8 points to compute a fundamental matrix,  $f$ , by using the normalized 8-point algorithm.
  - c Compute the fitness of  $f$  for all points in `matchedPoints1` and `matchedPoints2`.
  - d If the fitness of  $f$  is better than  $F$ , replace  $F$  with  $f$ .

For RANSAC and MSAC, update  $N$ .

**e**  $n = n + 1$

## Number of Random Samplings for RANSAC and MSAC Methods

The RANSAC and MSAC methods update the number of random trials  $N$ , for every iteration in the algorithm loop. The function resets  $N$ , according to the following:

$$N = \min\left(N, \frac{\log(1 - p)}{\log(1 - r^8)}\right).$$

Where,  $p$  represents the confidence parameter you specified, and  $r$  is calculated as follows:

$$\sum_i^N \text{sgn}(du_i, v_i, t) / N, \text{ where } \text{sgn}(a, b) = 1 \text{ if } a \leq b \text{ and } 0 \text{ otherwise.}$$

When you use RANSAC or MSAC, results may not be identical between runs because of the randomized nature of the algorithm.

## Distance Types

The function provides two distance types, algebraic distance and Sampson distance, to measure the distance of a pair of points according to a fundamental matrix. The following equations are used for each type, with  $u$  representing `matchedPoints1` and  $v$  representing `matchedPoints2`.

Algebraic distance:

$$d(u_i, v_i) = (v_i F u_i^T)^2$$

Sampson distance:

$$d(u_i, v_i) = (v_i F u_i^T)^2 \left[ \frac{1}{(F u_i^T)_1^2 + (F u_i^T)_2^2} + \frac{1}{(v_i F)_1^2 + (v_i F)_2^2} \right]$$

where  $i$  represents the index of the corresponding points, and  $(F u_i^T)_j^2$ , the square of the  $j$ -th entry of the vector  $F u_i^T$ .

## Fitness of Fundamental Matrix for Corresponding Points

The following table summarizes how each method determines the fitness of the computed fundamental matrix:

Method	Measure of Fitness
LMedS	$median(d(u_i, v_i); i = 1 : N)$ , the number of input points. The smaller the value, the better the fitness.
RANSA	$\sum_i^N \text{sgn}(d(u_i, v_i), t) / N$ , where $\text{sgn}(a, b) = 1$ if $a \leq b$ and $0$ otherwise, $t$ represents the specified threshold. The greater the value, the better the fitness.
MSAC	$\sum_i^N \min(d(u_i, v_i), t)$ . The smaller the value, the better the fitness.
LTS	$\sum_{i \in \Omega} d(u_i, v_i)$ , where $\Omega$ is the first lowest value of an $(N \times q)$ pair of points. Where $q$ represents the inlier percentage you specified. The smaller the value, the better the fitness.

- “Point Feature Types”
- “Structure from Motion”
- “Coordinate Systems”

## References

- [1] Hartley, R., A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, 2003.
- [2] Rousseeuw, P., A. Leroy, *Robust Regression and Outlier Detection*, John Wiley & Sons, 1987.
- [3] Torr, P. H. S., and A. Zisserman, *MLESAC: A New Robust Estimator with Application to Estimating Image Geometry*, Computer Vision and Image Understanding, 2000.

### See Also

`cameraPose` | `detectFASTFeatures` | `detectHarrisFeatures` |  
`detectMinEigenFeatures` | `detectMSERFeatures` | `detectSURFFeatures` |  
`epipolarline` | `estimateUncalibratedRectification` | `extractFeatures` |  
`matchFeatures`

**Introduced in R2012b**

# estimateGeometricTransform

Estimate geometric transform from matching point pairs

## Syntax

```
tform = estimateGeometricTransform(matchedPoints1,matchedPoints2,  
transformType)
```

```
[tform,inlierpoints1,inlierpoints2] = estimateGeometricTransform(  
matchedPoints1,matchedPoints2,transformType)
```

```
[ ____,status] = estimateGeometricTransform(matchedPoints1,  
matchedPoints2,transformType)
```

```
[ ____ ] = estimateGeometricTransform(matchedPoints1,matchedPoints2,  
transformType, Name,Value)
```

## Description

`tform = estimateGeometricTransform(matchedPoints1,matchedPoints2,transformType)` returns a 2-D geometric transform object, `tform`. The `tform` object maps the inliers in `matchedPoints1` to the inliers in `matchedPoints2`.

The function excludes outliers using the M-estimator Sample Consensus (MSAC) algorithm. The MSAC algorithm is a variant of the Random Sample Consensus (RANSAC) algorithm. Results may not be identical between runs because of the randomized nature of the MSAC algorithm.

```
[tform,inlierpoints1,inlierpoints2] = estimateGeometricTransform(  
matchedPoints1,matchedPoints2,transformType)
```

 returns the corresponding inlier points in `inlierpoints1` and `inlierpoints2`.

```
[ ____,status] = estimateGeometricTransform(matchedPoints1,  
matchedPoints2,transformType)
```

 returns a status code of 0, 1, or 2. If you do not request the `status` code output, the function returns an error for conditions that cannot produce results.

[ \_\_\_ ] = estimateGeometricTransform(matchedPoints1,matchedPoints2, transformType, Name,Value) uses additional options specified by one or more Name,Value pair arguments.

**Code Generation Support:**

Compile-time constant input: transformType

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Recover a Transformed Image Using SURF Feature Points

Read and display an image and a transformed image.

```
original = imread('cameraman.tif');  
imshow(original);  
title('Base image');  
distorted = imresize(original,0.7);  
distorted = imrotate(distorted,31);  
figure; imshow(distorted);  
title('Transformed image');
```

**Base image**



**Transformed image**



Detect and extract features from both images.

```
ptsOriginal = detectSURFFeatures(original);  
ptsDistorted = detectSURFFeatures(distorted);  
[featuresOriginal,validPtsOriginal] = extractFeatures(original, ptsOriginal);  
[featuresDistorted,validPtsDistorted] = extractFeatures(distorted,ptsDistorted);
```

Match features.

```
index_pairs = matchFeatures(featuresOriginal,featuresDistorted);  
matchedPtsOriginal = validPtsOriginal(index_pairs(:,1));  
matchedPtsDistorted = validPtsDistorted(index_pairs(:,2));  
figure; showMatchedFeatures(original,distorted,matchedPtsOriginal,matchedPtsDistorted)  
title('Matched SURF points,including outliers');
```



**Matched SURF points,including outliers**

Exclude the outliers, and compute the transformation matrix.

```
[tform,inlierPtsDistorted,inlierPtsOriginal] = estimateGeometricTransform(matchedPtsDistorted,matchedPtsOriginal);  
figure; showMatchedFeatures(original,distorted,inlierPtsOriginal,inlierPtsDistorted);  
title('Matched inlier points');
```

**Matched inlier points**



Recover the original image from the distorted image.

```
outputView = imref2d(size(original));  
Ir = imwarp(distorted,tform,'OutputView',outputView);  
figure; imshow(Ir);  
title('Recovered image');
```

Recovered image



- “Feature Based Panoramic Image Stitching”

## Input Arguments

### **matchedPoints1 — Matched points from image 1**

cornerPoints object | SURFPoints object | MSERRegions object |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Matched points from image 1, specified as either a cornerPoints object, SURFPoints object, MSERRegions object, or an  $M$ -by-2 matrix of  $[x,y]$  coordinates. The function excludes outliers using the M-estimator Sample Consensus (MSAC) algorithm. The MSAC algorithm is a variant of the Random Sample Consensus (RANSAC) algorithm.

### **matchedPoints2 — Matched points from image 2**

cornerPoints object | SURFPoints object | MSERRegions object |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Matched points from image 2, specified as either a `cornerPoints` object, `SURFPoints` object, `MSERegions` object, or an  $M$ -by-2 matrix of  $[x,y]$  coordinates. The function excludes outliers using the M-estimator Sample Consensus (MSAC) algorithm. The MSAC algorithm is a variant of the Random Sample Consensus (RANSAC) algorithm.

### **transformType** — Transform type

'similarity' | 'affine' | 'projective'

Transform type, specified as one of three character strings. You can set the transform type to either 'similarity', 'affine', or 'projective'. The greater the number of matched pairs of points, the greater the accuracy of the estimated transformation. The minimum number of matched pairs of points for each transform type:

Transform Type	Minimum Number of Matched Pairs of Points
'similarity'	2
'affine'	3
'projective'	4

Data Types: char

## **Name-Value Pair Arguments**

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

Example: 'Confidence',99 sets the confidence value to 99.

### **'MaxNumTrials'** — Maximum random trials

1000 (default) | positive integer

Maximum number of random trials for finding the inliers, specified as the comma-separated pair consisting of 'MaxNumTrials' and a positive integer scalar. Increasing this value improves the robustness of the results at the expense of additional computations.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**'Confidence' — Confidence of finding maximum number of inliers**

99 (default) | positive numeric scalar

Confidence of finding the maximum number of inliers, specified as the comma-separated pair consisting of 'Confidence' and a percentage numeric scalar in the range (0 100). Increasing this value improves the robustness of the results at the expense of additional computations.

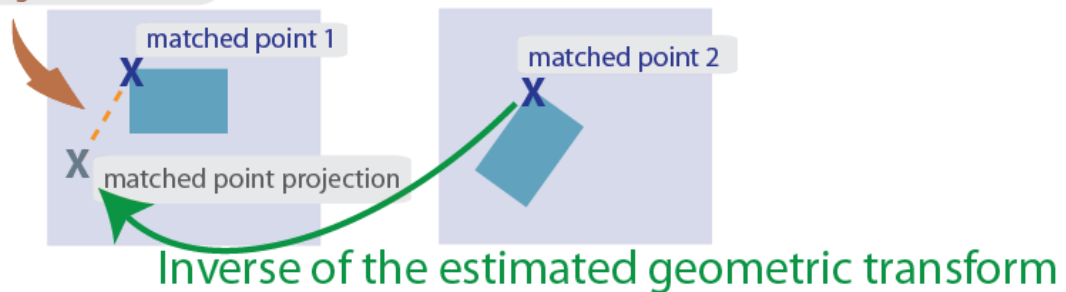
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**'MaxDistance' — Maximum distance from point to projection**

1.5 (default) | positive numeric scalar

Maximum distance in pixels, from a point to the projection of its corresponding point, specified as the comma-separated pair consisting of 'MaxDistance' and a positive numeric scalar. The corresponding projection is based on the estimated transform.

Distance in pixels between the point in image 1 and the projection of the corresponding point from image 2.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Output Arguments****tform — Geometric transformation**

affine2d object | projective2d object

Geometric transformation, returned as either an `affine2d` object or a `projective2d` object.

The returned geometric transformation matrix maps the inliers in `matchedPoints1` to the inliers in `matchedPoints2`. When you set the `transformType` input to either `'similarity'` or `'affine'`, the function returns an `affine2d` object. Otherwise, it returns a `projective2d` object.

#### **status** — Status code

0 | 1 | 2

Status code, returned as the value 0, 1, or 2.

<b>status</b>	<b>Description</b>
0	No error.
1	<code>matchedPoints1</code> and <code>matchedPoints2</code> inputs do not contain enough points.
2	Not enough inliers found.

If you do not request the `status` code output, the function will throw an error for the two conditions that cannot produce results.

Data Types: `double`

#### **inlierpoints1** — Inlier points in image 1

`inlier points`

Inlier points in image 1, returned as the same type as the input matching points.

#### **inlierpoints2** — Inlier points in image 2

`inlier points`

Inlier points in image 2, returned as the same type as the input matching points.

## More About

- “Point Feature Types”
- “Coordinate Systems”

## References

- [1] Hartley, R., and A. Zisserman, "Multiple View Geometry in Computer Vision," *Cambridge University Press*, 2003.
- [2] Torr, P. H. S., and A. Zisserman, "MLESAC: A New Robust Estimator with Application to Estimating Image Geometry," *Computer Vision and Image Understanding*, 2000.

## See Also

cornerPoints | MSERRegions | SURFPoints | detectFASTFeatures |  
detectMinEigenFeatures | detectMSERFeatures | detectSURFFeatures  
| estimateFundamentalMatrix | extractFeatures | fitgeotrans |  
matchFeatures

**Introduced in R2013a**

## estimateUncalibratedRectification

Uncalibrated stereo rectification

### Syntax

```
[T1,T2] = estimateUncalibratedRectification(F,inlierPoints1,  
inlierPoints2,imagesize)
```

### Description

`[T1,T2] = estimateUncalibratedRectification(F,inlierPoints1,inlierPoints2,imagesize)` returns projective transformations for rectifying stereo images. This function does not require either intrinsic or extrinsic camera parameters. The input points can be  $M$ -by-2 matrices of  $M$  number of [x y] coordinates, or SURFPoints, MSERRegions, or cornerPoints object.  $F$  is a 3-by-3 fundamental matrix for the stereo images.

#### Code Generation Support:

Compile-time constant input: No restrictions

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Find Fundamental Matrix Describing Epipolar Geometry

This example shows how to compute the fundamental matrix from corresponding points in a pair of stereo images.

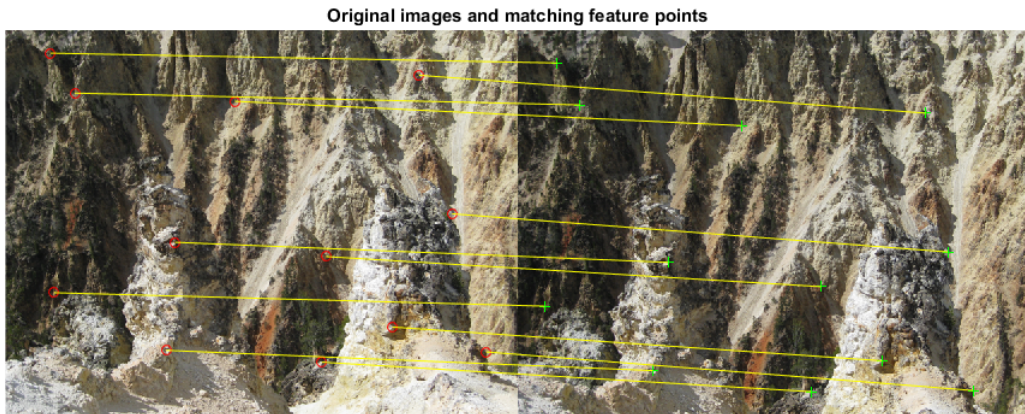
Load the stereo images and feature points which are already matched.

```
I1 = imread('yellowstone_left.png');  
I2 = imread('yellowstone_right.png');  
load yellowstone_inlier_points;
```



Display point correspondences. Notice that the matching points are in different rows, indicating that the stereo pair is not rectified.

```
showMatchedFeatures(I1, I2,inlier_points1,inlier_points2,'montage');
title('Original images and matching feature points');
```



Compute the fundamental matrix from the corresponding points.

```
f = estimateFundamentalMatrix(inlier_points1,inlier_points2,'Method','Norm8Point');
```

Compute the rectification transformations.

```
[t1, t2] = estimateUncalibratedRectification(f,inlier_points1,inlier_points2,size(I2));
```

Rectify the stereo images using projective transformations t1 and t2.

```
[I1Rect,I2Rect] = rectifyStereoImages(I1,I2,t1,t2);
```

Display the stereo anaglyph, which can also be viewed with 3-D glasses.

```
figure;
imshow(stereoAnaglyph(I1Rect,I2Rect));
```



- Image Rectification

## Input Arguments

### **F** — Fundamental matrix for the stereo images

3-by-3 matrix

Fundamental matrix for the stereo images, specified as a 3-by-3 fundamental matrix. The fundamental matrix satisfies the following criteria:

If  $P_1$ , a point in image 1, corresponds to  $P_2$ , a point in image 2, then:

$$[P_2,1] * F * [P_1,1]' = 0$$

F must be double or single.

### **inlierPoints1** — Coordinates of corresponding points

SURFPoints | cornerPoints | MSERRegions |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Coordinates of corresponding points in image one, specified as an  $M$ -by-2 matrix of  $M$  number of [x y] coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object.

**inlierPoints2 — Coordinates of corresponding points**

SURFPoints | cornerPoints | MSERRegions |  $M$ -by-2 matrix of [x,y] coordinates

Coordinates of corresponding points in image one, specified as an  $M$ -by-2 matrix of  $M$  number of [x y] coordinates, or as a SURFPoints, MSERRegions, or cornerPoints object.

**imagesize — Input image size**

single | double | integer

Second input image size, specified as a double, single, or integer value and in the format returned by the size function. The size of input image 2 corresponds to inlierPoints2.

## Output Arguments

**T1 — Projective transformation one**

3-by-3 matrix

Projective transformation, returned as a 3-by-3 matrix describing the projective transformations for input image T1.

**T2 — Projective transformation two**

3-by-3 matrix

Projective transformation, returned as a 3-by-3 matrix describing the projective transformations for input image T2.

## More About

### Tips

- An epipole may be located in the first image or the second image. Applying the output uncalibrated rectification of T1 (or T2) to image 1 (or image 2) may result in an undesired distortion. You can check for an epipole within an image by applying the isEpipoleInImage function.
- “Point Feature Types”

- “Coordinate Systems”

### References

- [1] Hartley, R. and A. Zisserman, "Multiple View Geometry in Computer Vision,"  
*Cambridge University Press*, 2003.

### See Also

stereoParameters | cameraParameters | Camera Calibrator | cameraMatrix |  
detectHarrisFeatures | detectHarrisFeatures | detectMinEigenFeatures |  
estimateCameraParameters | estimateFundamentalMatrix | extractFeatures  
| imwarp | isEpipoleInImage | matchFeatures | reconstructScene | size |  
Stereo Camera Calibrator | undistortImage

**Introduced in R2012b**

# evaluateImageRetrieval

Evaluate image search results

## Syntax

```
averagePrecision = evaluateImageRetrieval(queryImage,imageIndex,  
expectedIDs)  
[averagePrecision,imageIDs,scores] = evaluateImageRetrieval(  
queryImage,imageIndex,expectedIDs)  
[averagePrecision,imageIDs,scores] = evaluateImageRetrieval( ____,  
Name,Value)
```

## Description

`averagePrecision = evaluateImageRetrieval(queryImage,imageIndex,expectedIDs)` returns the average precision metric for measuring the accuracy of image search results for the `queryImage`. The `expectedIDs` input contains the indices of images within `imageIndex` that are known to be similar to the query image.

`[averagePrecision,imageIDs,scores] = evaluateImageRetrieval(queryImage,imageIndex,expectedIDs)` optionally returns the indices corresponding to images within `imageIndex` that are visually similar to the query image. It also returns the corresponding similarity scores.

`[averagePrecision,imageIDs,scores] = evaluateImageRetrieval( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

## Examples

### Evaluate Image Retrieval Results

Create an image set containing images of book covers.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata','bookCovers');
```

```
bookCovers = imageSet(dataDir);
```

Display the image set.

```
thumbnailGallery = [];  
for i = 1:bookCovers.Count  
    I = read(bookCovers, i);  
    thumbnail = imresize(I, [300 300]);  
    thumbnailGallery = cat(4, thumbnailGallery, thumbnail);  
end  
  
figure  
montage(thumbnailGallery);
```



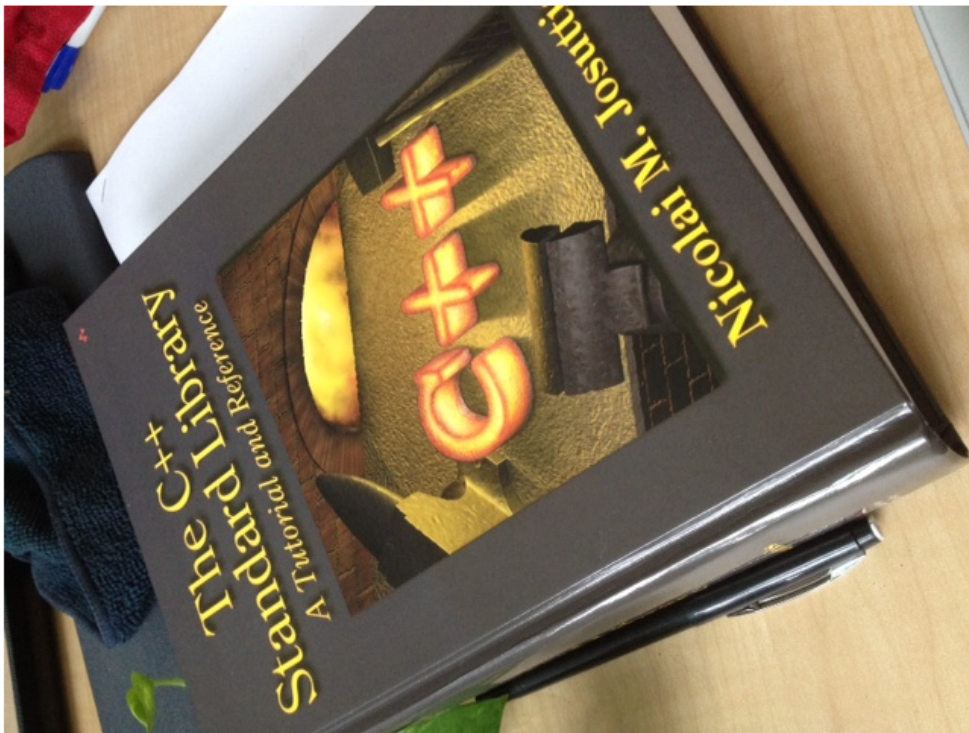


Index the image set. Indexing can take a few minutes.

```
imageIndex = indexImages(bookCovers, 'Verbose', false);
```

Select and display the query image.

```
queryDir = fullfile(dataDir, 'queries', filesep);  
query = imread([queryDir 'query2.jpg']);  
  
figure  
imshow(query)
```



Identify the expected image match. Evaluation requires knowing the expected results. Here, the query image is known to be the second book in the image index.

```
expectedID = 2;
```



Find and report the average precision score.

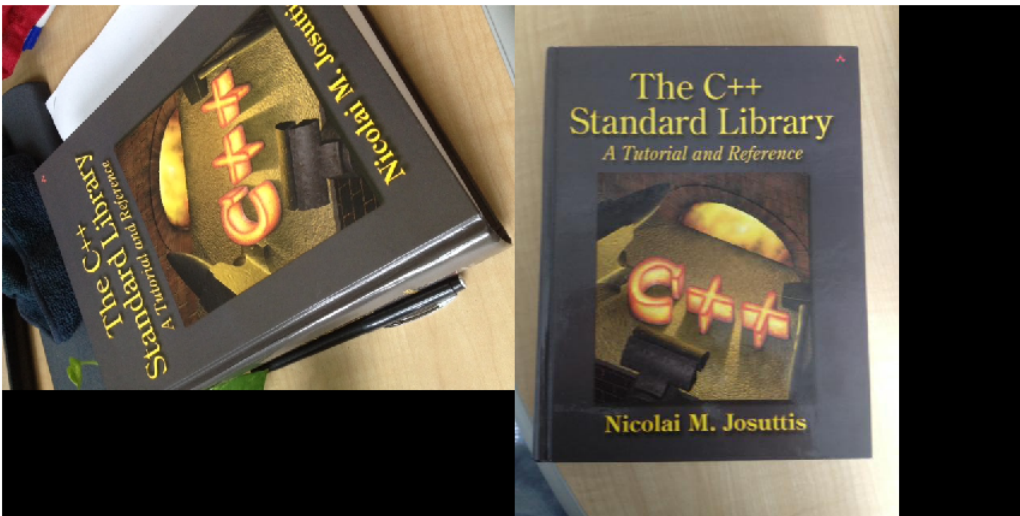
```
[averagePrecision, actualIDs] = evaluateImageRetrieval(query, imageIndex, expectedIDs);
fprintf('Average Precision: %f\n\n', averagePrecision)
```

Average Precision: 1.000000

Show the query and best match side-by-side.

```
bestMatch = actualIDs(1);
bestImage = imread(imageIndex.ImageLocation{bestMatch});

figure
imshowpair(query, bestImage, 'montage')
```



### Compute Mean Average Precision (MAP) for Image Retrieval

Create an image set of book covers.

```
dataDir = fullfile(toolboxdir('vision'), 'visiondata', 'bookCovers');
```

```
bookCovers = imageSet(dataDir);
```

Index the image set. The indexing may take a few minutes.

```
imageIndex = indexImages(bookCovers, 'Verbose', false);
```

Create a set of query images.

```
queryDir = fullfile(dataDir, 'queries', filesep);  
querySet = imageSet(queryDir);
```

Specify the expected search results for each query image.

```
expectedIDs = [1 2 3];
```

Evaluate each query image and collect average precision scores.

```
for i = 1:querySet.Count  
    query = read(querySet,i);  
    averagePrecision(i) = evaluateImageRetrieval(query, imageIndex, expectedIDs(i));  
end
```

Compute mean average precision (MAP).

```
map = mean(averagePrecision)
```

```
map =
```

```
1
```

- “Image Retrieval Using Customized Bag of Features”

## Input Arguments

### **queryImage** — Input query image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input query image, specified as either an *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-D grayscale image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

**imageIndex** — Image search index

`invertedImageIndex` object

Image search index, specified as an `invertedImageIndex` object. The `indexImages` function creates the `invertedImageIndex` object, which stores the data used for the image search.

**expectedIDs** — Image indices

row or column vector

Image indices, specified as a row or column vector. The indices correspond to the images within `imageIndex` that are known to be similar to the query image.

**Name-Value Pair Arguments**

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

Example: `'NumResults',25`

**'NumResults'** — Maximum number of search results to evaluate

`Inf` (default) | positive integer value

Maximum number of search results to evaluate, specified as the comma-separated pair consisting of `'NumResults'` and a positive integer value. The function evaluates the top `NumResults` and returns the average-precision-at-`NumResults` metric.

**'ROI'** — Rectangular search region

`[1 1 size(queryImage,2) size(queryImage,1)]` (default) | `[x y width height]` vector

Rectangular search region within the query image, specified as the comma-separated pair consisting of `'ROI'` and a `[x y width height]` formatted vector.

**Output Arguments****averagePrecision** — Average precision metric

scalar value in the range `[0 1]`

Average precision metric, returned as a scalar value in the range [0 1]. The average precision metric represents the accuracy of image search results for the query image.

### **imageIDs** — Ranked index of retrieved images

*M*-by-1 vector

Ranked index of retrieved images, returned as an *M*-by-1 vector. The image IDs are returned in ranked order, from the most to least similar matched image.

### **scores** — Similarity metric

*N*-by-1 vector

Similarity metric, returned as an *N*-by-1 vector. This output contains the scores that correspond to the retrieved images in the `imageIDs` output. The scores are computed using the cosine similarity and range from 0 to 1.

## More About

- “Image Retrieval with Bag of Visual Words”

## See Also

`imageSet` | `bagOfFeatures` | `invertedImageIndex` | `indexImages` | `retrieveImages`

**Introduced in R2015a**

## extractFeatures

Extract interest point descriptors

### Syntax

```
[features,validPoints] = extractFeatures(I,points)
[features,validPoints] = extractFeatures(I,points,Name,Value)
```

### Description

`[features,validPoints] = extractFeatures(I,points)` returns extracted feature vectors, also known as descriptors, and their corresponding locations, from a binary or intensity image.

The function derives the descriptors from pixels surrounding an interest point. The pixels represent and match features specified by a single-point location. Each single-point specifies the center location of a neighborhood. The method you use for descriptor extraction depends on the class of the input `points`.

`[features,validPoints] = extractFeatures(I,points,Name,Value)` uses additional options specified by one or more `Name, Value` pair arguments.

#### Code Generation Support:

Compile-time constant input: Method

Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries for BRISK, FREAK, and SURF Methods.

Supports MATLAB Function block: Yes, for Block method only.

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Extract Corner Features from an Image.

Read the image.

```
I = imread('cameraman.tif');
```

**Find and extract corner features.**

```
corners = detectHarrisFeatures(I);  
[features, valid_corners] = extractFeatures(I, corners);
```

**Display image.**

```
figure; imshow(I); hold on
```



**Plot valid corner points.**

```
plot(valid_corners);
```



### Extract SURF Features from an Image

#### Read image.

```
I = imread('cameraman.tif');
```

#### Find and extract features.

```
points = detectSURFFeatures(I);  
[features, valid_points] = extractFeatures(I, points);
```

#### Display and plot ten strongest SURF features.

```
figure; imshow(I); hold on;  
plot(valid_points.selectStrongest(10), 'showOrientation', true);
```



#### Extract MSER Features from an Image

Read image.

```
I = imread('cameraman.tif');
```

Find features using MSER with SURF feature descriptor.

```
regions = detectMSERFeatures(I);  
[features, valid_points] = extractFeatures(I,regions,'Upright',true);
```

Display SURF features corresponding to the MSER ellipse centers.

```
figure; imshow(I); hold on;  
plot(valid_points,'showOrientation',true);
```





## Input Arguments

### **I** — Input image

binary image |  $M$ -by- $N$  2-D grayscale image

Input image, specified as either a binary or 2-D grayscale image.

Data Types: logical | single | double | int16 | uint8 | uint16

### **points** — Center location point

BRISKPoints object | cornerPoints object | SURFPoints object | MSERRegions object |  $M$ -by-2 matrix of [x,y] coordinates

Center location point of a square neighborhood, specified as either a BRISKPoints, SURFPoints, MSERRegions, or cornerPoints object, or an  $M$ -by-2 matrix of  $M$  number of [x y] coordinates. The table lists the possible input classes of points that can be used for extraction.

Class of Points	
BRISKPoints	Binary Robust Invariant Scalable Keypoints (BRISK)
SURFPoints object	Speeded-Up Robust Features (SURF)
MSERRegions object	Maximally Stable Extremal Regions (MSER)
cornerPoints	Features from Accelerated Segment Test (FAST), Minimum eigen-value, or Harris
<i>M</i> -by-2 matrix of [x y] coordinates	Simple square neighborhood around [x y] point locations

### Name-Value Pair Arguments

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

Example: `'Method', 'Block'` specifies the `Block` method for descriptor extraction.

#### 'Method' — Descriptor extraction method

`'Auto'` (default) | `'BRISK'` | `'FREAK'` | `'SURF'` | `'Block'`

Descriptor extraction method, specified as a comma-separated pair consisting of `'Method'` and the string `'FREAK'`, `'SURF'`, `'Block'`, or `'Auto'`.

The table describes how the function implements the descriptor extraction methods.

Method	Feature Vector (Descriptor)
BRISK	Binary Robust Invariant Scalable Keypoints (BRISK). The function sets the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.
FREAK	Fast Retina Keypoint (FREAK). The function sets the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.
SURF	Speeded-Up Robust Features (SURF).

Method	Feature Vector (Descriptor)
	<p>The function sets the <b>Orientation</b> property of the <b>validPoints</b> output object to the orientation of the extracted features, in radians.</p> <p>When you use an <b>MSERRegions</b> object with the <b>SURF</b> method, the <b>Centroid</b> property of the object extracts SURF descriptors. The <b>Axes</b> property of the object selects the scale of the SURF descriptors such that the circle representing the feature has an area proportional to the MSER ellipse area. The scale is calculated as <math>1/4 * \text{sqrt}((\text{majorAxes}/2) .* (\text{minorAxes}/2))</math> and saturated to 1.6, as required by the <b>SURFPoints</b> object.</p>
Block	<p>Simple square neighborhood.</p> <p>The <b>Block</b> method extracts only the neighborhoods fully contained within the image boundary. Therefore, the output, <b>validPoints</b>, can contain fewer points than the input <b>POINTS</b>.</p>
Auto	<p>The function selects the <b>Method</b>, based on the class of the input points and implements:</p> <ul style="list-style-type: none"> <li>The <b>FREAK</b> method for a <b>cornerPoints</b> input object.</li> <li>The <b>SURF</b> method for a <b>SURFPoints</b> or <b>MSERRegions</b> input object.</li> <li>The <b>FREAK</b> method for a <b>BRISKPoints</b> input object.</li> </ul> <p>For an <math>M</math>-by-2 input matrix of <math>[x\ y]</math> coordinates, the function implements the <b>Block</b> method.</p>

**'BlockSize' — Block size**

11 (default) | odd integer scalar

Block size, specified as the comma-separated pair consisting of **'BlockSize'** and an odd integer scalar. This value defines the local square neighborhood **BlockSize**-by-**BlockSize**, centered at each interest point. This option applies only when the function implements the **Block** method.

**'Upright' — Rotation invariance flag**

false | logical scalar

Rotation invariance flag, specified as the comma-separated pair consisting of 'Upright' and a logical scalar. When you set this property to `true`, the orientation of the feature vectors are not estimated and the feature orientation is set to  $\pi/2$ . Set this to `true` when you do not need the image descriptors to capture rotation information. When you set this property to `false`, the orientation of the features is estimated and the features are then invariant to rotation.

**'SURFSize' — Length of feature vector**

64 (default) | 128

Length of the SURF feature vector (descriptor), specified as the comma-separated pair consisting of 'SURFSize' and the integer scalar 64 or 128. This option applies only when the function implements the SURF method. The larger SURFSize of 128 provides greater accuracy, but decreases the feature matching speed.

## Output Arguments

**features — Feature vectors**

*M*-by-*N* matrix | binaryFeatures object

Feature vectors, returned as a binaryFeatures object or an *M*-by-*N* matrix of *M* feature vectors, also known as descriptors. Each descriptor is of length *N*.

**validPoints — Valid points**

BRISKPoints object | cornerPoints object | SURFPoints object | MSERRegions object | *M*-by-2 matrix of [x,y] coordinates

Valid points associated with each output feature vector (descriptor) in `features`, returned in the same format as the input. Valid points can be a BRISKPoints, cornerPoints, SURFPoints, MSERRegions object, or an *M*-by-2 matrix of [x,y] coordinates.

The function extracts descriptors from a region around each interest point. If the region lies outside of the image, the function cannot compute a feature descriptor for that point. When the point of interest lies too close to the edge of the image, the function cannot compute the feature descriptor. In this case, the function ignores the point. The point is not included in the valid points output.

## More About

- “Point Feature Types”
- “Local Feature Detection and Extraction”

## References

- [1] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly, Sebastopol, CA, 2008.
- [2] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, *SURF: Speeded Up Robust Features*", *Computer Vision and Image Understanding (CVIU)*, Vol. 110, No. 3, pp. 346--359, 2008
- [3] Bay, Herbert, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool, "SURF: Speeded Up Robust Features", *Computer Vision and Image Understanding (CVIU)*, Vol. 110, No. 3, pp. 346--359, 2008.
- [4] Alahi, Alexandre, Ortiz, Raphael, and Pierre Vanderghenst, "FREAK: Fast Retina Keypoint", *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

## See Also

`SURFPoints` | `MSERRegions` | `binaryFeatures` | `detectBRISKFeatures` | `detectFASTFeatures` | `detectHarrisFeatures` | `detectMinEigenFeatures` | `detectMSERFeatures` | `detectSURFFeatures` | `extractHOGFeatures` | `extractLBPFeatures` | `matchFeatures`

**Introduced in R2011a**

## extractHOGFeatures

Extract histogram of oriented gradients (HOG) features

### Syntax

```
features = extractHOGFeatures(I)
[features,validPoints] = extractHOGFeatures(I,points)
[ ____, visualization] = extractHOGFeatures(I, ____)
[ ____ ] = extractHOGFeatures( ____,Name,Value)
```

### Description

`features = extractHOGFeatures(I)` returns extracted HOG features from a truecolor or grayscale input image, `I`. The features are returned in a 1-by- $N$  vector, where  $N$  is the HOG feature length. The returned features encode local shape information from regions within an image. You can use this information for many tasks including classification, detection, and tracking.

`[features,validPoints] = extractHOGFeatures(I,points)` returns HOG features extracted around specified point locations. The function also returns `validPoints`, which contains the input point locations whose surrounding region is fully contained within `I`. Scale information associated with the points is ignored.

`[ ____, visualization] = extractHOGFeatures(I, ____)` optionally returns a HOG feature visualization, using any of the preceding syntaxes. You can display this visualization using `plot(visualization)`.

`[ ____ ] = extractHOGFeatures( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

#### Code Generation Support:

Compile-time constant input: No

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Extract and Plot HOG Features

#### Read the image of interest.

```
img = imread('cameraman.tif');
```

#### Extract HOG features.

```
[featureVector, hogVisualization] = extractHOGFeatures(img);
```

#### Plot HOG features over the original image.

```
figure;  
imshow(img); hold on;  
plot(hogVisualization);
```



#### Extract HOG Features using CellSize

Read the image of interest.

```
I1 = imread('gantrycrane.png');
```

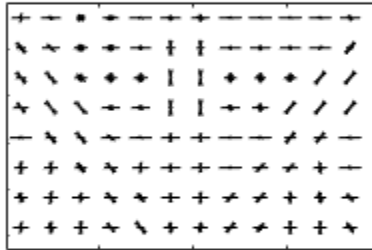
Extract HOG features.

```
[hog1, visualization] = extractHOGFeatures(I1, 'CellSize',[32 32]);
```

Display the original image and the HOG features.

```
subplot(1,2,1);  
imshow(I1);  
subplot(1,2,2);  
plot(visualization);
```





### Extract HOG Features Around Corner Points

Read in the image of interest.

```
I2 = imread('gantrycrane.png');
```

Detect and select the strongest corners in the image.

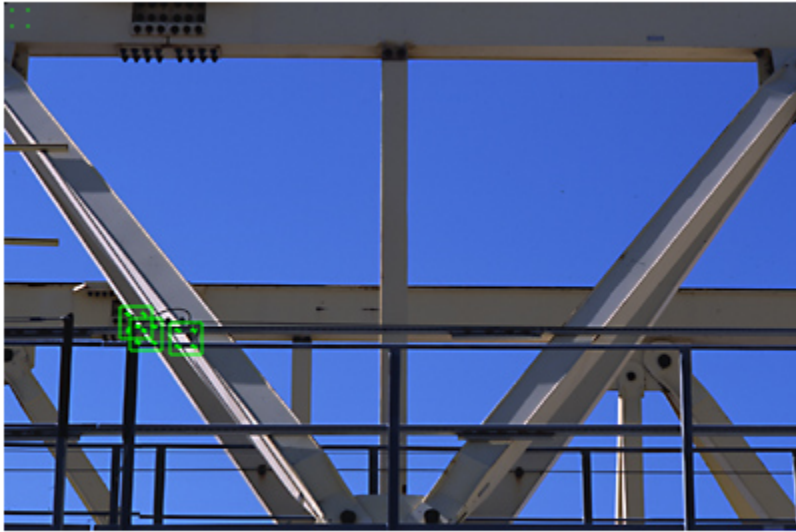
```
corners = detectFASTFeatures(rgb2gray(I2));
strongest = selectStrongest(corners,3);
```

Extract HOG features.

```
[hog2, validPoints, ptVis] = extractHOGFeatures(I2,strongest);
```

Display the original image with an overlay of HOG features around the strongest corners.

```
figure;  
imshow(I2); hold on;  
plot(ptVis, 'Color', 'green');
```



- “Digit Classification Using HOG Features”

## Input Arguments

### **I** — Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input image, specified in either *M*-by-*N*-by-3 truecolor or *M*-by-*N* 2-D grayscale. The input image must be a real, nonsparse value. If you have tightly cropped images, you may lose shape information that the HOG function can encode. You can avoid losing

this information by including an extra margin of pixels around the patch that contains background pixels.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

### **points** — Center location point

BRISKPoints object | cornerPoints object | SURFPoints object | MSERRegions object |  $M$ -by-2 matrix of  $[x, y]$  coordinates

Center location point of a square neighborhood, specified as either a BRISKPoints, SURFPoints, MSERRegions, or cornerPoints object, or an  $M$ -by-2 matrix of  $M$  number of  $[x, y]$  coordinates. The function extracts descriptors from the neighborhoods that are fully contained within the image boundary. You can set the size of the neighborhood with the `BlockSize` parameter. Only neighborhoods fully contained within the image are used to determine the valid output points. The function ignores scale information associated with these points.

## **Name-Value Pair Arguments**

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

Example: `'BlockSize', [2 2]` sets the `BlockSize` to be a 2-by-2 square block.

### **'CellSize'** — Size of HOG cell

`[8 8]` (default) | 2-element vector

Size of HOG cell, specified in pixels as a 2-element vector. To capture large-scale spatial information, increase the cell size. When you increase the cell size, you may lose small-scale detail.

### **'BlockSize'** — Number of cells in block

`[2 2]` (default) | 2-element vector

Number of cells in a block, specified as a 2-element vector. A large block size value reduces the ability to suppress local illumination changes. Because of the number of pixels in a large block, these changes may get lost with averaging. Reducing the block size helps to capture the significance of local pixels. Smaller block size can help suppress illumination changes of HOG features.

**'BlockOverlap' — Number of overlapping cells between adjacent blocks**`ceil(BlockSize/2)` (default)

Number of overlapping cells between adjacent blocks, specified as a 2-element vector. To ensure adequate contrast normalization, select an overlap of at least half the block size. Large overlap values can capture more information, but they produce larger feature vector size. This property applies only when you are extracting HOG features from regions and not from point locations. When you are extracting HOG features around a point location, only one block is used, and thus, no overlap occurs.

**'NumBins' — Number of orientation histogram bins**`9` (default) | positive scalar

Number of orientation histogram bins, specified as positive scalar. To encode finer orientation details, increase the number of bins. Increasing this value increases the size of the feature vector, which requires more time to process.

**'UseSignedOrientation' — Selection of orientation values**`false` (default) | logical scalar

Selection of orientation values, specified as a logical scalar. When you set this property to `true`, orientation values are evenly spaced in bins between -180 and 180 degrees. When you set this property to `false`, they are evenly spaced from 0 through 180. In this case, values of theta that are less than 0 are placed into a theta + 180 value bin. Using signed orientation can help differentiate light-to-dark versus dark-to-light transitions within an image region.

## Output Arguments

**features — Extracted HOG features**`1-by-N` vector | `P-by-Q` matrix

Extracted HOG features, returned as either a `1-by-N` vector or a `P-by-Q` matrix. The features encode local shape information from regions or from point locations within an image. You can use this information for many tasks including classification, detection, and tracking.

features output	Description
<code>1-by-<i>N</i></code> vector	HOG feature length, <i>N</i> , is based on the image size and the function parameter values. $N = \text{prod}([\textit{BlocksPerImage}, \textit{BlockSize}, \textit{NumBins}])$

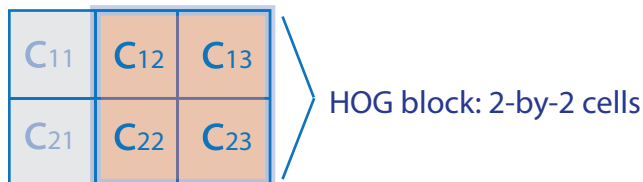
features output	Description
	$BlocksPerImage = \text{floor}((\text{size}(I)/\text{CellSize} - \text{BlockSize})/(\text{BlockSize} - \text{BlockOverlap}) + 1)$
<i>P</i> -by- <i>Q</i> matrix	<i>P</i> is the number of valid points whose surrounding region is fully contained within the input image. You provide the <code>points</code> input value for extracting point locations. The surrounding region is calculated as: $\text{CellSize} * \text{BlockSize}$ . The feature vector length, <i>Q</i> , is calculated as: $\text{prod}([\text{NumBins}, \text{BlockSize}])$ .

### Arrangement of Histograms in HOG Feature Vectors

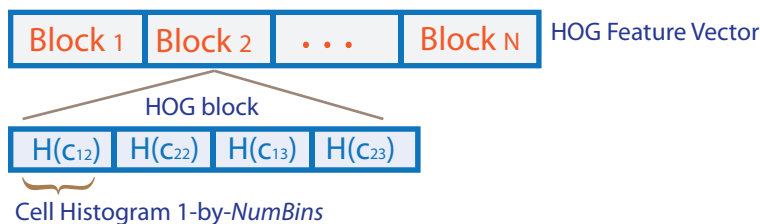
The figure below shows an image with six cells.

$C_{11}$	$C_{12}$	$C_{13}$
$C_{21}$	$C_{22}$	$C_{23}$

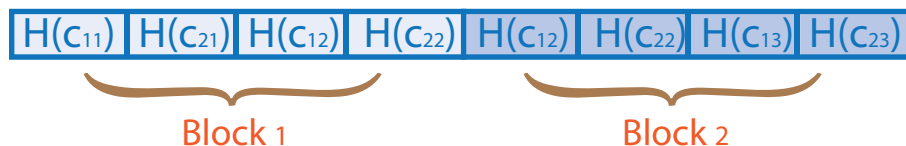
If you set the `BlockSize` to  $[2 \ 2]$ , it would make the size of each HOG block, 2-by-2 cells. The size of the cells are in pixels. You can set it with the `CellSize` property.



The HOG feature vector is arranged by HOG blocks. The cell histogram,  $H(C_{yx})$ , is 1-by-`NumBins`.



The figure below shows the HOG feature vector with a 1-by-1 cell overlap between blocks.



### **validPoints – Valid points**

cornerPoints object | BRISKPoints object | SURFPoints object | MSERRegions object |  $M$ -by-2 matrix of  $[x,y]$  coordinates

Valid points associated with each `features` descriptor vector output. This output can be returned as either a `cornerPoints` object, `BRISKPoints`, `SURFPoints` object, `MSERRegions` object, or an  $M$ -by-2 matrix of  $[x,y]$  coordinates. The function extracts  $M$  number of descriptors from valid interest points in a region of size equal to  $[CellSize * BlockSize]$ . The extracted descriptors are returned as the same type of object or matrix as the input. The region must be fully contained within the image.

### **visualization – HOG feature visualization**

object

HOG feature visualization, returned as an object. The function outputs this optional argument to visualize the extracted HOG features. You can use the `plot` method with the `visualization` output. See the “Extract and Plot HOG Features” on page 3-229 example.

HOG features are visualized using a grid of uniformly spaced `rose` plots. The cell size and the size of the image determines the grid dimensions. Each rose plot shows the distribution of gradient orientations within a HOG cell. The length of each petal of the rose plot is scaled to indicate the contribution each orientation makes within the cell histogram. The plot displays the edge directions, which are normal to the gradient directions. Viewing the plot with the edge directions allows you to better understand the shape and contours encoded by HOG. Each rose plot displays two times `NumBins` petals.

You can use the following syntax to plot the HOG features:

`plot(visualization)` plots the HOG features as an array of rose plots.

`plot(visualization,AX)` plots HOG features into the axes `AX`.

`plot(____, 'Color', Colorspec)` Specifies the color used to plot HOG features, where `Colorspec` represents the color.

## More About

- “Local Feature Detection and Extraction”
- “Point Feature Types”

## References

- [1] Dalal, N. and B. Triggs. "Histograms of Oriented Gradients for Human Detection", *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 1 (June 2005), pp. 886–893.

## See Also

[SURFPoints](#) | [MSERRegions](#) | [binaryFeatures](#) | [detectFASTFeatures](#) | [detectHarrisFeatures](#) | [detectMinEigenFeatures](#) | [detectMSERFeatures](#) | [detectSURFFeatures](#) | [extractFeatures](#) | [extractLBPFeatures](#) | [matchFeatures](#) | [rose](#)

**Introduced in R2013b**

## extrinsics

Compute location of calibrated camera

### Syntax

```
[rotationMatrix,translationVector] = extrinsics(imagePoints,  
worldPoints,cameraParams)
```

### Description

`[rotationMatrix,translationVector] = extrinsics(imagePoints, worldPoints,cameraParams)` returns the 3-D rotation matrix and the 3-D translation vector to allow you to transform points from the world coordinate to the camera coordinate system.

#### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Compute Extrinsics

Load calibration images.

```
numImages = 9;  
files = cell(1, numImages);  
for i = 1:numImages  
    files{i} = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'calibration',  
    end
```

Detect the checkerboard corners in the images.

```
[imagePoints, boardSize] = detectCheckerboardPoints(files);
```



Generate the world coordinates of the checkerboard corners in the pattern-centric coordinate system, with the upper-left corner at (0,0).

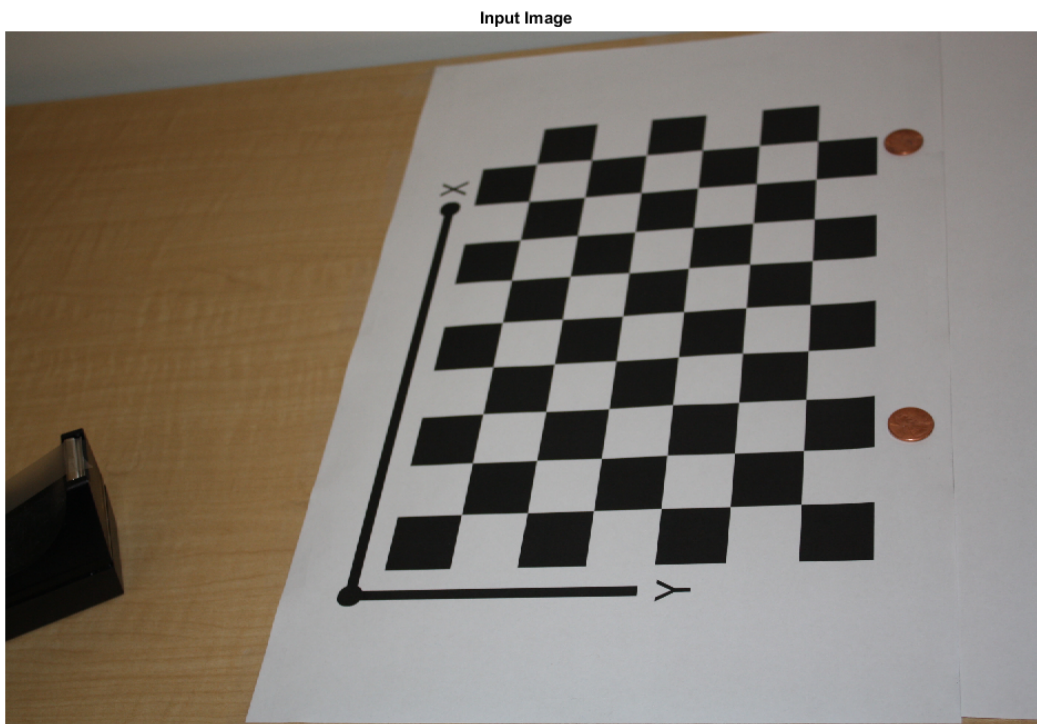
```
squareSize = 29; % in millimeters  
worldPoints = generateCheckerboardPoints(boardSize, squareSize);
```

Calibrate the camera.

```
cameraParams = estimateCameraParameters(imagePoints, worldPoints);
```

Load image at new location.

```
imOrig = imread(fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'calibration',  
figure; imshow(imOrig);  
title('Input Image');
```



Undistort image.

```
im = undistortImage(imOrig, cameraParams);
```

Find reference object in new image.

```
[imagePoints, boardSize] = detectCheckerboardPoints(im);
```

Compute new extrinsics.

```
[rotationMatrix, translationVector] = extrinsics(imagePoints, worldPoints, cameraParams);
```

```
rotationMatrix =
```

```
    0.1417    -0.7409     0.6565  
    0.9661    -0.0410    -0.2548  
    0.2157     0.6703     0.7100
```

```
translationVector =
```

```
   -29.2584    35.7824   725.5824
```

- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **imagePoints** — Image coordinates of points

*M*-by-2 array

Image coordinates of points, specified as an *M*-by-2 array. The array contains *M* number of  $[x, y]$  coordinates. The `imagePoints` and `worldPoints` inputs must both be double or both be single.

Data Types: `single` | `double`

### **worldPoints** — World coordinates corresponding to image coordinates

*M*-by-2 matrix | *M*-by-3 matrix

World coordinates corresponding to image coordinates, specified as an *M*-by-2 or an *M*-by-3 matrix. The `imagePoints` and `worldPoints` inputs must both be double or both be single.

When you specify an  $M$ -by-2 matrix of  $[x, y]$  coordinates, the function assumes that the points are coplanar with  $z=0$ . In this case, the number of points,  $M$ , must be greater than 3.

When you specify an  $M$ -by-3 matrix of  $[x, y, z]$  coordinates, then  $M$ , must be greater than 5.

Data Types: `single` | `double`

### **cameraParams** — Object for storing camera parameters

`cameraParameters` object

Object for storing camera parameters, specified as a `cameraParameters` returned by the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

## Output Arguments

### **rotationMatrix** — 3-D rotation

3-by-3 matrix

3-D rotation, returned as a 3-by-3 matrix. The rotation matrix together with the translation vector allows you to transform points from the world coordinate to the camera coordinate system..

$$\begin{array}{c}
 [x \ y \ z] = [X \ Y \ Z]R + t \\
 \text{camera coordinates} \quad \text{world coordinates} \quad \text{rotation matrix} \quad \text{translation vector}
 \end{array}$$

If you set the `imagePoints` and `worldPoints` inputs to class `double`, then the function returns the `rotationMatrix` and `translationVector` as `double`. Otherwise, they are `single`.

### **translationVector** — 3-D translation

3-D translation. returned as a 1-by-3 vector. The rotation matrix together with the translation vector allows you to transform points from the world coordinate to the camera coordinate system.

3-D translation, returned as a 1-by-3 vector.

$$[x \ y \ z] = [X \ Y \ Z]R + t$$

camera coordinates      world coordinates      translation vector      rotation matrix

If you set the `imagePoints` and `worldPoints` inputs to class `double`, then the function returns the `rotationMatrix` and `translationVector` as `double`. Otherwise, they are `single`.

## More About

### Algorithms

The `extrinsics` function uses two different algorithms to compute the extrinsics depending on whether `worldPoints` are specified as an  $M$ -by-2 or an  $M$ -by-3 matrix. The algorithm for the  $M$ -by-3 case requires the points to be non-coplanar. It will produce incorrect results if the `worldPoints` are coplanar. Use an  $M$ -by-2 matrix for coplanar points where  $z \neq 0$ .

The `extrinsics` function computes the rotation matrix and translation vector for a single image in closed form. During calibration, the extrinsics are estimated numerically to minimize the reprojection errors for all calibration images. Therefore, using the `extrinsics` function on one of the calibration images returns rotation matrix and translation vector slightly different from the ones obtained during calibration.

### See Also

`cameraParameters` | `Camera Calibrator` | `cameraMatrix` |  
`estimateCameraParameters` | `plotCamera`

**Introduced in R2014a**

# generateCheckerboardPoints

Generate checkerboard corner locations

## Syntax

```
[worldPoints] = generateCheckerboardPoints(boardSize,squareSize)
```

## Description

```
[worldPoints] = generateCheckerboardPoints(boardSize,squareSize)
```

returns an  $M$ -by-2 matrix containing  $M$   $[x, y]$  corner coordinates for the squares on a checkerboard. The point  $[0,0]$  corresponds to the lower-right corner of the top-left square of the board.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

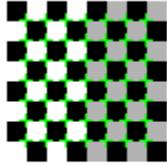
### Generate and Plot Corners of an 8-by-8 Checkerboard

Generate the checkerboard, and obtain world coordinates.

```
I = checkerboard;  
squareSize = 10;  
worldPoints = generateCheckerboardPoints([8 8], squareSize);
```

Offset the points, placing the first point at the lower-right corner of the first square.

```
imshow(insertMarker(I, worldPoints + squareSize));
```



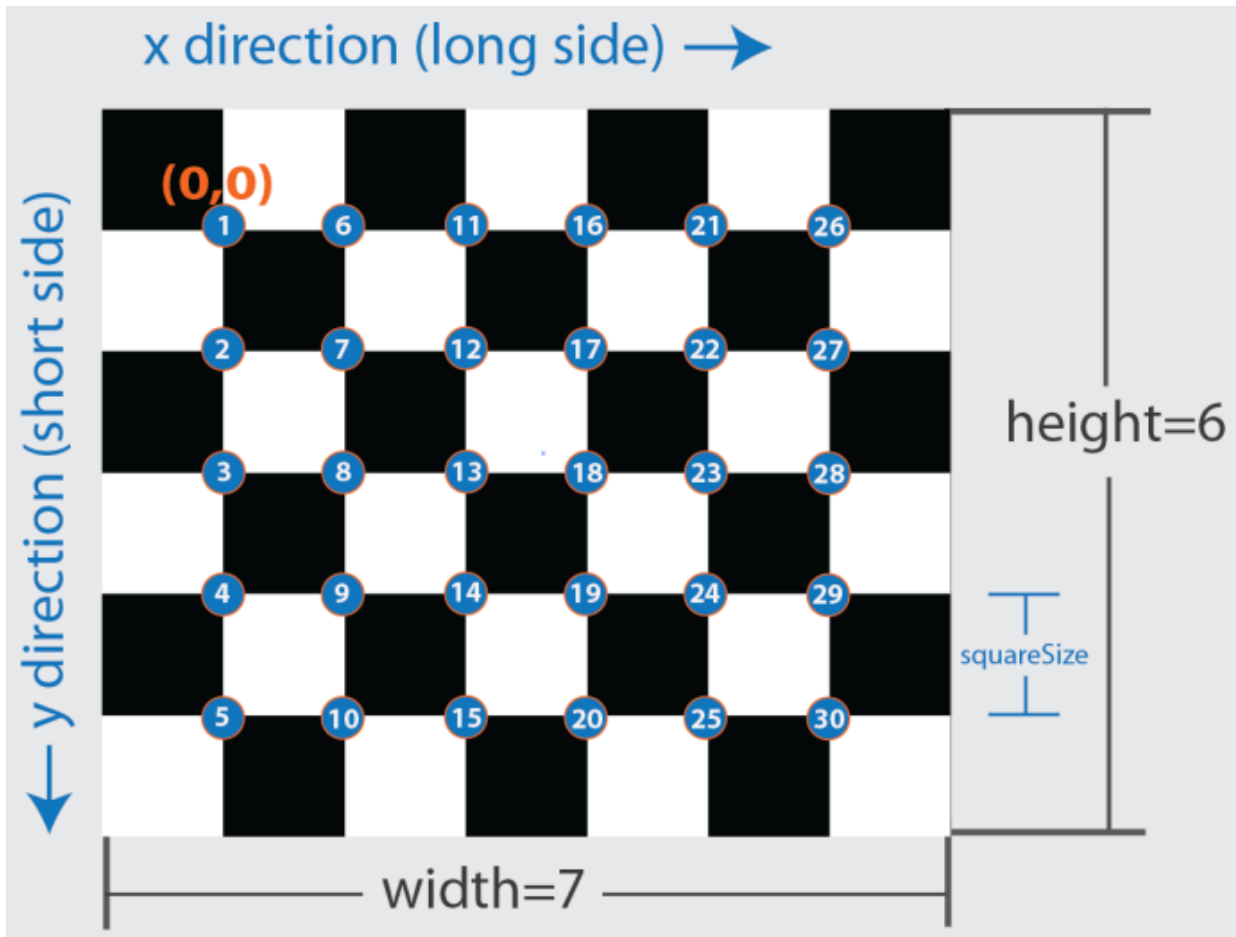
- “Measuring Planar Objects with a Calibrated Camera”

### Input Arguments

**boardSize** — Generated checkerboard dimensions

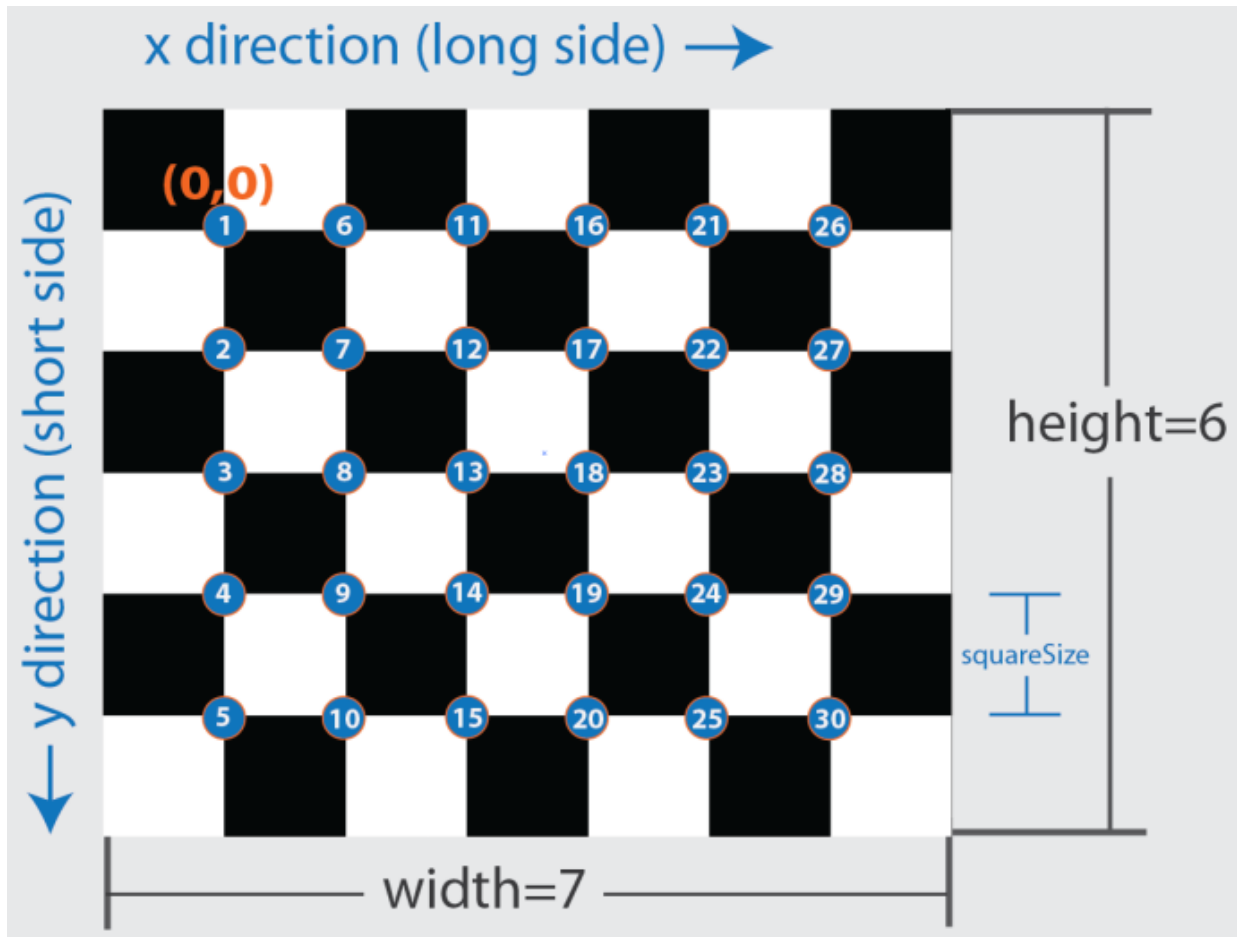
2-element [*height*, *width*] vector

Generated checkerboard dimensions, specified as a 2-element [*height*, *width*] vector. You express the dimensions of the checkerboard in number of squares.



**squareSize** — Generated checkerboard square side length  
 scalar

Checkerboard square side length, specified as a scalar in world units. You express world units as a measurement, such as millimeters or inches.



## Output Arguments

**worldPoints** — Generated checkerboard corner coordinates

*M*-by-2 matrix

Generated checkerboard corner coordinates, returned as an *M*-by-2 matrix of *M* number of  $[x \ y]$  coordinates. The coordinates represent the corners of the squares on the checkerboard. The point  $[0,0]$  corresponds to the lower-right corner of the top-left square



of the board. The number of points,  $M$ , that the function returns are based on the number of squares on the checkerboard. This value is set with the `boardSize` parameter.

$$M = (\text{boardSize}(1)-1) * (\text{boardSize}(2)-1)$$

## More About

- “Single Camera Calibration App”

## See Also

[cameraParameters](#) | [stereoParameters](#) | [Camera Calibrator](#) |  
[detectCheckerboardPoints](#) | [estimateCameraParameters](#)

**Introduced in R2013b**

## indexImages

Create image search index

### Syntax

```
imageIndex = indexImages(imgSet)
```

```
imageIndex = indexImages(imgSet,bag)
```

```
imageIndex = indexImages( ____,Name,Value)
```

### Description

`imageIndex = indexImages(imgSet)` creates an `invertedImageIndex` object, `imageIndex`, that contains a search index for `imgSet`. Use `imageIndex` with the `retrieveImages` function to search for images.

`imageIndex = indexImages(imgSet,bag)` returns a search index that uses a custom `bagOfFeatures` object, `bag`. Use this syntax with the `bag` you created when you want to modify the number of visual words or the feature type used to create the image search index for `imgSet`.

`imageIndex = indexImages( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

This object supports parallel computing using multiple MATLAB workers. Enable parallel computing from the “Computer Vision System Toolbox Preferences” dialog box. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision System Toolbox**.

### Examples

#### Search Image Set Using a Query Image

Create an image set.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets','cups');  
imgSets = imageSet(setDir,'recursive');
```

Index the image set.

```
imageIndex = indexImages(imgSets,'Verbose',false)
```

```
imageIndex =
```

```
invertedImageIndex with properties:
```

```
    ImageLocation: {6x1 cell}  
    ImageWords: [6x1 vision.internal.visualWords]  
    WordFrequency: [1x1366 double]  
    BagOfFeatures: [1x1 bagOfFeatures]  
    MatchThreshold: 0.0100  
    WordFrequencyRange: [0.0100 0.9000]
```

Display the image set using the `montage` function.

```
thumbnailGallery = [];  
for i = 1:imgSets.Count  
    I = read(imgSets,i);  
    thumbnail = imresize(I,[300 300]);  
    thumbnailGallery = cat(4,thumbnailGallery,thumbnail);  
end  
  
figure  
montage(thumbnailGallery);
```



Select a query image.

```
queryImage = read(imgSets,2);  
figure  
imshow(queryImage)
```



Search the image set for similar image using query image. The best result is first.

```
indices = retrieveImages(queryImage, imageIndex)
bestMatchIdx = indices(1);
```

```
indices =
```

```
    2
    1
    5
    4
    3
```

Display the best match from the image set.

```
bestMatch = imageIndex.ImageLocation{bestMatchIdx}
figure
imshow(bestMatch)
```

```
bestMatch =
```

```
B:\matlab\toolbox\vision\visiondata\imageSets\cups\blueCup.jpg
```



Create an image set.

**Create Search Index Using Custom Bag of Features**



```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets','cups');  
imgSets = imageSet(setDir, 'recursive');
```

Display image set.

```
thumbnailGallery = [];  
for i = 1:imgSets.Count  
    I = read(imgSets, i);  
    thumbnail = imresize(I, [300 300]);  
    thumbnailGallery = cat(4, thumbnailGallery, thumbnail);  
end  
  
figure  
montage(thumbnailGallery);
```



Train a bag of features using a custom feature extractor.



```

extractor = @exampleBagOfFeaturesExtractor;
bag = bagOfFeatures(imgSets, 'CustomExtractor', extractor);

```

Creating Bag-Of-Features from 1 image sets.

-----

\* Image set 1: cups.

\* Extracting features using a custom feature extraction function: exampleBagOfFeatures

\* Extracting features from 6 images in image set 1...done. Extracted 115200 features.

\* Keeping 80 percent of the strongest features from each image set.

\* Using K-Means clustering to create a 500 word visual vocabulary.

\* Number of features : 92160

\* Number of clusters (K) : 500

\* Initializing cluster centers...100.00%.

\* Clustering...completed 24/100 iterations (~0.26 seconds/iteration)...converged in 24

\* Finished creating Bag-Of-Features

Use the trained bag of features to index the image set.

```

imageIndex = indexImages(imgSets, bag, 'Verbose', false)

```

```

queryImage = read(imgSets, 4);

```

```

figure
imshow(queryImage)

```

```

imageIndex =

```

invertedImageIndex with properties:

```

    ImageLocation: {6x1 cell}
      ImageWords: [6x1 vision.internal.visualWords]
    WordFrequency: [1x500 double]
    BagOfFeatures: [1x1 bagOfFeatures]
    MatchThreshold: 0.0100
    WordFrequencyRange: [0.0100 0.9000]

```



Search for the image from image index using query image.

```
indices = retrieveImages(queryImage, imageIndex);  
  
bestMatch = imageIndex.ImageLocation{indices(1)};  
figure  
imshow(bestMatch)
```



- “Image Retrieval Using Customized Bag of Features”

## Input Arguments

### **imgSet** — Image set

imageSet object

Image set, specified as an imageSet object. The object stores a collection of images. You can use this object to manage your image data. The object contains image descriptions, locations of images, and the number of images in your collection.

### **bag** — Bag of visual words

bagOfFeatures object

Bag of visual words, specified as a bagOfFeatures 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`.

Example: `'Verbose',true` sets the `'Verbose'` property to `true`

### **'SaveFeatureLocations'** — Save feature locations

true (default) | false

Save feature locations, specified as the comma-separated pair consisting of `'SaveFeatureLocations'` and a logical scalar. When set to `true`, the image feature locations are saved in the `imageIndex` output object. Use location data to verify spatial or geometric image search results. If you do not require feature locations, set this property to `false` to reduce memory consumption.

### **'Verbose'** — Display progress information

true (default) | false

Display progress information, specified as the comma-separated pair consisting of `'Verbose'` and a logical scalar.

# Output Arguments

### **imageIndex** — Image search index

`invertedImageIndex` object

Image search index, returned as an `invertedImageIndex` object.

# More About

## Algorithms

`imageIndex` uses the bag-of-features framework with the speeded-up robust features (SURF) detector and extractor to learn a vocabulary of 20,000 visual words. The visual words are then used to create an index that maps visual words to the images in `imgSet`. You can use the index to search for images within `imgSet` that are similar to a given query image.

- “Image Retrieval with Bag of Visual Words”

## See Also

`imageSet` | `bagOfFeatures` | `invertedImageIndex` | `evaluateImageRetrieval` | `retrieveImages`

**Introduced in R2015a**

# integralFilter

Filter using integral image

## Syntax

```
J = integralFilter(intI,H)
```

## Description

`J = integralFilter(intI,H)` filters an image, given its integral image, `intI`, and filter object, `H`. The `integralKernel` function returns the filter object used for the input to the `integralFilter`.

This function uses integral images for filtering an image with box filters. You can obtain the integral image, `intI`, by calling the `integralImage` function. The filter size does not affect the speed of the filtering operation. Thus, the `integralFilter` function is ideally suited to use for fast analysis of images at different scales, as demonstrated by the Viola-Jones algorithm [1].

## Tips

Because the `integralFilter` function uses correlation for filtering, the filter is not rotated before computing the result.

## Input Arguments

### **intI**

Integral image. You can obtain the integral image, `intI`, by calling the `integralImage` function. The class for this value can be `double` or `single`.

### **H**

Filter object. You can obtain the filter object, `H`, by calling the `integralKernel` function.

## Output Arguments

### J

Filtered image. The filtered image, J, returns only the parts of correlation that are computed without padding. This results in `size(J) = size(intI) - H.Size` for an upright filter, and `size(J) = size(intI) - H.Size - [0 1]` for a rotated filter. This function uses correlation for filtering.

## Examples

### Blur an Image Using an Average Filter

Read and display the input image.

```
I = imread('pout.tif');  
imshow(I);
```





Compute the integral image.

```
intImage = integralImage(I);
```

Apply a 7-by-7 average filter.

```
avgH = integralKernel([1 1 7 7], 1/49);  
J = integralFilter(intImage, avgH);
```

Cast the result back to the same class as the input image.

```
J = uint8(J);  
figure  
imshow(J);
```



#### Find Vertical and Horizontal Edges in Image

Construct Haar-like wavelet filters to find vertical and horizontal edges in an image.

Read the input image and compute the integral image.

```
I = imread('pout.tif');  
intImage = integralImage(I);
```

Construct Haar-like wavelet filters. Use the dot notation to find the vertical filter from the horizontal filter.

```
horiH = integralKernel([1 1 4 3; 1 4 4 3],[-1, 1]);  
vertH = horiH.'
```

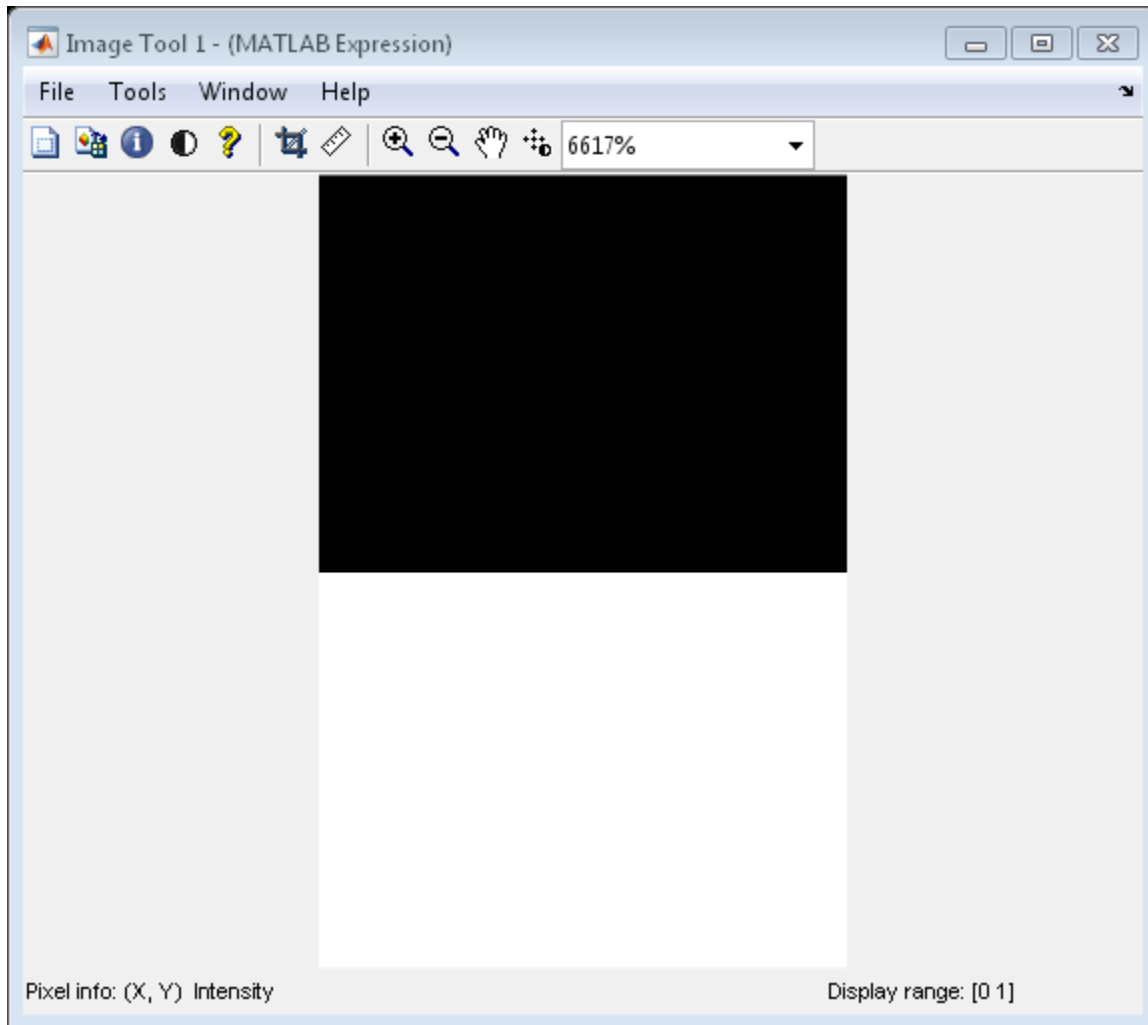
```
vertH =
```

```
integralKernel with properties:
```

```
  BoundingBoxes: [2x4 double]  
    Weights: [-1 1]  
  Coefficients: [4x6 double]  
    Center: [2 3]  
    Size: [4 6]  
  Orientation: 'upright'
```

Display the horizontal filter.

```
imtool(horiH.Coefficients, 'InitialMagnification','fit');
```



Compute the filter responses.

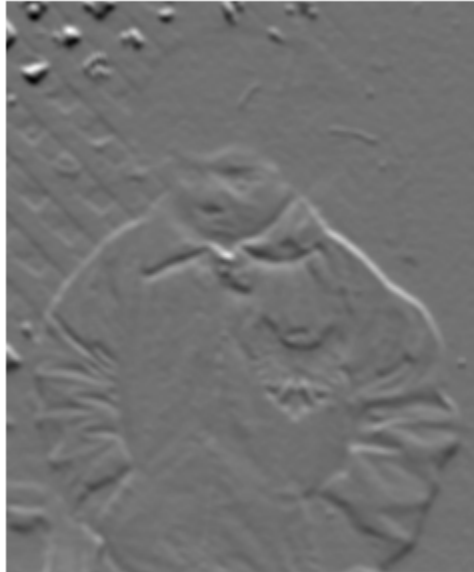
```
horiResponse = integralFilter(intImage, horiH);  
vertResponse = integralFilter(intImage, vertH);
```

Display the results.

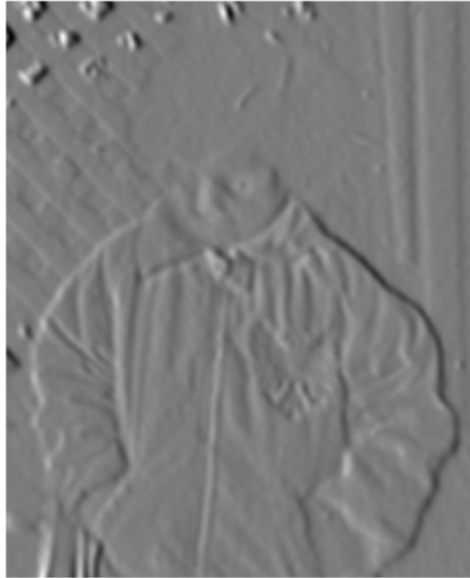
```
figure;
```

```
imshow(horiResponse,[]);  
title('Horizontal edge responses');  
figure;  
imshow(vertResponse,[]);  
title('Vertical edge responses');
```

**Horizontal edge responses**



#### Vertical edge responses



#### Compute a Rotated Edge Response Using Integral Filter

Read the input image.

```
I = imread('pout.tif');
```

Compute 45 degree edge responses of the image.

```
intImage = integralImage(I, 'rotated');  
figure;  
imshow(I);  
title('Original Image');
```

**Original Image**

Construct 45 degree rotated Haar-like wavelet filters.

```
rotH = integralKernel([2 1 2 2;4 3 2 2],[1 -1], 'rotated');  
rotHTrans = rotH.'
```

Visualize the filter rotH.

```
figure;  
imshow(rotH.Coefficients, [], 'InitialMagnification', 'fit');
```



Compute filter responses.

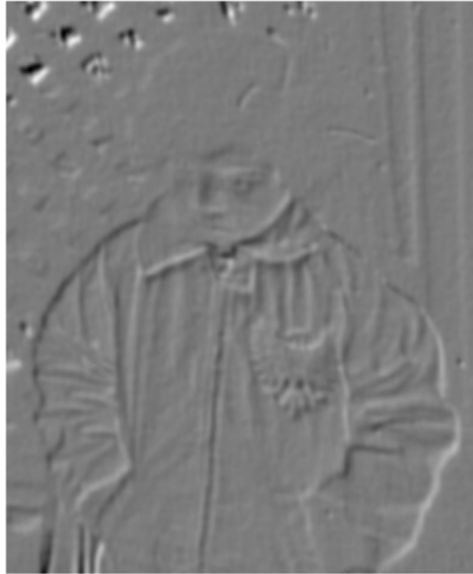
```
rothResponse = integralFilter(intImage,roth);  
rothTransResponse = integralFilter(intImage,rothTrans);
```

Display results.

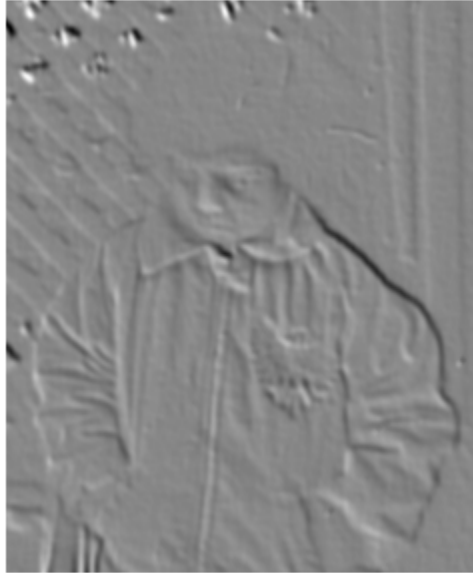
```
figure;  
imshow(rothResponse, []);  
title('Response for SouthWest-NorthEast edges');  
figure;  
imshow(rothTransResponse, []);  
title('Response for NorthWest-SouthEast edges');
```



**Response for SouthWest-NorthEast edges**



### Response for NorthWest-SouthEast edges



- “Compute an Integral Image” on page 3-274

### References

- [1] Viola, Paul and Michael J. Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features”, *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001. Volume: 1, pp.511–518.

### See Also

`integralKernel` | `cumsum` | `integralImage`

**Introduced in R2012a**

# integrallImage

Integral image

## Syntax

```
J = integrallImage(I)
J = integrallImage(I,orientation)
```

## Description

`J = integrallImage(I)` computes an integral image of the input intensity image, `I`. The function zero-pads the top and left side of the output integral image, `J`.

`J = integrallImage(I,orientation)` computes the integral image with the specified orientation.

An *integral image* lets you rapidly calculate summations over image subregions. Use of integral images was popularized by the Viola-Jones algorithm [1]. Integral images facilitate summation of pixels and can be performed in constant time, regardless of the neighborhood size.

### Code Generation Support:

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### **I** — Intensity image

Intensity image, specified as an  $M$ -by- $N$  grayscale image. This value can be any numeric class.

### **orientation** — Image orientation

'upright' (default) | 'rotated'

Image orientation, specified as 'upright' or 'rotated'. If you set the orientation to 'rotated', `integrallImage` returns the integral image for computing sums over

rectangles rotated by 45 degrees. To facilitate easy computation of pixel sums along all image boundaries, the output integral images are padded as follows:

Upright integral image — Zero-padded on top and left, resulting in `size(J) = size(I) + 1`

Rotated integral image — Zero-padded at the top, left, and right, resulting in `size(J) = size(I) + [1 2]`

## Output Arguments

### J

Integral image. The function zero-pads the top and left side of the integral image. The class of the output is `double`. The resulting size of the output integral image equals:

`size(J) = size(I) + 1`

Such sizing facilitates easy computation of pixel sums along all image boundaries. The integral image, J, is essentially a padded version of the value `cumsum(cumsum(I,2))`.

## Examples

### Compute an Integral Image

Compute the integral image and use it to compute the sum of pixels over a rectangular region of an intensity image.

Create an image matrix.

```
I = magic(5)
```

```
I =
```

```
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Define rectangular region as `[startingRow, startingColumn, endingRow, endingColumn]`.

```
[sR sC eR eC] = deal(1,3,2,4);
```

Compute the sum over the region using the integral image.

```
J = integralImage(I);
regionSum = J(eR+1,eC+1) - J(eR+1,sC) - J(sR,eC+1) + J(sR,sC)
```

```
regionSum =
```

```
30
```

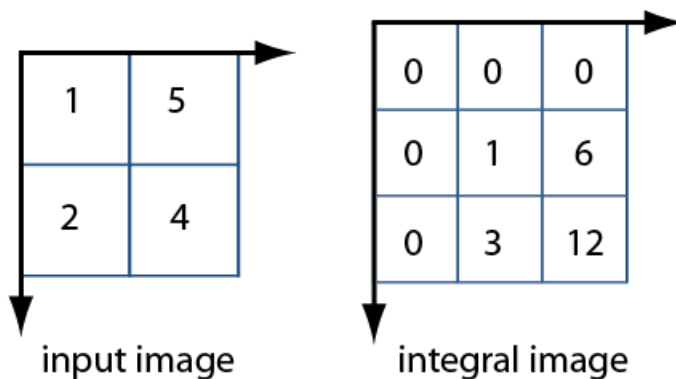
- “Blur an Image Using an Average Filter”
- “Find Vertical and Horizontal Edges in Image”

## More About

### Algorithms

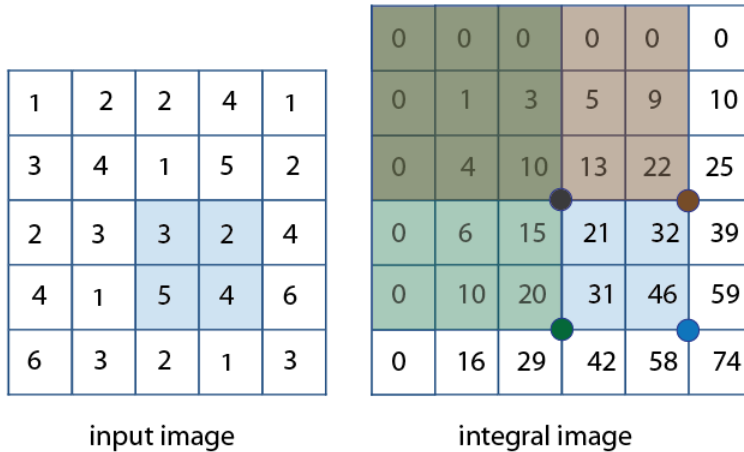
### How Integral Image Summation Works

An integral image helps you rapidly calculate summations over image subregions. Every pixel in an integral image is the summation of the pixels above and to the left of it.



To calculate the summation of a subregion of an image, you can use the corresponding region of its integral image. For example, in the input image below, the summation of the shaded region becomes a simple calculation using four reference values of the rectangular region in the corresponding integral image. The calculation becomes,  $46 - 22$

–  $20 + 10 = 14$ . The calculation subtracts the regions above and to the left of the shaded region. The area of overlap is added back to compensate for the double subtraction.



In this way, you can calculate summations in rectangular regions rapidly, irrespective of the filter size.

## References

- [1] Viola, Paul and Michael J. Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features”, *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001. Volume: 1, pp.511–518.

## See Also

`integralKernel` | `cumsum` | `integralFilter`

Introduced in R2012a

# insertMarker

Insert markers in image or video

## Syntax

```
RGB = insertMarker(I,position)
RGB = insertMarker(I,position,marker)
RGB = insertMarker(___,Name,Value)
```

## Description

`RGB = insertMarker(I,position)` returns a truecolor image with inserted plus (+) markers. The input image, `I`, can be either a truecolor or grayscale image. You draw the markers by overwriting pixel values. The input `position` can be either an  $M$ -by-2 matrix of  $M$  number of `[x y]` pairs or a `cornerPoints` object.

`RGB = insertMarker(I,position,marker)` returns a truecolor image with the `marker` type of markers inserted.

`RGB = insertMarker(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Compile-time constant input: `marker`

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Draw Markers on an Image

Read the image.

```
I = imread('peppers.png');
```

Insert a plus (+) marker.

```
RGB = insertMarker(I,[147 279]);
```

Draw four x-marks.

```
pos = [120 248;195 246;195 312;120 312];  
color = {'red','white','green','magenta'};  
RGB = insertMarker(RGB,pos,'x','color',color,'size',10);
```

Display the image.

```
imshow(RGB);
```



- “Insert a Circle and Filled Shapes on an Image” on page 3-290



- “Insert Numbers and Strings on an Image” on page 3-299

## Input Arguments

### **I** — Input image

*M*-by-*N*-by-3 truecolor | *M*-by-*N* 2-D grayscale image

Input image, specified in truecolor or 2-D grayscale.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

### **position** — Position of marker

*M*-by-2 matrix | vector

Position of marker, specified as either an *M*-by-2 matrix of *M* number of [*x* *y*] pairs or a `cornerPoints` object. The center positions for the markers are defined by either the [*xy*] pairs of the matrix or by the `position.Location` property of the `cornerPoints` object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **marker** — Type of marker

'plus' (default) | character string

Type of marker, specified as a string. The string can be full text or corresponding symbol.

String	Symbol String
'circle'	'o'
'x-mark'	'x'
'plus'	'+'
'star'	'*'
'square'	's'

Data Types: `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: `'Color', 'yellow'` specifies yellow for the marker color.

#### **'Size' — Size of marker**

3 (default) | scalar value

Size of marker in pixels, specified as the comma-separated pair consisting of `'Size'` and a scalar value in the range [1, inf).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **'Color' — Marker color**

'green' (default) | character string | cell array of character strings | vector | matrix

Marker color, specified as the comma-separated pair consisting of `'Color'` and either a string, cell array of strings, vector, or matrix. You can specify a different color for each marker or one color for all markers.

To specify a color for each marker, set `Color` to a cell array of color strings or an  $M$ -by-3 matrix of  $M$  number of RGB (red, green, and blue) color values.

To specify one color for all markers, set `Color` to either a color string or an [R G B] vector. The [R G B] vector contains the red, green, and blue values.

Supported color strings are: `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, `'black'`, and `'white'`.

Data Types: `cell` | `char` | `uint8` | `uint16` | `int16` | `double` | `single`

## Output Arguments

### **RGB — Output image**

$M$ -by- $N$ -by-3 truecolor

Output image, returned as a truecolor image.

### **See Also**

`cornerPoints` | `insertObjectAnnotation` | `insertShape` | `insertText`

**Introduced in R2013a**

## insertObjectAnnotation

Annotate truecolor or grayscale image or video stream

### Syntax

```
RGB = insertObjectAnnotation(I,shape,position,label)
RGB = insertObjectAnnotation(I,shape,position,label,Name,Value)

insertObjectAnnotation(I,'rectangle',position,label)
insertObjectAnnotation(I,'circle',position,label)
```

### Description

`RGB = insertObjectAnnotation(I,shape,position,label)` returns a truecolor image annotated with `shape` and `label` at the location specified by `position`.

`RGB = insertObjectAnnotation(I,shape,position,label,Name,Value)` uses additional options specified by one or more `Name, Value` pair arguments.

`insertObjectAnnotation(I,'rectangle',position,label)` inserts rectangles and labels at the location indicated by the position matrix.

`insertObjectAnnotation(I,'circle',position,label)` inserts circles and corresponding labels at the location indicated by the position matrix.

#### **Code Generation Support:**

Supports Code Generation: Yes

Limitation: Input image must be bounded, see “Specify Variable-Size Data Without Dynamic Memory Allocation”

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### **Annotate Image with Numbers and Strings**

Read image.

```
I = imread('board.tif');
```

Create labels of floating point numbers. The floating point numbers relate to confidence value labels.

```
label_str = cell(3,1);  
conf_val = [85.212 98.76 78.342];  
for ii=1:3  
    label_str{ii} = ['Confidence: ' num2str(conf_val(ii),'%0.2f') '%'];  
end
```

Set the position for the rectangles as [x y width height].

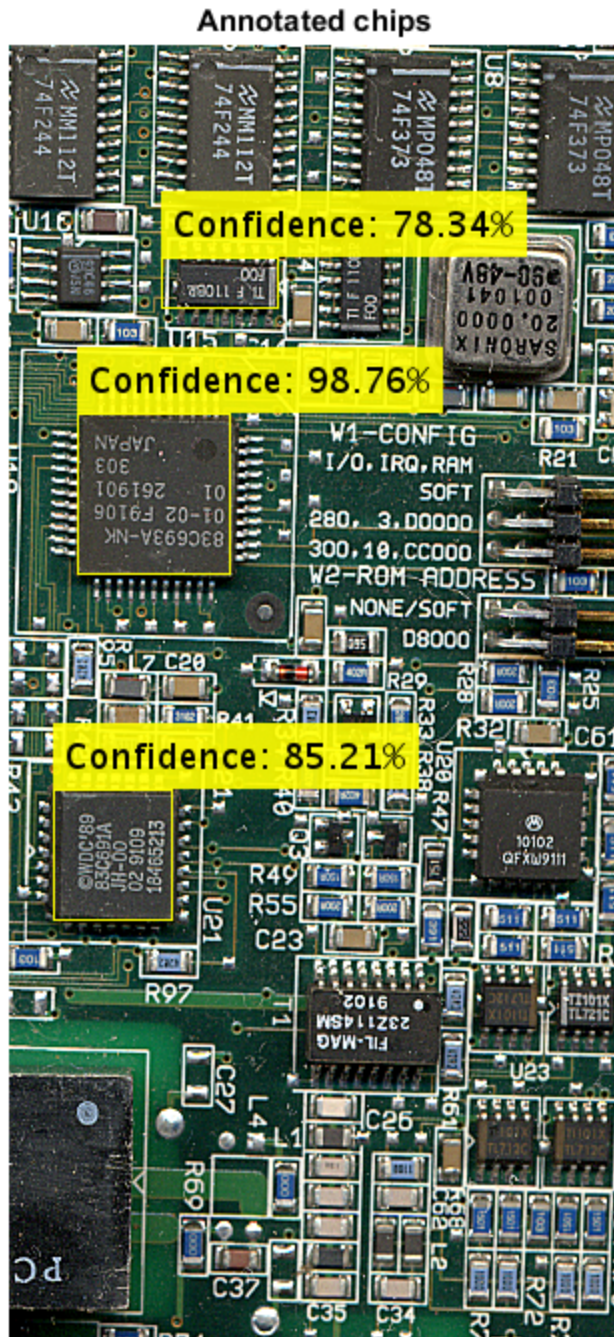
```
position = [23 373 60 66;35 185 77 81;77 107 59 26];
```

Insert the labels.

```
RGB = insertObjectAnnotation(I,'rectangle',position,label_str,'TextBoxOpacity',0.9,'For
```

Display the annotated image.

```
figure  
imshow(RGB)  
title('Annotated chips');
```



Read image.

```
I = imread('coins.png');
```

Set positions for the circles. The first two values represents the center at (x,y) and the third value is the radius.

```
position = [96 146 31;236 173 26];
```

Set the label to display the integers 5 and 10 (U.S. cents).

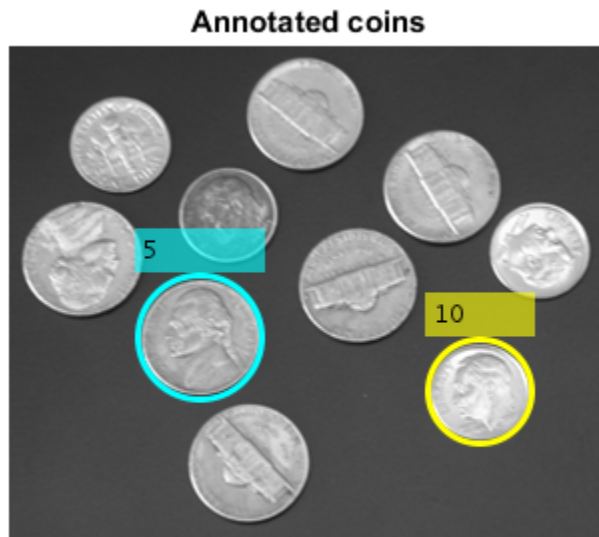
```
label = [5 10];
```

Insert the annotations.

```
RGB = insertObjectAnnotation(I, 'circle', position, label, 'LineWidth', 3, 'Color', {'cyan', 'y
```

Display.

```
figure  
imshow(RGB)  
title('Annotated coins');
```



## Input Arguments

### **I** — Truecolor or grayscale image

*M*-by-*N*-by-3 truecolor | *M*-by-*N* 2-D grayscale image

Truecolor or grayscale image, specified as an image or video stream. The input image can be either an *M*-by-*N*-by-3 truecolor or a *M*-by-*N* 2-D grayscale image.

Data Types: double | single | uint8 | uint16 | int16

### **shape** — Rectangle or circle annotation

'rectangle' | 'circle'

Rectangle or circle annotation, specified as a string indicating the annotation shape.

Data Types: char



**position — Location and size of the annotation shape***M*-by-3 matrix | *M*-by-4 matrix

Location and size of the annotation shape, specified as an *M*-by-3 or *M*-by-4 matrix.

When you specify a rectangle, the position input matrix must be an *M*-by-4 matrix. Each row, *M*, specifies a rectangle as a four-element vector, [*x y width height*]. The elements, *x* and *y*, indicate the upper-left corner of the rectangle, and the *width* and *height* specify the size.

When you specify a circle, the position input matrix must be an *M*-by-3 matrix, where each row, *M*, specifies a three-element vector [*x y r*]. The elements, *x* and *y*, indicate the center of the circle and *r* specifies the radius.

Example: `position = [50 120 75 75]`

A rectangle with top-left corner located at *x*=50, *y*=120, with a width and height of 75 pixels.

Example: `position = [96 146 31]`

A circle with center located at *x*=96, *y*=146 and a radius of 31 pixels.

Example: `position = [23 373 60 66;35 185 77 81;77 107 59 26]`

Location and size for three rectangles.

**label — A string label to associate with a shape**

numeric scalar | numeric vector | ASCII string | cell array of ASCII strings

A string label to associate with a shape, specified as a numeric vector or a cell array. The input can be a numeric vector of length *M*. It can also be a cell array of ASCII strings of length *M*, where *M* is the number of shape positions. You can also specify a single numeric scalar or string label for all shapes.

Example: `label = [5 10]`, where the function marks the first shape with the label, 5, and the second shape with the label, 10.

**Name-Value Pair Arguments**

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

Example: `'Color', 'white'` sets the color for the label text box to white.

#### **'Font' — Font face of text**

`'LucidaSansRegular'` (default) | string

Font face of text, specified as the comma-separated pair consisting of `'Font'` and a string. The font face must be one of the available truetype fonts installed on your system. To get a list of available fonts on your system, type `listTrueTypeFonts` at the MATLAB command prompt.

Data Types: char

#### **'FontSize' — Label text font size**

12 (default) | integer in the range of [8 72]

Label text font size, specified as the comma-separated pair consisting of `'FontSize'` and an integer corresponding to points in the range of [8 72].

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### **'LineWidth' — Shape border line width**

1 (default)

Shape border line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive scalar integer in pixels.

#### **'Color' — Color for shape and corresponding label text box**

`'yellow'` (default) | color string | [R G B] vector | cell array |  $M$ -by-3 matrix

Color for shape and for corresponding label text box, specified as the comma-separated pair consisting of `'Color'` and either a string, an [R G B] vector, a cell array, or an  $M$ -by-3 matrix.

To specify one color for all shapes, set this parameter to either a string or an [R G B] vector. To specify a color for each of the  $M$  shapes, set this parameter to a cell array of  $M$  strings. Alternatively, you can specify an  $M$ -by-3 matrix of RGB values for each annotation. RGB values must be in the range of the input image data type.

Supported color strings are: `'blue'`, `'green'`, `'cyan'`, `'red'`, `'magenta'`, `'black'`, and `'white'`.

Data Types: char | uint8 | uint16 | int16 | double | single | cell

**'TextColor' — Color of text in text label**

'black' (default) | color string | [R G B] vector | cell array |  $M$ -by-3 matrix

Color of text in text label, specified as the comma-separated pair consisting of 'TextColor' and either a string, an [R G B] vector, a cell array, or an  $M$ -by-3 matrix. To specify one color for all text, set this parameter to either a string or an [R G B] vector. To specify a color for each of the  $M$  text labels, set this parameter to a cell array of  $M$  strings. Alternatively, you can specify an  $M$ -by-3 matrix of RGB values for each annotation. RGB values must be in the range of the input image data type.

Supported color strings are: 'blue', 'green', 'cyan', 'red', 'magenta', 'yellow', and 'white'.

Data Types: char | uint8 | uint16 | int16 | double | single | cell

**'TextBoxOpacity' — Opacity of text label box background**

0.6 (default) | range of [0 1]

Opacity of text label box background, specified as the comma-separated pair consisting of 'TextBoxOpacity' and a scalar defining the opacity of the background of the label text box. Specify this value in the range of 0 to 1.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

**RGB — Truecolor or grayscale image with annotation**

$M$ -by- $N$ -by-3 truecolor

Truecolor image with annotation, returned as an image or video stream.

Data Types: double | single | uint8 | uint16 | int16

**See Also**

insertMarker | insertShape | insertText

**Introduced in R2012b**

# insertShape

Insert shapes in image or video

## Syntax

```
RGB = insertShape(I,shape,position)
RGB = insertShape( ____,Name,Value)
```

## Description

`RGB = insertShape(I,shape,position)` returns a truecolor image with `shape` inserted. The input image, `I`, can be either a truecolor or grayscale image. You draw the shapes by overwriting pixel values.

`RGB = insertShape( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Compile-time constant input: `shape` and `SmoothEdges`

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Insert a Circle and Filled Shapes on an Image

Read the image.

```
I = imread('peppers.png');
```

Draw a circle with a border line width of 5.

```
RGB = insertShape(I, 'circle', [150 280 35], 'LineWidth', 5);
```

Draw a filled triangle and a filled hexagon.

```
pos_triangle = [183 297 302 250 316 297];
pos_hexagon = [340 163 305 186 303 257 334 294 362 255 361 191];
RGB = insertShape(RGB, 'FilledPolygon', {pos_triangle, pos_hexagon}, 'Color', {'whi
```

**Display the image.**

```
imshow( RGB );
```



- “Draw Markers on an Image” on page 3-277
- “Insert Numbers and Strings on an Image” on page 3-299

## Input Arguments

**I** — Input image

*M*-by-*N*-by-3 truecolor | *M*-by-*N* 2-D grayscale image

Input image, specified in truecolor or 2-D grayscale.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

**shape — Type of shape**

character string

Type of shape, specified as a string. The string can be, 'Rectangle', 'FilledRectangle', 'Line', 'Polygon', 'FilledPolygon', 'Circle', or 'FilledCircle'.

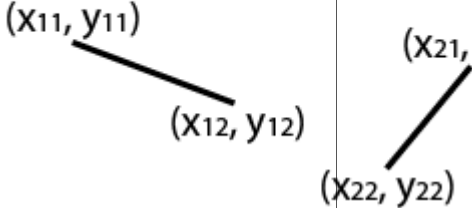
Data Types: `char`

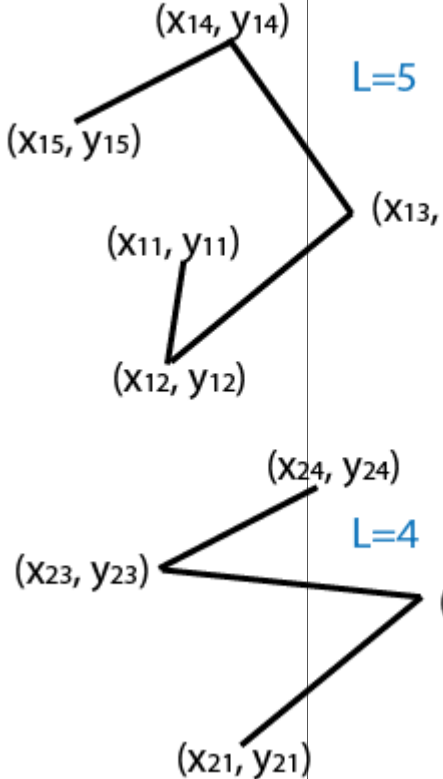
**position — Position of shape**

matrix | vector | cell array

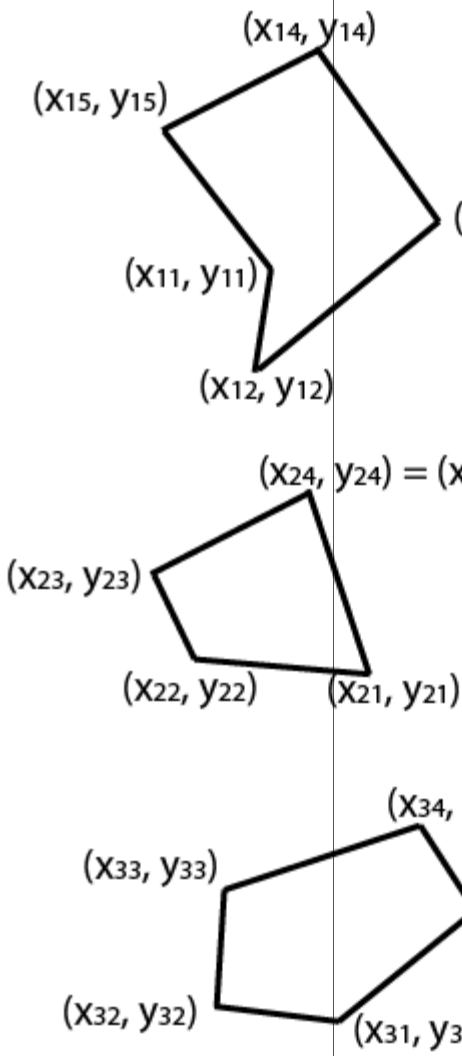
Position of shape, specified according to the type of shape, described in the table.

Shape	Position	Shape Drawn
'Rectangle' 'FilledRect	<p><math>M</math>-by-4 matrix where each row specifies a rectangle as <math>[x \ y \ width \ height]</math>.</p> $\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$	<p><u>For <math>M=2</math>:</u></p> <p>The diagram illustrates two rectangles. The first rectangle is positioned at coordinates <math>(x_1, y_1)</math> and has a width of <math>width_1</math> and a height of <math>height_1</math>. The second rectangle is positioned at coordinates <math>(x_2, y_2)</math> and has a width of <math>width_2</math> and a height of <math>height_2</math>.</p>

Shape	Position	Shape Drawn
'Line'	<p>For one or more disconnected lines, an <math>M</math>-by-4 matrix, where each four-element vector <math>[x_1, y_1, x_2, y_2]</math>, describe a line with endpoints, <math>[x_1, y_1]</math> and <math>[x_2, y_2]</math>.</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$	<p><b><u>For M=2:</u></b></p> 

Shape	Position	Shape Drawn
	<p>For one or more line segments, an <math>M</math>-by-<math>2L</math> matrix, where each row is a vector representing a polyline with <math>L</math> number of vertices.</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>The polyline always contains <math>(L-1)</math> number of segments because the first and last vertex points do not connect. For lines with fewer segments, repeat the ending coordinates to fill the matrix.</p> <p>You can also specify the shapes as a cell array of <math>M</math> vectors.</p> <p><math>\{[x_{11},y_{11},x_{12},y_{12},\dots,x_{1p},y_{1p}],</math>  <math>[x_{21},y_{21},x_{22},y_{22},\dots,x_{2q},y_{2q}], \dots</math>  <math>[x_{M1},y_{M1},x_{M2},y_{M2},\dots,x_{Mr},y_{Mr}]\}</math></p> <p><math>p</math>, <math>q</math>, and <math>r</math> specify the number of vertices.</p>	<p><u>For <math>M=2</math>:</u></p>  <p>The first diagram shows a closed shape with 5 vertices: <math>(X_{14}, Y_{14})</math>, <math>(X_{15}, Y_{15})</math>, <math>(X_{13}, Y_{13})</math>, <math>(X_{12}, Y_{12})</math>, and <math>(X_{11}, Y_{11})</math>. The label <math>L=5</math> is shown to the right.</p> <p>The second diagram shows a closed shape with 4 vertices: <math>(X_{24}, Y_{24})</math>, <math>(X_{23}, Y_{23})</math>, <math>(X_{21}, Y_{21})</math>, and <math>(X_{22}, Y_{22})</math>. The label <math>L=4</math> is shown to the right.</p>



Shape	Position	Shape Drawn
<p>'Polygon' 'FilledPoly</p>	<p>An <math>M</math>-by-<math>2L</math> matrix, where each row represents a polygon with <math>L</math> number of vertices. Each row of the matrix corresponds to a polygon. For polygons with fewer segments, repeat the ending coordinates to fill the matrix.</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>You can also specify the shapes as a cell array of <math>M</math> vectors:</p> <p><math>\{[x_{11}, y_{11}, x_{12}, y_{12}, \dots, x_{1p}, y_{1p}],</math>  <math>[x_{21}, y_{21}, x_{22}, y_{22}, \dots, x_{2q}, y_{2q}], \dots</math>  <math>[x_{M1}, y_{M1}, x_{M2}, y_{M2}, \dots, x_{Mr}, y_{Mr}]\}</math></p> <p><math>p</math>, <math>q</math>, and <math>r</math> specify the number of vertices.</p>	<p><u>For <math>M=3</math>: <math>L=5</math></u></p> 

Shape	Position	Shape Drawn
'Circle' 'FilledCircle'	<p>An <math>M</math>-by-3 matrix, where each row is a vector specifying a circle as <math>[x \ y \ radius]</math>. The <math>[x \ y]</math> coordinates represent the center of the circle.</p> $\begin{bmatrix} x_1 & y_1 & radius_1 \\ x_2 & y_2 & radius_2 \\ \vdots & \vdots & \vdots \\ x_M & y_M & radius_M \end{bmatrix}$	<p><u>For <math>M=2</math>:</u></p>

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### Name-Value Pair Arguments

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

Example: `'Color', 'yellow'` specifies yellow for the shape color.

#### 'LineWidth' — Shape border line width

1 (default) | positive scalar integer

Shape border line width, specified in pixels, as a positive scalar integer. This property only applies to the 'Rectangle', 'Line', 'Polygon', or 'Circle' shapes.

Data Types: uint8 | uint16 | int16 | double | single

**'Color' — Shape color**

'yellow' (default) | character string | cell array of character strings | [R G B] vector |  $M$ -by-3 matrix

Shape color, specified as the comma-separated pair consisting of 'Color' and either a string, cell array of strings, vector, or matrix. You can specify a different color for each shape, or one color for all shapes.

To specify a color for each shape, set Color to a cell array of color strings or an  $M$ -by-3 matrix of  $M$  number of RGB (red, green, and blue) color values.

To specify one color for all shapes, set Color to either a color string or an [R G B] vector. The [R G B] vector contains the red, green, and blue values.

Supported color strings are: 'blue', 'green', 'red', 'cyan', 'magenta', 'black', 'black', and 'white'.

Data Types: cell | char | uint8 | uint16 | int16 | double | single

**'Opacity' — Opacity of filled shape**

0.6 (default) | range of [0 1]

Opacity of filled shape, specified as the comma-separated pair consisting of 'Opacity' and a scalar value in the range [0 1]. The Opacity property applies for the FilledRectangle, FilledPolygon, and FilledCircle shapes.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**'SmoothEdges' — Smooth shape edges**

true (default) | false

Smooth shape edges, specified as the comma-separated pair consisting of 'SmoothEdges' and a logical value of true or false. A true value enables an anti-aliasing filter to smooth shape edges. This value applies only to nonrectangular shapes. Enabling anti-aliasing requires additional time to draw the shapes.

Data Types: logical

## Output Arguments

**RGB — Output image**

$M$ -by- $N$ -by-3 truecolor

Output image, returned as a truecolor image.

### **See Also**

`insertMarker` | `insertObjectAnnotation` | `insertText`

**Introduced in R2014a**

# insertText

Insert text in image or video

## Syntax

```
RGB = insertText(I,position,textString)
RGB = insertText(I,position,numericValue)
RGB = insertText( ____,Name,Value)
```

## Description

`RGB = insertText(I,position,textString)` returns a truecolor image with text strings inserted. The input image, `I`, can be either a truecolor or grayscale image. The text strings are drawn by overwriting pixel values.

`RGB = insertText(I,position,numericValue)` returns a truecolor image with numeric values inserted.

`RGB = insertText( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Compile-time constant input: `Font`, `FontSize`

Supports code generation: Yes

Supports non-ASCII characters: No

Generates platform-dependent library: No

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Insert Numbers and Strings on an Image

Read the image.

```
I = imread('board.tif');
```

Create texts that contain fractions.

```
text_str = cell(3,1);  
conf_val = [85.212 98.76 78.342];  
for ii=1:3  
    text_str{ii} = ['Confidence: ' num2str(conf_val(ii), '%0.2f') '%'];  
end
```

Define the positions and colors of the text boxes.

```
position = [23 373;35 185;77 107];  
box_color = {'red','green','yellow'};
```

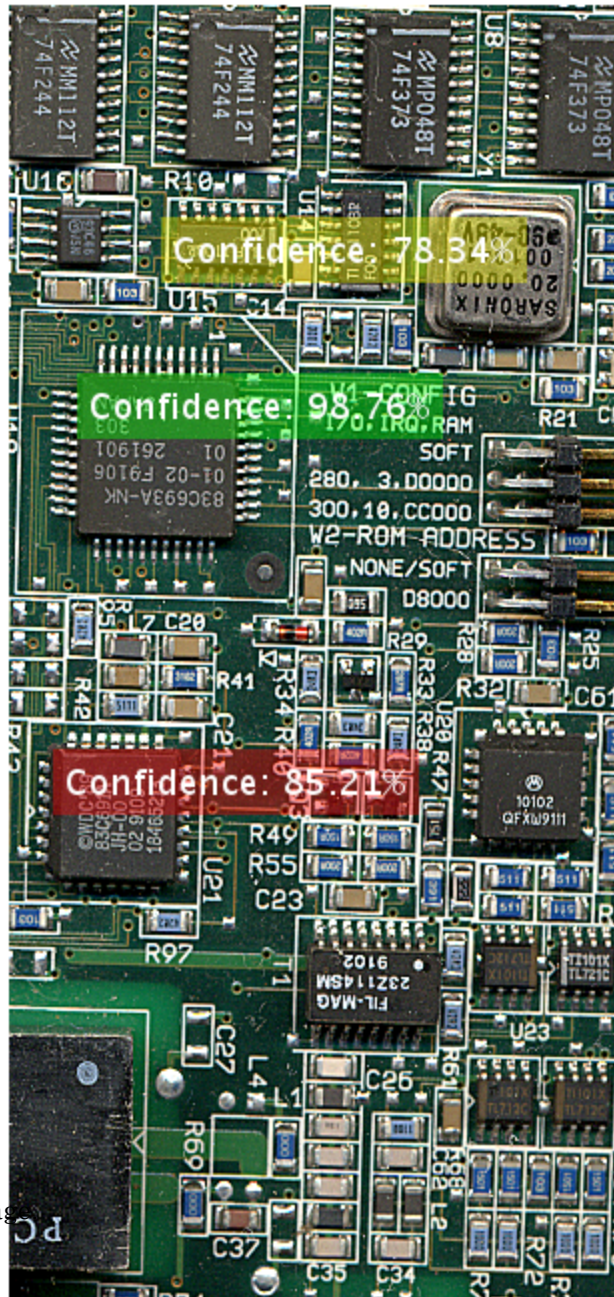
Insert the text with new font size, box color, opacity, and text color.

```
RGB = insertText(I,position,text_str,'FontSize',18,'BoxColor',box_color,'BoxOpacity',0
```

Display the image.

```
figure  
imshow(RGB)  
title('Board');
```

Board



Read the image

Insert Numeric Text on Image

```
I = imread('peppers.png');
```

Define the ( $x\_y\_$ ) position for the text and the value.

```
position = [1 50; 100 50];  
value = [555 pi];
```

Insert text using the bottom-left as the anchor point.

```
RGB = insertText(I,position,value,'AnchorPoint','LeftBottom');
```

Display the image with the numeric text inserted.

```
figure  
imshow(RGB),title('Numeric values');
```

**Numeric values**





Display non-ASCII character (U+014C)

```
OWithMacron=native2unicode([hex2dec('C5') hex2dec('8C')], 'UTF-8');  
RGB = insertText(RGB,[256 50],OWithMacron, 'Font', 'LucidaBrightRegular', 'BoxColor', 'w')
```

Display the image with the numeric text inserted.

```
figure  
imshow(RGB), title('Numeric values');
```



- “Draw Markers on an Image” on page 3-277
- “Insert a Circle and Filled Shapes on an Image” on page 3-290

## Input Arguments

### **I** — Input image

*M*-by-*N*-by-3 truecolor | *M*-by-*N* 2-D grayscale image

Input image, specified as *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-D grayscale image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

### **textString** — Unicode text string

text string | cell array of text strings

Unicode text string, specified as a text string or a cell array of Unicode text strings. The length of the cell array must equal the number of rows in the `position` matrix. If you specify a single text string, that string is used for all positions in the `position` matrix. Most unicode fonts contain ASCII characters. You can display non-English and English characters, including English numeric values, with a single font.

Data Types: `char`

### **numericValue** — Numeric value text

scalar | vector

Numeric value text, specified as a scalar or a vector. If you specify a scalar value, that value is used for all positions. The vector length must equal the number of rows in the `position` matrix. Numeric values are converted to a string using the `sprintf` format `'%0.5g'`.

Data Types: `char`

### **position** — Position of inserted text

vector | matrix

Position of inserted text, specified as a vector or an *M*-by-2 matrix of [*x* *y*] coordinates. Each row represents the [*x* *y*] coordinate for the `AnchorPoint` of the text bounding box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

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

Example: `'AnchorPoint','LeftTop'`

### 'Font' — Font face of text

`'LucidaSansRegular'` (default) | string

Font face of text, specified as the comma-separated pair consisting of 'Font' and a string. The font face must be one of the available truetype fonts installed on your system. To get a list of available fonts on your system, type `listTrueTypeFonts` at the MATLAB command prompt.

Data Types: char

### 'FontSize' — Font size

12 (default) | positive integer in the range [1,200]

Font size, specified as the comma-separated pair consisting of 'FontSize' and a positive integer in the range [1,200].

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### 'TextColor' — Text color

'black' (default) | character string | cell array of character strings | [R G B] vector |  $M$ -by-3 matrix

Text color, specified as the comma-separated pair consisting of 'TextColor' and a string, cell array of strings, vector, or matrix. You can specify a different color for each string or one color for all strings.

- To specify a color for each text string, set `TextColor` to a cell array of  $M$  number of color strings. Or, you can set it to an  $M$ -by-3 matrix of RGB string color values.
- To specify one color for all text strings, set `TextColor` to either a color string or an [R G B] vector of red, green, and blue values.
- RGB values must be in the range of the image data type. Supported color strings are: 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', and 'white'.

Data Types: cell | char | uint8 | uint16 | int16 | double | single

#### 'BoxColor' — Text box color

'yellow' (default) | character string | cell array of character strings | [R G B] vector |  $M$ -by-3 matrix

Text box color, specified as the comma-separated pair consisting of 'BoxColor' and a string, cell array of strings, vector, or matrix. You can specify a different color for each text box or one color for all the boxes.

- To specify a color for each text box, set **BoxColor** to a cell array of  $M$  number of color strings. Or, you can set it to an  $M$ -by-3 matrix of  $M$  number of RGB (red, green, and blue) string color values.
- To specify one color for all the text boxes, set **BoxColor** to either a color string or an [R G B] vector. The [R G B] vector contains the red, green, and blue values.
- RGB values must be in the range of the image data type. Supported color strings are: 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', and 'white'.

Data Types: cell | char | uint8 | uint16 | int16 | double | single

#### 'BoxOpacity' — Opacity of text box

0.6 (default) | scalar value in the range of [0 1]

Opacity of text box, specified as the comma-separated pair consisting of 'BoxOpacity' and a scalar value in the range [0,1]. A value of 0 corresponds to a fully transparent text box, or no box. A value of 1 corresponds to a fully opaque text box.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 'AnchorPoint' — Text box reference point

'LeftTop' (default) | 'LeftCenter' | 'LeftBottom' | 'CenterTop' | 'Center' | 'CenterBottom' | 'RightTop' | 'RightCenter' | 'RightBottom'

Text box reference point, specified as the comma-separated pair consisting of 'AnchorPoint' and a string value. The anchor point defines a relative location on the text box. You can position the text box by placing its anchor point at the  $[x,y]$  coordinate defined by the corresponding **position** for the text. For example, to place the center of the text box to be at the  $[x,y]$  coordinate you specified with the **position** input, then set **AnchorPoint** to **Center**.

Supported positions are **LeftTop**, **LeftCenter**, **LeftBottom**, **CenterTop**, **Center**, **CenterBottom**, **RightTop**, **RightCenter**, and **RightBottom**.

Data Types: char

## Output Arguments

### **RGB — Output image**

*M*-by-*N*-by-3 truecolor image

Output image, returned as an *M*-by-*N*-by-3 truecolor image with the specified text inserted.

## Limitations

- If you do not see characters in the output image, it means that the font did not contain the character. Select a different font. To get a list of available fonts on your system, at the MATLAB prompt, type `listTrueTypeFonts`.
- Increasing the font size also increases the preprocessing time and memory usage.
- The `insertText` function does not work for certain composite characters. For example, you cannot insert text when the rendering of one glyph corresponding to a character code influences the position, shape, or size of the adjacent glyph.

## See Also

`insertMarker` | `insertObjectAnnotation` | `insertShape` | `listTrueTypeFonts`

**Introduced in R2013a**

## isEpipoleInImage

Determine whether image contains epipole

### Syntax

```
isIn = isEpipoleInImage(F, imageSize)
isIn = isEpipoleInImage(F', imageSize)
[isIn, epipole] = isEpipoleInImage( ___ )
```

### Description

`isIn = isEpipoleInImage(F, imageSize)` determines whether the first stereo image associated with the fundamental matrix `F` contains an epipole. `imageSize` is the size of the first image, and `is` is in the format returned by the function `size`.

`isIn = isEpipoleInImage(F', imageSize)` determines whether the second stereo image associated with the fundamental matrix `F'` contains an epipole.

`[isIn, epipole] = isEpipoleInImage( ___ )` also returns the epipole.

#### Code Generation Support:

Compile-time constant input: No restrictions

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

### Input Arguments

#### **F**

A 3-by-3 fundamental matrix computed from stereo images. `F` must be double or single. If  $P_1$  represents a point in the first image  $I_1$  that corresponds to  $P_2$ , a point in the second image  $I_2$ , then:

$$[P_2, 1] * F * [P_1, 1]' = 0$$

In computer vision, the fundamental matrix is a 3-by-3 matrix which relates corresponding points in stereo images. When two cameras view a 3-D scene from two distinct positions, there are a number of geometric relations between the 3-D points and their projections onto the 2-D images that lead to constraints between the image points. Two images of the same scene are related by epipolar geometry.

### imageSize

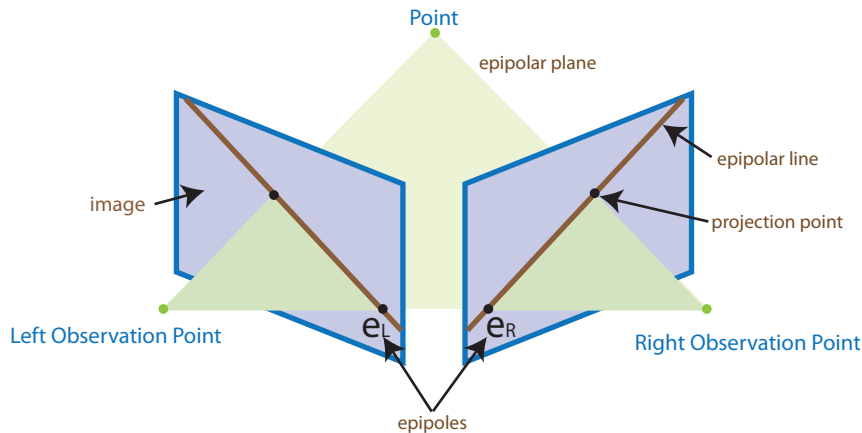
The size of the image, and in the format returned by the function `size`.

## Output Arguments

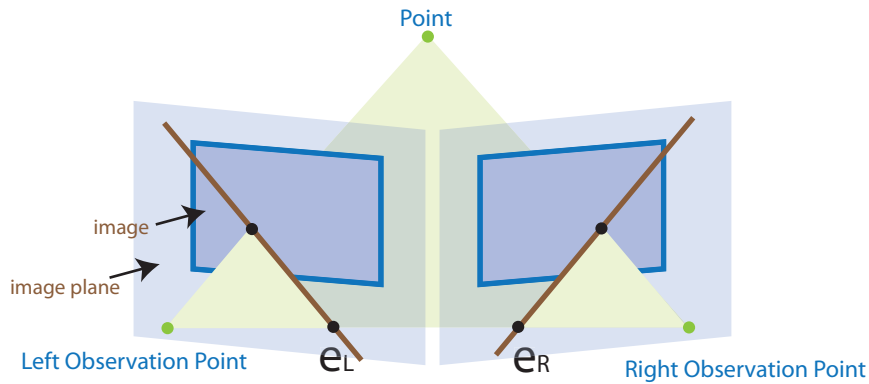
### isIn

Logical value set to `true` when the image contains an epipole, and set to `false` when the image does not contain an epipole.

When the image planes are at a great enough angle to each other, you can expect the epipole to be located in the image.



When the image planes are at a more subtle angle to each other, you can expect the epipole to be located outside of the image, (but still in the image plane).



## epipole

A 1-by-2 vector indicating the location of the epipole.

## Examples

### Determine Epipole Location in an Image

```
% Load stereo point pairs.
load stereoPointPairs
f = estimateFundamentalMatrix(matchedPoints1, matchedPoints2, 'NumTrials', 2000);
imageSize = [200 300];

% Determine whether the image contains epipole and epipole location.
[isIn,epipole] = isEpipoleInImage(f,imageSize)
```

```
isIn =
```

```
1
```

```
epipole =
```

```
256.2989 99.8517
```



### Determine Whether Image Contains Epipole

```
f = [0 0 1; 0 0 0; -1 0 0];  
imageSize = [200, 300];  
[isIn,epipole] = isEpipoleInImage(f',imageSize)
```

```
isIn =
```

```
0
```

```
epipole =
```

```
1.0e+308 *
```

```
0    1.7977
```

- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”

### See Also

[estimateFundamentalMatrix](#) | [epipolarLine](#) | [estimateUncalibratedRectification](#)

**Introduced in R2011a**

## isfilterseparable

Determine whether filter coefficients are separable

### Syntax

```
S = isfilterseparable(H)  
[S, HCOL, HROW] = isfilterseparable(H)
```

### Description

`S = isfilterseparable(H)` takes in the filter kernel  $H$  and returns 1 (true) when the filter is separable, and 0 (false) otherwise.

`[S, HCOL, HROW] = isfilterseparable(H)` uses the filter kernel,  $H$ , to return its vertical coefficients `HCOL` and horizontal coefficients `HROW` when the filter is separable. Otherwise, `HCOL` and `HROW` are empty.

### Input Arguments

#### **H**

$H$  numeric or logical, 2-D, and nonsparse.

### Output Arguments

#### **HCOL**

`HCOL` is the same data type as input  $H$  when  $H$  is either single or double floating point. Otherwise, `HCOL` becomes double floating point. If `S` is true, `HCOL` is a vector of vertical filter coefficients. Otherwise, `HCOL` is empty.

#### **HROW**

`HROW` is the same data type as input  $H$  when  $H$  is either single or double floating point. Otherwise, `HROW` becomes double floating point. If `S` is true, `HROW` is a vector of horizontal filter coefficients. Otherwise, `HROW` is empty.

**S**

Logical variable that is set to `true`, when the filter is separable, and `false`, when it is not.

## Examples

### Determine if Gaussian Filter is Separable

Determine if the Gaussian filter created using the `fspecial` function is separable.

Create a Gaussian filter.

```
twoDimensionalFilter = fspecial('gauss');
```

Test the filter.

```
[isseparable,hcol,hrow] = isfilterseparable(twoDimensionalFilter)
```

```
isseparable =
```

```
    1
```

```
hcol =
```

```
   -0.1065  
   -0.7870  
   -0.1065
```

```
hrow =
```

```
   -0.1065   -0.7870   -0.1065
```

## More About

### Separable two dimensional filters

*Separable two-dimensional filters* reflect the outer product of two vectors. Separable filters help reduce the number of calculations required.

A two-dimensional convolution calculation requires a number of multiplications equal to the *width* × *height* for each output pixel. The general case equation for a two-dimensional convolution is:

$$Y(m,n) = \sum_k \sum_l H(k,l)U(m-k,n-l)$$

If the filter  $H$  is separable then,

$$H(k,l) = H_{row}(k)H_{col}(l)$$

Shifting the filter instead of the image, the two-dimensional equation becomes:

$$Y(m,n) = \sum_k H_{row}(k) \sum_l H_{col}(l)U(m-k,n-l)$$

This calculation requires only (width + height) number of multiplications for each pixel.

#### Algorithms

The `isfilterseparable` function uses the singular value decomposition `svd` function to determine the rank of the matrix.

- MATLAB Central — Separable Convolution

#### See Also

2-D FIR Filter | `svd` | `rank`

**Introduced in R2006a**

# lineToBorderPoints

Intersection points of lines in image and image border

## Syntax

```
points = lineToBorderPoints(lines,imageSize)
```

## Description

`points = lineToBorderPoints(lines,imageSize)` computes the intersection points between one or more lines in an image with the image border.

### Code Generation Support:

Compile-time constant input: No restrictions

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Input Arguments

### **lines**

Line matrix. An  $M$ -by-3 matrix, where each row must be in the format,  $[A,B,C]$ . This matrix corresponds to the definition of the line:

$$A * x + B * y + C = 0.$$

$M$  represents the number of lines.

`lines` must be `double` or `single`.

### **imageSize**

Image size. This input must be in the format returned by the `size` function.

`imageSize` must be `double`, `single`, or `integer`.

## Output Arguments

### **points**

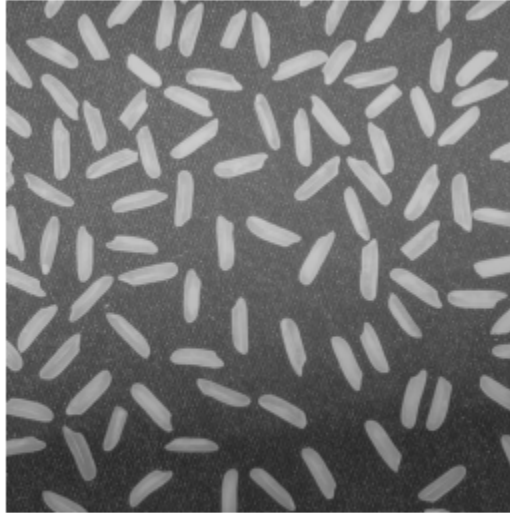
Output intersection points. An  $M$ -by-4 matrix. The function returns the matrix in the format of  $[x_1, y_1, x_2, y_2]$ . In this matrix,  $[x_1, y_1]$  and  $[x_2, y_2]$  are the two intersection points. When a line in the image and the image border do not intersect, the function returns  $[-1, -1, -1, -1]$ .

## Examples

### **Find Intersection Points Between a Line and Image Border**

Load and display an image.

```
I = imread('rice.png');  
figure;  
imshow(I);  
hold on;
```



Define a line with the equation,  $2 * x + y - 300 = 0$ .

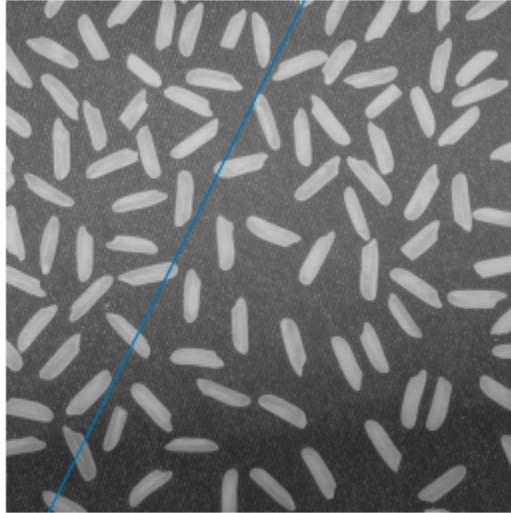
```
aLine = [2,1,-300];
```

Compute the intersection points of the line and the image border.

```
points = lineToBorderPoints(aLine,size(I))  
line(points([1,3]),points([2,4]));
```

```
points =
```

```
149.7500    0.5000    21.7500   256.5000
```



**See Also**

`vision.ShapeInserter` | `size` | `line` | `epipolarLine`

**Introduced in R2011a**



# matchFeatures

Find matching features

## Syntax

```
indexPairs = matchFeatures(features1,features2)
[indexPairs,matchmetric] = matchFeatures(features1,features2)
[indexPairs,matchmetric] = matchFeatures(features1,features2,
Name,Value)
```

## Description

`indexPairs = matchFeatures(features1,features2)` returns indices of the matching features in the two input feature sets. The input feature must be either `binaryFeatures` objects or matrices.

`[indexPairs,matchmetric] = matchFeatures(features1,features2)` also returns the distance between the matching features, indexed by `indexPairs`.

`[indexPairs,matchmetric] = matchFeatures(features1,features2, Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

- Generates platform-dependent library: For MATLAB host only when using the `Exhaustive` method.
- Generates portable C code for nonhost target only when using the `Exhaustive` method.
- Generates portable C code using a C++ compiler that links to user-provided OpenCV (Version 2.4.9) libraries when not using the `Exhaustive` method.

Compile-time constant input: `Method` and `Metric`

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Find Corresponding Interest Points Between Pair of Images

Find corresponding interest points between a pair of images using local neighborhoods and the Harris algorithm.

Read the stereo images.

```
I1 = rgb2gray(imread('viprectification_deskLeft.png'));  
I2 = rgb2gray(imread('viprectification_deskRight.png'));
```

Find the corners.

```
points1 = detectHarrisFeatures(I1);  
points2 = detectHarrisFeatures(I2);
```

Extract the neighborhood features.

```
[features1,valid_points1] = extractFeatures(I1,points1);  
[features2,valid_points2] = extractFeatures(I2,points2);
```

Match the features.

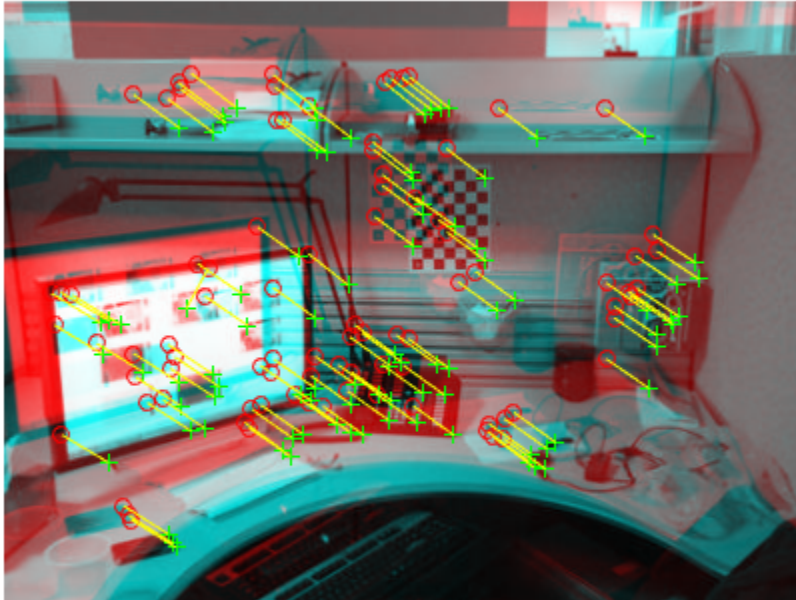
```
indexPairs = matchFeatures(features1,features2);
```

Retrieve the locations of the corresponding points for each image.

```
matchedPoints1 = valid_points1(indexPairs(:,1),:);  
matchedPoints2 = valid_points2(indexPairs(:,2),:);
```

Visualize the corresponding points. You can see the effect of translation between the two images despite several erroneous matches.

```
figure; showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);
```



### Find Corresponding Points Using SURF Features

Use the SURF local feature detector function to find the corresponding points between two images that are rotated and scaled with respect to each other.

Read the two images.

```
I1 = imread('cameraman.tif');  
I2 = imresize(imrotate(I1,-20),1.2);
```

Find the SURF features.

```
points1 = detectSURFFeatures(I1);  
points2 = detectSURFFeatures(I2);
```

Extract the features.

```
[f1,vpts1] = extractFeatures(I1,points1);  
[f2,vpts2] = extractFeatures(I2,points2);
```

Retrieve the locations of matched points.

```
indexPairs = matchFeatures(f1,f2) ;  
matchedPoints1 = vpts1(indexPairs(:,1));  
matchedPoints2 = vpts2(indexPairs(:,2));
```

Display the matching points. The data still includes several outliers, but you can see the effects of rotation and scaling on the display of matched features.

```
figure; showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);  
legend('matched points 1','matched points 2');
```



## Input Arguments

### **features1** — Feature set 1

binaryFeatures object |  $M_1$ -by- $N$  matrix

Features set 1, specified as a binaryFeatures object or an  $M_1$ -by- $N$  matrix. The matrix contains  $M_1$  features, and  $N$  corresponds to the length of each feature vector. You

can obtain the `binaryFeatures` object using the `extractFeatures` function with the fast retina keypoint (FREAK) or binary robust invariant scalable keypoints (BRISK) descriptor method.

#### **features2 — Feature set 2**

$M_2$ -by- $N$  matrix | `binaryFeatures` object

Features set 2, specified as a `binaryFeatures` object or an  $M_2$ -by- $N$  matrix. The matrix contains  $M_2$  features and  $N$  corresponds to the length of each feature vector. You can obtain the `binaryFeatures` object using the `extractFeatures` function with the fast retina keypoint (FREAK) or binary robust invariant scalable keypoints (BRISK) descriptor method.

## **Name-Value Pair Arguments**

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

Example: `'Metric', 'SSD'` specifies the sum of squared differences for the feature matching metric.

#### **'Method' — Matching method**

`'Exhaustive'` (default) | `'Approximate'`

Matching method, specified as the comma-separated pair consisting of `'Method'` and either `'Exhaustive'` or `'Approximate'`. The method specifies how nearest neighbors between `features1` and `features2` are found. Two feature vectors match when the distance between them is less than the threshold set by the `MatchThreshold` parameter.

`'Exhaustive'` Compute the pairwise distance between feature vectors in `features1` and `features2`.

`'Approximate'` Use an efficient approximate nearest neighbor search. Use this method for large feature sets. [3]

#### **'MatchThreshold' — Matching threshold**

10.0 or 1.0 (default) | percent value in the range (0, 100]

Matching threshold threshold, specified as the comma-separated pair consisting of 'MatchThreshold' and a scalar percent value in the range (0,100]. The default values are set to either 10.0 for binary feature vectors or to 1.0 for nonbinary feature vectors. You can use the match threshold for selecting the strongest matches. The threshold represents a percent of the distance from a perfect match.

Two feature vectors match when the distance between them is less than the threshold set by MatchThreshold. The function rejects a match when the distance between the features is greater than the value of MatchThreshold. Increase the value to return more matches.

Inputs that are binaryFeatures objects typically require a larger value for the match threshold. The binaryFeatures objects. The extractFeatures function returns the binaryFeatures objects when extracting FREAK or BRISK descriptors.

**'MaxRatio' — Ratio threshold**

0.6 (default) | ratio in the range (0,1]

Ratio threshold, specified as the comma-separated pair consisting of 'MaxRatio' and a scalar ratio value in the range (0,1]. Use the max ratio for rejecting ambiguous matches. Increase this value to return more matches.

**'Metric' — Feature matching metric**

'SSD' (default) | 'SAD'

Feature matching metric, specified as the comma-separated pair consisting of 'Metric' and either 'SAD' or 'SSD'.

'SAD' Sum of absolute differences

'SSD' Sum of squared differences

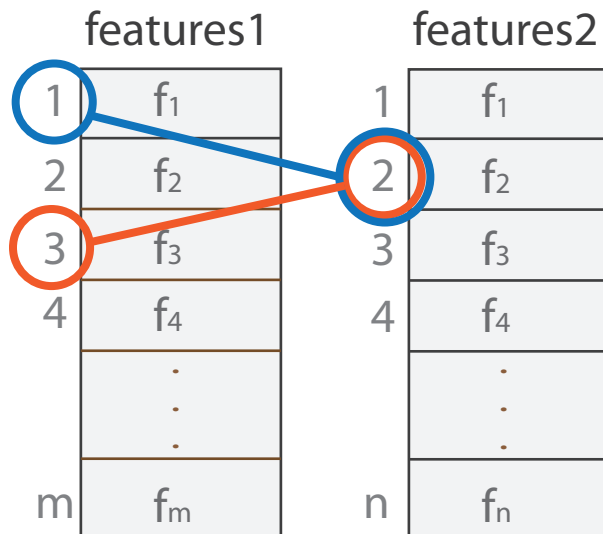
This property applies when the input feature sets, features1 and features2, are not binaryFeatures objects. When you specify the features as binaryFeatures objects, the function uses the Hamming distance to compute the similarity metric.

**'Unique' — Unique matches**

false (default) | true

Unique matches, specified as the comma-separated pair consisting of 'Unique' and either false or true. Set this value to true to return only unique matches between features1 and features2.

When you set `Unique` to `false`, the function returns all matches between `features1` and `features2`. Multiple features in `features1` can match to one feature in `features2`.



When you set `Unique` to `true`, the function performs a forward-backward match to select a unique match. After matching `features1` to `features2`, it matches `features2` to `features1` and keeps the best match.

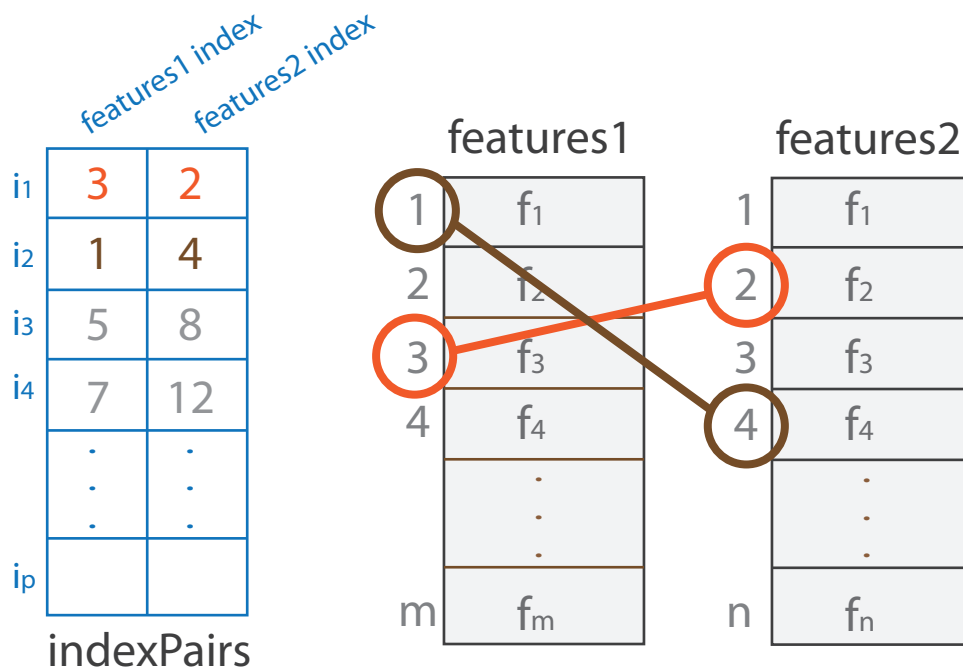
## Output Arguments

### **indexPairs** — Indices to corresponding features

*P*-by-2 matrix

Indices of corresponding features between the two input feature sets, returned as a *P*-by-2 matrix of *P* number of indices. Each index pair corresponds to a matched feature between the `features1` and `features2` inputs. The first element indexes the feature in `features1`. The second element indexes the matching feature in `features2`.





### matchmetric — Distance between matching features

$p$ -by-1 vector

Distance between matching features, returned as a  $p$ -by-1 vector. The value of the distances are based on the metric selected. Each  $i$ th element in **matchmetric** corresponds to the  $i$ th row in the **indexPairs** output matrix. When **Metric** is set to either **SAD** or **SSD**, the feature vectors are normalized to unit vectors before computation.

Metric	Range	Perfect Match Value
SAD	$[0, 2 * \sqrt{\text{size}(\text{features1}, 2)}]$ .	0
SSD	[0,4]	0
Hamming	[0, features1.NumBits]	0

**Note:** You cannot select the Hamming metric. It is invoked automatically when **features1** and **features2** inputs are **binaryFeatures**.

## More About

- “Structure from Motion”

## References

- [1] Lowe, David G. "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision*. Volume 60, Number 2, pp. 91–110.
- [2] Muja, M., and D. G. Lowe. "Fast Matching of Binary Features." *Conference on Computer and Robot Vision*. CRV, 2012.
- [3] Muja, M., and D. G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." *International Conference on Computer Vision Theory and Applications*. VISAPP, 2009.

## See Also

`binaryFeatures` | `detectBRISKFeatures` | `detectFASTFeatures` |  
`detectHarrisFeatures` | `detectMinEigenFeatures` | `detectMSERFeatures`  
| `detectSURFFeatures` | `estimateFundamentalMatrix` |  
`estimateGeometricTransform` | `extractFeatures`

**Introduced in R2011a**

# mplay

View video from MATLAB workspace, multimedia file, or Simulink model.

## Syntax

## Description

---

**Note:** The `mplay` function will be removed in a future release. Use the `implay` function with functionality identical to `mplay`.

---

**Introduced in R2006a**

## ocr

Recognize text using optical character recognition

### Syntax

```
txt = ocr(I)
txt = ocr(I, roi)

[ ___ ] = ocr( ___ ,Name,Value)
```

### Description

`txt = ocr(I)` returns an `ocrText` object containing optical character recognition information from the input image, `I`. The object contains recognized text, text location, and a metric indicating the confidence of the recognition result.

`txt = ocr(I, roi)` recognizes text in `I` within one or more rectangular regions. The `roi` input contains an  $M$ -by-4 matrix, with  $M$  regions of interest.

`[ ___ ] = ocr( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

#### Code Generation Support:

Compile-time constant input: `TextLayout`, `Language`, and `CharacterSet`.

Supports MATLAB Function block: No

Generated code for this function uses a precompiled platform-specific shared library. “Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Recognize Text Within an Image

```
businessCard = imread('businessCard.png');
ocrResults   = ocr(businessCard)
recognizedText = ocrResults.Text;
figure;
imshow(businessCard);
text(600, 150, recognizedText, 'BackgroundColor', [1 1 1]);
```

```
ocrResults =
```

```
    ocrText with properties:
```

```
                Text: ' MathWorks@...'  
CharacterBoundingBoxes: [103x4 double]  
CharacterConfidences: [103x1 single]  
                Words: {16x1 cell}  
WordBoundingBoxes: [16x4 double]  
WordConfidences: [16x1 single]
```



### Recognize Text in Regions of Interest (ROI)

Read image.

```
I = imread('handicapSign.jpg');
```

**Define one or more rectangular regions of interest within I.**

```
roi = [360 118 384 560];
```

**You may also use IMRECT to select a region using a mouse:**

```
figure; imshow(I); roi = round(getPosition(imrect))
```

```
ocrResults = ocr(I, roi);
```

**Insert recognized text into original image**

```
Iocr = insertText(I, roi(1:2), ocrResults.Text, 'AnchorPoint', 'RightTop', 'FontS
```

```
figure; imshow(Iocr);
```



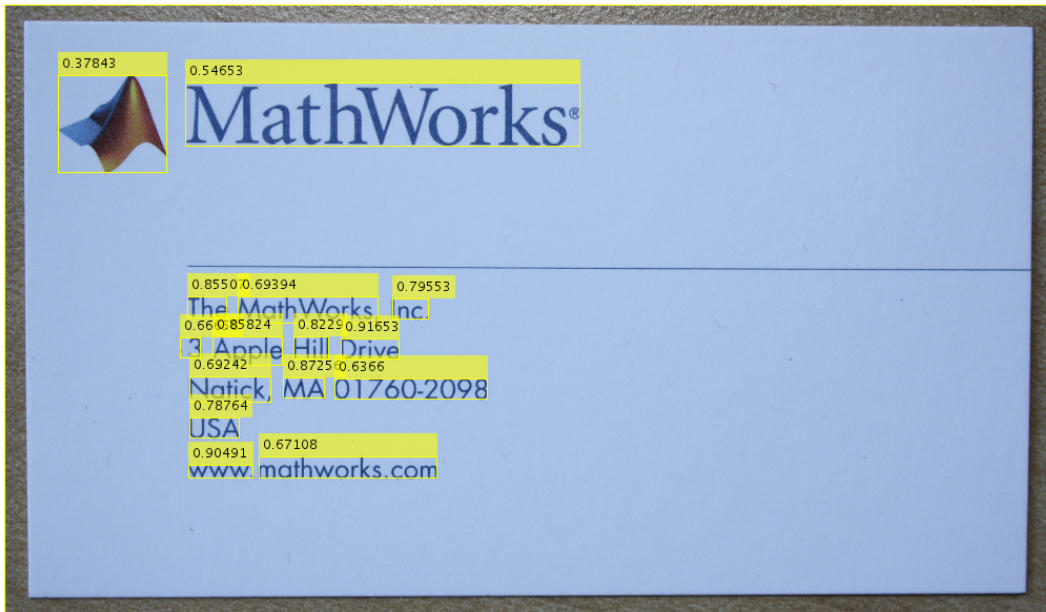
### Display Bounding Boxes of Words and Recognition Confidences

```
businessCard = imread('businessCard.png');
ocrResults   = ocr(businessCard)
Iocr         = insertObjectAnnotation(businessCard, 'rectangle', ...
                                      ocrResults.WordBoundingBoxes, ...
                                      ocrResults.WordConfidences);
figure; imshow(Iocr);
```

```
ocrResults =
```

ocrText with properties:

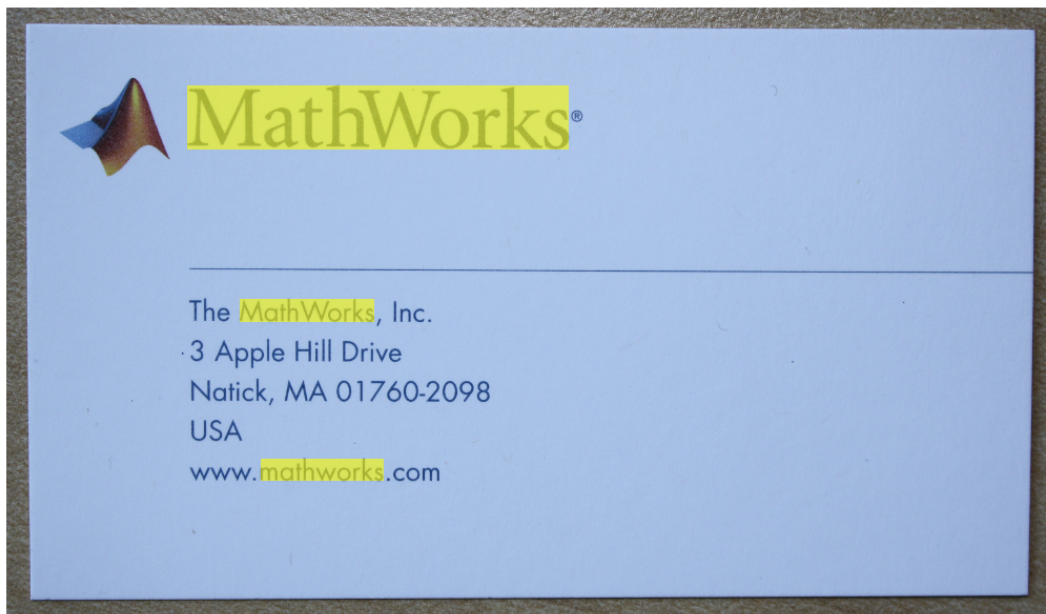
```
Text: ' ' MathWorks@...'
CharacterBoundingBoxes: [103x4 double]
CharacterConfidences: [103x1 single]
Words: {16x1 cell}
WordBoundingBoxes: [16x4 double]
WordConfidences: [16x1 single]
```



#### Find and Highlight Text in an Image

```
businessCard = imread('businessCard.png');
ocrResults = ocr(businessCard);
bboxes = locateText(ocrResults, 'MathWorks', 'IgnoreCase', true);
Iocr = insertShape(businessCard, 'FilledRectangle', bboxes);
figure; imshow(Iocr);
```





- “Automatically Detect and Recognize Text in Natural Images”
- “Recognize Text Using Optical Character Recognition (OCR)”

## Input Arguments

### **I** – Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image | *M*-by-*N* binary image

Input image, specified in *M*-by-*N*-by-3 truecolor, *M*-by-*N* 2-D grayscale, or binary format. The input image must be a real, nonsparse value. The function converts truecolor or grayscale input images to a binary image, before the recognition process. It uses the Otsu’s thresholding technique for the conversion. For best ocr results, the height of a lowercase ‘x’, or comparable character in the input image, must be greater than 20 pixels. From either the horizontal or vertical axes, remove any text rotations greater than +/- 10 degrees, to improve recognition results.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

### **roi** — Region of interest

*M*-by-4 element matrix

One or more rectangular regions of interest, specified as an *M*-by-4 element matrix. Each row, *M*, specifies a region of interest within the input image, as a four-element vector, [*x y width height*]. The vector specifies the upper-left corner location, [*x y*], and the size of a rectangular region of interest, [*width height*], in pixels. Each rectangle must be fully contained within the input image, *I*. Before the recognition process, the function uses the Otsu's thresholding to convert truecolor and grayscale input regions of interest to binary regions. The function returns text recognized in the rectangular regions as an array of objects.

## **Name-Value Pair Arguments**

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

Example:

### **'TextLayout'** — Input text layout

'Auto' (default) | 'Block' | 'Line' | 'Word'

Input text layout, specified as the comma-separated pair consisting of 'TextLayout' and the string 'Auto', 'Block', 'Line', or 'Word'.

The table lists how the function treats the text for each **TextLayout** setting.

<b>TextLayout</b>	<b>Text Treatment</b>
'Auto'	Determines the layout and reading order of text blocks within the input image.
'Block'	Treats the text in the image as a single block of text.
'Line'	Treats the text in the image as a single line of text.

TextLayout	Text Treatment
'Word'	Treats the text in the image as a single word of text.

Use the automatic layout analysis to recognize text from a scanned document that contains a specific format, such as a double column. This setting preserves the reading order in the returned text. You may get poor results if your input image contains a few regions of text or the text is located in a cluttered scene. If you get poor OCR results, try a different layout that matches the text in your image. If the text is located in a cluttered scene, try specifying an ROI around the text in your image in addition to trying a different layout.

### 'Language' — Language

'English' (default) | 'Japanese' | string | cell array of strings

Language to recognize, specified as the comma-separated pair consisting of 'Language' and the string 'English', 'Japanese', or a cell array of strings. You can also install the “Install OCR Language Data Support Package” package for additional languages or add a custom language. Specifying multiple languages enables simultaneous recognition of all the selected languages. However, selecting more than one language may reduce the accuracy and increase the time it takes to perform ocr.

To specify any of the additional languages which are contained in the “Install OCR Language Data Support Package” package, use the language string the same way as the built-in languages. You do not need to specify the path.

```
txt = ocr(img, 'Language', 'Finnish');
```

### List of Support Package OCR Languages

- 'Afrikaans'
- 'Albanian'
- 'AncientGreek'
- 'Arabic'
- 'Azerbaijani'
- 'Basque'
- 'Belarusian'
- 'Bengali'

- 'Bulgarian'
- 'Catalan'
- 'Cherokee'
- 'ChineseSimplified'
- 'ChineseTraditional'
- 'Croatian'
- 'Czech'
- 'Danish'
- 'Dutch'
- 'English'
- 'Esperanto'
- 'EsperantoAlternative'
- 'Estonian'
- 'Finnish'
- 'Frankish'
- 'French'
- 'Galician'
- 'German'
- 'Greek'
- 'Hebrew'
- 'Hindi'
- 'Hungarian'
- 'Icelandic'
- 'Indonesian'
- 'Italian'
- 'ItalianOld'
- 'Japanese'
- 'Kannada'
- 'Korean'
- 'Latvian'

- 'Lithuanian'
- 'Macedonian'
- 'Malay'
- 'Malayalam'
- 'Maltese'
- 'MathEquation'
- 'MiddleEnglish'
- 'MiddleFrench'
- 'Norwegian'
- 'Polish'
- 'Portuguese'
- 'Romanian'
- 'Russian'
- 'SerbianLatin'
- 'Slovakian'
- 'Slovenian'
- 'Spanish'
- 'SpanishOld'
- 'Swahili'
- 'Swedish'
- 'Tagalog'
- 'Tamil'
- 'Telugu'
- 'Thai'
- 'Turkish'
- 'Ukrainian'

To use your own custom languages, specify the path to the trained data file as the language string. You must name the file in the format, *<language>.traineddata*. The file must be located in a folder named 'tessdata'. For example:

```
txt = ocr(img, 'Language', 'path/to/tessdata/eng.traineddata');
```

You can load multiple custom languages as a cell array of strings:

```
txt = ocr(img, 'Language', ...
          {'path/to/tessdata/eng.traineddata', ...
           'path/to/tessdata/jpn.traineddata'});
```

The containing folder must always be the same for all the files specified in the cell array. In the preceding example, all of the `traineddata` files in the cell array are contained in the folder `'path/to/tessdata'`. Because the following code points to two different containing folders, it does not work.

```
txt = ocr(img, 'Language', ...
          {'path/one/tessdata/eng.traineddata', ...
           'path/two/tessdata/jpn.traineddata'});
```

Some language files have a dependency on another language. For example, Hindi training depends on English. If you want to use Hindi, the English `traineddata` file must also exist in the same folder as the Hindi `traineddata` file.

**For deployment targets generated by MATLAB Coder:** Generated ocr executable and language data file folder must be colocated. The `tessdata` folder must be named `tessdata`:

- For English: `C:/path/tessdata/eng.traineddata`
- For Japanese: `C:/path/tessdata/jpn.traineddata`
- For custom data files: `C:/path/tessdata/customlang.traineddata`
- `C:/path/ocr_app.exe`

You can copy the English and Japanese trained data files from:

```
fullfile(matlabroot, 'toolbox', 'vision', 'visionutilities', 'tessdata');
```

#### 'CharacterSet' — Character subset

' ' all characters (default) | string

Character subset, specified as the comma-separated pair consisting of `'CharacterSet'` and a character string. By default, `CharacterSet` is set to the empty string, `' '`. The empty string sets the function to search for all characters in the language specified by the `Language` property. You can set this property to a smaller set of known characters to constrain the classification process.

The `ocr` function selects the best match from the `CharacterSet`. Using deducible knowledge about the characters in the input image helps to improve text recognition accuracy. For example, if you set `CharacterSet` to all numeric digits, `'0123456789'`,

the function attempts to match each character to only digits. In this case, a non-digit character can incorrectly get recognized as a digit.

## Output Arguments

### **txt** — Recognized text and metrics

`ocrText` object

Recognized text and metrics, returned as an `ocrText` object. The object contains the recognized text, the location of the recognized text within the input image, and the metrics indicating the confidence of the results. The confidence values range is [0 1] and represents a percent probability. When you specify an  $M$ -by-4 `roi`, the function returns `ocrText` as an  $M$ -by-1 array of `ocrText` objects.

If your `OCR` results are not what you expect, try one or more of the following options:

- Increase the image 2-to-4 times the original size.
- If the characters in the image are too close together or their edges are touching, use `morphology` to thin out the characters. Using `morphology` to thin out the characters separates the characters.
- Use binarization to check for non-uniform lighting issues. Use the `graythresh` and `imbinarize` functions to binarize the image. If the characters are not visible in the results of the binarization, it indicates a potential non-uniform lighting issue. Try top hat, using the `imtophat` function, or other techniques that deal with removing non-uniform illumination.
- Use the region of interest `roi` option to isolate the text. Specify the `roi` manually or use text detection.
- If your image looks like a natural scene containing words, like a street scene, rather than a scanned document, try using an ROI input. Also, you can set the `TextLayout` property to 'Block' or 'Word'.

## More About

- “Train Optical Character Recognition for Custom Fonts”
- “Install OCR Language Data Support Package”
- “Install Computer Vision System Toolbox Add-on Support Files”

### References

- [1] R. Smith. *An Overview of the Tesseract OCR Engine*, Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2 (2007), pp. 629-633.
- [2] Smith, R., D. Antonova, and D. Lee. *Adapting the Tesseract Open Source OCR Engine for Multilingual OCR*. Proceedings of the International Workshop on Multilingual OCR, (2009).
- [3] R. Smith. *Hybrid Page Layout Analysis via Tab-Stop Detection*. Proceedings of the 10th international conference on document analysis and recognition. 2009.

### See Also

`ocrText` | `graythresh` | `imbinarize` | `imtophat` | `insertShape` | OCR Trainer

**Introduced in R2014a**



# pcdenoise

Remove noise from 3-D point cloud

## Syntax

```
ptCloudOut = pcdenoise(ptCloudIn)
[ptCloudOut,inlierIndices,outlierIndices] = pcdenoise(ptCloudIn)
[ptCloudOut, ___ ] = pcdenoise(___ Name,Value)
```

## Description

`ptCloudOut = pcdenoise(ptCloudIn)` returns a filtered point cloud that removes outliers.

`[ptCloudOut,inlierIndices,outlierIndices] = pcdenoise(ptCloudIn)` additionally returns the linear indices to the points that are identified as inliers and outliers.

`[ptCloudOut, ___ ] = pcdenoise(___ Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

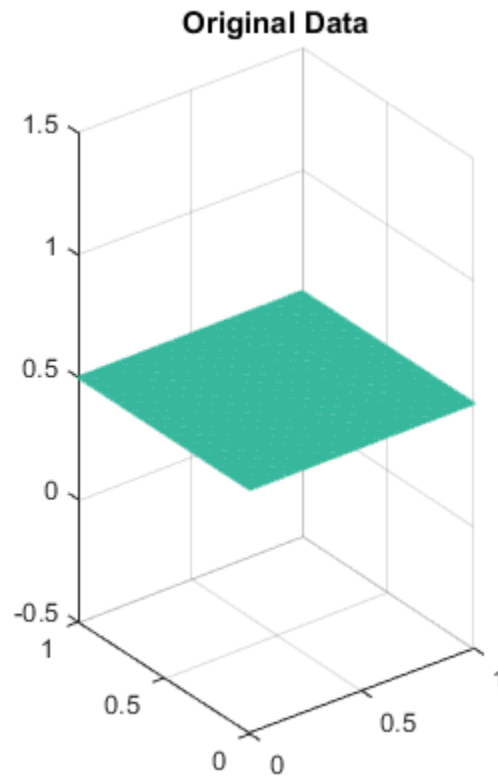
## Examples

### Remove Outliers from Noisy Point Cloud

Create a plane point cloud.

```
gv = 0:0.01:1;
[X,Y] = meshgrid(gv,gv);
ptCloud = pointCloud([X(:),Y(:),0.5*ones(numel(X),1)]);

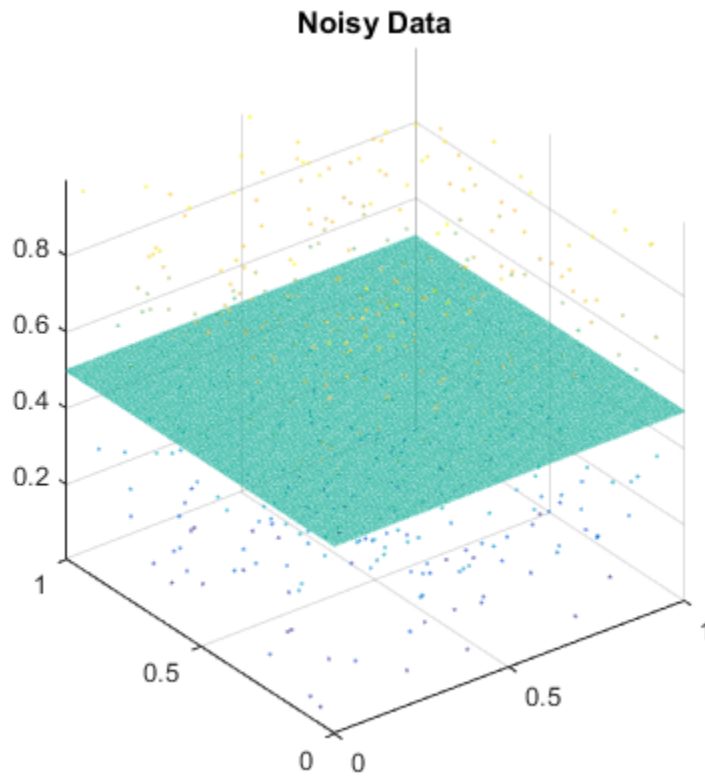
figure
pcshow(ptCloud);
title('Original Data');
```



Add uniformly distributed random noise.

```
noise = rand(500, 3);  
ptCloudA = pointCloud([ptCloud.Location; noise]);
```

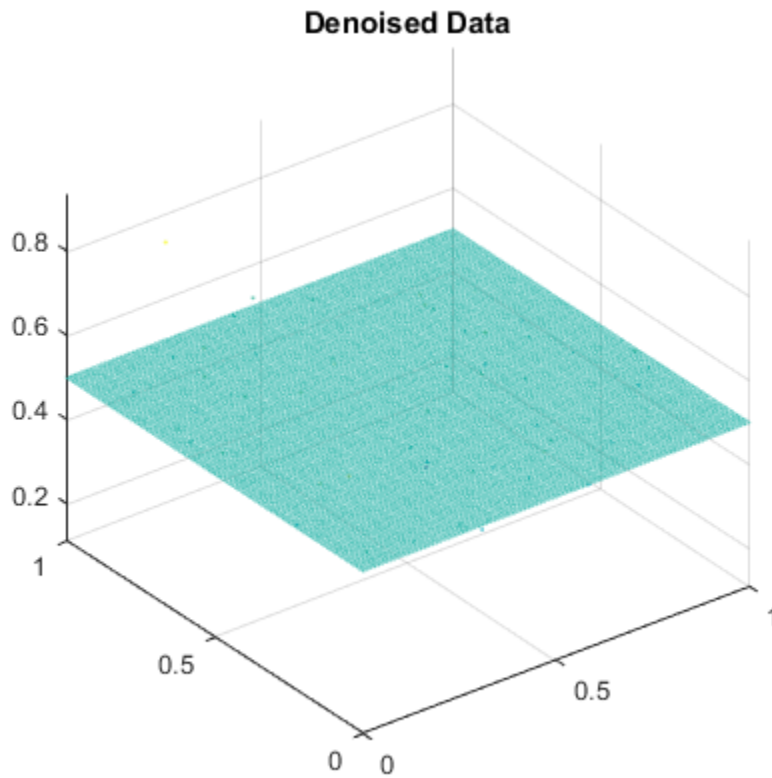
```
figure  
pcshow(ptCloudA);  
title('Noisy Data');
```



Remove outliers.

```
ptCloudB = pcdenoise(ptCloudA);
```

```
figure;  
pcshow(ptCloudB);  
title('Denoised Data');
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

**ptCloudIn** — Point cloud  
pointCloud object

Point cloud, specified as a pointCloud 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`.

Example: `'Threshold', '1.0'` sets the threshold to 1.0.

### 'NumNeighbors' — Number of nearest neighbor points

4 (default) | positive integer

Number of nearest neighbor points, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer in pixels. The value is used to estimate the mean of the average distance to neighbors of all points. Decreasing this value makes the filter more sensitive to noise. Increasing this value increases the number of computations.

### 'Threshold' — Outlier threshold

1.0 (default) | scalar

Outlier threshold, specified as the comma-separated pair consisting of 'Threshold' and a scalar. By default, the threshold is one standard deviation from the mean of the average distance to neighbors of all points. A point is considered to be an outlier if the average distance to its  $k$ -nearest neighbors is above the specified threshold.

## Output Arguments

### ptCloudOut — Filtered point cloud

pointCloud object

Filtered point cloud, returned as a pointCloud object.

### inlierIndices — Linear index of inlier points

1-by- $N$  vector

Linear index of inlier points, returned as a 1-by- $N$  vector.

Data Types: uint32

### outlierIndices — Linear index of outlier points

1-by- $N$  vector

Linear index of outlier points, returned as a 1-by- $N$  vector of linear indices.

Data Types: `uint32`

### References

- [1] Rusu, R. B., Z. C. Marton, N. Blodow, M. Dolha, and M. Beetz. “Towards 3D Point Cloud Based Object Maps for Household Environments”. *Robotics and Autonomous Systems Journal*. 2008.

### See Also

`pointCloud` | `pcplayer` | `planeModel` | `affine3d` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pctransform` | `pcwrite`

**Introduced in R2015a**

## pcmerge

Merge two 3-D point clouds

### Syntax

```
ptCloudOut = pcmerge(ptCloudA,ptCloudB,gridStep)
```

### Description

`ptCloudOut = pcmerge(ptCloudA,ptCloudB,gridStep)` returns a merged point cloud using a box grid filter. `gridStep` specifies the size of the 3-D box for the filter.

### Examples

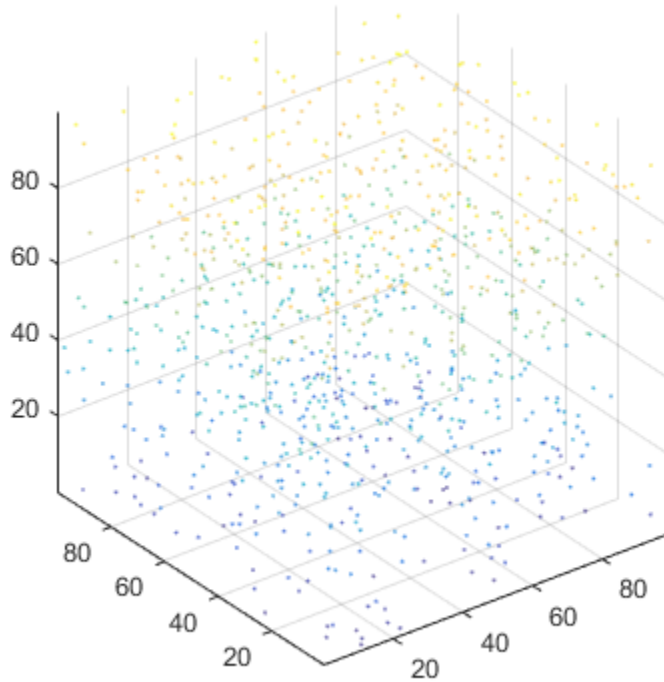
#### Merge Two Identical Point Clouds Using Box Grid Filter

Create two identical point clouds.

```
ptCloudA = pointCloud(100*rand(1000,3));  
ptCloudB = copy(ptCloudA);
```

Merge the two point clouds.

```
ptCloud = pcmerge(ptCloudA,ptCloudB,1);  
pcshow(ptCloud);
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **ptCloudA — Point cloud A**

pointCloud object

Point cloud A, specified as a pointCloud object.

### **ptCloudB — Point cloud B**

pointCloud object



Point cloud B, specified as a pointCloud object.

**gridStep** — Size of 3-D box for grid filter

numeric value

Size of 3-D box for grid filter, specified as a numeric value. Increase the size of `gridStep` when there are not enough resources to construct a large fine-grained grid.

Data Types: `single` | `double`

## Output Arguments

**ptCloudOut** — Merged point cloud

pointCloud object

Merged point cloud, returned as a pointCloud object. The function computes the axis-aligned bounding box for the overlapped region between two point clouds. The bounding box is divided into grid boxes of the size specified by `gridStep`. Points within each grid box are merged by averaging their locations, colors, and normals. Points outside the overlapped region are untouched.

### See Also

`pointCloud` | `pcplayer` | `planeModel` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcread` | `pcregrid` | `pcshow` | `pctransform` | `pcwrite`

**Introduced in R2015a**

## pcdownsample

Downsample a 3-D point cloud

### Syntax

```
ptCloudOut = pcdownsample(ptCloudIn, 'random', percentage)
ptCloudOut = pcdownsample(ptCloudIn, 'gridAverage', gridStep)
ptCloudOut = pcdownsample(ptCloudIn, 'nonuniformGridSample',
maxNumPoints)
```

### Description

`ptCloudOut = pcdownsample(ptCloudIn, 'random', percentage)` returns a downsampled point cloud with random sampling and without replacement. The `percentage` input specifies the portion of the input to return to the output.

`ptCloudOut = pcdownsample(ptCloudIn, 'gridAverage', gridStep)` returns a downsampled point cloud using a box grid filter. The `gridStep` input specifies the size of a 3-D box.

`ptCloudOut = pcdownsample(ptCloudIn, 'nonuniformGridSample', maxNumPoints)` returns a downsampled point cloud using nonuniform box grid filter. You must set the maximum number of points in the grid box, `maxNumPoints`, to at least 6.

### Examples

#### Downsample Point Cloud Using Box Grid Filter

Read a point cloud.

```
ptCloud = pcread('teapot.ply');
```

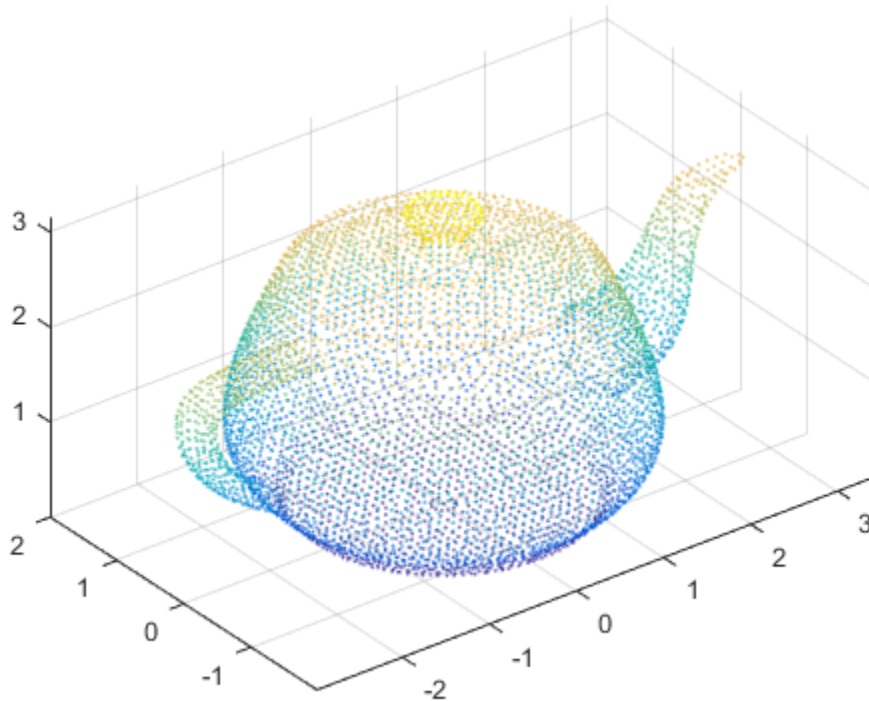
Set the 3-D resolution to be (0.1 x 0.1 x 0.1).

```
gridStep = 0.1;
```

```
ptCloudA = pcdownsample(ptCloud, 'gridAverage', gridStep);
```

Visualize the downsampled data.

```
figure;  
pcshow(ptCloudA);
```

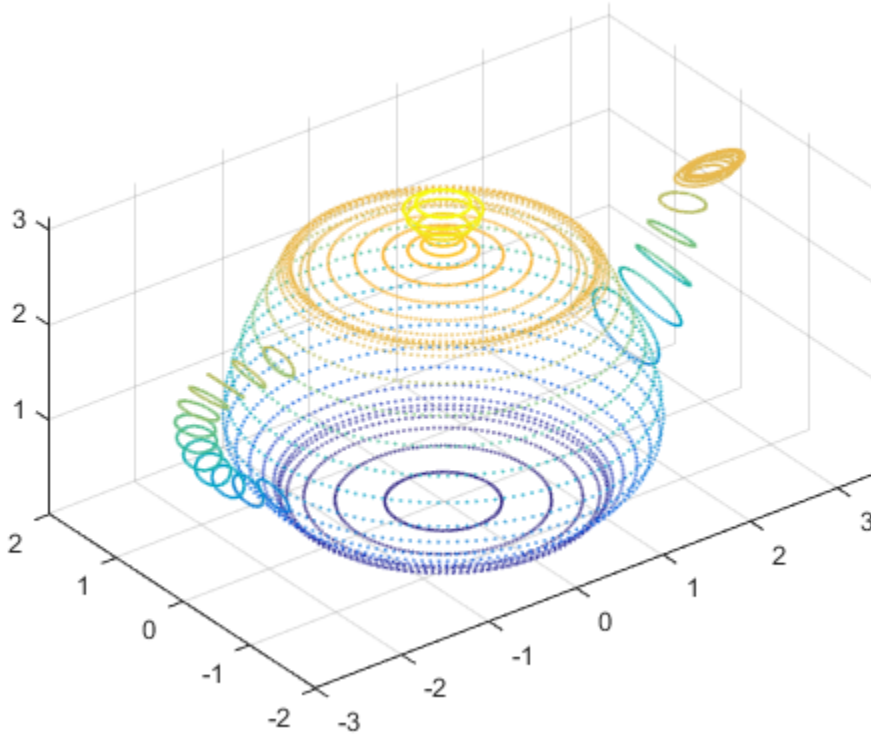


Compare the point cloud to data that is downsampled using a fixed step size.

```
stepSize = floor(ptCloud.Count/ptCloudA.Count);  
indices = 1:stepSize:ptCloud.Count;  
ptCloudB = select(ptCloud, indices);
```

```
figure;
```

```
pcshow(ptCloudB);
```



#### Remove Redundant Points from Point Cloud

Create a point cloud with all points sharing the same coordinates.

```
ptCloud = pointCloud(ones(100,3));
```

Set the 3-D resolution to a small value.

```
gridStep = 0.01;
```

The output now contains only one unique point.

```
ptCloudOut = pcdownsampling(ptCloud,'gridAverage',gridStep)
```

```
ptCloudOut =
    pointCloud with properties:
        Location: [1 1 1]
        Color: [0x3 uint8]
        Normal: [0x3 double]
        Count: 1
        XLimits: [1 1]
        YLimits: [1 1]
        ZLimits: [1 1]
```

- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **ptCloudIn** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **'random'** — Random downsample method

'random'

Random downsample method, specified as the string, 'random'. This method is more efficient than the 'gridAverage' downsample method, especially when it is applied before point cloud registration.

Downsample the point cloud using 'random', 'gridAverage', or 'nonuniformGridSample' inputs, according to the Metric you use in the pcregrid function for registration.

Metric	Point Cloud A Downsample Method	Point Cloud B Downsample Method
'pointToPoint'	'random'	'random'
	'gridAverage'	'gridAverage'
'pointToPlane'	'gridAverage'	'gridAverage'

Metric	Point Cloud A Downsample Method	Point Cloud B Downsample Method
	'random'	'nonuniformGridSample'

#### **percentage** — Percentage of input

positive scalar

Percentage of input, specified as a positive scalar in the range [0, 1]. The `percentage` input specifies the portion of the input for the function to return.

#### **'gridAverage'** — Grid average downsample method

'gridAverage'

Grid average downsample method, specified as the string, 'gridAverage'. Points within the same box are merged to a single point in the output. Their color and normal properties are averaged accordingly. This method preserves the shape of the point cloud better than the 'random' downsample method.

The function computes the axis-aligned bounding box for the entire point cloud. The bounding box is divided into grid boxes of size specified by `gridStep`. Points within each grid box are merged by averaging their locations, colors, and normals.

Downsample the point cloud using 'random', 'gridAverage', or 'nonuniformGridSample' inputs, according to the `Metric` you use in the `pcregrid` function for registration.

Metric	Point Cloud A Downsample Method	Point Cloud B Downsample Method
'pointToPoint'	'random'	'random'
	'gridAverage'	'gridAverage'
'pointToPlane'	'gridAverage'	'gridAverage'
	'random'	'nonuniformGridSample'

#### **gridStep** — Size of 3-D box for grid filter

numeric value

Size of 3-D box for grid filter, specified as a numeric value. Increase the size of `gridStep` when there are not enough resources to construct a large fine-grained grid.

Data Types: `single` | `double`

**'nonuniformGridSample' — Nonuniform grid sample method**  
`'nonuniformGridSample'`

Nonuniform grid sample method, specified as the string `'nonuniformGridSample'`. The best use of this method is to apply it as a preprocessing step to the `pcregrigid` function for point cloud registration, when you use the `'pointToPlane'` metric. When you use the `'nonuniformGridSample'` algorithm, the normals are computed on the original data prior to downsampling. The downsampled output preserves more accurate normals.

Downsample the point cloud using `'random'`, `'gridAverage'`, or `'nonuniformGridSample'` inputs, according to the `Metric` you use in the `pcregrigid` function for registration.

<b>Metric</b>	<b>Point Cloud A Downsample Method</b>	<b>Point Cloud B Downsample Method</b>
<code>'pointToPoint'</code>	<code>'random'</code>	<code>'random'</code>
	<code>'gridAverage'</code>	<code>'gridAverage'</code>
<code>'pointToPlane'</code>	<code>'gridAverage'</code>	<code>'gridAverage'</code>
	<code>'random'</code>	<code>'nonuniformGridSample'</code>

**maxNumPoints — Maximum number of points in grid box**  
`integer`

Maximum number of points in grid box, specified as an integer greater than 6. The method randomly selects a single point from each box. If the normal was not provided in the input point cloud, this method automatically fills in the normal property in the `ptCloudOut` output.

## Output Arguments

**ptCloudOut — Filtered point cloud**  
`pointCloud` object

Filtered point cloud, returned as a `pointCloud` object.

**See Also**

pointCloud | pcplayer | planeModel | affine3d | pcdenoise | pcfitplane | pcmerge  
| pcread | pcregrigid | pcshow | pctransform | pcwrite

**Introduced in R2015a**



# pcread

Read 3-D point cloud from PLY file

## Syntax

```
ptCloud = pcread(filename)
```

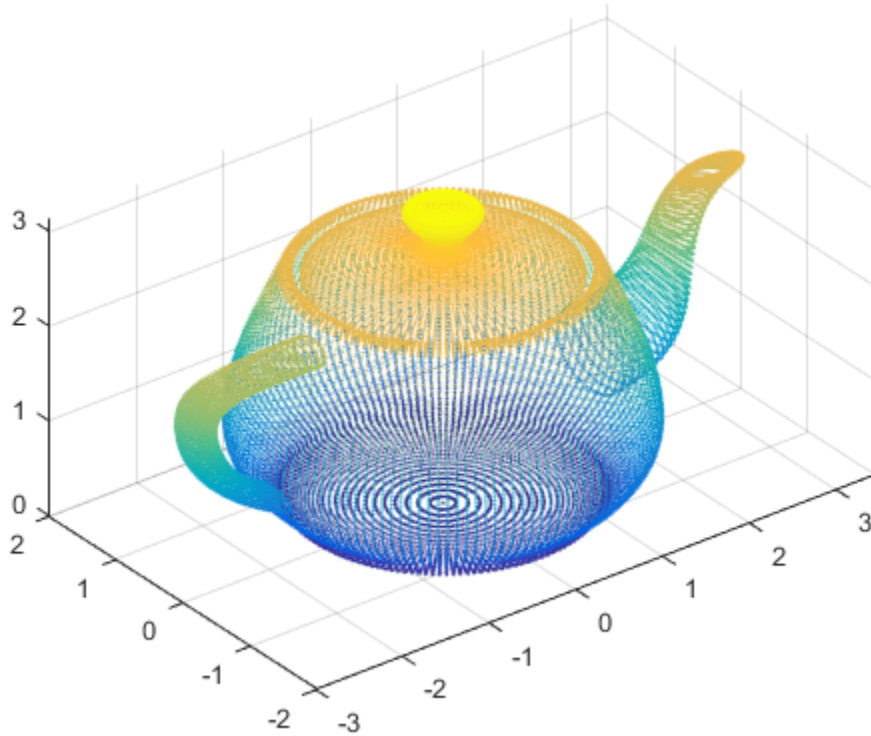
## Description

`ptCloud = pcread(filename)` reads a point cloud from the PLY file specified by the input `filename` string. The function returns a `pointCloud` object, `ptCloud`.

## Examples

### Read Point Cloud from a PLY File

```
ptCloud = pcread('teapot.ply');  
pcshow(ptCloud);
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

**filename** — PLY file name

string

PLY file name, specified as a string. The input file type must be a PLY format file.

The polygon file format (PLY), also known as the Stanford triangle format stores three-dimensional data from 3-D scanners. It is a format for storing graphical objects that are described as a collection of polygons. A PLY file consists of a header, followed by a

list of vertices and then, a list of polygons. The header specifies how many vertices and polygons are in the file. It also states what properties are associated with each vertex, such as  $(x,y,z)$  coordinates, normals, and color. The file format has two sub-formats: an ASCII representation and a binary version for compact storage and for rapid saving and loading. The header of both ASCII and binary files is ASCII text. Only the numeric data that follows the header is different between the two versions. See [The PLY Format](#) for details on the contents of a PLY file.

## Output Arguments

### **ptCloud** — Object for storing point cloud

`pointCloud` object

Object for storing point cloud, returned as a `pointCloud` object that contains the following PLY fields:

- `Location` property, stores the  $x$ ,  $y$ , and  $z$  values.
- `Color` property, stores the red, green, and blue values.
- `Normal` property, stores the normal vectors for each point.

## More About

- [The PLY Format](#)

## See Also

`pointCloud` | `pcplayer` | `planeModel` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcregrigid` | `pcshow` | `pctransform` | `pcwrite`

**Introduced in R2015a**

## pcregrigid

Register two point clouds using ICP algorithm

### Syntax

```
tform = pcregrigid(moving, fixed)
[tform, movingReg] = pcregrigid(moving, fixed)
[ ___, rmse] = pcregrigid(moving, fixed)
[ ___ ] = pcregrigid(moving, fixed, Name, Value)
```

### Description

`tform = pcregrigid(moving, fixed)` returns a rigid transformation that registers a moving point cloud to a fixed point cloud.

The registration algorithm is based on the "iterative closest point" (ICP) algorithm. Best performance of this iterative process requires adjusting properties for your data. Consider downsampling point clouds using `pcdsample` before using `pcregrigid` to improve accuracy and efficiency of registration.

Point cloud normals are required by the registration algorithm when you select the 'pointToPlane' metric. Therefore, if the input point cloud's `Normal` property is empty, the function fills it. When the function fills the `Normal` property, it uses 6 points to fit the local plane. Six points may not work under all circumstances. If registration with the 'pointToPlane' metric fails, consider calling the `pcnormals` function which allows you to select the number of points to use.

`[tform, movingReg] = pcregrigid(moving, fixed)` additionally returns the transformed point cloud that aligns with the fixed point cloud.

`[ ___, rmse] = pcregrigid(moving, fixed)` additionally returns the root mean squared error of the Euclidean distance between the aligned point clouds, using any of the preceding syntaxes.

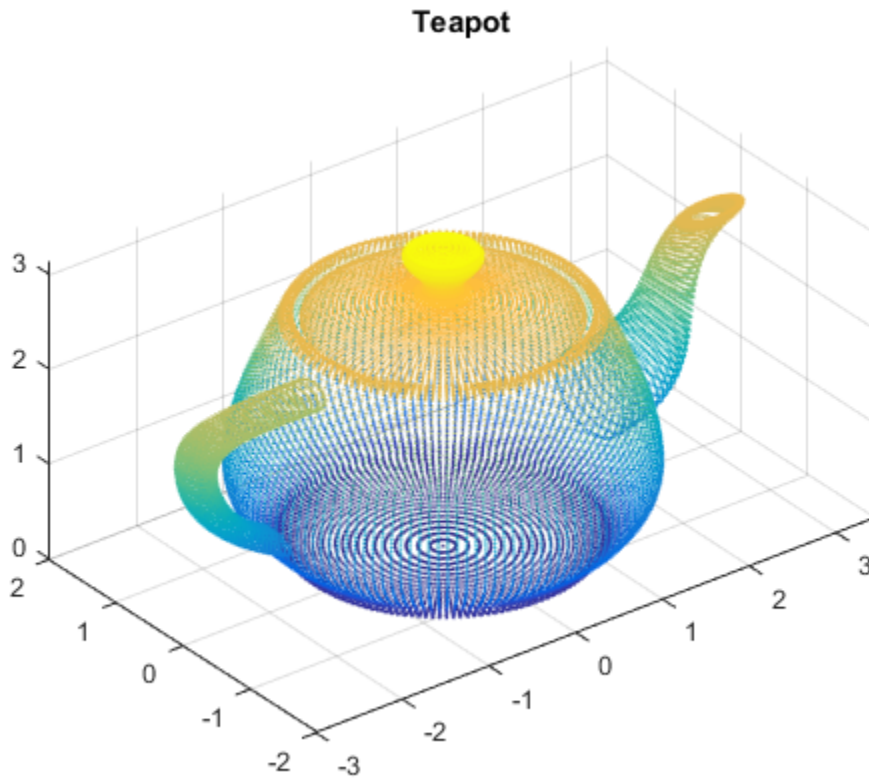
`[ ___ ] = pcregrigid(moving, fixed, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Align Two Point Clouds

Load point cloud data.

```
ptCloud = pcread('teapot.ply');  
figure  
pcshow(ptCloud);  
title('Teapot');
```



Create a transform object with 30 degree rotation along  $z$ -axis and translation  $[5,5,10]$ .

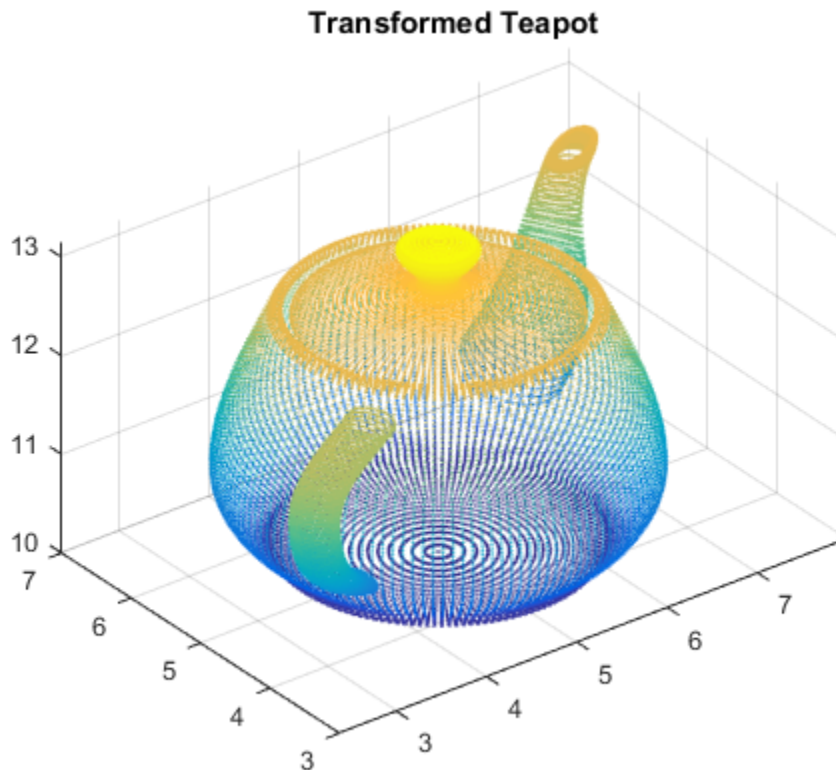
```
A = [cos(pi/6) sin(pi/6) 0 0; ...
```

```
    -sin(pi/6) cos(pi/6) 0 0; ...  
        0         0 1 0; ...  
        5         5 10 1];  
tform1 = affine3d(A);
```

Transform the point cloud.

```
ptCloudTformed = pctransform(ptCloud,tform1);
```

```
figure  
pcshow(ptCloudTformed);  
title('Transformed Teapot');
```



Apply the rigid registration.

```
tform = pcregrigid(ptCloudTformed,ptCloud,'Extrapolate',true);
```

Compare the result with the true transformation.

```
disp(tform1.T);
tform2 = invert(tform);
disp(tform2.T);

    0.8660    0.5000         0         0
   -0.5000    0.8660         0         0
         0         0    1.0000         0
    5.0000    5.0000   10.0000    1.0000

    0.8660    0.5000   -0.0000         0
   -0.5000    0.8660    0.0000         0
    0.0000   -0.0000    1.0000         0
    5.0000    5.0000   10.0000    1.0000
```

- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **moving** — Moving point cloud

pointCloud object

Moving point cloud, specified as a pointCloud object.

### **fixed** — Fixed point cloud

pointCloud object

Fixed point cloud, specified as a pointCloud 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**.

Example: 'Metric','pointToPoint' sets the metric for the ICP algorithm to the 'pointToPoint' string.

**'Metric' — Minimization metric**

'pointToPoint' (default) | 'pointToPlane'

Minimization metric, specified as the comma-separated pair consisting of 'Metric' and the 'pointToPoint' or 'pointToPlane' string. The rigid transformation between the moving and fixed point clouds are estimated by the iterative closest point (ICP) algorithm. The ICP algorithm minimizes the distance between the two point clouds according to the given metric.

Setting 'Metric' to 'pointToPlane' can reduce the number of iterations to process. However, this metric requires extra algorithmic steps within each iteration. The 'pointToPlane' metric improves the registration of planar surfaces.

**Downsample Method Selection:**

Downsample the point clouds using the `pcdownsample` function. Use either the 'random' or 'gridAverage' input for the `pcdownsample` function according to the Metric table below.

Metric	Point Cloud A Downsample Method	Point Cloud B Downsample Method
'pointToPoint'	'random'	'random'
	'gridAverage'	'gridAverage'
'pointToPlane'	'gridAverage'	'gridAverage'
	'random'	'nonuniformGridSample'

**'Extrapolate' — Extrapolation**

false (default) | true

Extrapolation, specified as the comma-separated pair consisting of 'Extrapolate' and the boolean `true` or `false`. When you set this property to `true`, the function adds an extrapolation step that traces out a path in the registration state space, that is described in [2]. Setting this property to `true` can reduce the number of iterations to converge.

**'InlierRatio' — Percentage of inliers**

1 (default) | scalar

Percentage of inliers, specified as the comma-separated pair consisting of 'InlierRatio' and a scalar value. Use this value to set a percentage of matched pairs as inliers. A pair of matched points is considered an inlier if its Euclidean distance falls within the percentage set of matching distances. By default, all matching pairs are used.



**'MaxIterations' — Maximum number of iterations**

20 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIterations' and a positive integer. This value specifies the maximum number of iterations before ICP stops.

**'Tolerance' — Tolerance between consecutive ICP iterations**

[0.01, 0.009] (default) | 2-element vector

Tolerance between consecutive ICP iterations, specified as the comma-separated pair consisting of 'Tolerance' and a 2-element vector. The 2-element vector, [*Tdiff*, *Rdiff*], represents the tolerance of absolute difference in translation and rotation estimated in consecutive ICP iterations. *Tdiff* measures the Euclidean distance between two translation vectors. *Rdiff* measures the angular difference in radians. The algorithm stops when the average difference between estimated rigid transformations in the three most recent consecutive iterations falls below the specified tolerance value.

**'InitialTransform' — Initial rigid transformation**

affine3d() object (default)

Initial rigid transformation, specified as the comma-separated pair consisting of 'InitialTransform' and an affine3d object. The initial rigid transformation is useful when you provide an external coarse estimation.

**'Verbose' — Display progress information**

true (default) | false

Display progress information, specified as the comma-separated pair consisting of 'Verbose' and a logical scalar. Set `Verbose` to true to display progress information.

## Output Arguments

**tform — Rigid transformation**

affine3d object

Rigid transformation, returned as an affine3d object. The rigid transformation registers a moving point cloud to a fixed point cloud. The `affine3d` object describes the rigid 3-D transform. The iterative closest point (ICP) algorithm estimates the rigid transformation between the moving and fixed point clouds.

### **movingReg — Transformed point cloud**

pointCloud object

Transformed point cloud, returned as a pointCloud object. The transformed point cloud is aligned with the fixed point cloud.

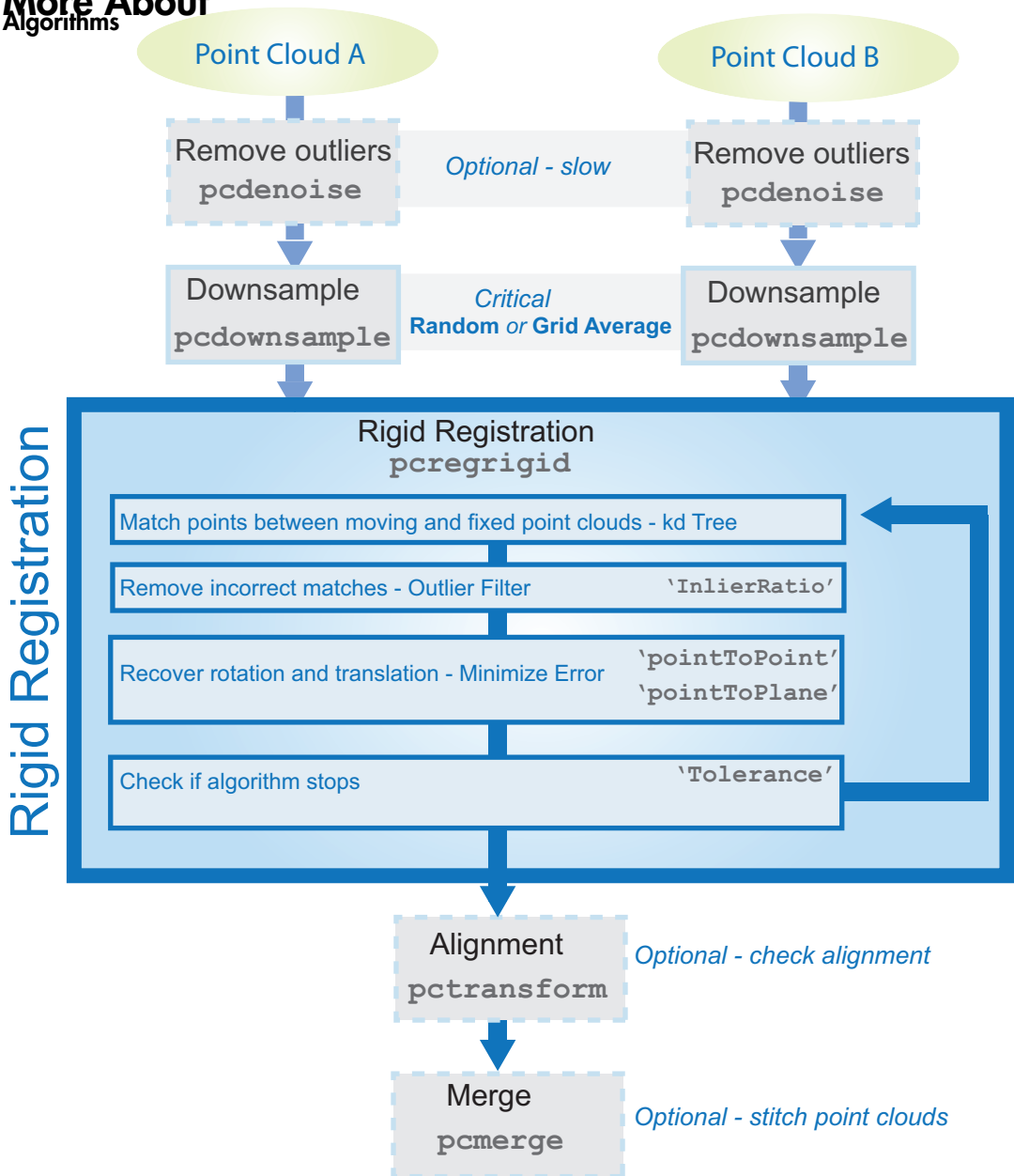
### **rmse — Root mean square error**

positive numeric

Root mean square error, returned as the Euclidean distance between the aligned point clouds.

**More About Algorithms**

Complete Point Cloud Registration Workflow



### References

- [1] Chen, Y. and G. Medioni. "Object Modelling by Registration of Multiple Range Images." *Image Vision Computing*. Butterworth-Heinemann . Vol. 10, Issue 3, April 1992, pp. 145-155.
- [2] Besl, Paul J., N. D. McKay. "A Method for Registration of 3-D Shapes." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Los Alamitos, CA: IEEE Computer Society. Vol. 14, Issue 2, 1992, pp. 239-256.

### See Also

`pointCloud` | `pcplayer` | `planeModel` | `affine3d` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcshow` | `pctransform` | `pcwrite`

**Introduced in R2015a**

# pcwrite

Write 3-D point cloud to PLY file

## Syntax

```
pcwrite(filename)
pcwrite(ptCloud,filename,'PLYFormat',format)
```

## Description

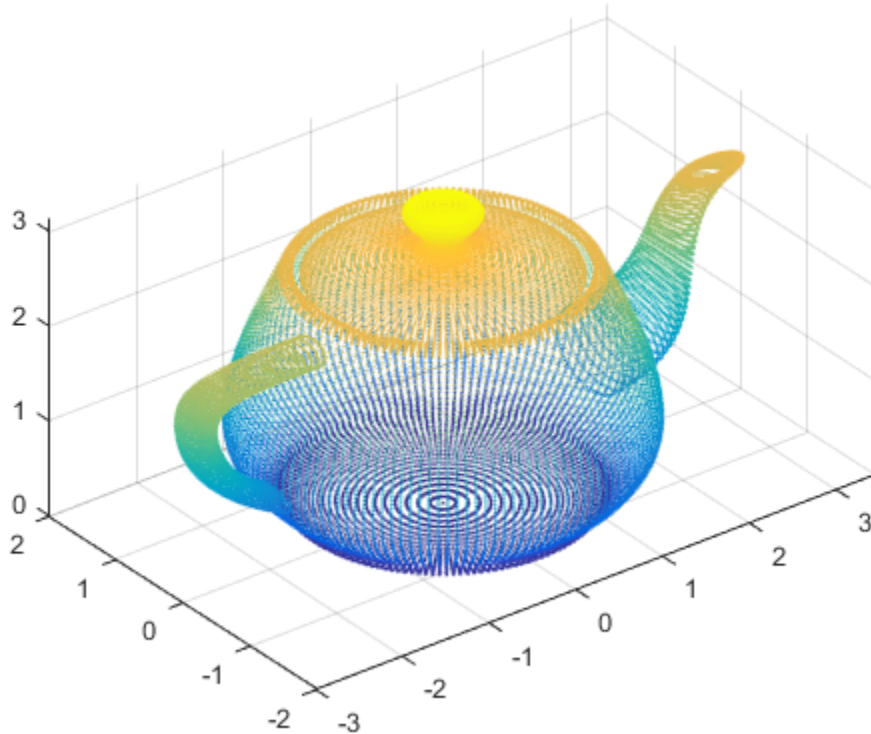
`pcwrite(filename)` writes a point cloud object to the PLY file specified by the input `filename` string.

`pcwrite(ptCloud,filename,'PLYFormat',format)` writes a `pointCloud` object, `ptCloud`, to a PLY file that is in the specified format.

## Examples

### Write 3-D Point Cloud to PLY File

```
ptCloud = pcread('teapot.ply');
pcshow(ptCloud);
pcwrite(ptCloud,'teapotOut','PLYFormat','binary');
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

**filename** — PLY file name

string

PLY file name, specified as a string. The input file type must be a PLY format file. The `pcwrite` function converts an organized  $M$ -by- $N$ -by-3 point cloud to an unorganized  $M$ -by-3 format. It converts the format because PLY files do not support organized point clouds. To preserve the organized format, you can save the point cloud as a MAT-file.

**ptCloud — Object for storing point cloud**

pointCloud object

Object for storing point cloud, specified as a pointCloud object.

**format — PLY format**

'ascii' (default) | 'ascii' | 'binary'

PLY format, specified as the comma-separated pair consisting of the string `format` and the string `'ascii'` or `'binary'`. Use the binary format to save space when storing the point cloud file on disk.

**See Also**

pointCloud | pcplayer | planeModel | pcdnoise | pcdsample | pcfplane |  
pcmerge | pcread | pcregrigid | pcshow | pctransform

**Introduced in R2015a**

## pctransform

Rigid transform of 3-D point cloud

### Syntax

```
ptCloudOut = pctransform(ptCloudIn,tform)
```

### Description

`ptCloudOut = pctransform(ptCloudIn,tform)` applies the specified forward rigid transform to the input point cloud.

### Examples

#### Rotate 3-D Point Cloud

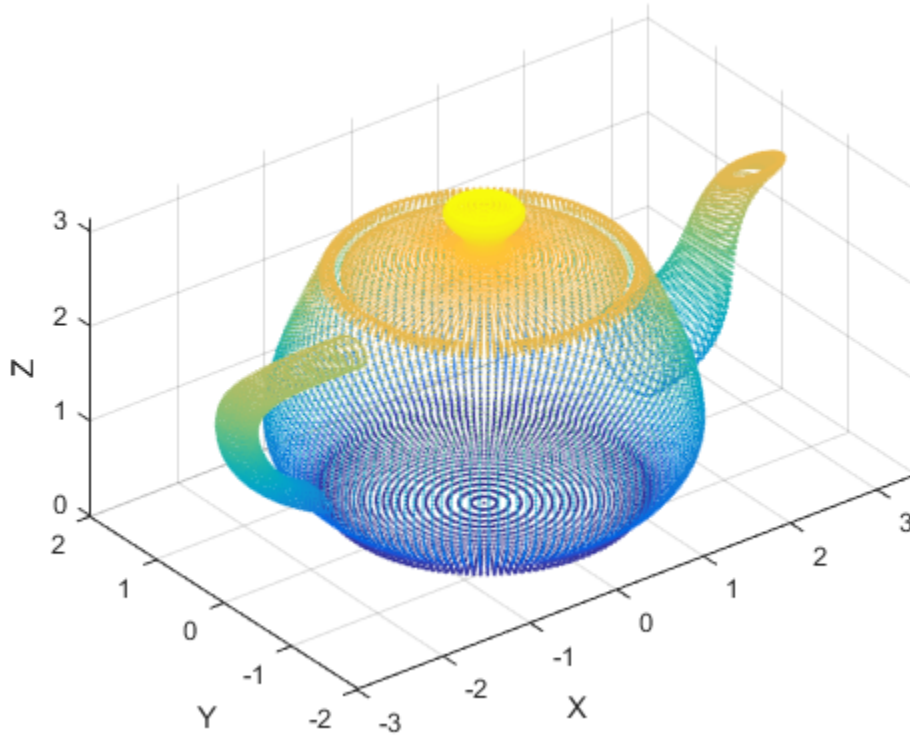
Read a point cloud.

```
ptCloud = pcread('teapot.ply');
```

Plot the original data.

```
figure  
pcshow(ptCloud);  
xlabel('X');  
ylabel('Y');  
zlabel('Z');
```





Create a transform object with a 45 degrees rotation along the  $z$ -axis.

```
A = [cos(pi/4) sin(pi/4) 0 0; ...  
     -sin(pi/4) cos(pi/4) 0 0; ...  
     0 0 1 0; ...  
     0 0 0 1];  
tform = affine3d(A);
```

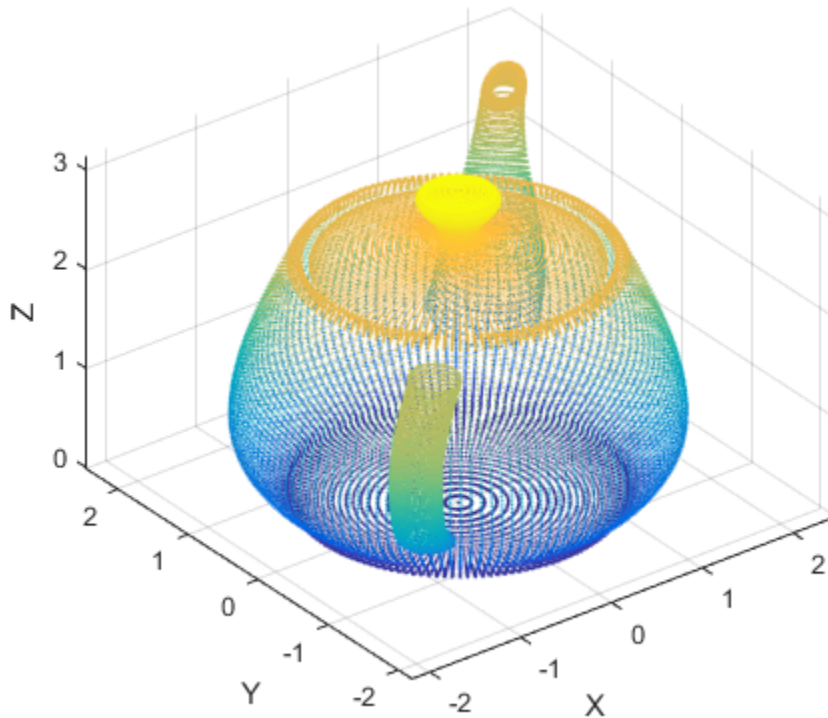
Transform the point cloud.

```
ptCloudOut = pctransform(ptCloud,tform);
```

Plot the transformed point cloud.

```
figure
```

```
pcshow(ptCloudOut);  
xlabel('X');  
ylabel('Y');  
zlabel('Z');
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

**ptCloudIn** — Point cloud  
pointCloud object

Point cloud, specified as a `pointCloud` object.

**tform** — 3-D affine geometric transformation

`affine3d` object

3-D affine geometric transformation, specified as a rigid transform `affine3d` object. The `tform` input must be a valid rigid transform (rotation and translation only). See “Using a Transformation Matrix”, for details on how to set up the `tform` object.

## Output Arguments

**ptCloudOut** — Transformed point cloud

`pointCloud` object

Transformed point cloud, returned as a `pointCloud` object. The transformation applies to the coordinates of points and their normal vectors.

## See Also

`pointCloud` | `pcplayer` | `planeModel` | `affine3d` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcwrite`

**Introduced in R2015a**

## pcnormals

Estimate normals for point cloud

### Syntax

```
normals = pcnormals(ptCloud)
normals = pcnormals(ptCloud,k)
```

### Description

`normals = pcnormals(ptCloud)` returns a matrix that stores a normal for each point in the input `ptCloud`. The function uses six neighboring points to fit a local plane to determine each normal vector.

`normals = pcnormals(ptCloud,k)` additionally specifies `k`, the number of points used for local plane fitting.

### Examples

#### Estimate Normals of Point Cloud

Load a point cloud.

```
load('object3d.mat');
```

Estimate the normal vectors.

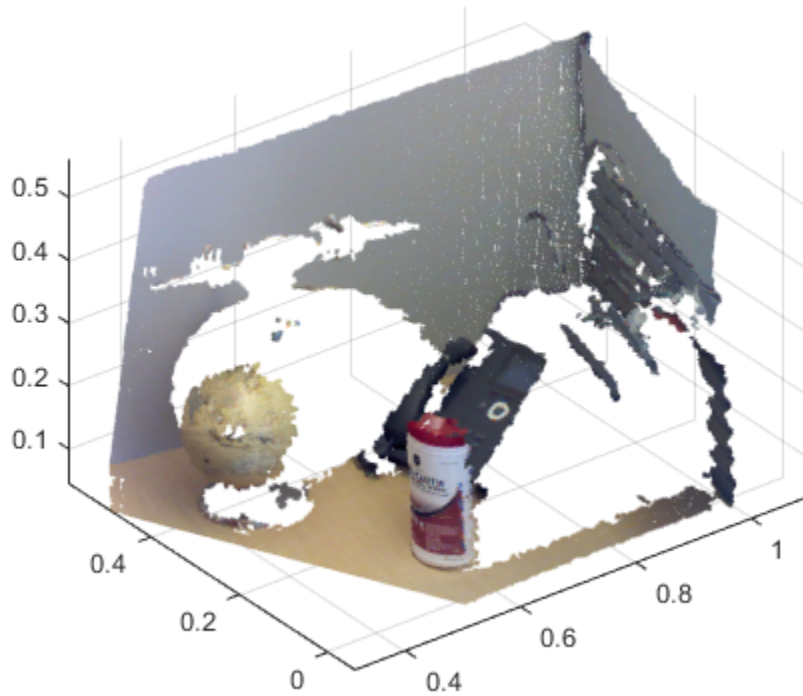
```
normals = pcnormals(ptCloud);

figure
pcshow(ptCloud)
title('Estimated Normals of Point Cloud')
hold on

x = ptCloud.Location(1:10:end,1:10:end,1);
```

```
y = ptCloud.Location(1:10:end,1:10:end,2);  
z = ptCloud.Location(1:10:end,1:10:end,3);  
u = normals(1:10:end,1:10:end,1);  
v = normals(1:10:end,1:10:end,2);  
w = normals(1:10:end,1:10:end,3);
```

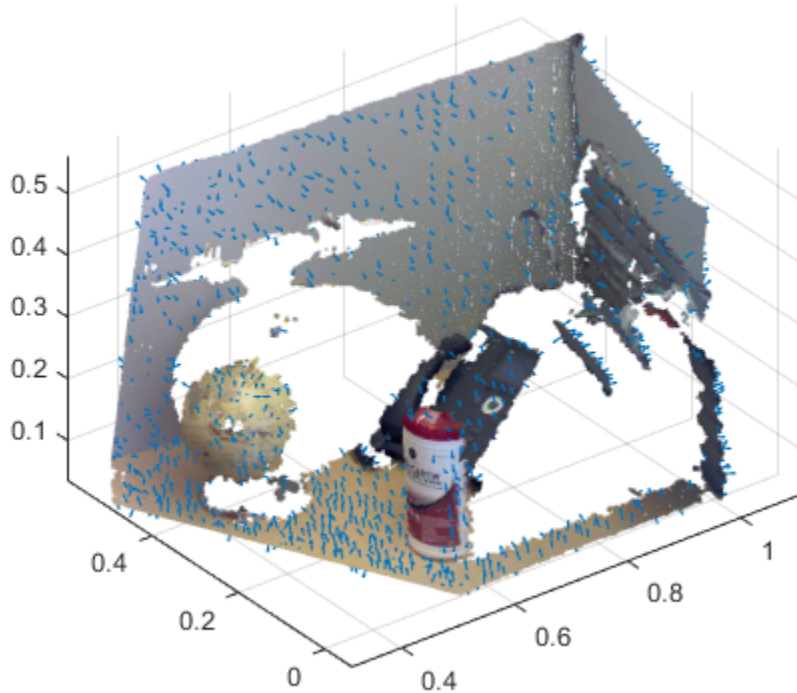
### Estimated Normals of Point Cloud



Plot the normal vectors.

```
quiver3(x,y,z,u,v,w);  
hold off
```

### Estimated Normals of Point Cloud



Flip the normals to point towards the sensor location. This step necessary only for determining the inward or outward direction of the surface. The sensor center is set in  $x$ ,  $y$ ,  $z$  coordinates.

```

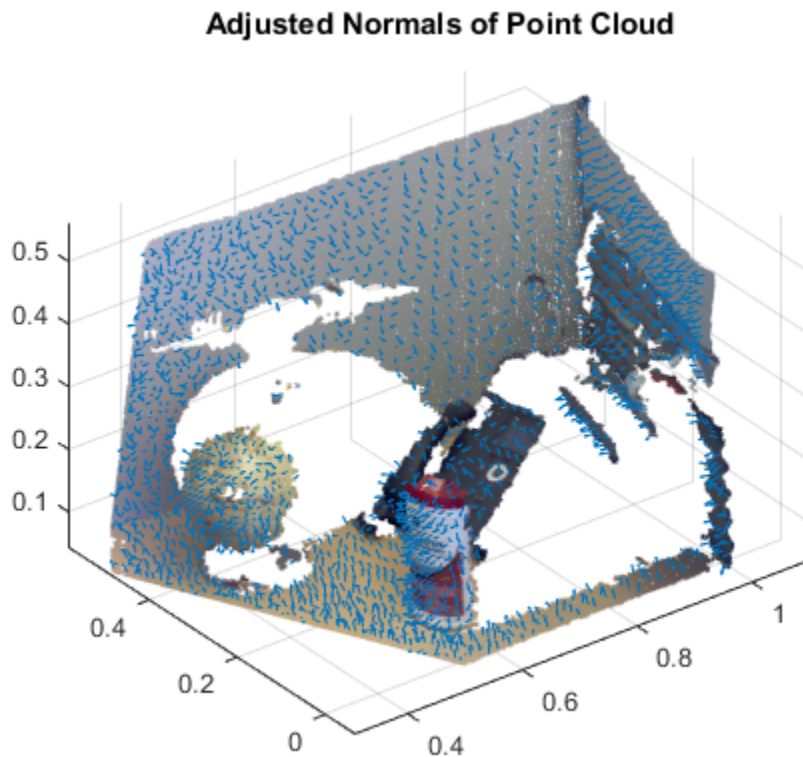
sensorCenter = [0, -0.3, 0.3];
for k = 1 : numel(x)
    p1 = sensorCenter - [x(k), y(k), z(k)];
    p2 = [u(k), v(k), w(k)];
    % Flip the normal vector if it is not pointing towards the sensor.
    angle = atan2(norm(cross(p1, p2)), p1*p2');
    if angle > pi/2 || angle < -pi/2
        u(k) = -u(k);
        v(k) = -v(k);
        w(k) = -w(k);
    end
end

```

```
end  
end
```

Plot the adjusted normals.

```
figure  
pcshow(ptCloud)  
title('Adjusted Normals of Point Cloud')  
hold on  
quiver3(x, y, z, u, v, w);  
hold off
```



- “3-D Point Cloud Registration and Stitching”

### Input Arguments

#### **ptCloud** — Object for storing point cloud

pointCloud object

Object for storing point cloud, returned as a pointCloud object.

#### **k** — Number of points used for local plane fitting

integer greater than or equal to 3

Number of points used for local plane fitting, specified as an integer greater than or equal to 3. Increasing this value improves accuracy but slows down computation time.

### Output Arguments

#### **normals** — Normals used to fit a local plane

$M$ -by-3 |  $M$ -by- $N$ -by-3

Normals used to fit a local plane, returned as an  $M$ -by-3 or an  $M$ -by- $N$ -by-3 vector. The normal vectors are computed locally using six neighboring points. The direction of each normal vector can be set based on how you acquired the points. The “Estimate Normals of Point Cloud” on page 3-378 example, shows how to set the direction when the normal vectors are pointing towards the sensor.

### More About

- The PLY Format

### References

- [1] Hoppe, H., T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. "Surface Reconstruction from Unorganized Points". *Computer Graphics (SIGGRAPH 1992 Proceedings)*. 1992, pp. 71–78.

### See Also

pointCloud | pcplayer | planeModel | pcdnoise | pcdsample | pcfplane | pcmerge | pcregrid | pcshow | pctransform | pcwrite



**Introduced in R2015b**

## pcfitcylinder

Fit cylinder to 3-D point cloud

### Syntax

```
model = pcfitcylinder(ptCloudIn,maxDistance)
model = pcfitcylinder(ptCloudIn,maxDistance,referenceVector)
model = pcfitcylinder(ptCloudIn,maxDistance,referenceVector,
maxAngularDistance)
```

```
[model,inlierIndices,outlierIndices] = pcfitcylinder(ptCloudIn,
maxDistance)
[ ____,rmse] = pcfitcylinder(ptCloudIn,maxDistance)
[ ____ ] = pcfitcylinder( ____,Name,Value)
```

### Description

`model = pcfitcylinder(ptCloudIn,maxDistance)` fits a cylinder to a point cloud with a maximum allowable distance from an inlier point to the cylinder.

`model = pcfitcylinder(ptCloudIn,maxDistance,referenceVector)` fits a cylinder to the point cloud with additional orientation constraints specified by the 1-by-3 reference orientation input vector.

`model = pcfitcylinder(ptCloudIn,maxDistance,referenceVector,maxAngularDistance)` additionally specifies the maximum allowed absolute angular distance.

`[model,inlierIndices,outlierIndices] = pcfitcylinder(ptCloudIn,maxDistance)` additionally returns linear indices to the inlier and outlier points in the point cloud input.

`[ ____,rmse] = pcfitcylinder(ptCloudIn,maxDistance)` additionally returns the root mean square error of the distance of the inlier points to the model.

`[ ____ ] = pcfitcylinder( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Examples

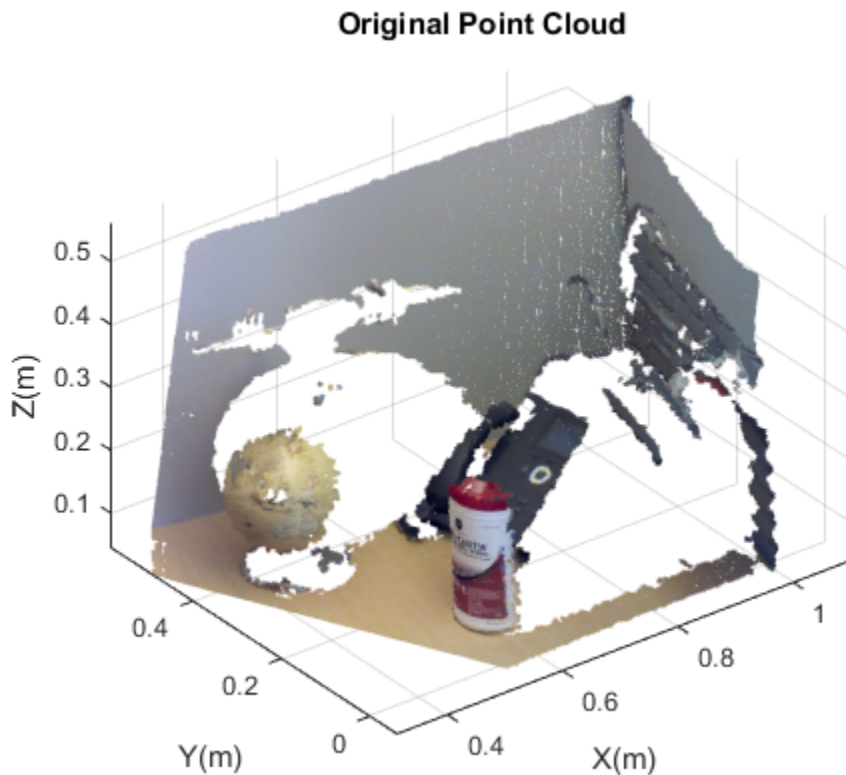
### Extract Cylinder from Point Cloud

Load the point cloud.

```
load('object3d.mat');
```

Display the point cloud.

```
figure  
pcshow(ptCloud)  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
title('Original Point Cloud')
```



Set the maximum point-to-cylinder distance (5 mm) for cylinder fitting.

```
maxDistance = 0.005;
```

Set the region of interest to constrain the search.

```
roi = [0.4,0.6,-inf,0.2,0.1,inf];  
sampleIndices = findPointsInROI(ptCloud,roi);
```

Set the orientation constraint.

```
referenceVector = [0,0,1];
```

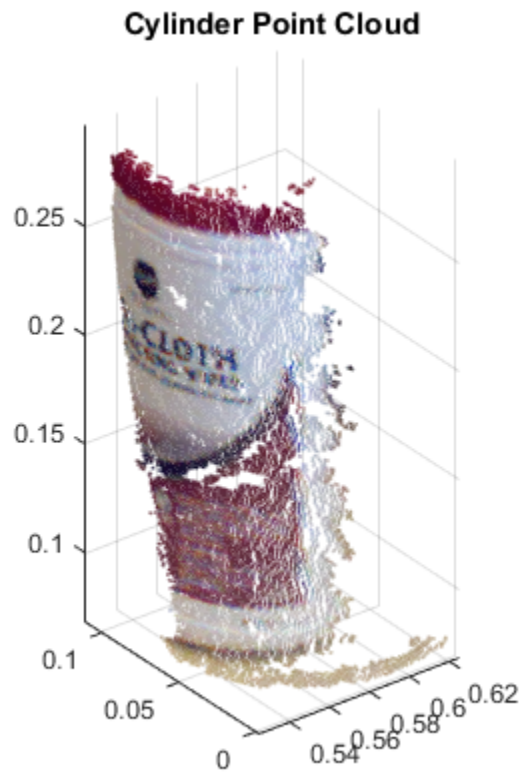
Detect the cylinder and extract it from the point cloud by specifying the inlier points.

```
[model,inlierIndices] = pcfitcylinder(ptCloud,maxDistance,referenceVector,'SampleIndices');
```

```
pc = select(ptCloud,inlierIndices);
```

Plot the extracted cylinder.

```
figure  
pcshow(pc)  
title('Cylinder Point Cloud')
```



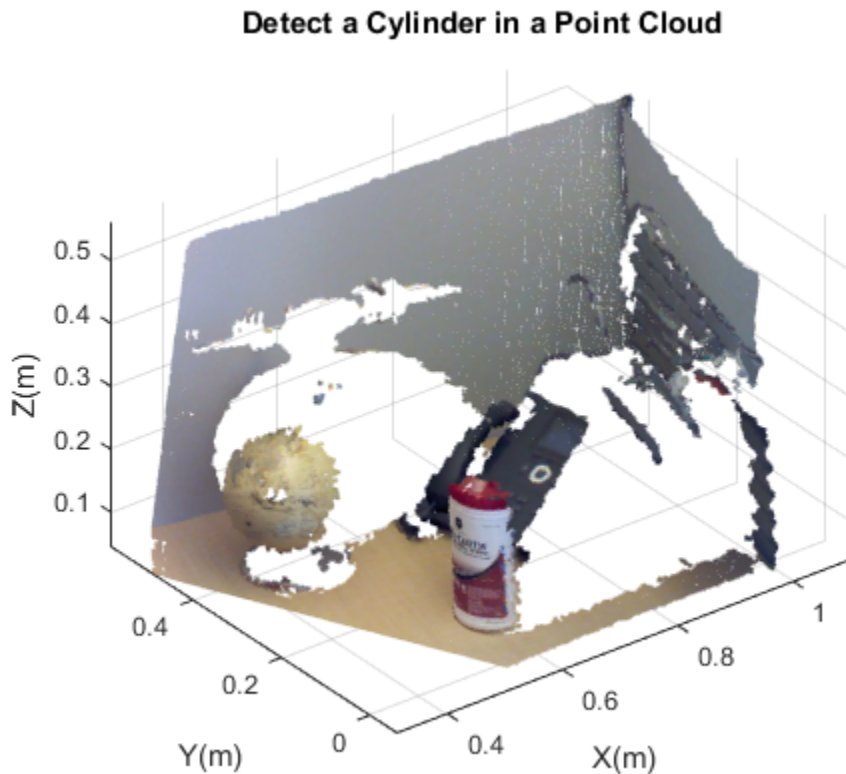
### Detect Cylinder in Point Cloud

Load the point cloud.

```
load('object3d.mat');
```

Display point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Detect a Cylinder in a Point Cloud')
```



Set the maximum point-to-cylinder distance (5 mm) for the cylinder fitting.

```
maxDistance = 0.005;
```

Set the region of interest to constrain the search.

```
roi = [0.4,0.6;-inf,0.2;0.1,inf];
sampleIndices = findPointsInROI(ptCloud,roi);
```

Set the orientation constraint.

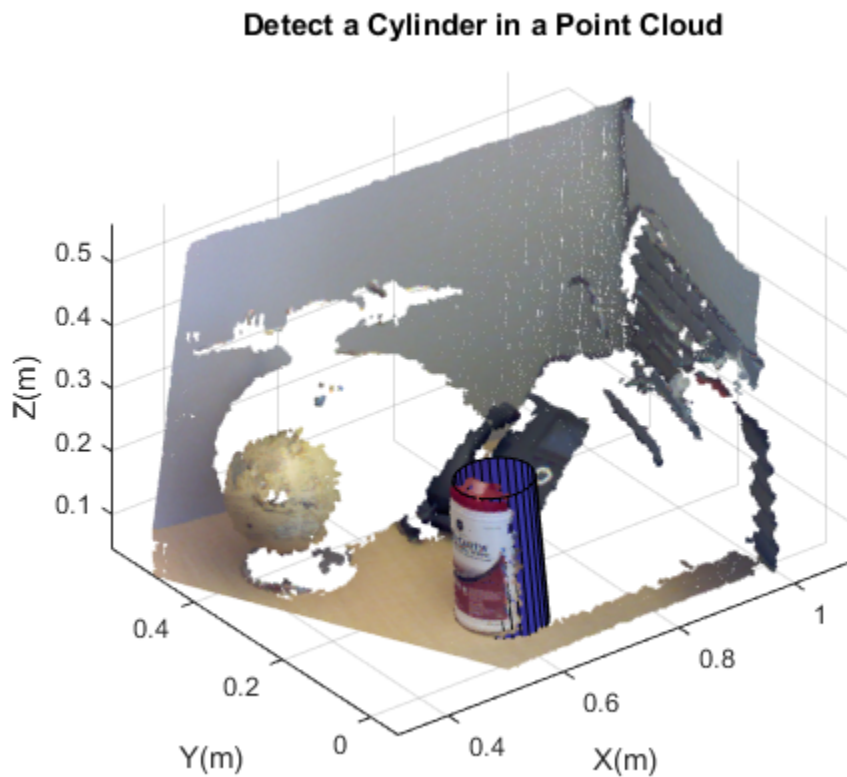
```
referenceVector = [0,0,1];
```

Detect the cylinder in the point cloud and extract it.

```
model = pcfitcylinder(ptCloud,maxDistance,referenceVector,'SampleIndices',sampleIndices)
```

Plot the cylinder.

```
hold on  
plot(model)
```



- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **ptCloudIn** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object. If the `Normal` property of the input `ptCloud` is empty, the function populates it with values to meet the requirements of the fitting algorithm.

### **maxDistance** — Maximum distance from an inlier point to the cylinder

scalar value

Maximum distance from an inlier point to the cylinder, specified as a scalar value. Specify the distance in units that are consistent with the units you are using for the point cloud.

Data Types: `single` | `double`

### **referenceVector** — Reference orientation

1-by-3 vector

Reference orientation, specified as a 1-by-3 vector.

### **maxAngularDistance** — Maximum absolute angular distance

5 (default) | scalar value

Maximum absolute angular distance, specified as a scalar value. The maximum angular distance is measured in degrees between the direction of the fitted cylinder and the reference orientation.

## Name-Value Pair Arguments

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

Example: `'SampleIndices', []`.

### **'SampleIndices'** — Linear indices of points to sample

`[]` (default) | column vector



Linear indices of points to sample in the input point cloud, specified as the comma-separated pair consisting of 'SampleIndices' and a column vector. An empty vector means that all points are candidates to sample when fitting the cylinder during the RANSAC iteration. If you specify a subset of points, the function fits the model by sampling only those points in the subset. Providing a subset of points can significantly speed up the process by reducing the number of trials. You can generate the indices vector using the findPointsInROI method of the pointCloud object.

**'MaxNumTrials' — Maximum number of random trials**

1000 (default) | positive integer

Maximum number of random trials for finding inliers, specified as the comma-separated pair consisting of 'MaxNumTrials' and a positive integer. To improve robustness of the output, increase this value. However, doing so adds additional computations.

**'Confidence' — Percentage for finding maximum number of inliers**

99 (default) | numeric scalar in the range (0,100)

Percentage for finding maximum number of inliers, specified as the comma-separated pair consisting of 'Confidence' and a numeric scalar, in the range (0 100). To improve the robustness of the output, increase this value. However, doing so adds additional computations.

## Output Arguments

**model — Geometric model of cylinder**

cylinderModel object.

Geometric model of cylinder, returned as a cylinderModel object.

The coefficients for the output model are set to zero when:

- The input point cloud does not contain enough valid points.
- The algorithm cannot find enough inlier points.

**inlierIndices — Linear indices of inlier points**

column vector

Linear indices of the inlier points in the input point cloud, returned as a column vector.

### **outlierIndices** — Linear indices of outlier points

column vector

Linear indices of the outlier points in the input point cloud returned as a column vector.

### **rmse** — Root mean square error

scalar value

Root mean square error, returned as a scalar value.

## More About

### Algorithms

The function returns a geometric model that describes the cylinder. This function uses the M-estimator SAMple Consensus (MSAC) algorithm to find the cylinder. The MSAC algorithm is a variant of the RANdom SAMple Consensus (RANSAC) algorithm.

The fitting algorithm for the `pcfitcylinder` function requires point cloud normals. Therefore, if the `Normal` property for the input point cloud is empty, the function fills it. When the function fills the `Normal` property, it uses six points to fit the local cylinder. If six points do not work and the fitting fails, consider calling the `pcnormals` function which enables you to select the number of points to use.

### References

- [1] Torr, P. H. S., and A. Zisserman. “MLE-SAC: A New Robust Estimator with Application to Estimating Image Geometry.” *Computer Vision and Image Understanding*. Volume 78, Issue 1, April 2000, pp. 138-156.

### See Also

`pointCloud` | `pcplayer` | `cylinderModel` | `affine3d` | `pcdenoise` | `pcfitplane` | `pcfitsphere` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pctransform` | `pcwrite`

Introduced in R2015b

# pcfitplane

Fit plane to 3-D point cloud

## Syntax

```
model = pcfitplane(ptCloudIn,maxDistance)
model = pcfitplane(ptCloudIn,maxDistance,referenceVector)
model = pcfitplane(ptCloudIn,maxDistance,referenceVector,
maxAngularDistance)

[model,inlierIndices,outlierIndices] = pcfitplane(ptCloudIn,
maxDistance)
[ ____,rmse] = pcfitplane(ptCloudIn,maxDistance)
[ ____ ] = pcfitplane(ptCloudIn,maxDistance,Name,Value)
```

## Description

`model = pcfitplane(ptCloudIn,maxDistance)` fits a plane to a point cloud that has a maximum allowable distance from an inlier point to the plane. The function returns a geometrical model that describes the plane.

This function uses the M-estimator SAmple Consensus (MSAC) algorithm to find the plane. The MSAC algorithm is a variant of the RANdom SAmple Consensus (RANSAC) algorithm.

`model = pcfitplane(ptCloudIn,maxDistance,referenceVector)` fits a plane to a point cloud that has additional orientation constraints specified by the 1-by-3 `referenceVector` input.

`model = pcfitplane(ptCloudIn,maxDistance,referenceVector,maxAngularDistance)` fits a plane to a point cloud that has a specified maximum angular distance.

`[model,inlierIndices,outlierIndices] = pcfitplane(ptCloudIn,maxDistance)` additionally returns the linear indices to the inlier and outlier points in the point cloud input.

[ \_\_\_\_,rmse] = `pcfitplane`(ptCloudIn,maxDistance) additionally returns root mean square error of the distance of inlier points to the model, using any of the preceding syntaxes.

[ \_\_\_\_ ] = `pcfitplane`(ptCloudIn,maxDistance,Name,Value) uses additional options specified by one or more Name,Value pair arguments.

## Examples

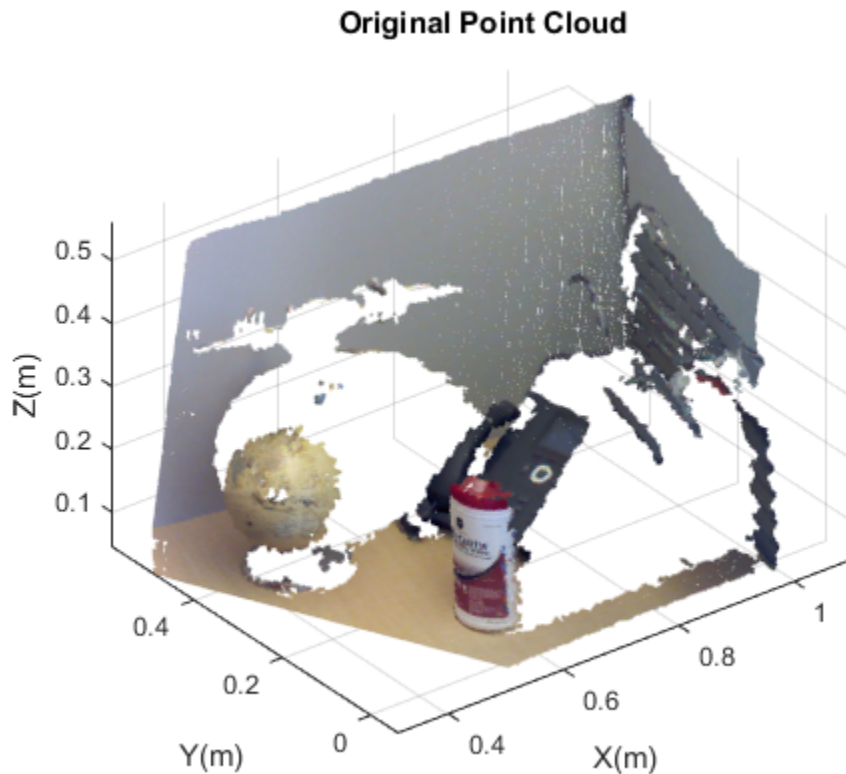
### Detect Multiple Planes from Point Cloud

Load the point cloud.

```
load('object3d.mat')
```

Display and label the point cloud.

```
figure  
pcshow(ptCloud)  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
title('Original Point Cloud')
```



Set the maximum point-to-plane distance (2cm) for plane fitting.

```
maxDistance = 0.02;
```

Set the normal vector of the plane.

```
referenceVector = [0,0,1];
```

Set the maximum angular distance to 5 degrees.

```
maxAngularDistance = 5;
```

Detect the first plane, the table, and extract it from the point cloud.

```
[model1,inlierIndices,outlierIndices] = pcfitplane(ptCloud,maxDistance,referenceVector,
```

```
plane1 = select(ptCloud,inlierIndices);  
remainPtCloud = select(ptCloud,outlierIndices);
```

Set the region of interest to constrain the search for the second plane, left wall.

```
roi = [-inf,inf;0.4,inf;-inf,inf];  
sampleIndices = findPointsInROI(ptCloud,roi);
```

Detect the left wall and extract it from the remaining point cloud.

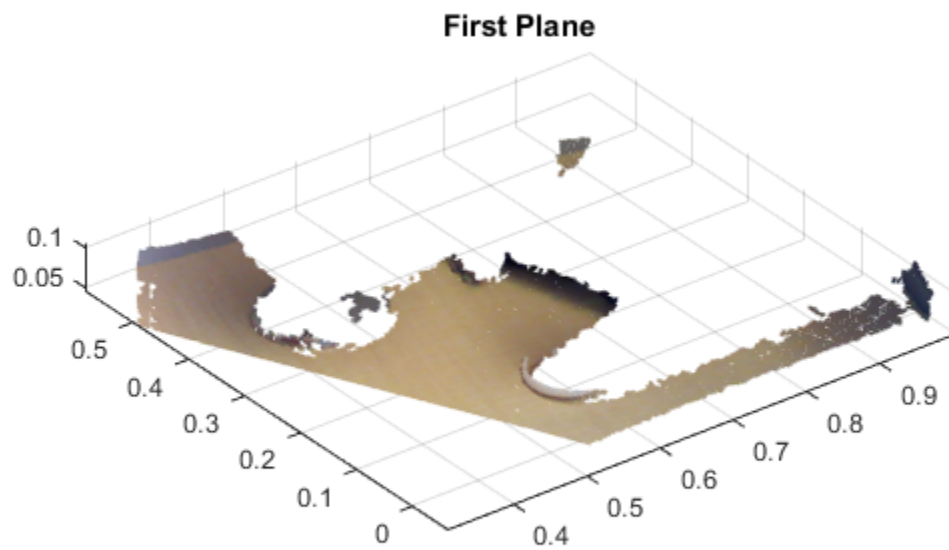
```
[model2,inlierIndices,outlierIndices] = pcfitplane(remainPtCloud,maxDistance, 'SampleIndices');  
plane2 = select(remainPtCloud,inlierIndices);  
remainPtCloud = select(remainPtCloud,outlierIndices);
```

Plot the two planes and the remaining points.

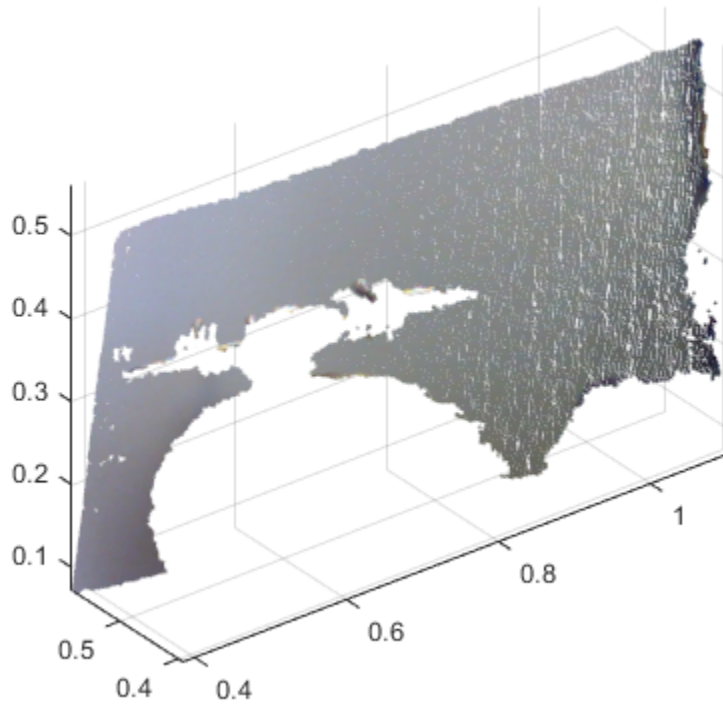
```
figure  
pcshow(plane1)  
title('First Plane')
```

```
figure  
pcshow(plane2)  
title('Second Plane')
```

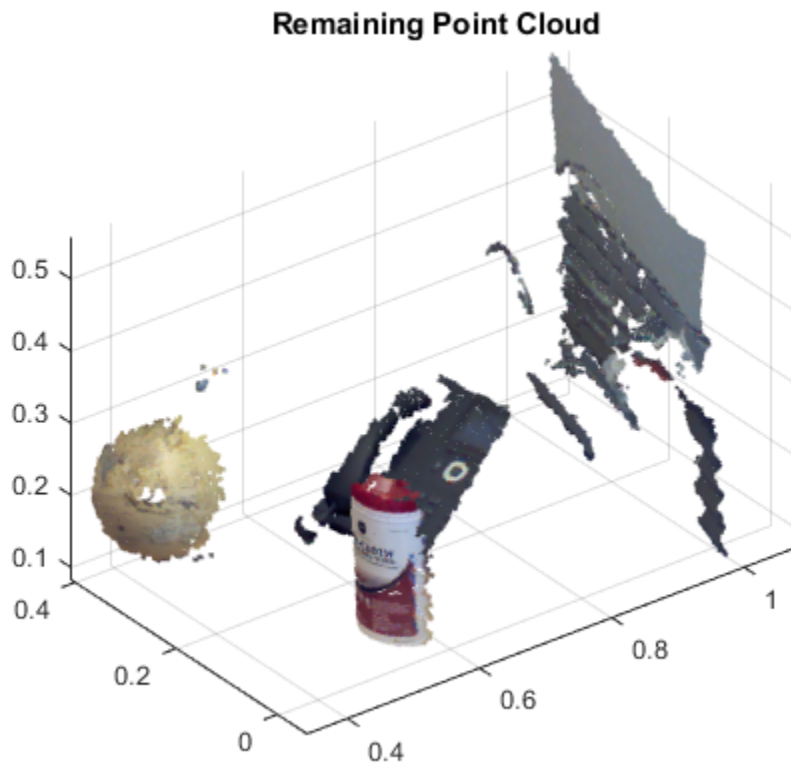
```
figure  
pcshow(remainPtCloud)  
title('Remaining Point Cloud')
```



### Second Plane







- “3-D Point Cloud Registration and Stitching”

## Input Arguments

**ptCloudIn** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

**maxDistance** — Maximum distance from an inlier point to the plane

scalar value

Maximum distance from an inlier point to the plane, specified as a scalar value. Specify the distance in units that are consistent with the units you are using for the point cloud.

Data Types: `single` | `double`

#### **referenceVector** — Reference orientation constraint

1-by-3 vector

Reference orientation constraint, specified as a 1-by-3 vector.

Data Types: `single` | `double`

#### **maxAngularDistance** — Maximum absolute angular distance

5 degrees (default) | scalar value

Maximum absolute angular distance between the normal vector of the fitted plane and the reference orientation, specified as a scalar value in degrees.

Data Types: `single` | `double`

## Name-Value Pair Arguments

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

Example: `'SampleIndices', []`.

#### **'SampleIndices'** — Linear indices of points to be sampled

`[]` (default) | column vector

Linear indices of points to sample in the input point cloud, specified as the comma-separated pair consisting of `'SampleIndices'` and a column vector. An empty vector means that all points are candidates to sample in the RANSAC iteration to fit the plane. When you specify a subset, only points in the subset are sampled to fit a model.

Providing a subset of points can significantly speed up the process and reduce the number of trials. You can generate the indices vector using the `findPointsInROI` method of the `pointCloud` object.

#### **'MaxNumTrials'** — Maximum number of random trials

1000 (default) | positive integer

Maximum number of random trials for finding inliers, specified as the comma-separated pair consisting of 'MaxNumTrials' and a positive integer. Increasing this value makes the output more robust but adds additional computations.

**'Confidence' — Confidence percentage for finding maximum number of inliers**

99 (default) | numeric scalar

Confidence percentage for finding maximum number of inliers, specified as the comma-separated pair consisting of 'Confidence' and a numeric scalar, in the range [0 100]. Increasing this value makes the output more robust but adds additional computations.

## Output Arguments

**model — Geometric model of plane**

planeModel object

Geometric model of plane, returned as a planeModel object.

When the input point cloud does not contain enough valid points, or when the function cannot find enough inlier points, the coefficients for the output model are set to zero.

**inlierIndices — Linear indices of inlier points**

column vector

Linear indices of inlier points within the input point cloud, returned as a column vector.

**outlierIndices — Linear indices of outlier points**

column vector

Linear indices of outlier points within the input point cloud, returned as a column vector.

**rmse — Root mean square error**

scalar value

Root mean square error of the distance of inlier points to the model, returned as a scalar value.

## References

- [1] Torr, P. H. S., and A. Zisserman. "MLE-SAC: A New Robust Estimator with Application to Estimating Image Geometry." *Computer Vision and Image Understanding*. 2000.

### **See Also**

pointCloud | pcplayer | planeModel | affine3d | pcdenoise | pcfitylinder |  
pcfitsphere | pcmerge | pcread | pcregrigid | pcshow | pctransform |  
pcwrite

**Introduced in R2015b**

# pcfitsphere

Fit sphere to 3-D point cloud

## Syntax

```
model = pcfitsphere(ptCloudIn,maxDistance)
[model,inlierIndices,outlierIndices] = pcfitsphere(ptCloudIn,
maxDistance)
[ ___,rmse] = pcfitsphere(ptCloudIn,maxDistance)
[ ___ ] = pcfitsphere( ___,Name,Value)
```

## Description

`model = pcfitsphere(ptCloudIn,maxDistance)` fits a sphere to a point cloud that has a maximum allowable distance from an inlier point to the sphere. The function returns a geometrical model that describes the sphere.

This function uses the M-estimator SAMPLE Consensus (MSAC) algorithm to find the sphere. The MSAC algorithm is a variant of the RANDOM SAMPLE Consensus (RANSAC) algorithm.

`[model,inlierIndices,outlierIndices] = pcfitsphere(ptCloudIn,maxDistance)` additionally returns linear indices to the inlier and outlier points in the point cloud input.

`[ ___,rmse] = pcfitsphere(ptCloudIn,maxDistance)` additionally returns the root mean square error of the distance of inlier points to the model, using any of the preceding syntaxes.

`[ ___ ] = pcfitsphere( ___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Examples

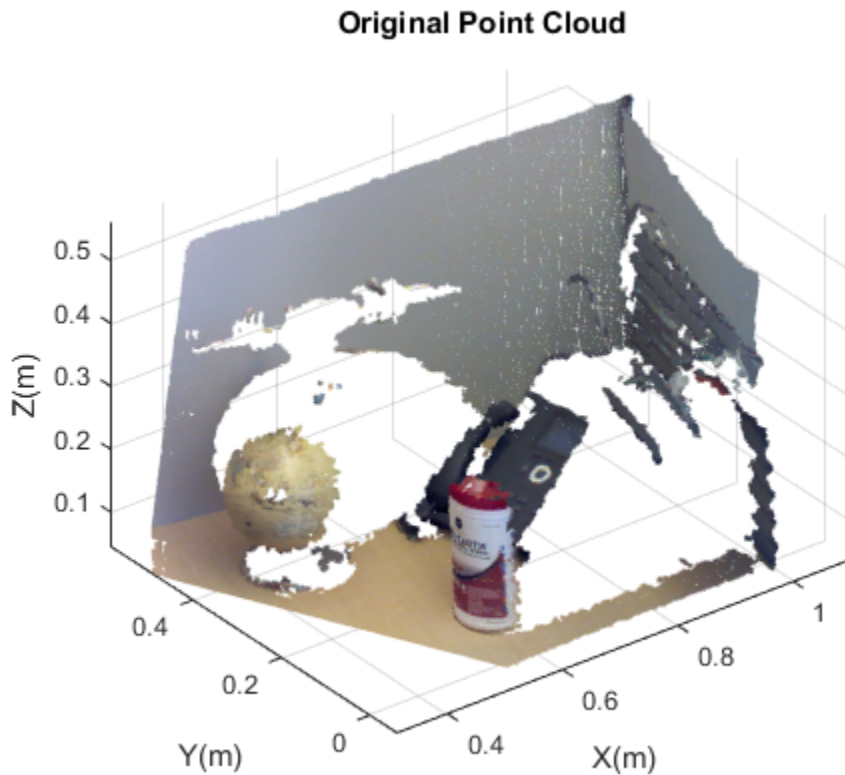
### Detect Sphere from Point Cloud

Load data file.

```
load('object3d.mat');
```

Display original point cloud.

```
figure  
pcshow(ptCloud)  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
title('Original Point Cloud')
```



Set a maximum point-to-sphere distance of 1cm for sphere fitting.

```
maxDistance = 0.01;
```

Set the roi to constrain the search.

```
roi = [-inf,0.5,0.2,0.4,0.1,inf];  
sampleIndices = findPointsInROI(ptCloud,roi);
```

Detect the sphere, a globe, and extract it from the point cloud.

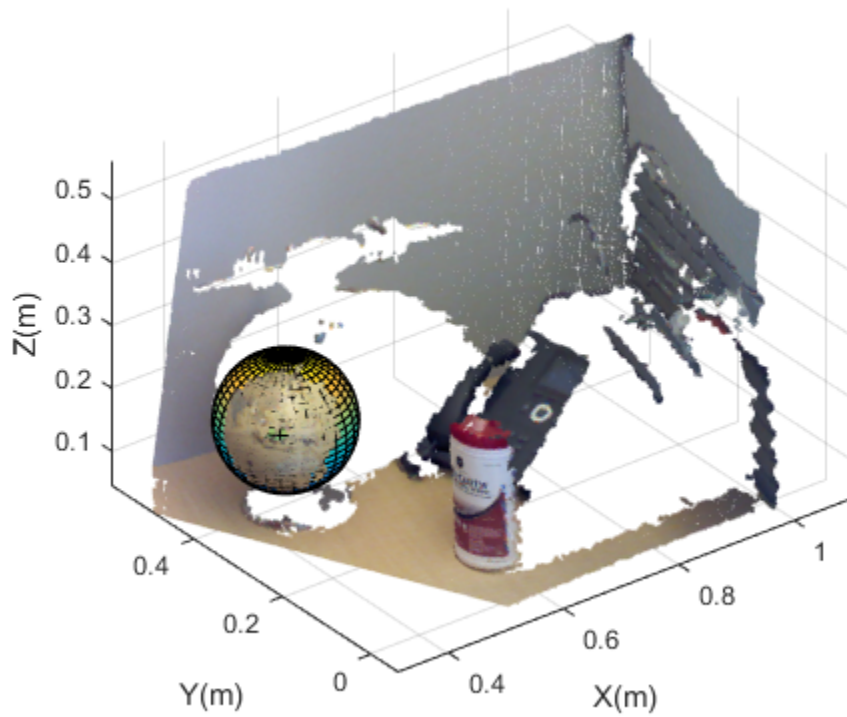
```
[model,inlierIndices] = pcfitsphere(ptCloud,maxDistance,'SampleIndices',sampleIndices)  
globe = select(ptCloud,inlierIndices);
```

Plot the globe.

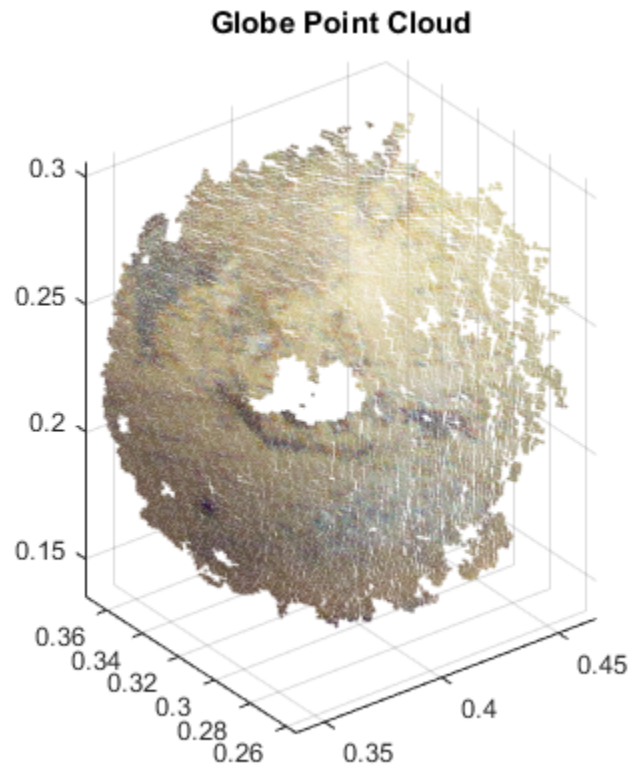
```
hold on  
plot(model)
```

```
figure  
pcshow(globe)  
title('Globe Point Cloud')
```

Original Point Cloud







- “3-D Point Cloud Registration and Stitching”

## Input Arguments

### **ptCloudIn** — Point cloud

pointCloud object

Point cloud, specified as a pointCloud object.

### **maxDistance** — Maximum distance from an inlier point to the sphere

scalar value

Maximum distance from an inlier point to the sphere, specified as a scalar value. Specify the distance in units that are consistent with the units you are using for the point cloud.

Data Types: `single` | `double`

## Name-Value Pair Arguments

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

Example: `'SampleIndices', []`.

### **'SampleIndices'** — Linear indices of points to be sampled

`[]` (default) | column vector

Linear indices of points to sample in the input point cloud, specified as the comma-separated pair consisting of `'SampleIndices'` and a column vector. An empty vector means that all points are candidates to sample in the RANSAC iteration to fit the sphere. When you specify a subset, only points in the subset are sampled to fit a model. Providing a subset of points can significantly speed up the process and reduce the number of trials. You can generate the indices vector using the `findPointsInROI` method of the `pointCloud` object.

### **'MaxNumTrials'** — Maximum number of random trials

1000 (default) | positive integer

Maximum number of random trials for finding inliers, specified as the comma-separated pair consisting of `'MaxNumTrials'` and a positive integer. Increasing this value makes the output more robust but adds additional computations.

### **'Confidence'** — Confidence percentage for finding maximum number of inliers

99 (default) | numeric scalar in the range [0,100]

Confidence percentage for finding maximum number of inliers, specified as the comma-separated pair consisting of `'Confidence'` and a numeric scalar representing percentage, in the range [0,100]. Increasing this value makes the output more robust but adds additional computations.



## plotCamera

Plot a camera in 3-D coordinates

### Syntax

```
cam = plotCamera()  
cam = plotCamera(Name,Value)
```

### Description

`cam = plotCamera()` creates a camera visualization object rendered in the current axes.

`cam = plotCamera(Name,Value)` creates a camera visualization object with the property values specified by one or more `Name,Value` pair arguments.

### Examples

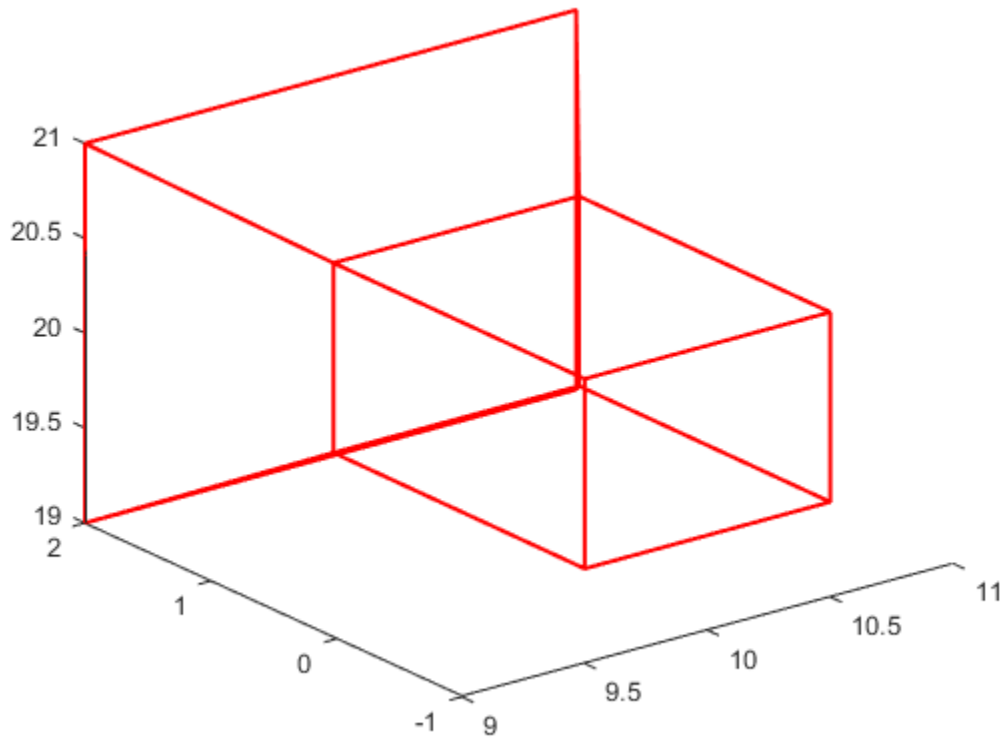
#### Create Animated Camera Plot

Plot a camera pointing along the *y*-axis.

```
R = [1    0    0;  
     0    0   -1;  
     0    1    0];
```

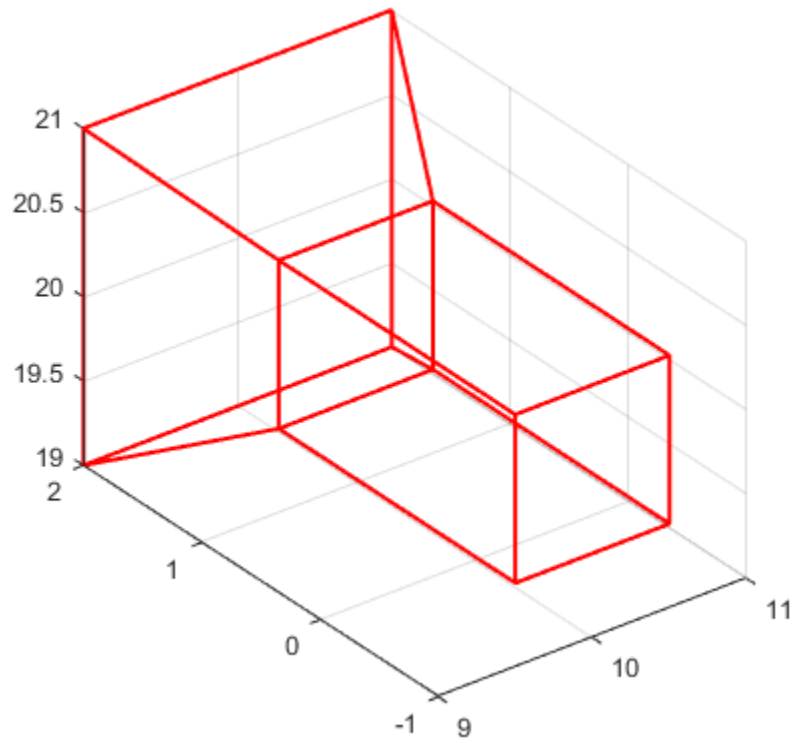
Set the opacity of the camera to zero for faster animation.

```
cam = plotCamera('Location',[10 0 20],'Orientation',R,'Opacity',0);
```



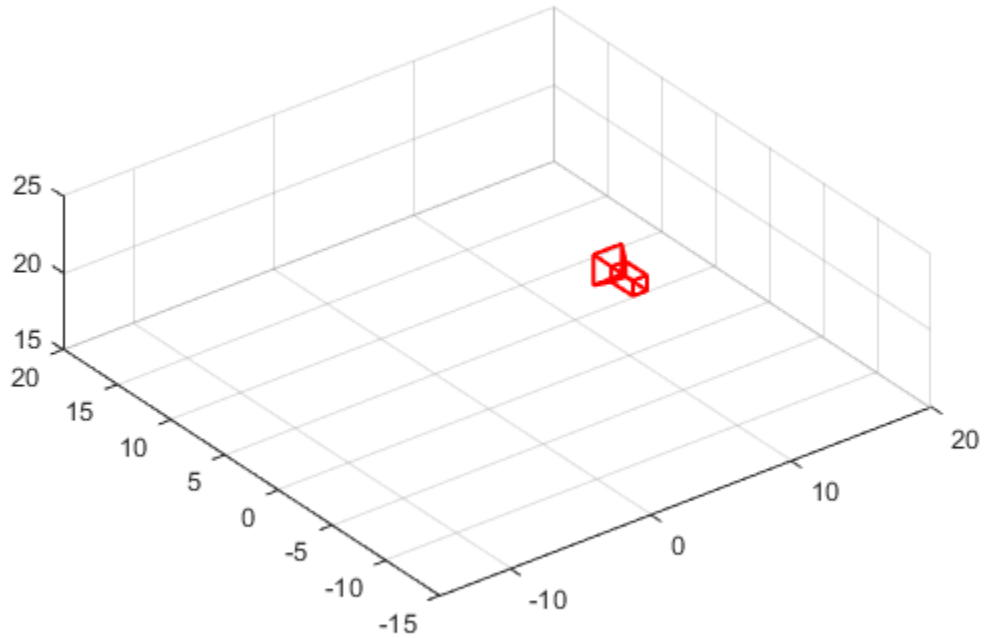
Set the view properties.

```
grid on  
axis equal  
axis manual
```



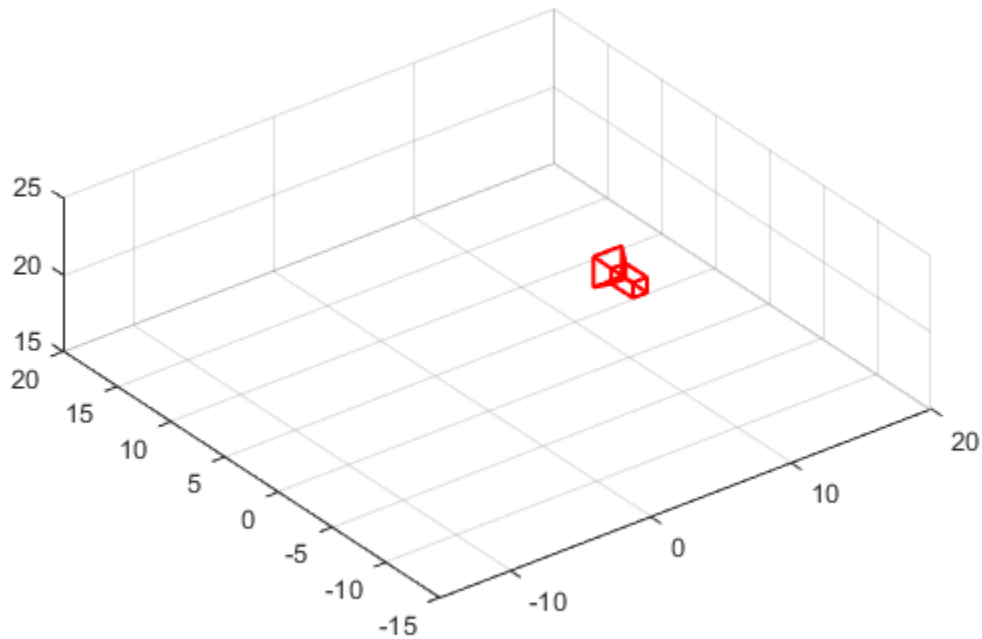
Make the space large enough for the animation.

```
xlim([-15,20]);  
ylim([-15,20]);  
zlim([15,25]);
```



Rotate the camera about the camera's y -axis.

```
for theta = 0:pi/64:10*pi
    T = [cos(theta) 0 sin(theta);
         0 1 0;
         -sin(theta) 0 cos(theta)];
    cam.Orientation = T * R;
    cam.Location = [10 * cos(theta), 10 * sin(theta), 20];
    drawnow();
end
```



#### Visualize Camera Extrinsic

Create a set of calibration images.

```
images = imageSet(fullfile(toolboxdir('vision'),'visiondata','calibration','slr'));
```

Detect the checkerboard corners in the images.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images.ImageLocation);
```

Generate the world coordinates of the checkerboard corners in the pattern-centric coordinate system, with the upper-left corner at (0,0). Set the square size to 29 mm.

```
squareSize = 29;
```



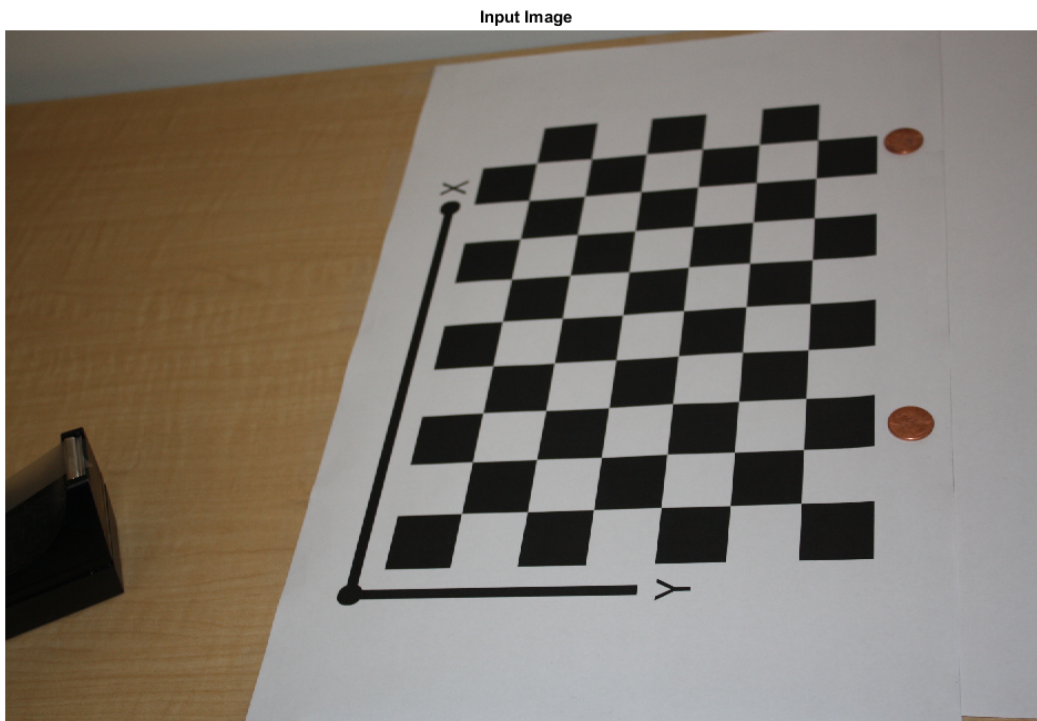
```
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the camera.

```
cameraParams = estimateCameraParameters(imagePoints,worldPoints);
```

Load an image at its new location.

```
imOrig = imread(fullfile(toolboxdir('vision'),'visiondata','calibration','slr','image9  
figure; imshow(imOrig,'InitialMagnification',50);  
title('Input Image');
```



Undistort the image.

```
im = undistortImage(imOrig,cameraParams);
```

Find the reference object in the new image.

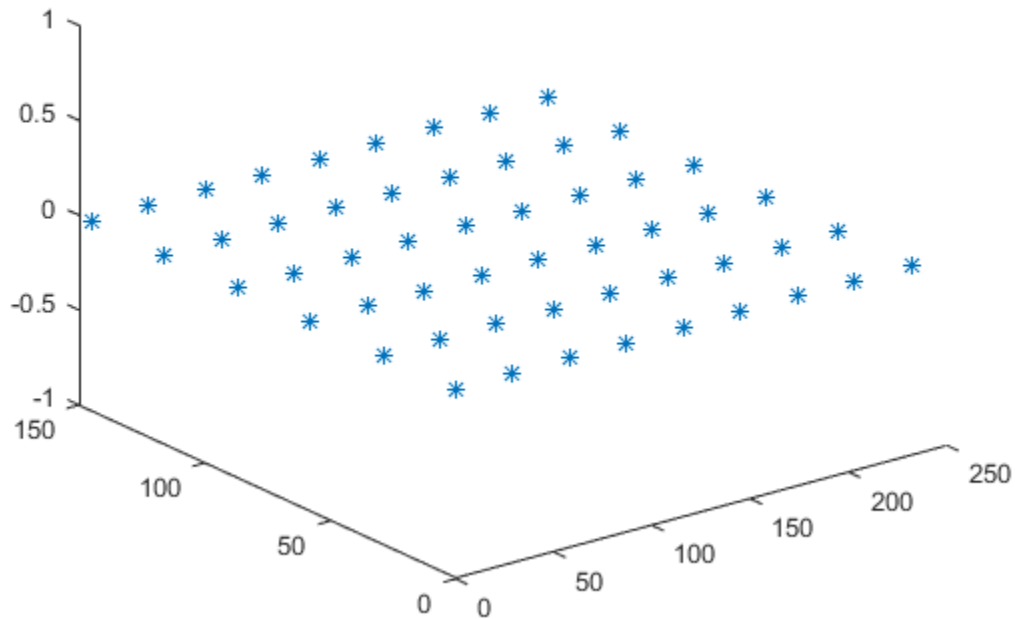
```
[imagePoints,boardSize] = detectCheckerboardPoints(im);
```

Compute the new extrinsics.

```
[rotationMatrix,translationVector] = extrinsics(imagePoints,worldPoints,cameraParams);
```

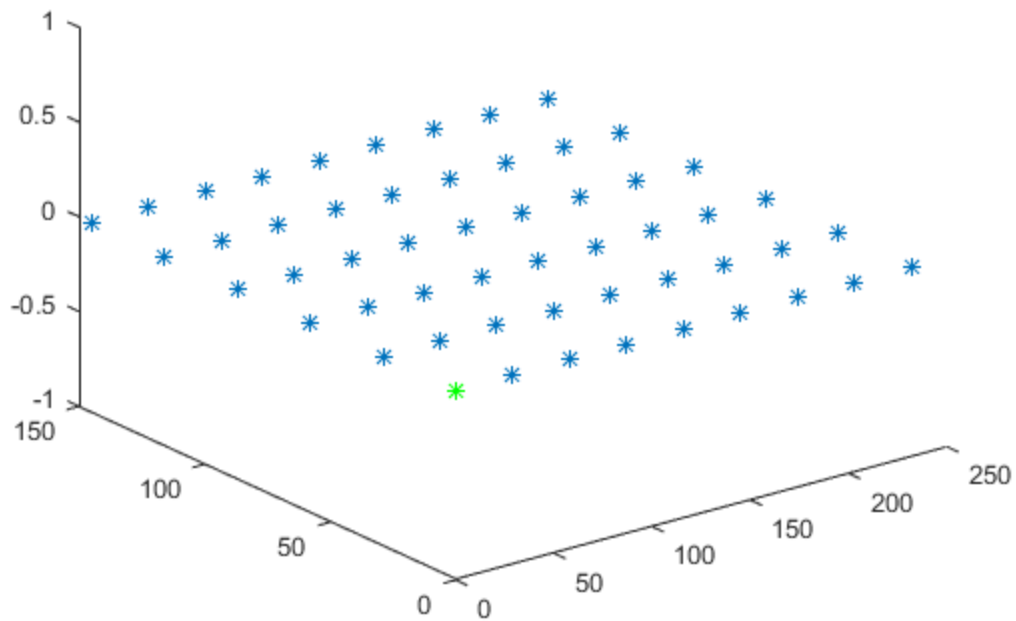
Plot the world points.

```
figure;  
plot3(worldPoints(:,1), worldPoints(:,2),zeros(size(worldPoints, 1),1),'*');  
hold on
```



Mark the origin.

```
plot3(0,0,0, 'g*');
```

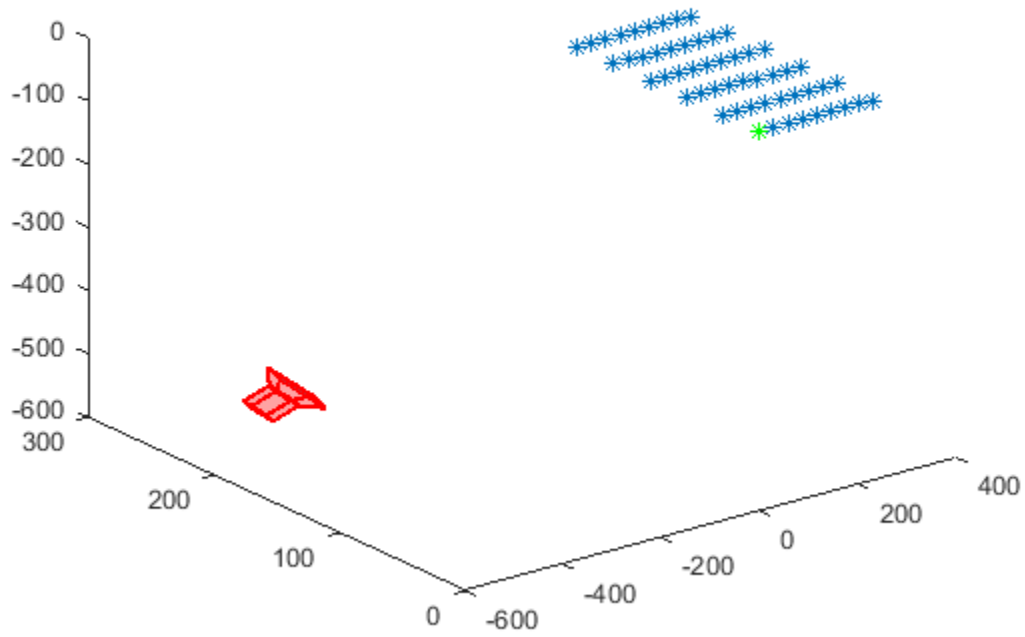


Compute the camera location and orientation.

```
orientation = rotationMatrix';  
location = -translationVector * orientation;
```

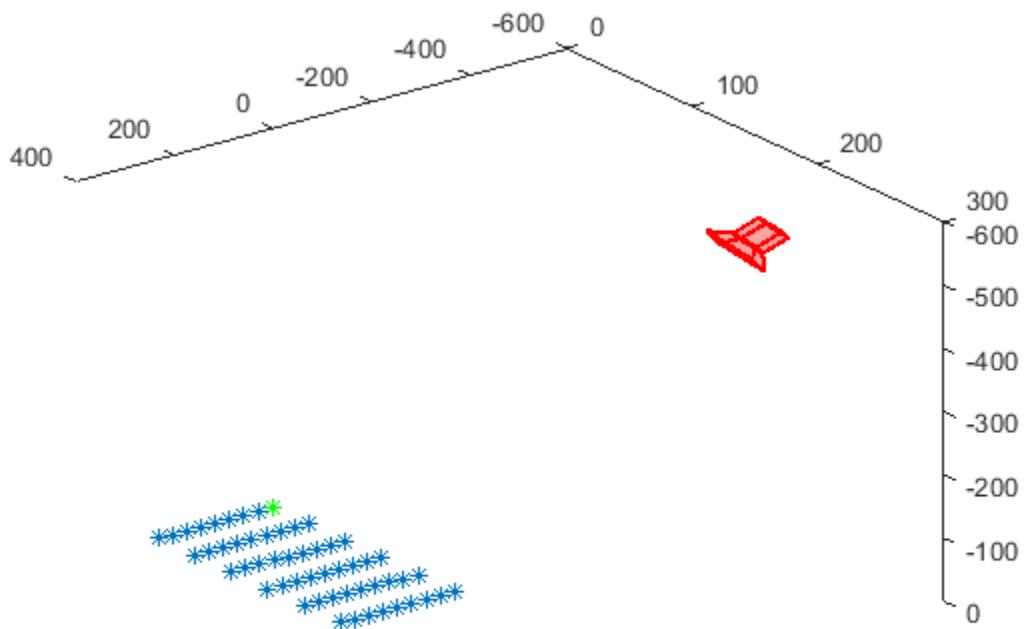
Plot the camera.

```
cam = plotCamera('Location',location,'Orientation',orientation,'Size',20);
```



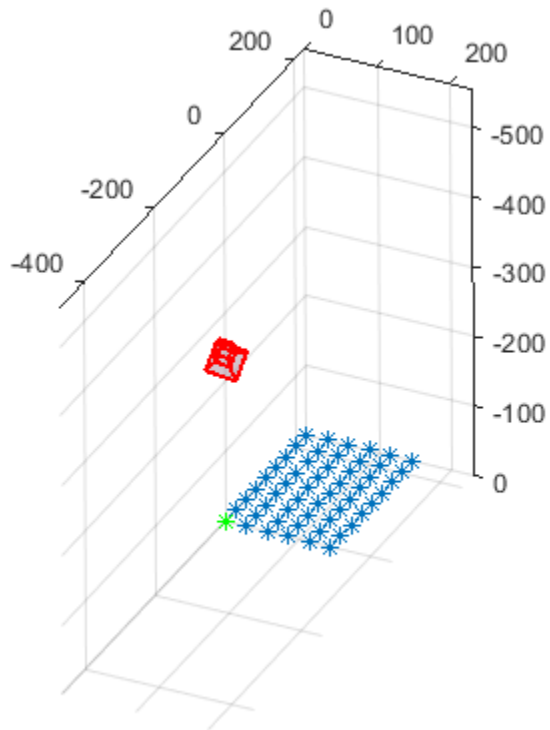
Make the  $z$ -axis point down.

```
set(gca, 'CameraUpVector', [0 0 -1]);
```



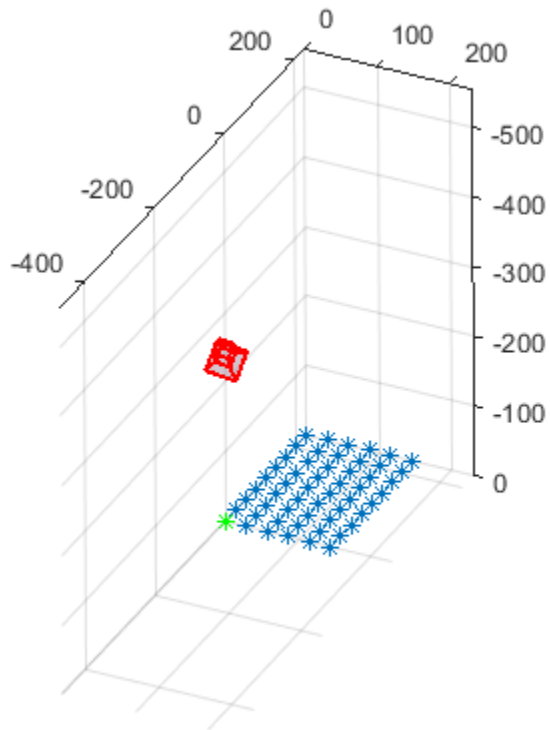
Set the view parameters.

```
camorbit(gca, -110, 60, 'data', [0 0 1]);  
axis equal  
grid on
```



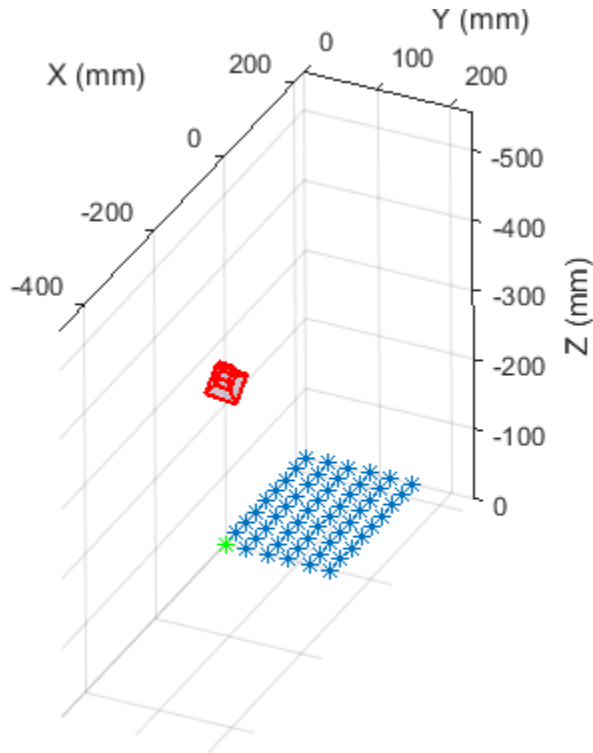
Turn on 3-D rotation.

```
cameratoolbar('SetMode','orbit');
```



Label the axes.

```
xlabel('X (mm)');  
ylabel('Y (mm)');  
zlabel('Z (mm)');
```



- “Structure From Motion From Two Views”

## Input Arguments

### Name-Value Pair Arguments

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



Example: 'Location', [0,0,0]

**'Location' — Camera location coordinates**

[0,0,0] (default) | 3-element vector

Camera location coordinates, specified as the comma-separated pair consisting of 'Location' and a 3-element vector. The  $[x, y, z]$  coordinates are specified in the data units of the parent axes.

**'Orientation' — Matrix orientation**

eye(3) (default) | 3-by-3 3-D rotation matrix

Matrix orientation, specified as the comma-separated pair consisting of 'Orientation' and a 3-by-3 3-D rotation matrix.

**'Size' — Camera base width**

1 | scalar

Camera base width, specified as the comma-separated pair consisting of 'Size' and a scalar.

**'Label' — Camera label**

' ' (default) | string

Camera label, specified as the comma-separated pair consisting of 'Label' and a string.

**'Color' — Camera color**

[1 0 0] (default) | string | 3-element vector

Camera color, specified as the comma-separated pair consisting of 'Color' and a string or a 3-element vector of RGB values in the range [0 1]. See `colspec` for more information on how to specify an RGB color.

**'Opacity' — Camera opacity**

0.2 | scalar in the range [0 1]

Camera opacity, specified as the comma-separated pair consisting of 'Opacity' and a scalar in the range [0 1].

**'Visible' — Camera visibility**

true (default) | false

Camera visibility, specified as the comma-separated pair consisting of 'Visible' and the logical `true` or `false`.

**'AxesVisible' — Camera axes visibility**

`false` (default) | `true`

Camera axes visibility, specified as the comma-separated pair consisting of 'AxesVisible' and the logical `true` or `false`.

**'ButtonDownFcn' — Callback function**

`' '` | function name

Callback function, specified as the comma-separated pair consisting of 'ButtonDownFcn' and a function name that executes when you click the camera.

**'Parent' — Output axes**

`gca` handle

Output axes, specified as the comma-separated pair consisting of 'Parent' and an axes handle. The default is set to the current axes handle, `gca`.

## More About

- “3-D Coordinate Systems”
- “Single Camera Calibration App”
- “Stereo Calibration App”

## See Also

`extrinsics` | `showExtrinsics`

**Introduced in R2015a**

# reconstructScene

Reconstruct 3-D scene from disparity map

## Syntax

```
pointCloud = reconstructScene(disparityMap, stereoParams)
```

## Description

`pointCloud = reconstructScene(disparityMap, stereoParams)` returns an array of 3-D world point coordinates that reconstruct a scene from a disparity map. The `stereoParams` input must be the same input that you use to rectify the stereo images corresponding to the disparity map.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Reconstruct 3-D Scene from Disparity Map

**Load the stereo parameters.**

```
load('webcamsSceneReconstruction.mat');
```

**Read in the stereo pair of images.**

```
I1 = imread('sceneReconstructionLeft.jpg');  
I2 = imread('sceneReconstructionRight.jpg');
```

**Rectify the images.**

```
[J1, J2] = rectifyStereoImages(I1, I2, stereoParams);
```

**Display the images after rectification.**

```
figure; imshow(cat(3, J1(:,:,1), J2(:,:,2:3)), 'InitialMagnification', 50);
```



**Compute the disparity.**

```
disparityMap = disparity(rgb2gray(J1), rgb2gray(J2));  
figure; imshow(disparityMap, [0, 64], 'InitialMagnification', 50);
```



**Reconstruct the 3-D world coordinates of points corresponding to each pixel from the disparity map.**

```
pointCloud = reconstructScene(disparityMap, stereoParams);
```

**Segment out a person located between 3.2 and 3.7 meters away from the camera.**

```
Z = pointCloud(:, :, 3);  
mask = repmat(Z > 3200 & Z < 3700, [1, 1, 3]);  
J1(~mask) = 0;  
imshow(J1, 'InitialMagnification', 50);
```



- “Structure From Motion From Two Views”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **disparityMap** — Disparity image

*2-D array*

Disparity image, specified as a 2-D array of disparity values for pixels in image 1 of a stereo pair. The `disparity` function provides the input disparity map.

The map can contain invalid values marked by `-realmax('single')`. These values correspond to pixels in image 1, which the `disparity` function did not match in image 2. The function sets the world coordinates corresponding to invalid disparity to `NaN`.

Pixels with zero disparity correspond to world points that are too far away to measure, given the resolution of the camera. The function sets the world coordinates corresponding to zero disparity to `Inf`.

When you specify the `disparityMap` input as a `double`, the function returns the `pointCloud` as a `double`. Otherwise, the function returns the coordinates as `single`.

Data Types: `single` | `double`

### **stereoParams** — Stereo camera system parameters

`stereoParameters` object

Stereo camera system parameters, specified as a `stereoParameters` object.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

## Output Arguments

### **pointCloud** — Coordinates of world points

$M$ -by- $N$ -by-3 array

Coordinates of world points, returned as an  $M$ -by- $N$ -by-3 array. The 3-D world coordinates are relative to the optical center of camera 1 in the stereo system represented by `stereoParams`.

The output array contains the  $[x, y, z]$  coordinates of world points that correspond to the pixels in the `disparityMap` input. `pointCloud(:, :, 1)` contains the  $x$  world coordinates of points corresponding to the pixels in the disparity map. `pointCloud(:, :, 2)` contains the  $y$  world coordinates, and `pointCloud(:, :, 3)` contains the  $z$  world coordinates. The 3-D world coordinates are relative to the optical center of camera 1 in the stereo system.

When you specify the `disparityMap` input as `double`, the function returns the `pointCloud` output as `double`. Otherwise, the function returns it as `single`.

Data Types: `single` | `double`

## More About

- “Coordinate Systems”

## References

- [1] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, Sebastopol, CA: O'Reilly, 2008.

**See Also**

`cameraParameters` | `stereoParameters` | `disparity` | `estimateCameraParameters` | `lineToBorderPoints` | `rectifyStereoImages` | `size`

**Introduced in R2014a**



# rectifyStereoImages

Rectify a pair of stereo images

## Syntax

```
[J1,J2] = rectifyStereoImages(I1,I2,stereoParams)
[J1,J2] = rectifyStereoImages(I1,I2,tform1,tform2)

[J1,J2] = rectifyStereoImages( ____,interp)
[J1,J2] = rectifyStereoImages( ____,Name,Value)
```

## Description

`[J1,J2] = rectifyStereoImages(I1,I2,stereoParams)` returns undistorted and rectified versions of `I1` and `I2` input images using the stereo parameters stored in the `stereoParams` object.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates. This image projection makes the image appear as though the two cameras are parallel. Use the `disparity` function to compute a disparity map from the rectified images for 3-D scene reconstruction.

`[J1,J2] = rectifyStereoImages(I1,I2,tform1,tform2)` returns rectified versions of `I1` and `I2` input images by applying projective transformations `tform1` and `tform2`. The projective transformations are returned by the `estimateUncalibratedRectification` function.

`[J1,J2] = rectifyStereoImages( ____,interp)` additionally specifies the interpolation method to use for rectified images. You can specify the method as `'nearest'`, `'linear'`, or `'cubic'`.

`[J1,J2] = rectifyStereoImages( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Supports Code Generation: Yes  
Supports MATLAB Function block: No  
“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Rectify Stereo Images

Specify calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');  
leftImages = imageSet(fullfile(imageDir,'left'));  
rightImages = imageSet(fullfile(imageDir,'right'));
```

*%Detect the checkerboards.*

```
[imagePoints,boardSize] = detectCheckerboardPoints(leftImages.ImageLocation,...  
    rightImages.ImageLocation);
```

Specify the world coordinates of checkerboard keypoints.

```
worldPoints = generateCheckerboardPoints(boardSize,108);
```

Calibrate the stereo camera system.

```
stereoParams = estimateCameraParameters(imagePoints,worldPoints);
```

Read in the images.

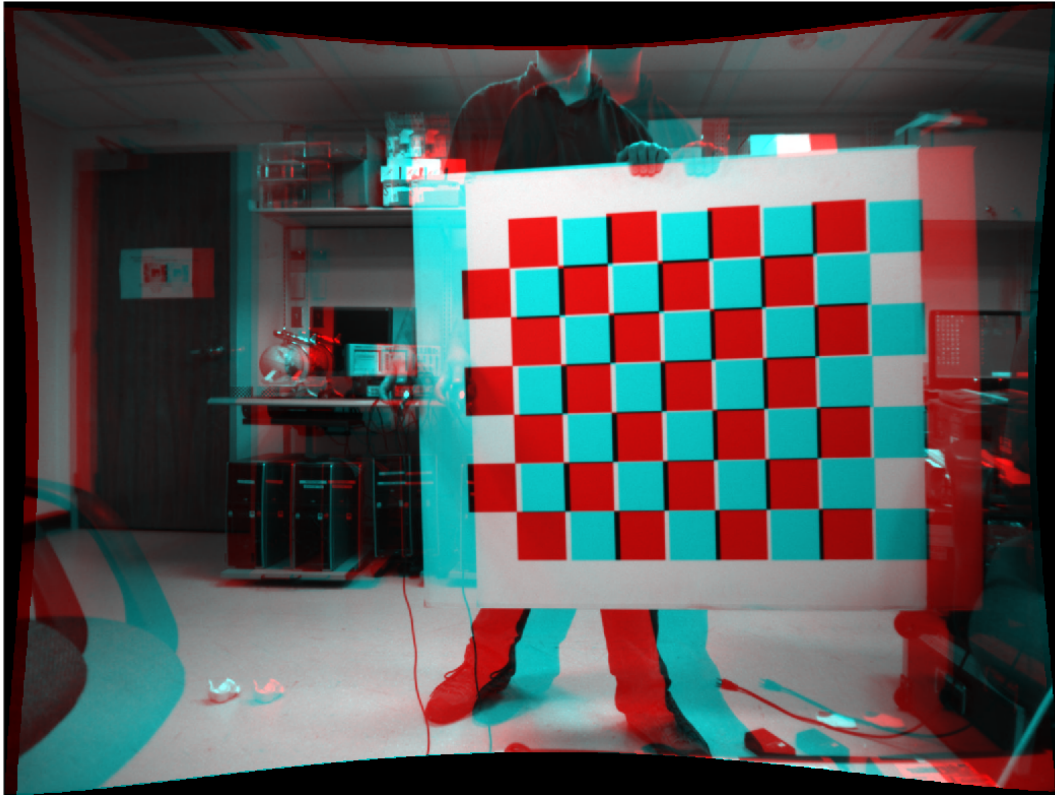
```
I1 = read(leftImages, 1);  
I2 = read(rightImages, 1);
```

Rectify the images using the 'full' output view.

```
[J1_full,J2_full] = rectifyStereoImages(I1,I2, stereoParams,'OutputView','full');
```

Display the result.

```
figure;  
imshowpair(J1_full,J2_full,'falsecolor','ColorChannels','red-cyan');
```

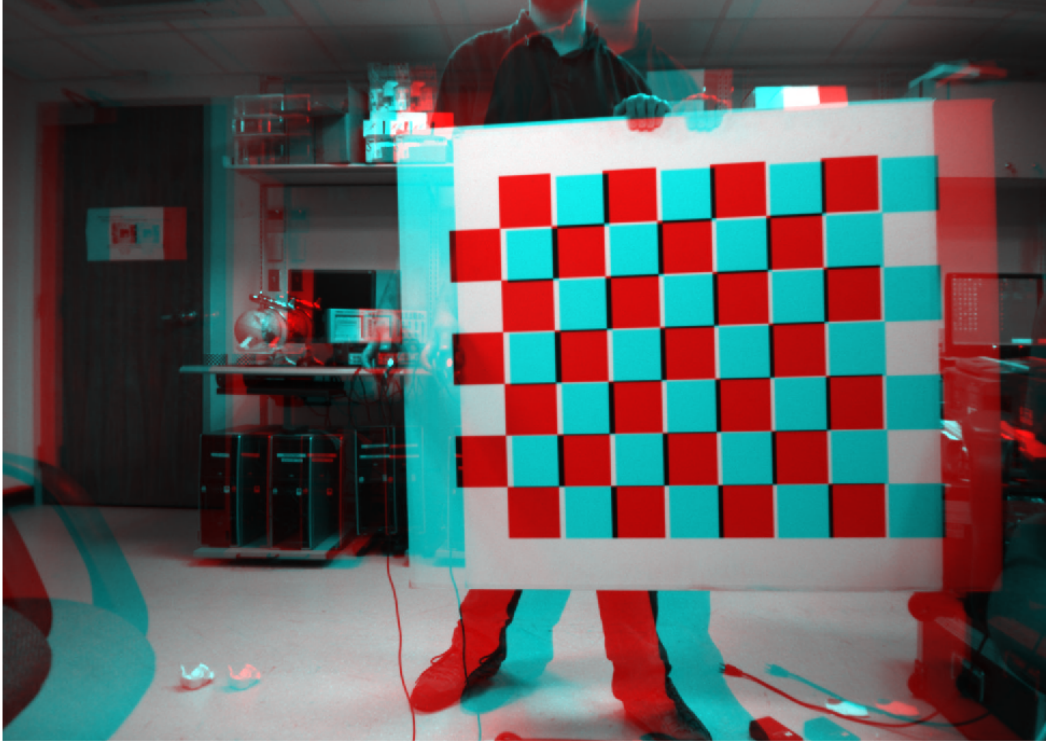


Rectify the images using the 'valid' output view

```
[J1_valid,J2_valid] = rectifyStereoImages(I1,I2,stereoParams,'OutputView','valid');
```

Display the result.

```
figure;  
imshowpair(J1_valid,J2_valid,'falsecolor','ColorChannels','red-cyan');
```



- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”
- “Uncalibrated Stereo Image Rectification”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **I1** — Input image 1

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input image corresponding to camera 1, specified as an  $M$ -by- $N$ -by-3 truecolor image or an  $M$ -by- $N$  2-D grayscale array. Input images **I1** and **I2** must also be real, finite, and nonsparse. The input images must be the same class.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

### **I2** — Input image 2

$M$ -by- $N$ -by-3 truecolor image |  $M$ -by- $N$  2-D truecolor image

Input image corresponding to camera 2, specified as an  $M$ -by- $N$ -by-3 truecolor image or an  $M$ -by- $N$  2-D grayscale array. Input images **I1** and **I2** must be real, finite, and nonsparse. The input images must also be the same class.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

### **stereoParams** — Stereo camera system parameters

`stereoParameters` object

Stereo camera system parameters, specified as a `stereoParameters` object.

Data Types: `uint8` | `uint16` | `int16` | `single` | `double`

### **tform1** — Projective transformation

3-by-3 matrix | `projective2d` object

Projective transformations for image 1, specified as a 3-by-3 matrix returned by the `estimateUncalibratedRectification` function or a `projective2d` object.

### **tform2** — Projective transformation

3-by-3 matrix | `projective2d` object

Projective transformations for image 2, specified as a 3-by-3 matrix returned by the `estimateUncalibratedRectification` function or a `projective2d` object.

### **interp** — Interpolation method

'linear' (default) | 'nearest' | 'cubic'

Interpolation method, specified as the string 'linear', 'nearest', or 'cubic' string.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single

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

Example: `'OutputView', 'valid'` sets the `'OutputView'` property to the string `'valid'`.

#### **'OutputView' — Size of rectified images**

`'valid'` (default) | string

Size of rectified images, specified as the comma-separated pair consisting of `'OutputView'` and the string `'full'` or `'valid'`. When you set this parameter to `'full'`, the rectified images include all pixels from the original images. When you set this value to `'valid'`, the output images are cropped to the size of the largest common rectangle containing valid pixels.

#### **'FillValues' — Output pixel fill values**

array of scalar values

Output pixel fill values, specified as the comma-separated pair consisting of `'FillValues'` and an array of one or more scalar values. When the corresponding inverse-transformed location in the input image is completely outside the input image boundaries, use the fill values for output pixels. If `I1` and `I2` are 2-D grayscale images, then you must set `'FillValues'` to a scalar. If `I1` and `I2` are truecolor images, then you can set `'FillValues'` to a scalar or a 3-element vector of RGB values.

## Output Arguments

### **J1 — Undistorted and rectified image 1**

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Undistorted and rectified version of `I1`, returned as an *M*-by-*N*-by-3 truecolor image or as an *M*-by-*N* 2-D grayscale image.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates. This image projection makes the image appear as though the two cameras are parallel. Use the `disparity` function to compute a disparity map from the rectified images for 3-D scene reconstruction.

### **J2 — Undistorted and rectified image 2**

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Undistorted and rectified version of `I2`, returned as an  $M$ -by- $N$ -by-3 truecolor image or as an  $M$ -by- $N$  2-D grayscale image.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates. This image projection makes the image appear as though the two cameras are parallel. Use the `disparity` function to compute a disparity map from the rectified images for 3-D scene reconstruction.

## More About

- “Coordinate Systems”

## References

[1] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA: O'Reilly, 2008.

## See Also

`stereoParameters` | `Camera Calibrator` | `disparity` | `estimateCameraParameters`  
| `estimateUncalibratedRectification` | `reconstructScene` | `Stereo Camera Calibrator`

**Introduced in R2014a**

## retrieveImages

Search image set for similar image

### Syntax

```
imageIDs = retrieveImages(queryImage,imageIndex)
[imageIDs,scores] = retrieveImages(queryImage,imageIndex)
[imageIDs,scores,imageWords] = retrieveImages(queryImage,imageIndex)
[imageIDs, ___ ] = retrieveImages(queryImage,imageIndex,Name,Value)
```

### Description

`imageIDs = retrieveImages(queryImage,imageIndex)` returns the indices corresponding to images within `imageIndex` that are visually similar to the query image. The `imageIDs` output contains the indices in ranked order, from the most to least similar match.

`[imageIDs,scores] = retrieveImages(queryImage,imageIndex)` optionally returns the similarity scores used to rank the image retrieval results. The `scores` output contains the corresponding scores from 0 to 1.

`[imageIDs,scores,imageWords] = retrieveImages(queryImage,imageIndex)` optionally returns the visual words in `queryImage` that are used to search for similar images.

`[imageIDs, ___ ] = retrieveImages(queryImage,imageIndex,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

### Examples

#### Search Image Set Using Query Image

Create an image set of book covers.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata','bookCovers');
```



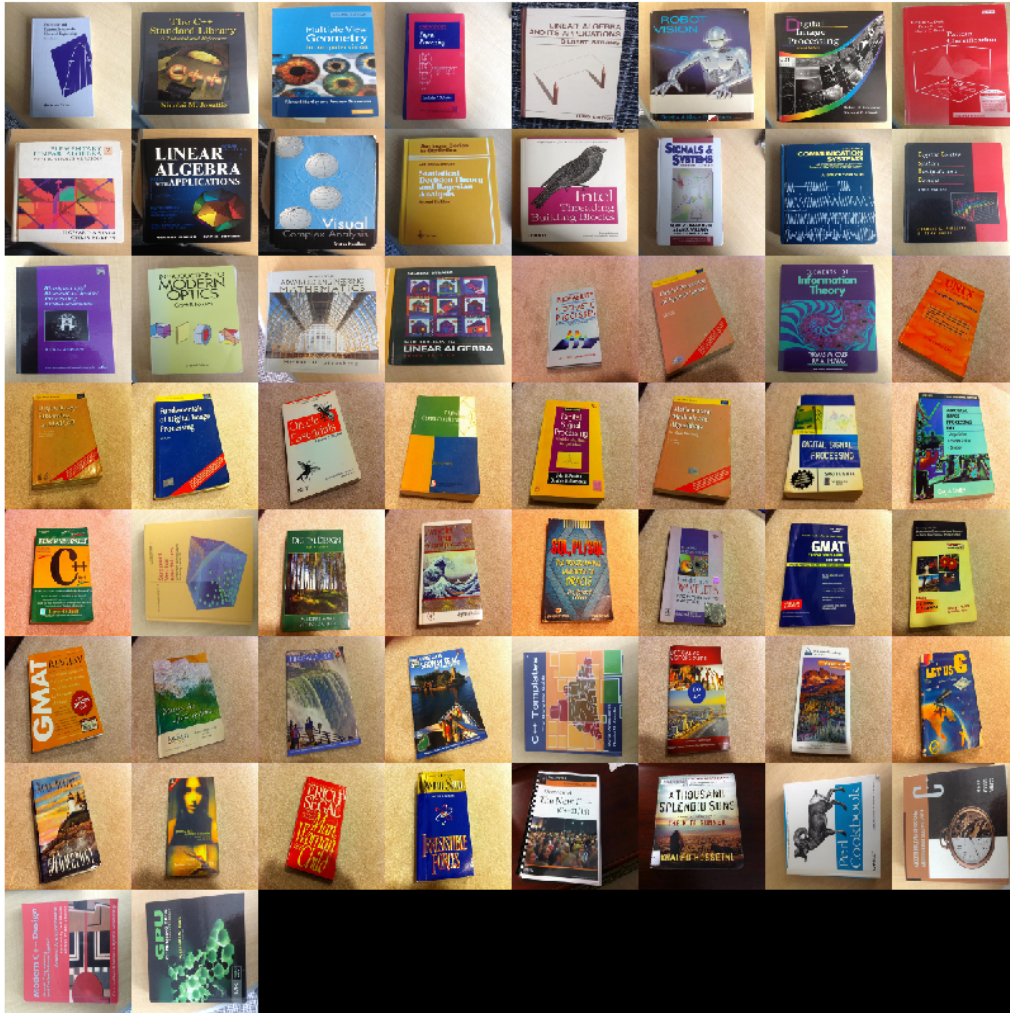
```
bookCovers = imageSet(dataDir);
```

**Display image set using the montage functions.**

```
thumbnailGallery = [];  
for i = 1:bookCovers.Count  
    I = read(bookCovers, i);  
    thumbnail = imresize(I, [300 300]);  
    thumbnailGallery = cat(4, thumbnailGallery, thumbnail);  
end  
  
figure  
montage(thumbnailGallery);
```

### 3 Functions Alphabetical

---



Index the image set. This step may take a few minutes.

```
imageIndex = indexImages(bookCovers, 'Verbose', false);
```

**Select the query image.**

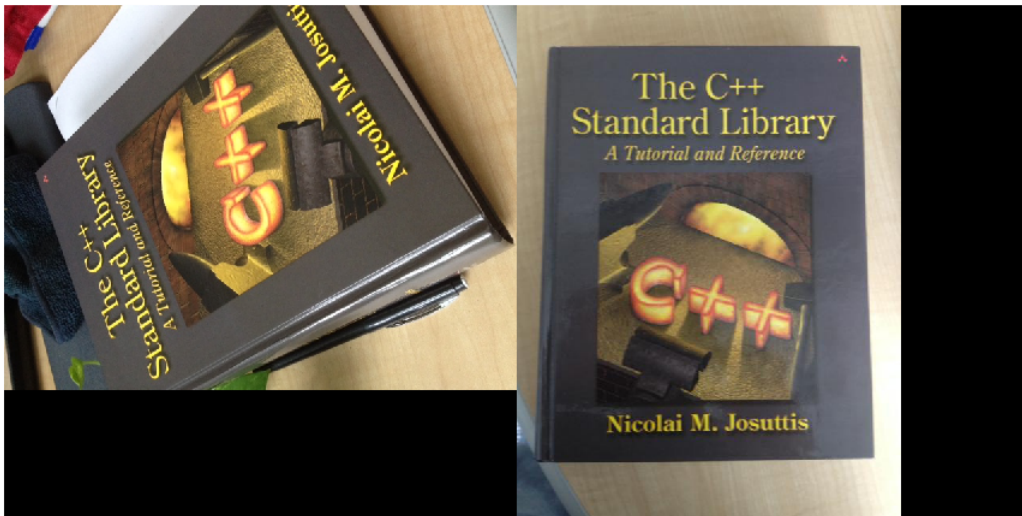
```
queryDir = fullfile(dataDir, 'queries', filesep);
queryImage = imread([queryDir 'query2.jpg']);

imageIDs = retrieveImages(queryImage, imageIndex);
```

**Show the query image and its best match, side-by-side.**

```
bestMatch = imageIDs(1);
bestImage = imread(imageIndex.ImageLocation{bestMatch});

figure
imshowpair(queryImage, bestImage, 'montage')
```

**Search Image Set for Specific Object Using ROIs**

Search an image set for an object using a region of interest (ROI) for the query image.

Define a set of images to search.

```
imageFiles = ...
    {'elephant.jpg', 'cameraman.tif', ...
     'peppers.png', 'saturn.png',...
     'pears.png', 'stapleRemover.jpg', ...
     'football.jpg', 'mandi.tif',...
     'kids.tif', 'liftingbody.png', ...
     'office_5.jpg', 'gantrycrane.png',...
     'moon.tif', 'circuit.tif', ...
     'tape.png', 'coins.png'};
```

```
imgSet = imageSet(imageFiles);
```

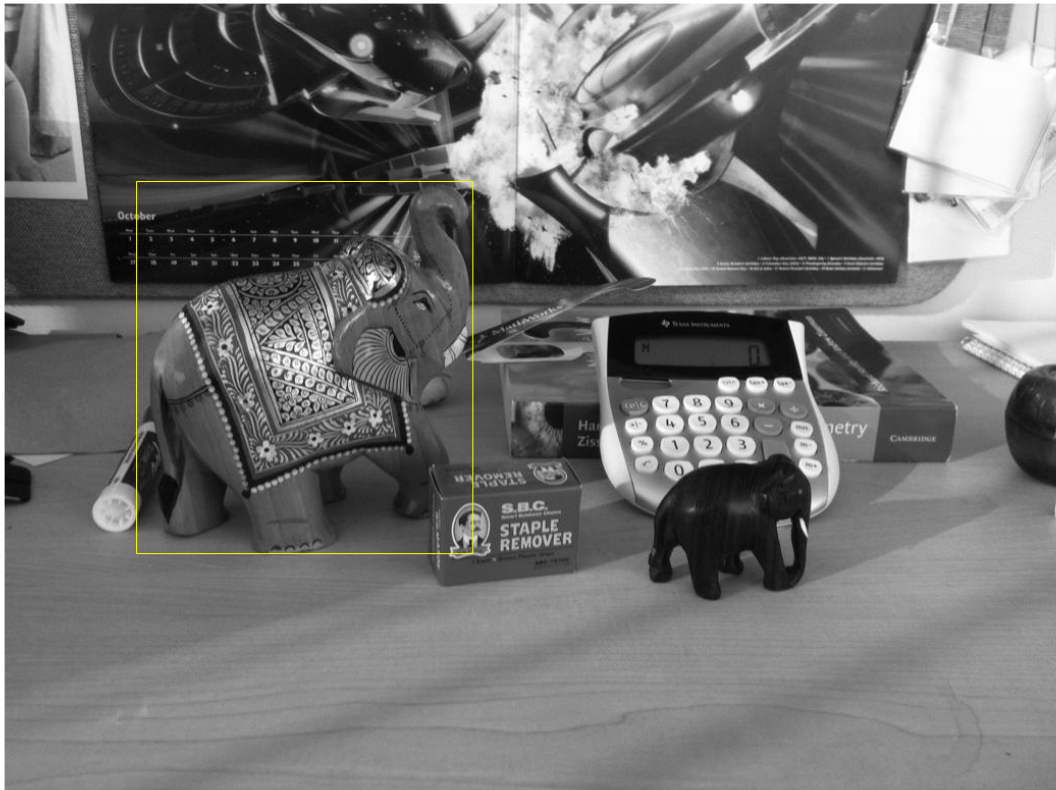
Create a search index.

```
imageIndex = indexImages(imgSet, 'Verbose', false);
```

Specify a query image and an ROI. The ROI outlines the object, an elephant, for the search.

```
queryImage = imread('clutteredDesk.jpg');
queryROI = [130 175 330 365];

figure
imshow(queryImage)
rectangle('Position', queryROI, 'EdgeColor', 'yellow')
```



You can also use the `imrect` function to select an ROI interactively. For example,  
`queryROI = getPosition(imrect)`

Find images that contain the object.

```
imageIDs = retrieveImages(queryImage, imageIndex, 'ROI', queryROI)
```

```
imageIDs =
```

```
1  
11
```

```
6  
12  
3  
13  
2  
14  
8  
16  
5  
10
```

Display the best match.

```
bestMatch = imageIDs(1);
```

```
figure  
imshow(imageIndex.ImageLocation{bestMatch})
```



### Geometric Verification Using `estimateGeometricTransform` Function

Use the locations of visual words to verify the best search result. To rerank the search results based on geometric information, repeat this procedure for the top  $N$  search results.

Specify the location of the images.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata','bookCovers');  
bookCovers = imageSet(dataDir);
```

Index the image set. This process can take a few minutes.

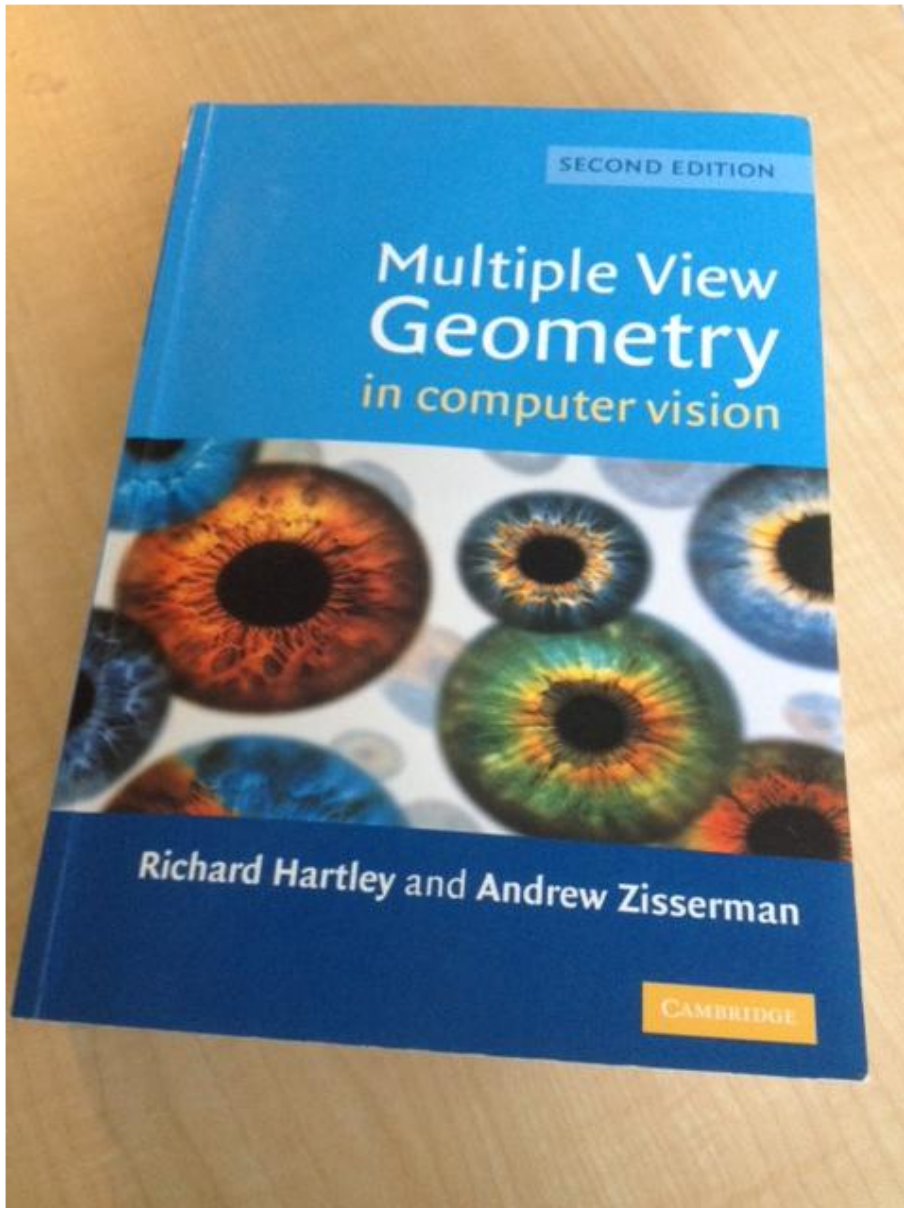
```
imageIndex = indexImages(bookCovers, 'Verbose', false);
```

Select and display the query image.

```
queryDir = fullfile(dataDir, 'queries', filesep);  
queryImage = imread([queryDir 'query3.jpg']);
```

```
figure  
imshow(queryImage)
```





Retrieve the best matches. The `queryWords` output contains visual word locations information for the query image. Use this information to verify the search results.

```
[imageIDs, ~, queryWords] = retrieveImages(queryImage, imageIndex);
```

Find the best match for the query image by extracting the visual words from the image index. The image index contains the visual word information for all images in the index.

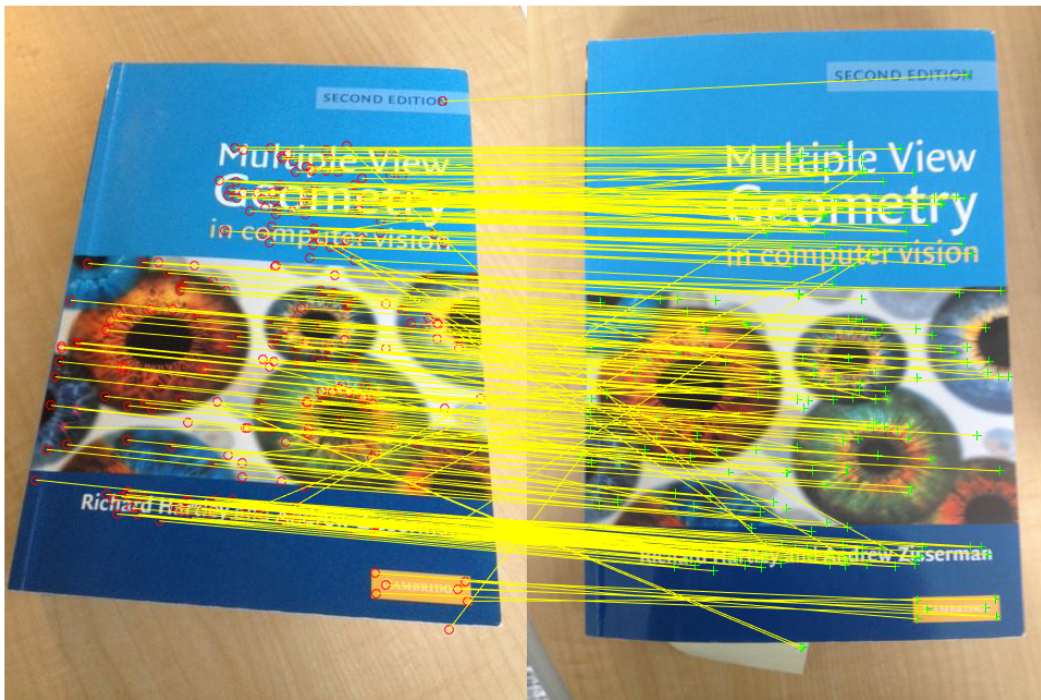
```
bestMatch = imageIDs(1);  
bestImage = imread(imageIndex.ImageLocation{bestMatch});  
bestMatchWords = imageIndex.ImageWords(bestMatch);
```

Generate a set of tentative matches based on visual word assignments. Each visual word in the query can have multiple matches due to the hard quantization used to assign visual words.

```
queryWordsIndex      = queryWords.WordIndex;  
bestMatchWordIndex  = bestMatchWords.WordIndex;  
  
tentativeMatches = [];  
for i = 1:numel(queryWords.WordIndex)  
    idx = find(queryWordsIndex(i) == bestMatchWordIndex);  
    matches = [repmat(i, numel(idx), 1) idx];  
    tentativeMatches = [tentativeMatches; matches];  
end
```

Show the point locations for the tentative matches. There are many poor matches.

```
points1 = queryWords.Location(tentativeMatches(:,1),:);  
points2 = bestMatchWords.Location(tentativeMatches(:,2),:);  
  
figure  
showMatchedFeatures(queryImage,bestImage,points1,points2,'montage')
```



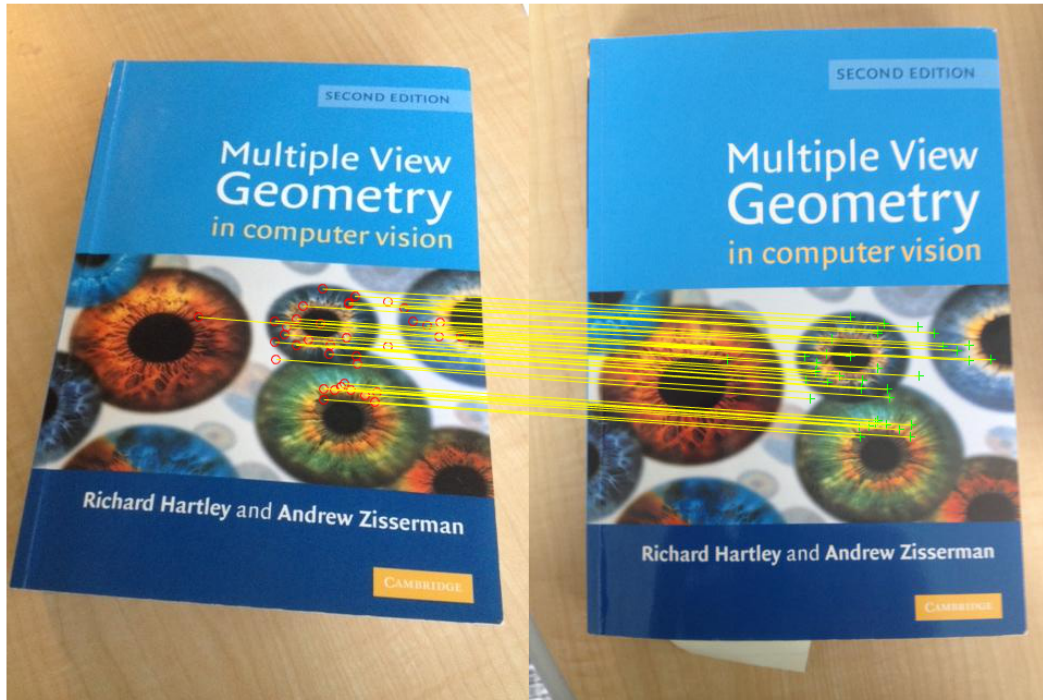
Remove poor visual word assignments using `estimateGeometricTransform` function. Keep the assignments that fit a valid geometric transform.

```
[tform,inlierPoints1,inlierPoints2] = estimateGeometricTransform(points1,points2,'affine');
```

Rerank the search results by the percentage of inliers. Do this when the geometric verification procedure is applied to the top  $N$  search results. Those images with a higher percentage of inliers are more likely to be relevant.

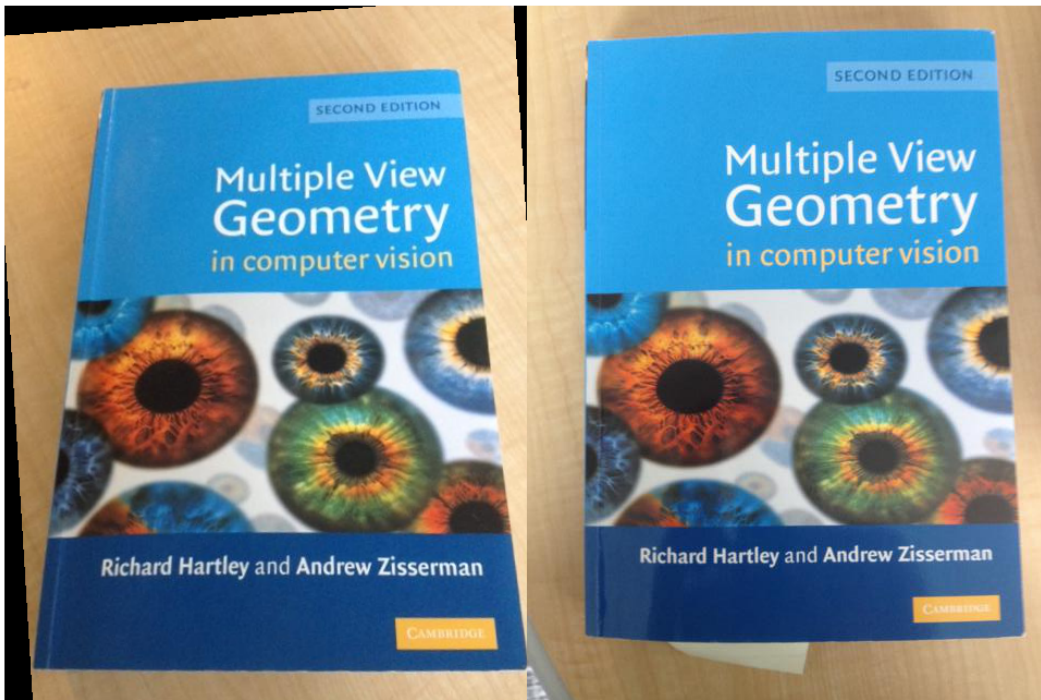
```
percentageOfInliers = size(inlierPoints1,1)./size(points1,1);
```

```
figure
showMatchedFeatures(queryImage,bestImage,inlierPoints1,inlierPoints2,'montage')
```



Apply the estimated transform.

```
outputView = imref2d(size(bestImage));  
Ir = imwarp(queryImage, tform, 'OutputView', outputView);  
  
figure  
imshowpair(Ir, bestImage, 'montage')
```



### Modify Search Parameters For Image Search

Use the `evaluateImageRetrieval` function to help select proper search parameters.

Create an image set.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets','cups');  
imgSets = imageSet(setDir, 'recursive');
```

Index the image set.

```
imageIndex = indexImages(imgSets, 'Verbose', false);
```

Tune image search parameters.

```
imageIndex.MatchThreshold = 0.2;
```



```
imageIndex.WordFrequencyRange = [0 1]

queryImage = read(imgSets, 1);
indices = retrieveImages(queryImage, imageIndex);

imageIndex =
    invertedImageIndex with properties:
        ImageLocation: {6x1 cell}
        ImageWords: [6x1 vision.internal.visualWords]
        WordFrequency: [1x1366 double]
        BagOfFeatures: [1x1 bagOfFeatures]
        MatchThreshold: 0.2000
        WordFrequencyRange: [0 1]
```

- “Image Retrieval Using Customized Bag of Features”

## Input Arguments

### **queryImage** — Input query image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input query image, specified as either an *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-D grayscale image.

Data Types: single | double | int16 | uint8 | uint16 | logical

### **imageIndex** — Image search index

*invertedImageIndex* object

Image search index, specified as an *invertedImageIndex* object. The *indexImages* function creates the *invertedImageIndex* object, which stores the data used for the image search.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single

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

Example: `'NumResults',25` sets the `'NumResults'` property to 25

### 'NumResults' — Maximum number of results

20 (default) | numeric value

Maximum number of results to return, specified as the comma-separated pair consisting of `'NumResults'` and a numeric value. Set this value to `Inf` to return as many matching images as possible.

### 'ROI' — Query image search region

`[1 1 size(queryImage,2) size(queryImage,1)]` (default) | `[x y width height]` vector

Query image search region, specified as the comma-separated pair consisting of `'ROI'` and an `[x y width height]` vector.

## Output Arguments

### **imageIDs** — Ranked index of retrieved images

*M*-by-1 vector

Ranked index of retrieved images, returned as an *M*-by-1 vector. The image IDs are returned in ranked order, from the most to least similar matched image.

### **scores** — Similarity metric

*N*-by-1 vector

Similarity metric, returned as an *N*-by-1 vector. This output contains the scores that correspond to the retrieved images in the `imageIDs` output. The scores are computed using the cosine similarity and range from 0 to 1.

### **imageWords** — Object for storing visual word assignments

`visualWords` object

Object for storing visual word assignments, returned as a `visualWords` object. The object stores the visual word assignments of `queryImage` and their locations within that image.

### More About

- “Image Retrieval with Bag of Visual Words”

### References

- [1] Sivic, J. and A. Zisserman. *Video Google: A text retrieval approach to object matching in videos*. ICCV (2003) pg 1470-1477.
- [2] Philbin, J., O. Chum, M. Isard, J. Sivic, and A. Zisserman. *Object retrieval with large vocabularies and fast spatial matching*. CVPR (2007).

### See Also

`imageSet` | `bagOfFeatures` | `invertedImageIndex` | `evaluateImageRetrieval` | `indexImages`

**Introduced in R2015a**



# rotationMatrixToVector

Convert 3-D rotation matrix to rotation vector

## Syntax

```
rotationVector = rotationMatrixToVector(rotationMatrix)
```

## Description

`rotationVector = rotationMatrixToVector(rotationMatrix)` returns an axis-angle rotation vector that corresponds to the input 3-D rotation matrix. The function uses the Rodrigues formula for the conversion.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Convert Rotation Matrix to Rotation Vector

Create a matrix representing a 90-degree rotation about the  $Z$ -axis.

```
rotationMatrix = [0, -1, 0; 1, 0, 0; 0, 0, 1];
```

Find the equivalent rotation vector.

```
rotationVector = rotationMatrixToVector(rotationMatrix)
```

```
rotationVector =
```

```
0 0 -1.5708
```

- “Evaluating the Accuracy of Single Camera Calibration”

- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”

## Input Arguments

### **rotationMatrix** — Rotation of camera

3-by-3 matrix

Rotation of camera, specified as a 3-by-3 matrix. You can obtain this matrix by using the `extrinsics` function.

## Output Arguments

### **rotationVector** — Rotation vector

three-element vector

Rotation vector, returned as a three-element vector. The vector represents the axis of rotation in 3-D, where the magnitude corresponds to the rotation angle in radians.

Data Types: `single` | `double`

## References

[1] Trucco, E., and A. Verri. *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, 1998.

## See Also

`cameraPose` | `extrinsics` | `rotationVectorToMatrix` | `triangulate`

Introduced in R2016a

# rotationVectorToMatrix

Convert 3-D rotation vector to rotation matrix

## Syntax

```
rotationMatrix = rotationVectorToMatrix(rotationVector)
```

## Description

`rotationMatrix = rotationVectorToMatrix(rotationVector)` returns a 3-D rotation matrix that corresponds to the input axis-angle rotation vector. The function uses the Rodrigues formula for the computation.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: Yes

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Convert Rotation Vector to Rotation Matrix

Create a vector representing a 90-degree rotation about the  $Z$ -axis.

```
rotationVector = pi/2 * [0, 0, 1];
```

Find the equivalent rotation matrix.

```
rotationMatrix = rotationVectorToMatrix(rotationVector)
```

```
rotationMatrix =
```

```
    0.0000    1.0000         0
   -1.0000    0.0000         0
         0         0    1.0000
```

- “Evaluating the Accuracy of Single Camera Calibration”
- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”

## Input Arguments

**rotationVector** — **Rotation vector**  
three-element vector

Rotation vector, specified as a three-element vector. The vector represents the axis of rotation in 3-D, where the magnitude corresponds to the rotation angle in radians.

Data Types: `single` | `double`

## Output Arguments

**rotationMatrix** — **Rotation of camera**  
3-by-3 matrix

Rotation of camera, returned as a 3-by-3 matrix that corresponds to the input axis-angle rotation vector.

## References

- [1] Trucco, E., and A. Verri. *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, 1998.

## See Also

`cameraPose` | `extrinsics` | `rotationMatrixToVector` | `triangulate`

Introduced in R2016a

# selectStrongestBbox

Select strongest bounding boxes from overlapping clusters

## Syntax

```
[selectedBbox,selectedScore] = selectStrongestBbox(bbox,score)
[selectedBbox,selectedScore,index] = selectStrongestBbox(bbox,score)
[selectedBbox,selectedScore,index] = selectStrongestBbox( ____,
Name,Value)
```

## Description

`[selectedBbox,selectedScore] = selectStrongestBbox(bbox,score)` returns selected bounding boxes that have a high confidence score. The function uses nonmaximal suppression to eliminate overlapping bounding boxes from the `bbox` input.

`[selectedBbox,selectedScore,index] = selectStrongestBbox(bbox,score)` additionally returns the `index` vector associated with `selectedBbox`. This vector contains the indices of the selected boxes in the `bbox` input.

`[selectedBbox,selectedScore,index] = selectStrongestBbox( ____, Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Code Generation Support:

Compile-time constant input: No restriction

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Run Nonmaximal Suppression on Bounding Boxes Using People Detector

Load the pretrained people detector and disable bounding box merging.

```
peopleDetector = vision.PeopleDetector('ClassificationThreshold',0,'MergeDetections
```

**Read an image, run the people detector, and then insert bounding boxes with confidence scores.**

```
I = imread('visionteam1.jpg');  
[bbox, score] = step(peopleDetector, I);  
I1 = insertObjectAnnotation(I, 'rectangle', bbox, cellstr(num2str(score)), 'Color',
```

**Run nonmaximal suppression on the bounding boxes.**

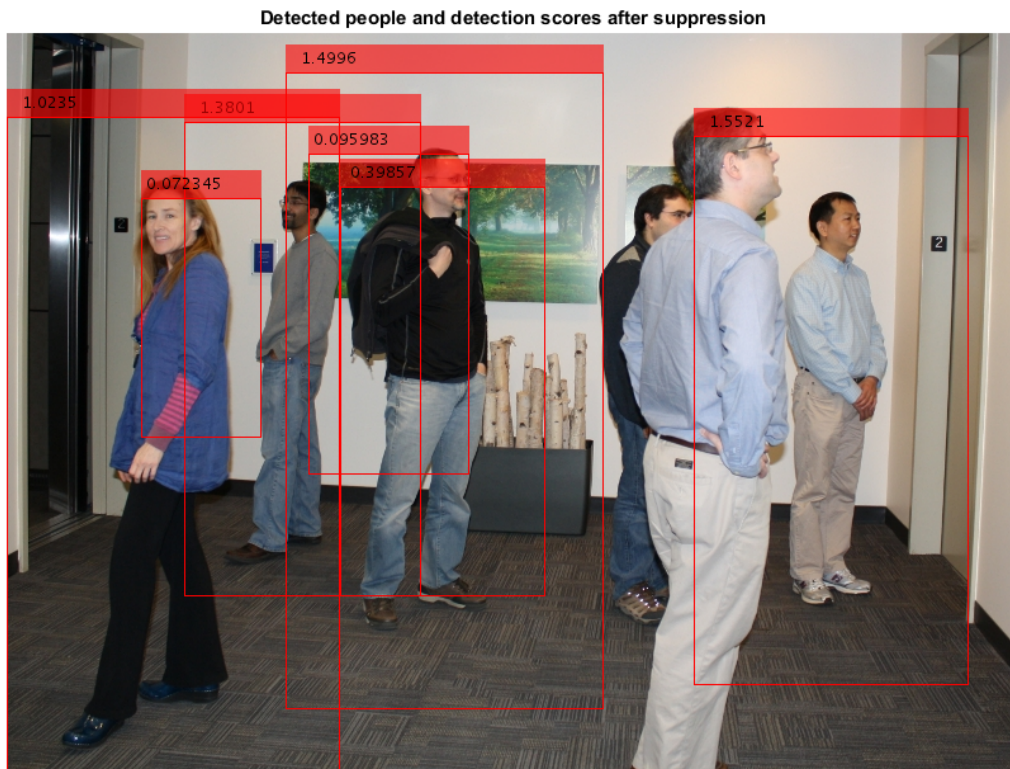
```
[selectedBbox, selectedScore] = selectStrongestBbox(bbox, score);  
I2 = insertObjectAnnotation(I, 'rectangle', selectedBbox, cellstr(num2str(selectedScore)), 'Color',
```

**Display detection before and after suppression.**

```
figure, imshow(I1); title('Detected people and detection scores before suppression');  
figure, imshow(I2); title('Detected people and detection scores after suppression');
```

Detected people and detection scores before suppression





## Input Arguments

### **bbox** — Bounding boxes

*M*-by-4 matrix

Bounding boxes, specified as an *M*-by-4 matrix with *M* bounding boxes. Each row of the **bbox** input represents a bounding box, specified in the format  $[x \ y \ width \ height]$ , where *x* and *y* correspond to the upper left corner of the bounding box. The **bbox** input must be real, finite, and nonspase.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32



**score — Confidence score***M*-by-1 vector

Confidence score, specified as an *M*-by-1 vector. The *M*th score in the `score` input corresponds to the *M*th bounding box in the `bbox` input. The `score` input must be real, finite, and nonsparse. The function uses nonmaximal suppression to eliminate overlapping bounding boxes and associate the confidence score with the boxes. A higher score represents a higher confidence in keeping the bounding box.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Name-Value Pair Arguments**

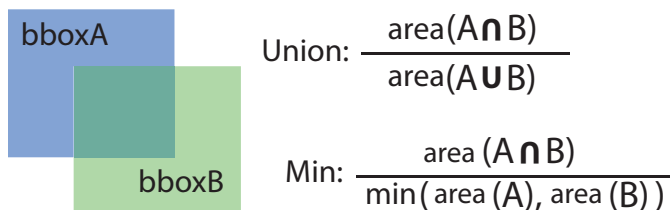
Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `'RatioType','Union'` sets the `'RatioType'` property to `'Union'`.

**'RatioType' — Bounding box overlap ratio denominator**`'Union'` (default) | `'Min'`

Ratio type, specified as the string `'Union'` or `'Min'`.

- Set the ratio type to `'Union'` to compute the ratio as the area of intersection between `bboxA` and `bboxB`, divided by the area of the union of the two.
- Set the ratio type to `'Min'` to compute the ratio as the area of intersection between `bboxA` and `bboxB`, divided by the minimum area of the two bounding boxes.



Data Types: char

**'OverlapThreshold' — Overlap ratio threshold**

0.5 (default) | scalar in the range [0 1]

Overlap ratio threshold, specified as the comma-separated pair consisting of 'OverlapThreshold' and a scalar in the range [0 1]. When the overlap ratio is above the threshold you set, the function removes bounding boxes around the reference box. Decrease this value to reduce the number of selected bounding boxes. However, if you decrease the overlap ratio too much, you might eliminate boxes that represent objects close to each other in the image.

Data Types: `single` | `double`

## Output Arguments

### **selectedBbox** — Selected bounding boxes

*M*-by-4 matrix

Selected bounding boxes, returned as an *M*-by-4 matrix. The `selectedBbox` output returns the selected bounding boxes from the `bbox` input that have the highest confidence score. The function uses nonmaximal suppression to eliminate overlapping bounding boxes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### **selectedScore** — Scores of selected bounding boxes

*M*-by-1 vector

Scores of selected bounding boxes, returned as an *M*-by-1 vector. The *M*th score in the `selectedScore` output corresponds to the *M*th bounding box in the `selectedBbox` output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### **index** — Index of selected bounding boxes

*M*-by-1 vector

Index of selected bounding boxes, returned as an *M*-by-1 vector. The `index` vector contains the indices to the selected boxes in the `bbox` input.

Data Types: `double`

## See Also

`bboxOverlapRatio`

**Introduced in R2014b**

## showExtrinsics

Visualize extrinsic camera parameters

### Syntax

```
showExtrinsics(cameraParams)
showExtrinsics(cameraParams,view)
showExtrinsics( ____,Name,Value)

ax = showExtrinsics( ____ )
```

### Description

`showExtrinsics(cameraParams)` renders a 3-D visualization of extrinsic parameters of a single calibrated camera or a calibrated stereo pair. The function plots a 3-D view of the calibration patterns with respect to the camera. The `cameraParams` input contains either a `cameraParameters` or a `stereoParameters` object, which the `estimateCameraParameters` function returns.

`showExtrinsics(cameraParams,view)` displays visualization of the camera extrinsic parameters using the style specified by the `view` input.

`showExtrinsics( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

`ax = showExtrinsics( ____ )` returns the plot axis, using any of the preceding syntaxes.

### Examples

#### Visualize Single Camera Extrinsic Parameters

Create a cell array of file names of calibration images.

```
for i = 1:5
    imageFileName = sprintf('image%d.tif', i);
    imageFileNames{i} = fullfile(matlabroot,'toolbox','vision','visiondata','calibrati
end
```

Detect calibration pattern.

```
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate world coordinates of the corners of the squares,(squares are in mm).

```
squareSide = 25;
```

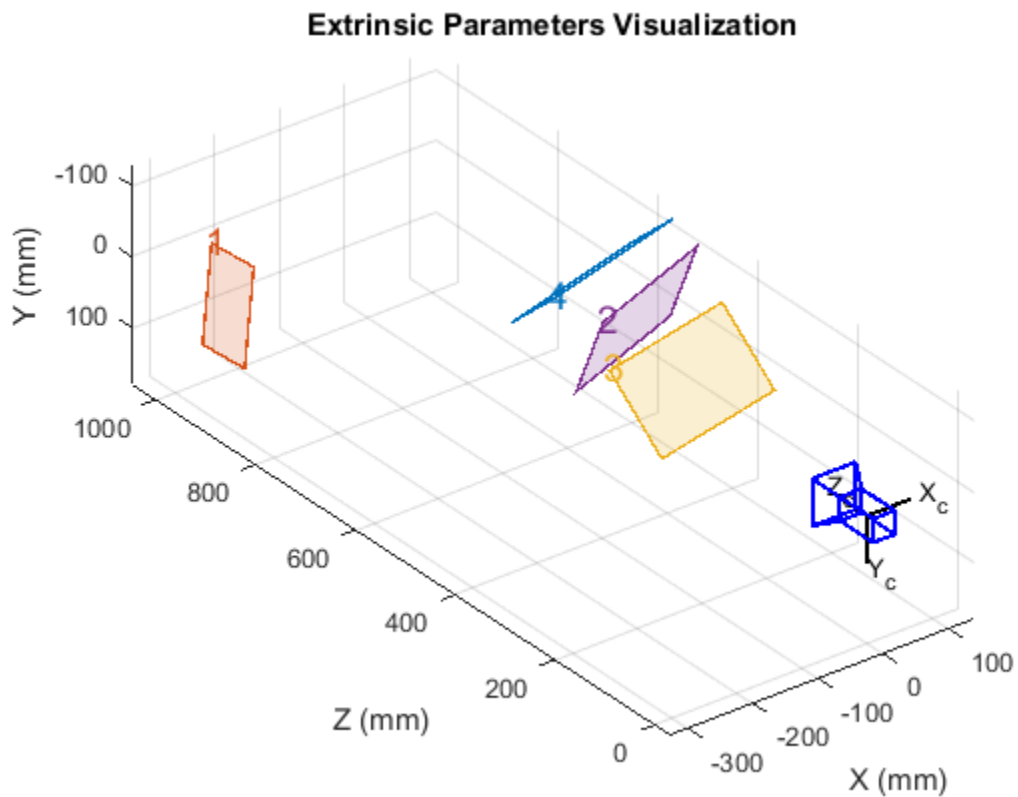
```
worldPoints = generateCheckerboardPoints(boardSize, squareSide);
```

Calibrate the camera.

```
cameraParams = estimateCameraParameters(imagePoints, worldPoints);
```

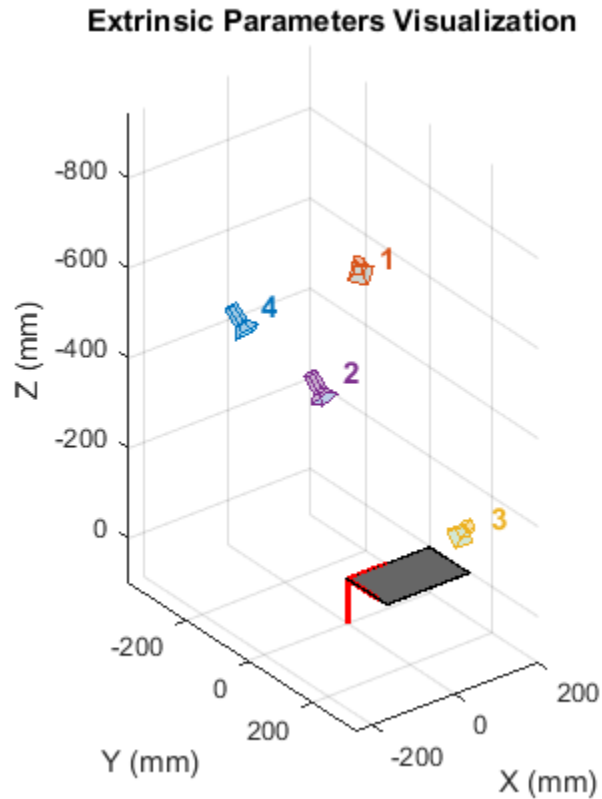
Visualize pattern locations.

```
figure; showExtrinsics(cameraParams);
```



Visualize camera locations.

```
figure; showExtrinsics(cameraParams,'patternCentric');
```



### Visualize Stereo Pair of Camera Extrinsic Parameters

Specify calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');  
leftImages = imageSet(fullfile(imageDir,'left'));  
rightImages = imageSet(fullfile(imageDir,'right'));  
images1 = leftImages.ImageLocation;  
images2 = rightImages.ImageLocation;
```

Detect the checkerboards.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images1, images2);
```

Specify world coordinates of checkerboard keypoints, (squares are in mm).

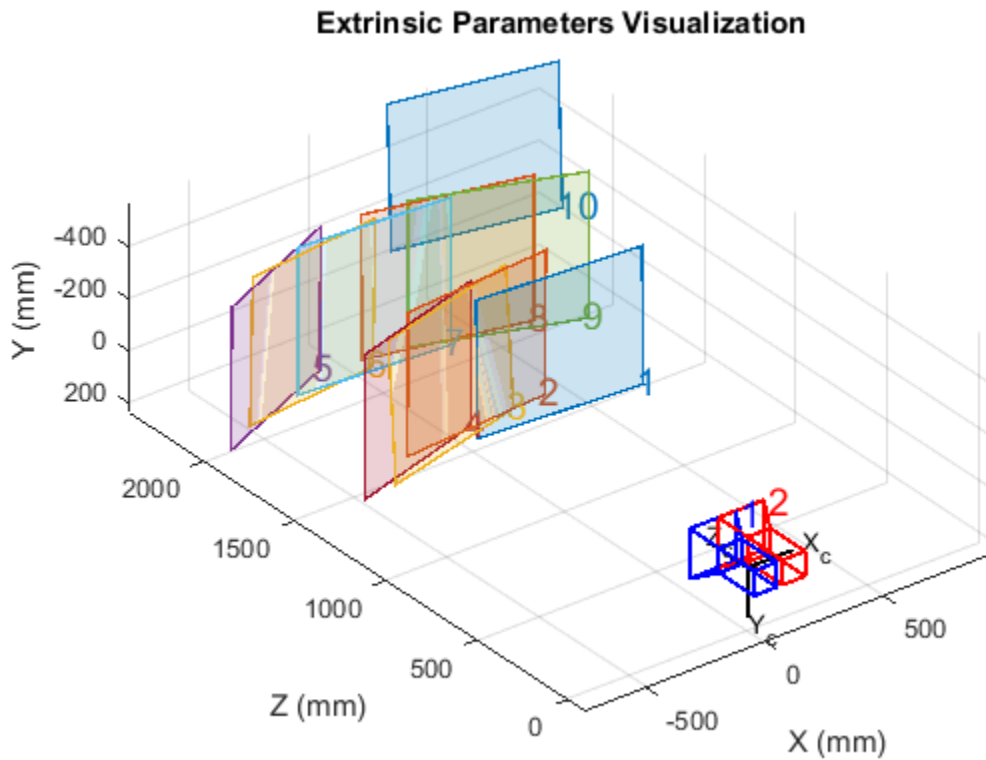
```
squareSizeInMM = 108;  
worldPoints = generateCheckerboardPoints(boardSize, squareSizeInMM);
```

Calibrate the stereo camera system.

```
cameraParams = estimateCameraParameters(imagePoints, worldPoints);
```

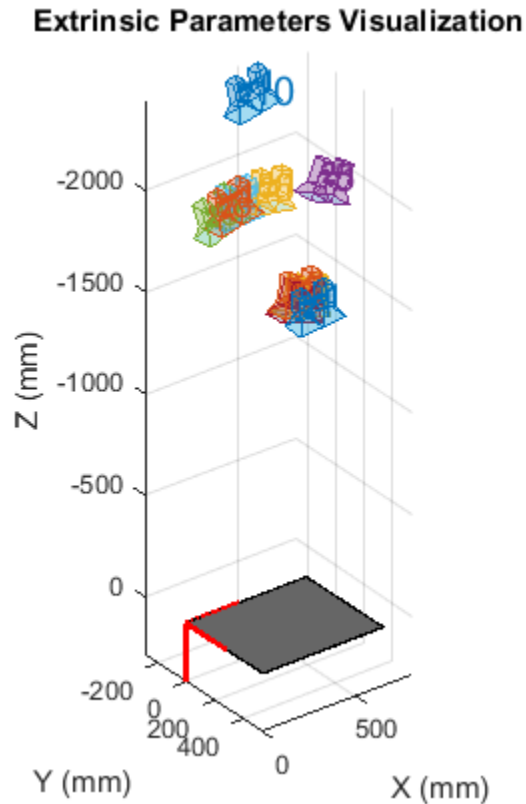
Visualize pattern locations.

```
figure;  
showExtrinsics(cameraParams);
```



Visualize camera locations.

```
figure;  
showExtrinsics(cameraParams, 'patternCentric');
```



## Input Arguments

**cameraParams** — Object containing parameters of single camera or stereo pair  
cameraParameters object | stereoParameters object

Object containing parameters of single camera or stereo pair, specified as either a cameraParameters or stereoParameters object. You can create the single camera or stereo pair input object using the estimateCameraParameters function.



You can also use the Camera Calibrator app to create the cameraParameters input object, or use Stereo Camera Calibrator app to create the stereoParameters input object. See “Single Camera Calibration App” and “Stereo Calibration App”.

### **view — Camera- or pattern-centric view**

'CameraCentric' | 'PatternCentric'

Camera or pattern-centric view, specified as the character string 'CameraCentric' or 'PatternCentric'. The view input sets the visualization for the camera extrinsic parameters. If you keep your camera stationary while moving the calibration pattern, set view to 'CameraCentric'. If the pattern is stationary while you move your camera, set it to 'PatternCentric'.

## **Name-Value Pair Arguments**

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

Example: 'HighlightIndex', [1, 4] sets the 'HighlightIndex' to patterns 1 and 4.

### **'HighlightIndex' — Highlight selection index**

[] (default) | vector | scalar

Highlight selection index, specified as a scalar or a vector of integers. For example, if you want to highlight patterns 1 and 4, use [1, 4]. Doing so increases the opacity of patterns 1 and 4 in contrast to the rest of the patterns.

### **'Parent' — Output axes**

current axes (default) | scalar value

Output axes, specified as the comma-separated pair consisting of 'Parent' and a scalar value. Specify an output axes for displaying the visualization. You can obtain the current axes handle by returning the function to an output variable:

```
ax = showExtrinsics(cameraParams)
```

You can also use the gca function to get the current axes handle.

Example: showExtrinsics(cameraParams, 'Parent', ax)

### Output Arguments

#### **ax** — Current axes handle

scalar value

Current axes handle, returned as a scalar value. The function returns the handle to the current axes for the current figure.

Example: `ax = showExtrinsics(cameraParams)`

### More About

- “Single Camera Calibration App”

### See Also

`cameraParameters` | `stereoParameters` | `Camera Calibrator` | `detectCheckerboardPoints` | `estimateCameraParameters` | `generateCheckerboardPoints` | `plotCamera` | `showReprojectionErrors` | `Stereo Camera Calibrator` | `undistortImage`

**Introduced in R2014a**

# showMatchedFeatures

Display corresponding feature points

## Syntax

```
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2)  
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2,method)
```

```
showMatchedFeatures( ____,PlotOptions, {MarkerStyle1, MarkerStyle2,  
LineStyle})
```

```
H = showMatchedFeatures( ____ )
```

## Description

`showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2)` displays a falsecolor overlay of images `I1` and `I2` with a color-coded plot of corresponding points connected by a line. `matchedPoints1` and `matchedPoints2` contain the coordinates of corresponding points in `I1` and `I2`. The input points can be  $M$ -by-2 matrices of  $M$  number of  $[x\ y]$  coordinates, or `SURFPoints`, `MSERRegions`, or `cornerPoints` object.

`showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2,method)` displays images `I1` and `I2` using the visualization style specified by the `method` parameter.

`showMatchedFeatures( ____,PlotOptions, {MarkerStyle1, MarkerStyle2, LineStyle})` lets you specify custom plot options in a cell array containing three string values. The `MarkerStyle1`, `MarkerStyle2`, and `LineStyle` string values correspond to the marker specification in `I1`, marker specification in `I2`, and line style and color. The `LineStyle` syntax of the `plot` function defines each of the specifiers.

`H = showMatchedFeatures( ____ )` returns the handle to the image object returned by `showMatchedFeatures`.

## Examples

### Find Corresponding Points Between Two Images Using Harris Features

#### Read Images.

```
I1 = rgb2gray(imread('parkinglot_left.png'));  
I2 = rgb2gray(imread('parkinglot_right.png'));
```

#### Detect SURF features

```
points1 = detectHarrisFeatures(I1);  
points2 = detectHarrisFeatures(I2);
```

#### Extract features

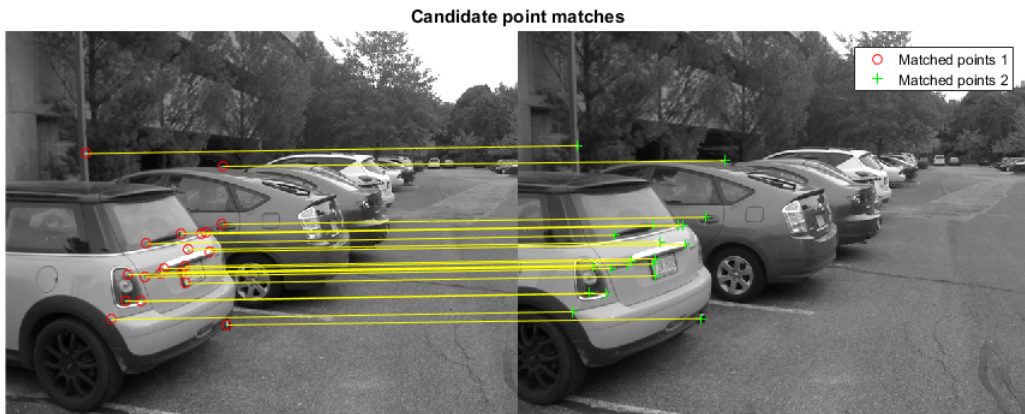
```
[f1, vpts1] = extractFeatures(I1, points1);  
[f2, vpts2] = extractFeatures(I2, points2);
```

#### Match features.

```
indexPairs = matchFeatures(f1, f2) ;  
matchedPoints1 = vpts1(indexPairs(1:20, 1));  
matchedPoints2 = vpts2(indexPairs(1:20, 2));
```

#### Visualize candidate matches.

```
figure; ax = axes;  
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2,'montage','Parent',ax);  
title(ax, 'Candidate point matches');  
legend(ax, 'Matched points 1','Matched points 2');
```



### Display Corresponding Points Between Two Rotated and Scaled Images

Use SURF features to find corresponding points between two images rotated and scaled with respect to each other.

#### Read images.

```
I1 = imread('cameraman.tif');
I2 = imresize(imrotate(I1,-20), 1.2);
```

#### Detect SURF features.

```
points1 = detectSURFFeatures(I1);
points2 = detectSURFFeatures(I2);
```

#### Extract features.

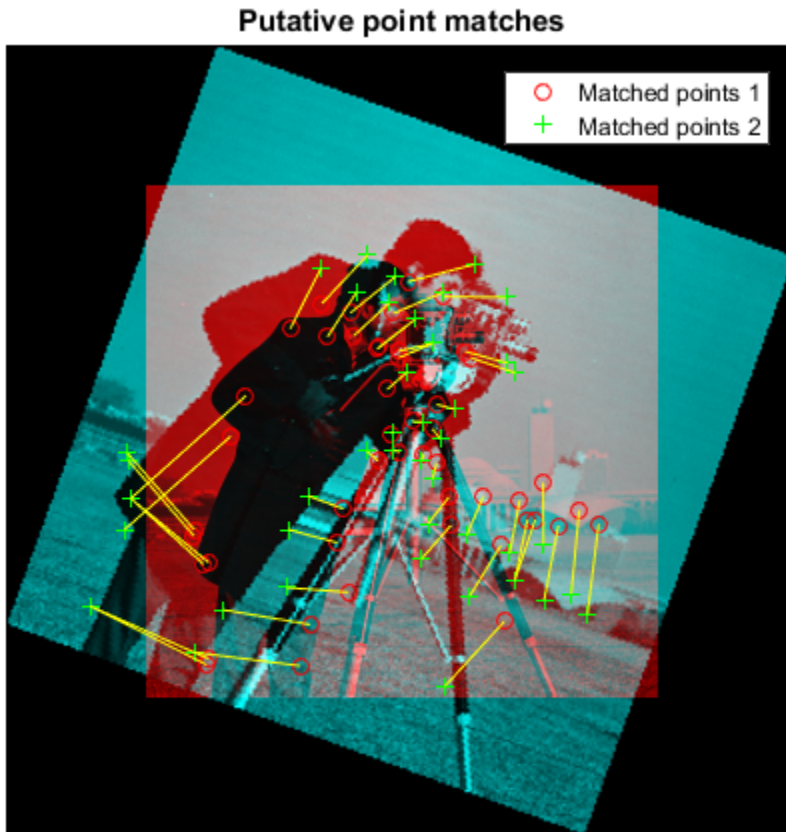
```
[f1, vpts1] = extractFeatures(I1, points1);
[f2, vpts2] = extractFeatures(I2, points2);
```

#### Match features.

```
indexPairs = matchFeatures(f1, f2) ;
matchedPoints1 = vpts1(indexPairs(:, 1));
matchedPoints2 = vpts2(indexPairs(:, 2));
```

#### Visualize candidate matches.

```
figure; ax = axes;  
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2,'Parent',ax);  
title(ax, 'Putative point matches');  
legend(ax,'Matched points 1','Matched points 2');
```



## Input Arguments

**I1** — Input image  
numeric array

Input image one, specified as a numeric array.

### **I2 — Input image**

numeric array

Input image two, specified as a numeric array.

### **matchedPoints1 — Coordinates of points**

*M*-by-2 matrix | SURFPoints object | MSERRegions object | cornerPoints object | BRISKPoints object

Coordinates of points in image one, specified as an *M*-by-2 matrix of *M* number of [x y] coordinates, or as a SURFPoints, MSERRegions, cornerPoints, or BRISKPoints object.

### **matchedPoints2 — Coordinates of points**

*M*-by-2 matrix | SURFPoints object | MSERRegions object | cornerPoints object | BRISKPoints object

Coordinates of points in image two, specified as an *M*-by-2 matrix of *M* number of [x y] coordinates, or as a SURFPoints, MSERRegions, cornerPoints, or BRISKPoints object.

### **method — Display method**

falsecolor (default) | blend | montage

Display style method, specified as one of the following:

**falsecolor:** Overlay the images by creating a composite red-cyan image showing I1 as red and I2 as cyan.

**blend:** Overlay I1 and I2 using alpha blending.

**montage:** Place I1 and I2 next to each other in the same image.

## **Name-Value Pair Arguments**

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

Example:

### **'PlotOptions' — Line style and color**

{ 'ro', 'g+', 'y-' } (default) | cell array



Line style and color options, specified as a cell array containing three string values, *{MarkerStyle1, MarkerStyle2, LineStyle}*, corresponding to a marker specification in I1, marker specification in I2, and line style and color. The `LineStyle` syntax of the `plot` function defines each of the specifiers.

**'Parent' — Output axes**

axes graphics object

Output axes for displaying visualization, specified as an `axes` graphics object.

## Output Arguments

**H — Handle to image object**

handle

Handle to image object, returned as the handle to the image object returned by `showMatchedFeatures`.

## See Also

`SURFPoints` | `MSERRegions` | `cornerPoints` | `estimateGeometricTransform` | `imshowpair` | `legend` | `matchFeatures`

**Introduced in R2012b**

## **showPointCloud**

Plot 3-D point cloud

### **Syntax**

```
showPointCloud
```

### **Description**

`showPointCloud` was renamed to `pcshow`. Please use `pcshow` in place of `showPointCloud`.

**Introduced in R2014b**

# pcshow

Plot 3-D point cloud

## Syntax

```
pcshow(ptCloud)
```

```
pcshow(xyzPoints)  
pcshow(xyzPoints,C)
```

```
pcshow( ____,Name,Value)
```

```
ax = pshow( ____)
```

## Description

`pcshow(ptCloud)` displays points using the locations and colors stored in the point cloud object.

`pcshow(xyzPoints)` displays points specified by the `xyzPoints` matrix.

`pcshow(xyzPoints,C)` displays points contained in the `xyzPoints` matrix, with colors specified by `C`.

`pcshow( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

`ax = pshow( ____)` returns the plot axes.

## Examples

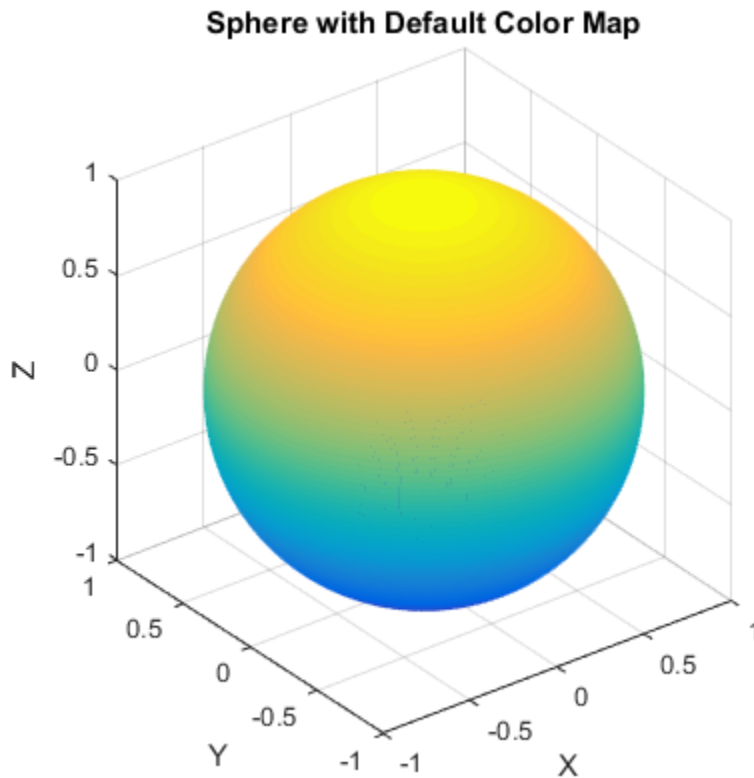
### Plot Spherical Point Cloud with Texture Mapping

Generate a sphere consisting of 600-by-600 faces.

```
numFaces = 600;  
[x,y,z] = sphere(numFaces);
```

Plot the sphere using the default color map.

```
figure;  
pcshow([x(:),y(:),z(:)]);  
title('Sphere with Default Color Map');  
xlabel('X');  
ylabel('Y');  
zlabel('Z');
```



Load an image for texture mapping.

```
I = im2double(imread('visionteam1.jpg'));  
imshow(I);
```



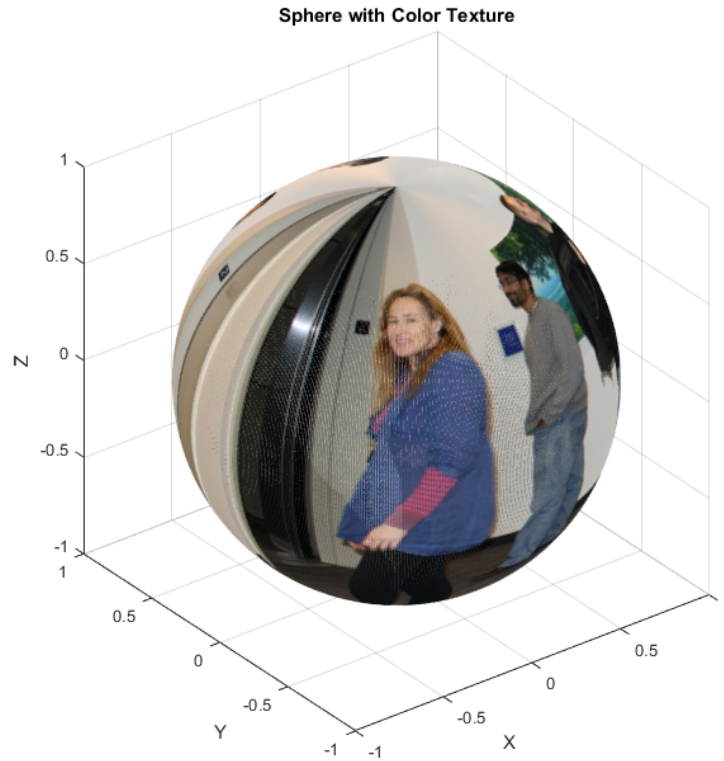
Resize and flip the image for mapping the coordinates.

```
J = flipud(imresize(I,size(x)));
```

Plot the sphere with the color texture.

```
pcshow([x(:),y(:),z(:)],reshape(J,[],3));  
title('Sphere with Color Texture');  
xlabel('X');  
ylabel('Y');
```

```
zlabel('Z');
```



- “Structure From Motion From Two Views”
- “Depth Estimation From Stereo Video”

## Input Arguments

**ptCloud** — Point cloud  
pointCloud object

Point cloud, specified as a pointCloud object.

### xyzPoints — Point cloud *x*, *y*, and *z* locations

*M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud *x*, *y*, and *z* locations, specified as either an *M*-by-3 or an *M*-by-*N*-by-3 numeric matrix. The xyzPoints numeric matrix contains *M* or *M*-by-*N* [*x*,*y*,*z*] points. The *z* values in the matrix, which generally corresponds to depth or elevation, determine the color of each point. When you do not specify the **C** input color, the function maps the *z* value to a color in the current colormap.

### **C** — Point cloud color

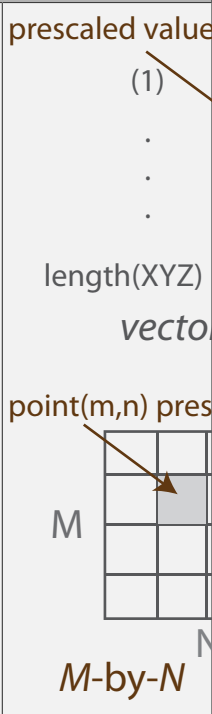
ColorSpec (Color Specification) color string | 1-by-3 RGB vector | *M*-by-1 vector  
| *M*-by-*N* matrix | *M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud color of points, specified as one of:

- ColorSpec (Color Specification) color string, a 1-by-3 RGB vector
- 1-by-3 RGB vector
- *M*-by-1 vector
- *M*-by-*N* matrix
- *M*-by-3 matrix
- *M*-by-*N*-by-3 matrix

You can specify the same color for all points or a different color for each point. When you set **C** to **single** or **double**, the RGB values range between [0, 1]. When you set **C** to **uint8**, the values range between [0, 255].

Points Input	Color Selection	Valid Values of C			
xyzPoints	Same color for all points	ColorSpec (Color Specification) color string or a 1-by-3 RGB vector <div style="text-align: center;"> <span style="color: red;">r</span>   <span style="color: green;">g</span>   <span style="color: blue;">b</span>  <table border="1" style="display: inline-table; margin: 0 auto;"> <tr> <td style="width: 20px; height: 20px;"></td> <td style="width: 20px; height: 20px;"></td> <td style="width: 20px; height: 20px;"></td> </tr> </table>   <b>1-by-3</b> </div>			

Points Input	Color Selection	Valid Values of C	
	Different color for each point	Vector or $M$ -by- $N$ matrix. The matrix must contain values that are linearly mapped to a color in the current colormap.	<p>prescaled value</p> <p>(1)</p> <p>.</p> <p>.</p> <p>.</p> <p>length(XYZ)</p> <p>vector</p> <p>point(m,n) prescaled value</p> <p>M</p> <p>N</p> <p><math>M</math>-by-<math>N</math></p> 



Points Input	Color Selection	Valid Values of C
		<p><math>M</math>-by-3 matrix or <math>M</math>-by-<math>N</math>-by-3 matrix containing RGB values for each point.</p> <p>The diagram illustrates the data flow from a point matrix to a grid. On the left, a column vector of points is shown as <math>\begin{bmatrix} x_1 &amp; y_1 &amp; z_1 \\ \vdots &amp; \vdots &amp; \vdots \\ x_m &amp; y_m &amp; z_m \end{bmatrix}</math>, labeled <math>M</math>-by-3. Arrows point from this vector to a grid with three columns labeled <math>r</math>, <math>g</math>, and <math>b</math>. Below this, a 2D grid of size <math>M</math>-by-<math>N</math> is shown. A specific point is labeled <math>\text{point}(m,n)</math> and is highlighted with a gray square. The grid is also labeled <math>M</math>-by-<math>N</math>-by-3.</p>

## Name-Value Pair Arguments

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

Example: 'VerticalAxisDir','Up' sets the vertical axis direction to up.

### 'MarkerSize' — Diameter of marker

6 (default) | positive scalar

Diameter of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive scalar. The value specifies the approximate diameter of the point marker.

MATLAB graphics define the unit as points. A marker size larger than six can reduce the rendering performance.

### 'VerticalAxis' — Vertical axis

'Z' (default) | 'X' | 'Y'

Vertical axis, specified as the comma-separated pair consisting of 'VerticalAxis' and a string specifying the vertical axis: 'X', 'Y', or 'Z'.

### 'VerticalAxisDir' — Vertical axis direction

'Up' (default) | 'Down'

Vertical axis direction, specified as the comma-separated pair consisting of 'VerticalAxisDir' and a string specifying the direction of the vertical axis: 'Up' or 'Down'.

### 'Parent' — Output axes

axes graphics object

Output axes, specified as the comma-separated pair consisting of 'Parent' and an axes graphics object that displays the point cloud visualization.

## Output Arguments

### ax — Plot axes

axes graphics object

Plot axes, returned as an axes graphics object.

## More About

### Tips

To improve performance, `pcshow` automatically downsamples the rendered point cloud during interaction with the figure. The downsampling occurs only for rendering the point cloud and does not affect the saved points.

### Algorithms

The `pcshow` function supports the 'opengl' option for the `Renderer` figure property only.

- “Coordinate Systems”

## See Also

`pointCloud` | `pcplayer` | `planeModel` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcwrite` | `plot3` | `reconstructScene` | `scatter3` | `triangulate`

**Introduced in R2015b**

## pcshowpair

Visualize difference between two point clouds

### Syntax

```
pcshowpair(ptCloudA,ptCloudB)
```

```
pcshowpair(ptCloudA,ptCloudB,Name,Value)
```

```
ax = pshowpair( ___ )
```

### Description

`pcshowpair(ptCloudA,ptCloudB)` creates a visualization depicting the differences between the two input point clouds. The differences are displayed using a blending of magenta for point cloud A and green for point cloud B.

`pcshowpair(ptCloudA,ptCloudB,Name,Value)` visualizes the differences using additional options specified by one or more `Name,Value` pair arguments.

`ax = pshowpair( ___ )` returns the plot axes to the visualization of the differences, using any of the preceding syntaxes.

### Examples

#### Visualize the Difference Between Two Point Clouds

Load two point clouds that were captured using a Kinect device in a home setting.

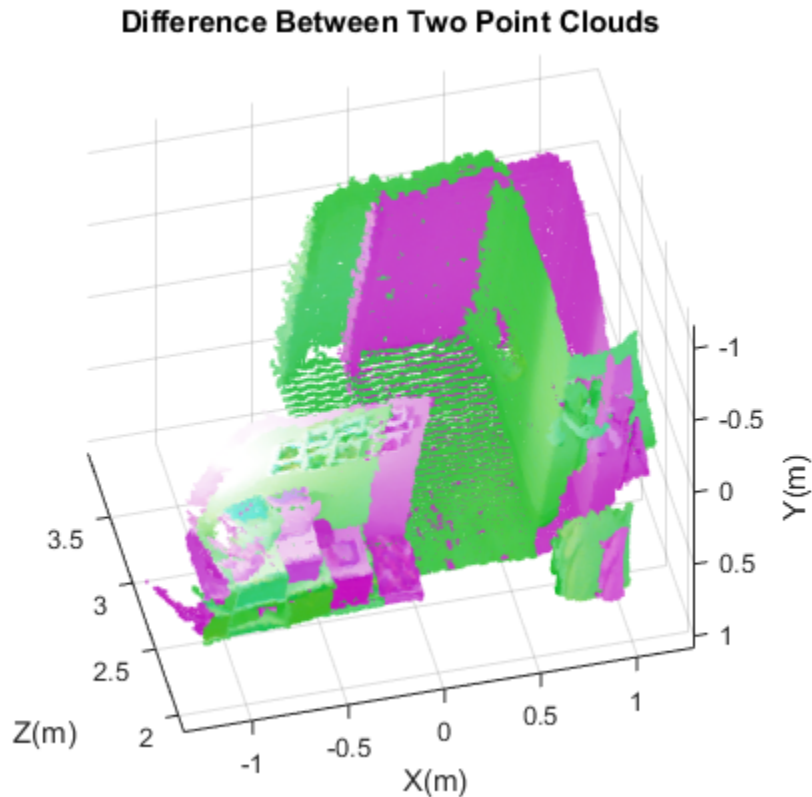
```
load('livingRoom');
```

```
pc1 = livingRoomData{1};  
pc2 = livingRoomData{2};
```

Plot and set the viewpoint of point clouds.

```
figure  
pcshowpair(pc1,pc2,'VerticalAxis','Y','VerticalAxisDir','Down')
```

```
title('Difference Between Two Point Clouds')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
```



- “Structure From Motion From Two Views”
- “Depth Estimation From Stereo Video”

## Input Arguments

**ptCloudA** — Point cloud  
pointCloud object

Point cloud A, specified as a `pointCloud` object. The function uses levels of magenta to represent `ptCloudA` and a pure magenta when the point cloud contains no color information.

#### **ptCloudB** — Point cloud

`pointCloud` object

Point cloud B, specified as a `pointCloud` object. The function uses levels of green to represent `ptCloudB` and a pure green when the point cloud contains no color information.

## Name-Value Pair Arguments

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

Example: `'VerticalAxisDir', 'Up'` sets the vertical axis direction to up.

#### **'MarkerSize'** — Diameter of marker

6 (default) | positive scalar

Approximate diameter of the point marker, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive scalar. The units are in points. A marker size larger than six can reduce the rendering performance.

#### **'VerticalAxis'** — Vertical axis

'Z' (default) | 'X' | 'Y'

Vertical axis, specified as the comma-separated pair consisting of `'VerticalAxis'` and a string specifying the vertical axis: 'X', 'Y', or 'Z'.

#### **'VerticalAxisDir'** — Vertical axis direction

'Up' (default) | 'Down'

Vertical axis direction, specified as the comma-separated pair consisting of `'VerticalAxisDir'` and a string specifying the direction of the vertical axis: 'Up' or 'Down'.

#### **'Parent'** — Output axes

`axes` graphics object

Output axes, specified as the comma-separated pair consisting of 'Parent' and an `axes` graphics object that displays the point cloud visualization.

## Output Arguments

### **ax** — Plot axes

`axes` graphics object

Plot axes, returned as an `axes` graphics object. Points with NaN or Inf coordinates are not displayed.

## More About

### Tips

To improve performance, `pcshowpair` automatically downsamples the rendered point cloud during interaction with the figure. The downsampling occurs only for rendering the point cloud and does not affect the saved points.

- “Coordinate Systems”

### See Also

`pointCloud` | `pcplayer` | `planeModel` | `pcdenoise` | `pcdownsample` | `pcfitplane` | `pcmerge` | `pcread` | `pcregrigid` | `pcshow` | `pcwrite` | `plot3` | `reconstructScene` | `scatter3` | `triangulate`

**Introduced in R2015b**

## showReprojectionErrors

Visualize calibration errors

### Syntax

```
showReprojectionErrors(cameraParams)  
showReprojectionErrors(cameraParams,view)  
showReprojectionErrors( ____,Name,Value)
```

```
ax = showReprojectionErrors( ____)
```

### Description

`showReprojectionErrors(cameraParams)` displays a bar graph that represents the calibration accuracy for a single camera or for a stereo pair. The bar graph displays the mean reprojection error per image. The `cameraParams` input contains either a `cameraParameters` or a `stereoParameters` object, which the `estimateCameraParameters` function returns.

`showReprojectionErrors(cameraParams,view)` displays the reprojection errors using the visualization style specified by the `view` input.

`showReprojectionErrors( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments, using any of the preceding syntaxes.

`ax = showReprojectionErrors( ____)` returns the plot axis, using any of the preceding syntaxes.

### Examples

#### Visualize Reprojection Errors for a Single Camera

Create a cell array of calibration image file names.

```
for i = 1:5
```



---

```
imageFileName = sprintf('image%d.tif', i);  
imageFileNames{i} = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', 'calibration'  
end
```

Detect the calibration pattern.

```
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate world coordinates for the corners of the squares.

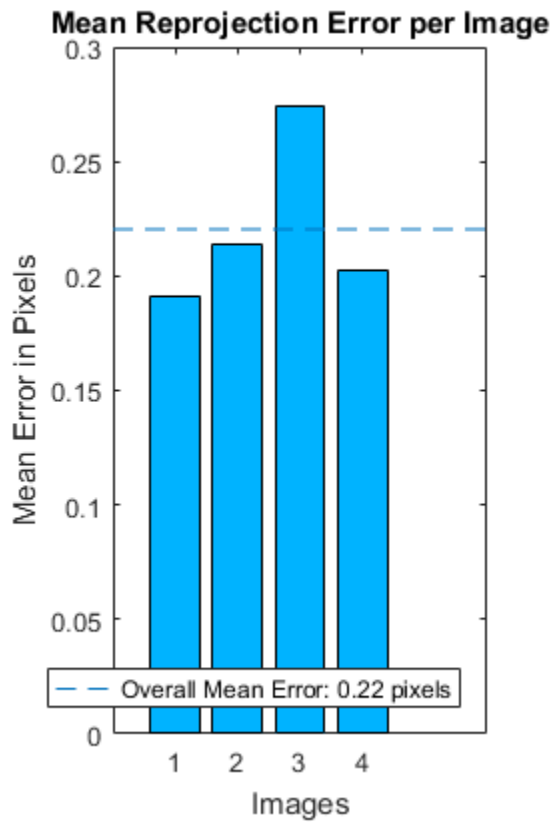
```
squareSide = 25; % (millimeters)  
worldPoints = generateCheckerboardPoints(boardSize, squareSide);
```

Calibrate the camera.

```
params = estimateCameraParameters(imagePoints, worldPoints);
```

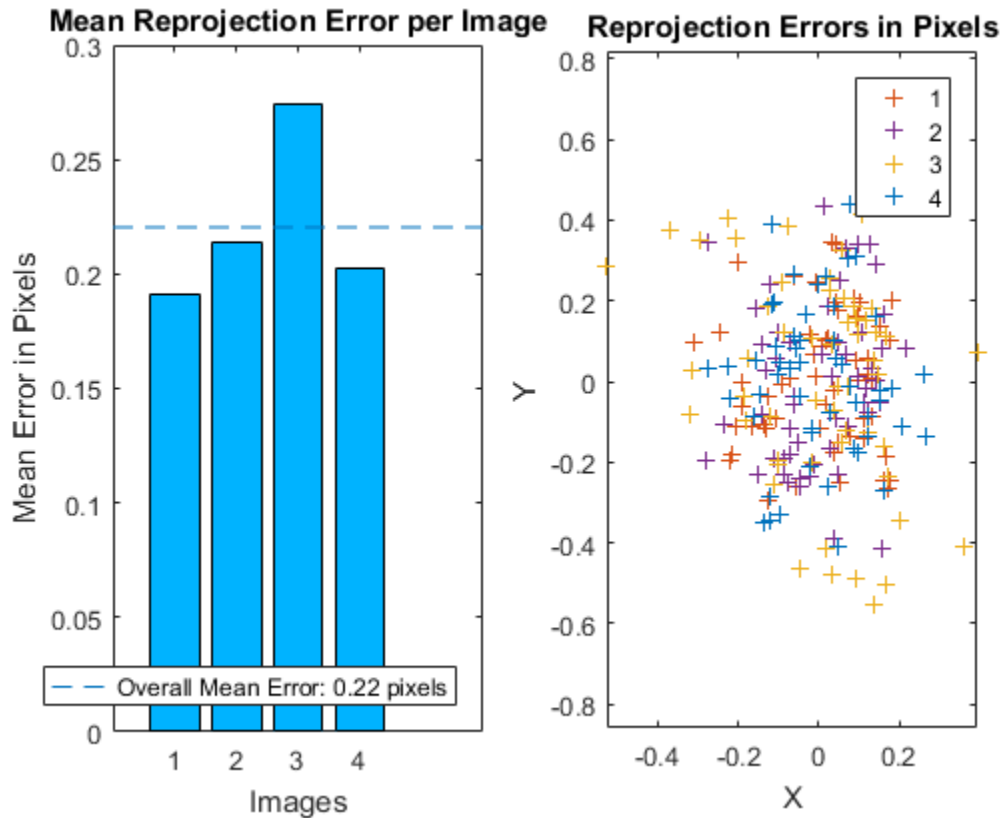
Visualize errors as a bar graph.

```
subplot(1, 2, 1);  
showReprojectionErrors(params);
```



Visualize errors as a scatter plot.

```
subplot(1,2,2);  
showReprojectionErrors(params, 'ScatterPlot');
```



### Visualize Reprojection Errors for Stereo Pair of Cameras

Specify the calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');
leftImages = imageSet(fullfile(imageDir,'left'));
rightImages = imageSet(fullfile(imageDir,'right'));
images1 = leftImages.ImageLocation;
images2 = rightImages.ImageLocation;
```

Detect the checkerboard patterns.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images1,images2);
```

Specify the world coordinates of the checkerboard keypoints.

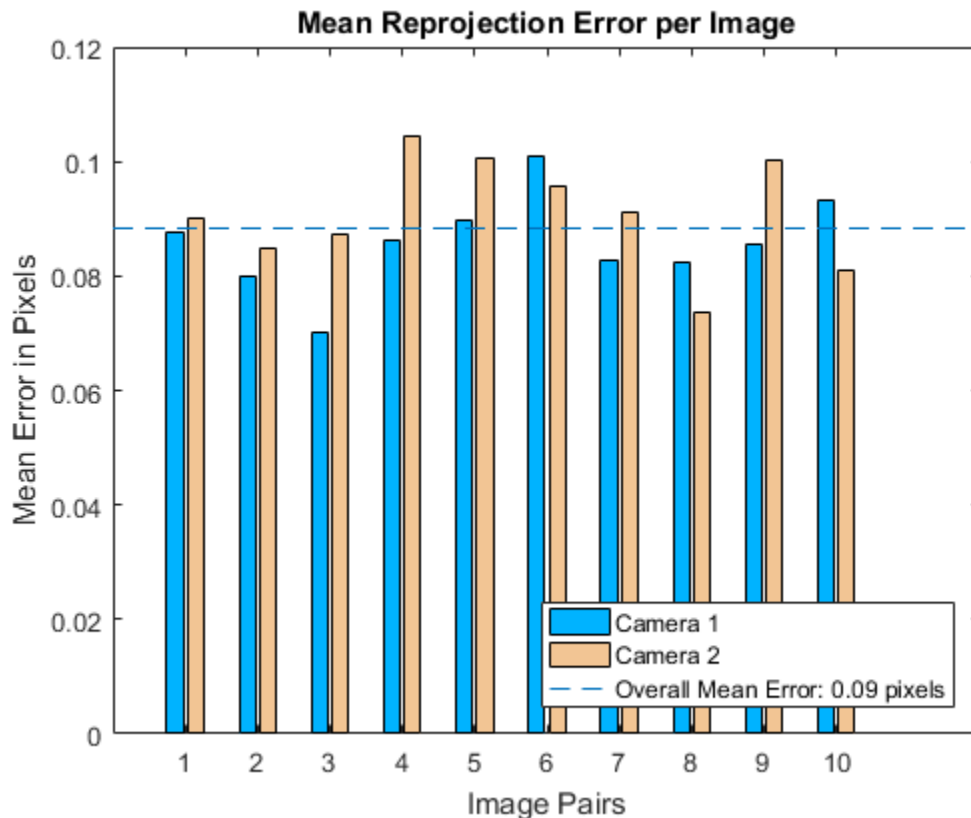
```
squareSize = 108; % millimeters  
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the stereo camera system.

```
params = estimateCameraParameters(imagePoints,worldPoints);
```

Visualize the calibration accuracy.

```
showReprojectionErrors(params);
```



## Input Arguments

**cameraParams** — Object containing parameters of single camera or stereo pair  
 cameraParameters object | stereoParameters object

Object containing parameters of single camera or stereo pair, specified as either a cameraParameters or stereoParameters object. You can create the single camera or stereo pair input object using the `estimateCameraParameters` function.

You can also use the Camera Calibrator app to create the cameraParameters input object, or use Stereo Camera Calibrator app to create the stereoParameters input object. See “Single Camera Calibration App” and “Stereo Calibration App”.

#### **view — Bar graph or scatter plot view**

'BarGraph' | 'ScatterPlot'

Bar graph or scatter plot view, specified as the character string 'BarGraph' or 'ScatterPlot'. The `view` input sets the visualization for the camera extrinsic parameters. Set `view` to 'BarGraph' to display the mean error per image as a bar graph. Set `view` to 'ScatterPlot' to display the error for each point as a scatter plot. The 'ScatterPlot' option applies only to the single camera case.

### **Name-Value Pair Arguments**

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

Example: 'view', 'BarGraph' displays the mean error per image as a bar graph.

#### **'HighlightIndex' — Highlight selection index**

[] (default) | vector | scalar

Highlight selection index, specified as a scalar or a vector of integers. When you set the `view` to 'BarGraph', the function highlights the bars corresponding to the selected images. When you set the `view` to 'ScatterPlot', the function highlights the points corresponding to the selected images with circle markers.

#### **'Parent' — Output axes**

current axes (default) | scalar value

Output axes, specified as the comma-separated pair consisting of 'Parent' and a scalar value. Specify output axes to display the visualization. You can obtain the current axes handle by returning the function to an output variable:

```
ax = showReprojectionErrors(cameraParams)
```

You can also use the `gca` function to get the current axes handle.

Example: `showReprojectionErrors(cameraParams, 'Parent', ax)`

### **Output Arguments**

#### **ax — Current axes handle**

scalar value

Current axes handle, returned as a scalar value. The function returns the handle to the current axes for the current figure.

Example: `ax = showReprojectionErrors(cameraParams)`

## More About

- “Single Camera Calibration App”

## See Also

[cameraParameters](#) | [stereoParameters](#) | [Camera Calibrator](#) |  
[detectCheckerboardPoints](#) | [estimateCameraParameters](#) |  
[generateCheckerboardPoints](#) | [showExtrinsics](#) | [Stereo Camera Calibrator](#) |  
[undistortImage](#)

**Introduced in R2014a**

## stereoAnaglyph

Create red-cyan anaglyph from stereo pair of images

### Syntax

```
J = stereoAnaglyph(I1,I2)
```

### Description

`J = stereoAnaglyph(I1,I2)` combines images `I1` and `I2` into a red-cyan anaglyph. When the inputs are rectified stereo images, you can view the output image with red-blue stereo glasses to see the stereo effect.

### Examples

#### Create 3-D Stereo Display

**Load parameters for a calibrated stereo pair of cameras.**

```
load('webcamsSceneReconstruction.mat')
```

**Load a stereo pair of images.**

```
I1 = imread('sceneReconstructionLeft.jpg');  
I2 = imread('sceneReconstructionRight.jpg');
```

**Rectify the stereo images.**

```
[J1, J2] = rectifyStereoImages(I1, I2, stereoParams);
```

**Create the anaglyph.**

```
A = stereoAnaglyph(J1, J2);
```

**Display the anaglyph. Use red-blue stereo glasses to see the stereo effect.**

```
figure; imshow(A);
```





- “Structure From Motion From Two Views”
- “Structure From Motion From Multiple Views”

## Input Arguments

### I1 — Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-*D* grayscale image

Input image I1, specified as an *M*-by-*N*-by-3 truecolor image or an *M*-by-*N* 2-*D* grayscale image. I1 and I2 must be real, finite, and nonsparse, and the images must be the same size. If the images are not the same size, use `imfuse` to pad the smaller image dimension with zeros before creating the anaglyph.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

### I2 — Input image

*M*-by-*N* 2-*D* grayscale image | *M*-by-*N*-by-3 truecolor image

Input image **I2**, specified as an  $M$ -by- $N$ -by-3 truecolor image or an  $M$ -by- $N$  2- $D$  grayscale image. **I1** and **I2** must be real, finite, and nonsparse, and the images must be the same size. If the images are not the same size, use `imfuse` to pad the smaller image dimension with zeros before creating the anaglyph.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## Output Arguments

### **J** — Stereo anaglyph output image

$M$ -by- $N$ -by-3 truecolor image

Stereo anaglyph output image, returned as an  $M$ -by- $N$ -by-3 truecolor image. Output image **J** is the same size as input images **I1** and **I2**.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

## See Also

`estimateUncalibratedRectification` | `imfuse` | `imshowpair` | `reconstructScene` | `rectifyStereoImages`

**Introduced in R2014b**

# trainCascadeObjectDetector

Train cascade object detector model

## Syntax

```
trainCascadeObjectDetector(outputXMLFilename, positiveInstances,  
negativeImages)  
trainCascadeObjectDetector(outputXMLFilename, 'resume')
```

```
trainCascadeObjectDetector( ____, Name, Value)
```

## Description

`trainCascadeObjectDetector(outputXMLFilename, positiveInstances, negativeImages)` writes a trained cascade detector XML file with the name, `outputXMLFilename`. The name specified by the `outputXMLFilename` input must have an XML extension. For a more detailed explanation on how this function works, refer to “Train a Cascade Object Detector”.

`trainCascadeObjectDetector(outputXMLFilename, 'resume')` resumes an interrupted training session. The `outputXMLFilename` input must match the output file name from the interrupted session. All arguments saved from the earlier session are reused automatically.

`trainCascadeObjectDetector( ____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Train A Stop Sign Detector

This example shows you the steps involved in training a cascade object detector. It trains a 5-stage detector from a very small training set. In reality, an accurate detector requires many more stages and thousands of positive samples and negative images.

Load the positive samples data from a .mat file. The file names and bounding boxes are contained in an array of structures named 'data'.

```
load('stopSigns.mat');
```

Add the images location to the MATLAB path.

```
imDir = fullfile(matlabroot,'toolbox','vision','visiondata','stopSignImages');  
addpath(imDir);
```

Specify the folder for negative images.

```
negativeFolder = fullfile(matlabroot,'toolbox','vision','visiondata','nonStopSigns');
```

Train a cascade object detector called 'stopSignDetector.xml' using HOG features. The following command may take several minutes to run:

```
trainCascadeObjectDetector('stopSignDetector.xml',data,negativeFolder,'FalseAlarmRate',
```

```
Automatically setting ObjectTrainingSize to [ 35, 32 ]  
Using at most 42 of 42 positive samples per stage  
Using at most 84 negative samples per stage
```

```
Training stage 1 of 5  
[.....]  
Used 42 positive and 84 negative samples  
Time to train stage 1: 0 seconds
```

```
Training stage 2 of 5  
[.....]  
Used 42 positive and 84 negative samples  
Time to train stage 2: 0 seconds
```

```
Training stage 3 of 5  
[.....]  
Used 42 positive and 84 negative samples  
Time to train stage 3: 3 seconds
```

```
Training stage 4 of 5  
[.....]  
Used 42 positive and 84 negative samples  
Time to train stage 4: 5 seconds
```

```
Training stage 5 of 5
[.....]
Very low false alarm rate 0.000308187 reached in stage.
Training will halt and return cascade detector with 4 stages
Time to train stage 5: 17 seconds
```

Training complete

Use the newly trained classifier to detect a stop sign in an image.

```
detector = vision.CascadeObjectDetector('stopSignDetector.xml');
```

Read the test image.

```
img = imread('stopSignTest.jpg');
```

Detect a stop sign.

```
bbox = step(detector, img);
```

Insert bounding boxes and return marked image.

```
detectedImg = insertObjectAnnotation(img, 'rectangle', bbox, 'stop sign');
```

Display the detected stop sign.

```
figure;
imshow(detectedImg);
```



Remove the image directory from the path.

```
rmpath(imDir);
```

- “Image Category Classification Using Bag of Features”

## Input Arguments

**positiveInstances** — Positive samples

array of structs

Positive samples, specified as an array of structs containing string image file names, and an  $M$ -by-4 matrix of bounding boxes specifying object locations in the images. You can use the Training Image Labeler app to label objects of interest with bounding boxes. The app outputs an array of structs to use for `positiveInstances`.

The `struct` fields are defined as follows:

<code>imageFilename</code>	A string that specifies the image name. The image can be true color, grayscale, or indexed, in any of the formats supported by <code>imread</code> .
<code>objectBoundingBoxes</code>	An $M$ -by-4 matrix of $M$ bounding boxes. Each bounding box is in the format, <code>[x y width height]</code> and specifies an object location in the corresponding image.

The function automatically determines the number of positive samples to use at each of the cascade stages. This value is based on the number of stages and the true positive rate. The true positive rate specifies how many positive samples can be misclassified

Data Types: `struct`

### **negativeImages** — Negative images

cell array | string

Negative images, specified as either a path to a folder containing images or as a cell array of image file names. Because the images are used to generate negative samples, they must not contain any objects of interest. Instead, they should contain backgrounds associated with the object.

Data Types: `char` | `cell`

### **outputXMLFilename** — Trained cascade detector file name

string

Trained cascade detector file name, specified as a string with an XML extension.

Data Types: `char`

## **Name-Value Pair Arguments**

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

Example: `'FeatureType', 'Haar'` specifies Haar for the type of features to use.

### **'ObjectTrainingSize'** — Object size for training

'Auto' (default) | 2-element vector

Training object size, specified as the comma-separated pair. This pair contains 'ObjectTrainingSize' and either a 2-element vector [*height*, *width*] or as the string 'Auto'. Before training, the function resizes the positive and negative samples to ObjectTrainingSize in pixels. If you select 'Auto', the function determines the size automatically based on the median width-to-height ratio of the positive instances. For optimal detection accuracy, specify an object training size close to the expected size of the object in the image. However, for faster training and detection, set the object training size to be smaller than the expected size of the object in the image.

Data Types: char | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 'NegativeSamplesFactor' — Negative sample factor

2 (default) | real-valued scalar

Negative sample factor, specified as the comma-separated pair consisting of 'NegativeSamplesFactor' and a real-valued scalar. The number of negative samples to use at each stage is equal to NegativeSamplesFactor x [*the number of positive samples used at each stage*].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 'NumCascadeStages' — Number of cascade stages

20 (default) | positive integer

Number of cascade stages to train, specified as the comma-separated pair consisting of 'NumCascadeStages' and a positive integer. Increasing the number of stages may result in a more accurate detector but also increases training time. More stages may require more training images, because at each stage, some number of positive and negative samples may be eliminated. This value depends on the FalseAlarmRate and the TruePositiveRate. More stages may also allow you to increase the FalseAlarmRate. See the “Train a Cascade Object Detector” tutorial for more details.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### 'FalseAlarmRate' — Acceptable false alarm rate

0.5 (default) | value in the range (0 1]

Acceptable false alarm rate at each stage, specified as the comma-separated pair consisting of 'FalseAlarmRate' and a value in the range (0 1]. The false alarm rate is the fraction of negative training samples incorrectly classified as positive samples.



The overall false alarm rate is calculated using the `FalseAlarmRate` per stage and the number of cascade stages, `NumCascadeStages`:

$FalseAlarmRate^{NumCascadeStages}$

Lower values for `FalseAlarmRate` increase complexity of each stage. Increased complexity can achieve fewer false detections but may result in longer training and detection times. Higher values for `FalseAlarmRate` may require a greater number of cascade stages to achieve reasonable detection accuracy.

Data Types: `single` | `double`

### 'TruePositiveRate' — Minimum true positive rate

0.995 (default)

Minimum true positive rate required at each stage, specified as the comma-separated pair consisting of 'TruePositiveRate' and a value in the range (0 1]. The true positive rate is the fraction of correctly classified positive training samples.

The overall resulting target positive rate is calculated using the `TruePositiveRate` per stage and the number of cascade stages, `NumCascadeStages`:

$TruePositiveRate^{NumCascadeStages}$

Higher values for `TruePositiveRate` increase complexity of each stage. Increased complexity can achieve a greater number of correct detections but may result in longer training and detection times.

Data Types: `single` | `double`

### 'FeatureType' — Feature type

'HOG' (default) | 'LBP' | 'Haar'

Feature type, specified as the comma-separated pair consisting of 'FeatureType' and one of three strings. The possible features types are:

'Haar' [1]	Haar-like features
'LBP' [2]	Local Binary Patterns
'HOG' [3]	Histogram of Oriented Gradients

The function allocates a large amount of memory, this is especially the case for the Haar features. To avoid running out of memory, use this function on a 64-bit operating system with a sufficient amount of RAM.

Data Types: `char`

## More About

### Tips

Training a good detector requires thousands of training samples. Processing time for a large amount of data varies. It is likely to take on the order of hours or even days. During training, the function displays the time it took to train each stage in the MATLAB command window.

- “Label Images for Classification Model Training”
- “Train a Cascade Object Detector”
- “Multiple Object Tracking”
- Cascade Training GUI

### References

- [1] Viola, P., and M. J. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features". *Proceedings of the 2001 IEEE Computer Society Conference*. Volume 1, 15 April 2001, pp. 1-511–1-518.
- [2] Ojala, T., M. Pietikainen, and T. Maenpaa, “Multiresolution Gray-scale and Rotation Invariant Texture Classification With Local Binary Patterns”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Volume 24, No. 7 July 2002, pp. 971–987.
- [3] Dalal, N., and B. Triggs, “Histograms of Oriented Gradients for Human Detection”. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Volume 1, (2005), pp. 886–893.

### See Also

vision.CascadeObjectDetector | imrect | insertObjectAnnotation | Training Image Labeler

**Introduced in R2013a**

# trainImageCategoryClassifier

Train an image category classifier

## Syntax

```
classifier = trainImageCategoryClassifier(imgSets,bag)
classifier = trainImageCategoryClassifier(imgSets,bag,Name,Value)
```

## Description

`classifier = trainImageCategoryClassifier(imgSets,bag)` returns an image category classifier. The classifier contains the number of categories and the category labels for the input `imgSets`, an `imageSet` object. The function trains a support vector machine (SVM) multiclass classifier using the input `bag`, a `bagOfFeatures` object.

You must have a Statistics and Machine Learning Toolbox license to use this function.

This function supports parallel computing using multiple MATLAB workers. Enable parallel computing using the “Computer Vision System Toolbox Preferences” dialog. To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Select **Computer Vision System Toolbox**.

`classifier = trainImageCategoryClassifier(imgSets,bag,Name,Value)` returns a `Classifier` object with optional input properties specified by one or more `Name,Value` pair arguments.

## Examples

### Train, Evaluate, and Apply Image Category Classifier

Load two image categories.

```
setDir = fullfile(toolboxdir('vision'),'visiondata','imageSets');
imgSets = imageSet(setDir, 'recursive');
```

Separate the two sets into training and test data. Pick 30% of images from each set for the training data and the remainder 70% for the test data.

```
[trainingSets, testSets] = partition(imgSets, 0.3, 'randomize');
```

Create bag of visual words.

```
bag = bagOfFeatures(trainingSets, 'Verbose', false);
```

Train a classifier with the training sets.

```
categoryClassifier = trainImageCategoryClassifier(trainingSets, bag);
```

Training an image category classifier for 2 categories.

```
-----  
* Category 1: books  
* Category 2: cups
```

```
* Encoding features for category 1...done.  
* Encoding features for category 2...done.
```

```
* Finished training the category classifier. Use evaluate to test the classifier on a t
```

Evaluate the classifier using test images. Display the confusion matrix.

```
confMatrix = evaluate(categoryClassifier, testSets)
```

Evaluating image category classifier for 2 categories.

```
-----  
* Category 1: books  
* Category 2: cups
```

```
* Evaluating 4 images from category 1...done.  
* Evaluating 4 images from category 2...done.
```

```
* Finished evaluating all the test sets.
```

```
* The confusion matrix for this test set is:
```

	PREDICTED	
KNOWN	books	cups
books	0.75	0.25
cups	0.25	0.75

```
* Average Accuracy is 0.75.
```

```
confMatrix =  
    0.7500    0.2500  
    0.2500    0.7500
```

Find the average accuracy of the classification.

```
    mean(diag(confMatrix))  
  
ans =  
  
    0.7500
```

Apply the newly trained classifier to categorize new images.

```
img = imread(fullfile(setDir, 'cups', 'bigMug.jpg'));  
[labelIdx, score] = predict(categoryClassifier, img);
```

Display the classification label.

```
    categoryClassifier.Labels(labelIdx)  
  
ans =  
  
    'cups'
```

- “Image Category Classification Using Bag of Features”

## Input Arguments

### **imgSets** — Image sets

array of `imageSet` object

Image sets, specified as an array of `imageSet` objects. Each element of `imgSets` defines an image category.

### **bag — Bag of features**

bagOfFeatures object

Bag of features, specified as a bagOfFeatures object. The object contains a visual vocabulary of extracted feature descriptors from representative images of each image category.

## **Name-Value Pair Arguments**

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

Example: `'Verbose',true` sets `'Verbose'` to the logical `true`.

### **'Verbose' — Enable progress display to screen**

`true` (default) | `false`

Enable progress display to screen, specified as the comma-separated pair consisting of `'Verbose'` and the logical `true` or `false`.

### **'LearnerOptions' — Classifier options**

default values of `templateSVM` function

Classifier options, specified as the comma-separated pair consisting of `'LearnerOptions'` and the learner options output returned by the `templateSVM` function.

### **Example**

To adjust the regularization parameter of `templateSVM` and to set a custom kernel function, use the following syntax:

```
opts = templateSVM('BoxConstraint', 1.1, 'KernelFunction', 'gaussian');  
classifier = trainImageCategoryClassifier(imgSets, bag, 'LearnerOptions', opts);
```

## **Output Arguments**

### **classifier — Image category classifier**

imageCategoryClassifier object

Image category classifier, returned as an `imageCategoryClassifier` object. The function trains a support vector machine (SVM) multiclass classifier using the error correcting output codes (ECOC) framework.

## More About

- “Image Classification with Bag of Visual Words”

## References

- [1] Csurka, G., C. R. Dance, L. Fan, J. Willamowski, and C. Bray *Visual Categorization with Bag of Keypoints*, Workshop on Statistical Learning in Computer Vision, ECCV 1 (1-22), 1-2.

## See Also

`imageSet` | `bagOfFeatures` | `imageCategoryClassifier` | `fitcecoc` | `templateSVM`

**Introduced in R2014b**

# Training Image Labeler

Label images for training a classifier

## Description

The **Training Image Labeler** app allows you to interactively specify rectangular Regions of Interest (ROIs).

## Open the Training Image Labeler App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `trainingImageLabeler`.

## Examples

### Open Training Image Labeler App

Type `trainingImageLabeler` on the MATLAB command line or select it from the MATLAB desktop **Apps** tab.

### Programmatic Use

`trainingImageLabeler` invokes an app for labeling ground truth data in images. This app allows you to interactively specify rectangular Regions of Interest (ROIs). The ROIs define locations of objects, which are used to train a classifier. It outputs training data in a format supported by the `trainCascadeObjectDetector` function. The function trains a model to use with the `vision.CascadeObjectDetector` detector.

`trainingImageLabeler` `CLOSE` closes all open apps.

## More About

- “Label Images for Classification Model Training”



- “Train a Cascade Object Detector”

## See Also

`vision.CascadeObjectDetector` | `imrect` | `insertObjectAnnotation` | `trainCascadeObjectDetector`

**Introduced in R2014a**

## triangulate

3-D locations of undistorted matching points in stereo images

### Syntax

```
worldPoints = triangulate(matchedPoints1,matchedPoints2,  
stereoParams)  
worldPoints = triangulate(matchedPoints1,matchedPoints2,  
cameraMatrix1,cameraMatrix2)  
[worldPoints,reprojectionErrors] = triangulate( ___ )
```

### Description

`worldPoints = triangulate(matchedPoints1,matchedPoints2, stereoParams)` returns 3-D locations of matching pairs of undistorted image points from two stereo images.

`worldPoints = triangulate(matchedPoints1,matchedPoints2, cameraMatrix1,cameraMatrix2)` returns the 3-D locations of the matching pairs in a world coordinate system. These locations are defined by camera projection matrices.

`[worldPoints,reprojectionErrors] = triangulate( ___ )` additionally returns reprojection errors for the world points using any of the input arguments from previous syntaxes.

#### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Measure Distance from Stereo Camera to a Face

Load stereo parameters.

```
load('webcamsSceneReconstruction.mat');
```

**Read in the stereo pair of images.**

```
I1 = imread('sceneReconstructionLeft.jpg');  
I2 = imread('sceneReconstructionRight.jpg');
```

**Undistort the images.**

```
I1 = undistortImage(I1, stereoParams.CameraParameters1);  
I2 = undistortImage(I2, stereoParams.CameraParameters2);
```

**Detect a face in both images.**

```
faceDetector = vision.CascadeObjectDetector;  
face1 = step(faceDetector, I1);  
face2 = step(faceDetector, I2);
```

**Find the center of the face.**

```
center1 = face1(1:2) + face1(3:4)/2;  
center2 = face2(1:2) + face2(3:4)/2;
```

**Compute the distance from camera 1 to the face.**

```
point3d = triangulate(center1, center2, stereoParams);  
distanceInMeters = norm(point3d)/1000;
```

**Display the detected face and distance.**

```
distanceAsString = sprintf('%0.2f meters', distanceInMeters);  
I1 = insertObjectAnnotation(I1, 'rectangle', face1, distanceAsString, 'FontSize', 18);  
I2 = insertObjectAnnotation(I2, 'rectangle', face2, distanceAsString, 'FontSize', 18);  
I1 = insertShape(I1, 'FilledRectangle', face1);  
I2 = insertShape(I2, 'FilledRectangle', face2);  
  
imshowpair(I1, I2, 'montage');
```



- “Structure From Motion From Two Views”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **matchedPoints1 — Coordinates of points in image 1**

*M*-by-2 matrix | SURFPoints object | MSERRegions object | cornerPoints object | BRISKPoints object

Coordinates of points in image 1, specified as an *M*-by-2 matrix of *M* number of [*x y*] coordinates, or as a SURFPoints, MSERRegions, cornerPoints, or BRISKPoints object. The `matchedPoints1` and `matchedPoints2` inputs must contain points that are matched using a function such as `matchFeatures`.

### **matchedPoints2 — Coordinates of points**

*M*-by-2 matrix | SURFPoints object | MSERRegions object | cornerPoints object | BRISKPoints object

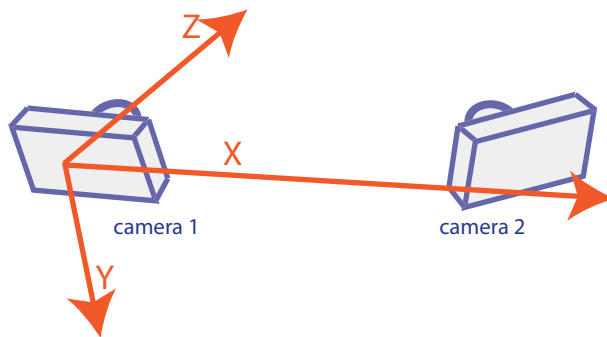
Coordinates of points in image 2, specified as an *M*-by-2 matrix of *M* number of [*x y*] coordinates, or as a SURFPoints, MSERRegions, cornerPoints, or BRISKPoints object. The `matchedPoints1` and `matchedPoints2` inputs must contain points that are matched using a function such as `matchFeatures`.

### **stereoParams — Camera parameters for stereo system**

stereoParameters object

Camera parameters for stereo system, specified as a `stereoParameters` object. The object contains the intrinsic, extrinsic, and lens distortion parameters of the stereo camera system. You can use the `estimateCameraParameters` function to estimate camera parameters and return a `stereoParameters` object.

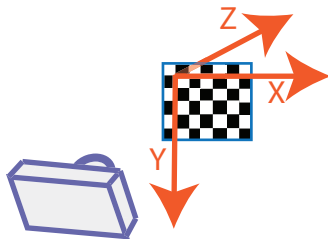
When you pass a `stereoParameters` object to the function, the origin of the world coordinate system is located at the optical center of camera 1. The  $x$ -axis points to the right, the  $y$ -axis points down, and the  $z$ -axis points away from the camera.



### **cameraMatrix1** — Projection matrix

4-by-3 matrix

Projection matrix for camera 1, specified as a 4-by-3 matrix. The matrix maps a 3-D point in homogeneous coordinates onto the corresponding point in the camera's image. This input describes the location and orientation of camera 1 in the world coordinate system. `cameraMatrix1` must be a real and nonsparse numeric matrix. You can obtain the camera matrix using the `cameraMatrix` function.

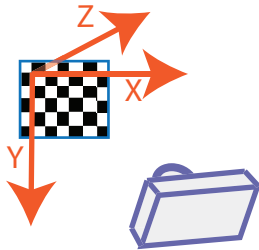


Camera matrices passed to the function, define the world coordinate system.

#### **cameraMatrix2 — Projection matrix**

4-by-3 projection matrix

Projection matrix for camera 1, specified as a 4-by-3 matrix. The matrix maps a 3-D point in homogeneous coordinates onto the corresponding point in the camera's image. This input describes the location and orientation of camera 1 in the world coordinate system. `cameraMatrix1` must be a real and nonsparse numeric matrix. You can obtain the camera matrix using the `cameraMatrix` function.



camera matrices passed to the function, define the world coordinate system.

## Output Arguments

#### **worldPoints — 3-D locations of matching pairs of undistorted image points**

$M$ -by-3 matrix

3-D locations of matching pairs of undistorted image points, specified as an  $M$ -by-3 matrix. The matrix contains  $M$  number of  $[x, y, z]$  locations of matching pairs of undistorted image points from two stereo images. The coordinates are relative to the optical center of camera 1. The function returns `worldPoints` as `double`, if `matchedPoints1` and `matchedPoints2` are `double`. Otherwise the function returns `worldPoints` as `single`.

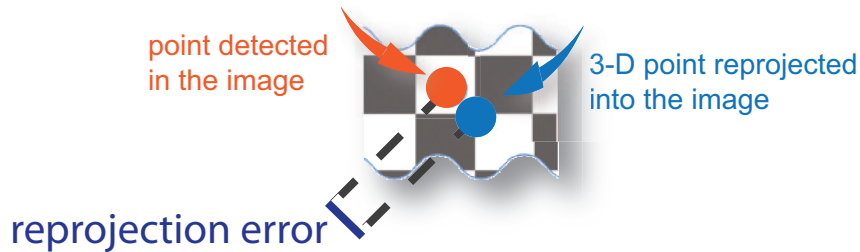
Data Types: `single` | `double`

#### **reprojectionErrors — Reprojection errors**

$M$ -by-1 vector

Reprojection errors, returned as an  $M$ -by-1 vector. The function projects each world point back into both images. Then in each image, the function calculates the reprojection error as the distance between the detected and the reprojected point. The

`reprojectionErrors` vector contains the average reprojection error for each world point.



## More About

### Tips

The `triangulate` function does not account for lens distortion. You can undistort the images using the `undistortImage` function before detecting the points. Alternatively, you can undistort the points themselves using the `undistortPoints` function.

- “Coordinate Systems”

## References

- [1] Hartley, R. and A. Zisserman. "Multiple View Geometry in Computer Vision."  
*Cambridge University Press*, 2003.

## See Also

`stereoParameters` | `cameraParameters` | Camera Calibrator | `cameraMatrix` | `cameraPose` | `estimateCameraParameters` | `reconstructScene` | Stereo Camera Calibrator | `undistortImage` | `undistortPoints`

Introduced in R2014b

## triangulateMultiview

3-D locations of undistorted points matched across multiple images

### Syntax

```
xyzPoints = triangulateMultiview(pointTracks, cameraPoses,  
cameraParams)  
[xyzPoints, reprojectionErrors] = triangulateMultiview(pointTracks,  
cameraPoses, cameraParams)
```

### Description

`xyzPoints = triangulateMultiview(pointTracks, cameraPoses, cameraParams)` returns locations of 3-D world points that correspond to points matched across multiple images taken with a calibrated camera.

`[xyzPoints, reprojectionErrors] = triangulateMultiview(pointTracks, cameraPoses, cameraParams)` also returns reprojection errors for the world points.

#### Code Generation Support:

Supports Code Generation: No

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

### Examples

#### Find 3-D World Points Across Multiple Images Using Triangulation

Load images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata',...  
    'structureFromMotion');  
images = imageSet(imageDir);
```

Load precomputed camera parameters.



```
load(fullfile(imageDir, 'cameraParams.mat'));
```

Compute features for the first image.

```
I = rgb2gray(read(images,1));
I = undistortImage(I,cameraParams);
pointsPrev = detectSURFFeatures(I);
[featuresPrev,pointsPrev] = extractFeatures(I,pointsPrev);
```

Load camera locations and orientations.

```
load(fullfile(imageDir, 'cameraPoses.mat'));
```

Create a viewSet object.

```
vSet = viewSet;
vSet = addView(vSet, 1, 'Points',pointsPrev,'Orientation',...
    orientations(:,:,1),'Location',locations(1,:));
```

Compute features and matches for the rest of the images.

```
for i = 2:images.Count
    I = rgb2gray(read(images, i));
    I = undistortImage(I, cameraParams);
    points = detectSURFFeatures(I);
    [features, points] = extractFeatures(I, points);
    vSet = addView(vSet,i, 'Points',points, 'Orientation',...
        orientations(:,:,i), 'Location',locations(i,:));
    pairsIdx = matchFeatures(featuresPrev,features, 'MatchThreshold',5);
    vSet = addConnection(vSet,i-1,i, 'Matches',pairsIdx);
    featuresPrev = features;
end
```

Find point tracks.

```
tracks = findTracks(vSet);
```

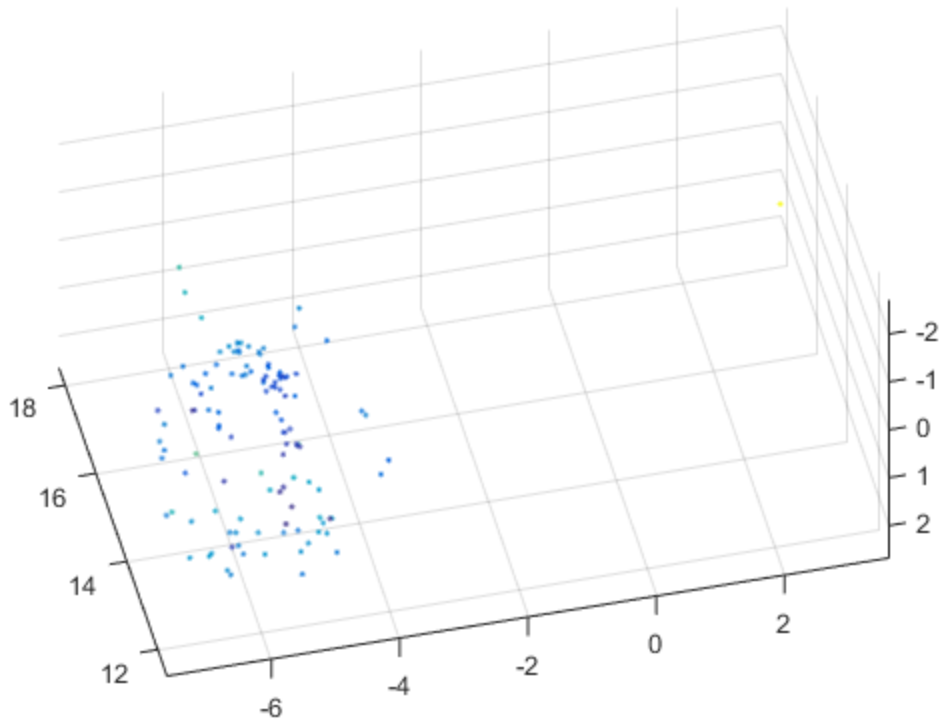
Get camera poses.

```
cameraPoses = poses(vSet);
```

Find 3-D world points.

```
[xyzPoints,errors] = triangulateMultiview(tracks,cameraPoses,cameraParams);
z = xyzPoints(:,3);
```

```
idx = errors < 5 & z > 0 & z < 20;  
pcshow(xyzPoints(idx, :), 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...  
       'MarkerSize', 30);
```



- “Structure From Motion From Two Views”
- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

**pointTracks** — Matching points across multiple images

*N*-element array of pointTrack objects

Matching points across multiple images, specified as an  $N$ -element array of `pointTrack` objects. Each element contains two or more points that match across multiple images.

### **cameraPoses** — Camera pose information

three-column table

Camera pose information, specified as a three-column table. The table contains columns for `ViewId`, `Orientation`, and `Location`. The view IDs correspond to the IDs in the `pointTracks` object. Specify the orientations as 3-by-3 rotation matrices and the locations as three-element vectors. You can obtain `cameraPoses` from a `viewSet` object by using its `poses` method.

### **cameraParams** — Camera parameters

`cameraParameters` object

Camera parameters, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

## Output Arguments

### **xyzPoints** — 3-D world points

$N$ -by-3 array

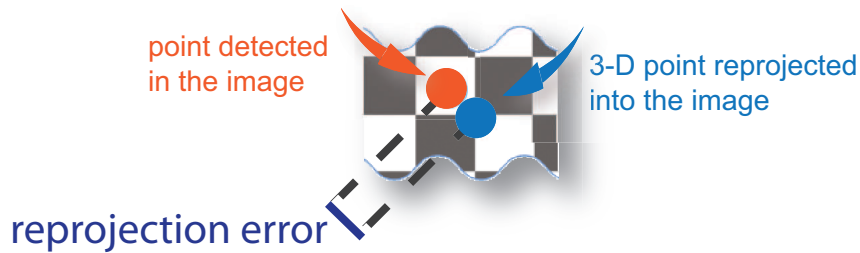
3-D world points, specified as an  $N$ -by-3 array of  $[x,y,z]$  coordinates.

Data Types: `single` | `double`

### **reprojectionErrors** — Reprojection errors

$N$ -by-1 vector

Reprojection errors, returned as an  $N$ -by-1 vector. The function projects each world point back into both images. Then in each image, the function calculates the reprojection error as the distance between the detected and the reprojected point. The `reprojectionErrors` vector contains the average reprojection error for each world point.



## More About

### Tips

Because `triangulateMultiview` does not account for lens distortion, you can undistort the images before detecting the points by using `undistortImage`. Alternatively, you can undistort the points directly using `undistortPoints`.

- “Structure from Motion”
- “Coordinate Systems”

### References

[1] Hartley, R. and A. Zisserman. "Multiple View Geometry in Computer Vision."  
*Cambridge University Press*, 2003.

### See Also

`viewSet` | `cameraParameters` | `pointTrack` | `bundleAdjustment` | `Camera Calibrator` | `cameraPose` | `estimateCameraParameters` | `undistortImage` | `undistortPoints`

**Introduced in R2016a**

# undistortImage

Correct image for lens distortion

## Syntax

```
[J,newOrigin] = undistortImage(I,cameraParams)
[J,newOrigin] = undistortImage(I,cameraParams,interp)
[J,newOrigin] = undistortImage( ____,Name,Value)
```

## Description

`[J,newOrigin] = undistortImage(I,cameraParams)` returns an image, `J`, containing the input image, `I`, with lens distortion removed. The function also returns the `[x,y]` location of the output image origin. The location is set in terms of the input intrinsic coordinates specified in `cameraParams`.

`[J,newOrigin] = undistortImage(I,cameraParams,interp)` specifies the interpolation method for the function to use on the input image.

`[J,newOrigin] = undistortImage( ____,Name,Value)` specifies one or more `Name,Value` pair arguments, using any of the preceding syntaxes. Unspecified properties have their default values.

### Code Generation Support:

Supports Code Generation: Yes

Supports MATLAB Function block: No

“Code Generation Support, Usage Notes, and Limitations”

## Examples

### Correct Image for Lens Distortion

Create a set of calibration images.

```
images = imageSet(fullfile(toolboxdir('vision'),'visiondata','calibration','fishEye'))
```

#### Detect the calibration pattern.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images.ImageLocation);
```

#### Generate the world coordinates of the corners of the squares. The square size is in millimeters.

```
squareSize = 29;  
worldPoints = generateCheckerboardPoints(boardSize, squareSize);
```

#### Calibrate the camera.

```
cameraParams = estimateCameraParameters(imagePoints, worldPoints);
```

#### Remove lens distortion and display the results.

```
I = images.read(1);  
J1 = undistortImage(I, cameraParams);
```

#### Display the original and corrected images.

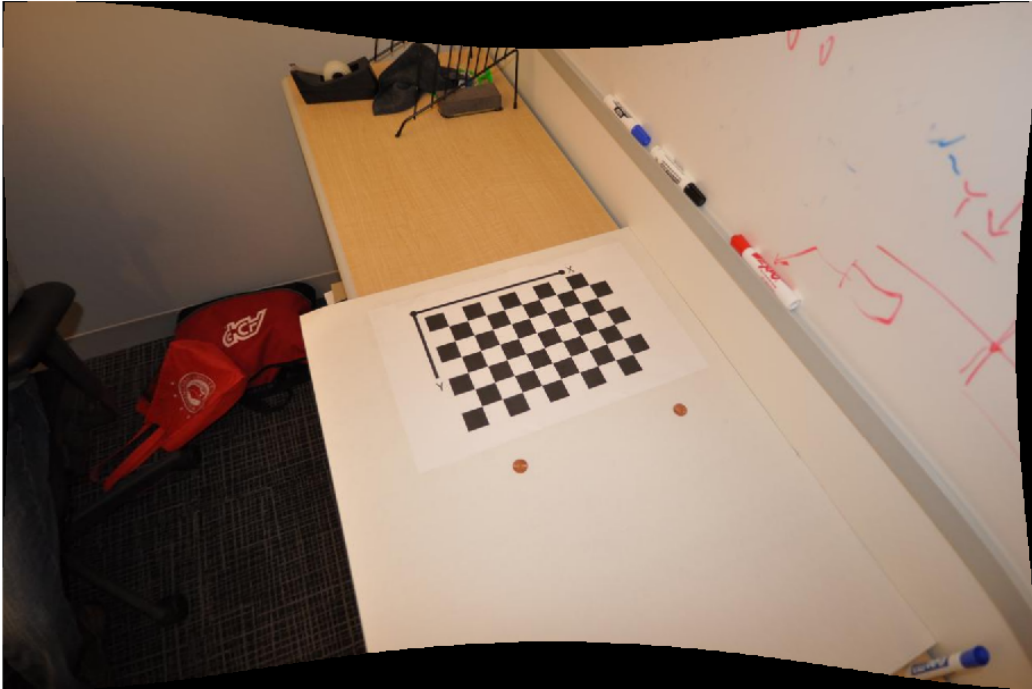
```
figure; imshowpair(I, J1, 'montage');  
title('Original Image (left) vs. Corrected Image (right)');
```

```
J2 = undistortImage(I, cameraParams, 'OutputView', 'full');  
figure; imshow(J2);  
title('Full Output View');
```

Original Image (left) vs. Corrected Image (right)



Full Output View



- “Code Generation for Depth Estimation From Stereo Video”

## Input Arguments

### **I** — Input image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Input image, specified in either *M*-by-*N*-by-3 truecolor or *M*-by-*N* 2-D grayscale. The input image must be real and nonsparse.

Data Types: single | double | int16 | uint8 | uint16 | logical

### **cameraParams** — Camera parameters

cameraParameters object

Camera parameters, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function or the Camera Calibrator app. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

#### **interp** — Interpolation method

'linear' (default) | 'nearest' | 'cubic'

Interpolation method to use on the input image, specified as the string 'linear', 'nearest', or 'cubic'.

## **Name-Value Pair Arguments**

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

Example: 'FillValues',0 sets the sets the output pixel fill values to 0.

#### **'FillValues'** — Output pixel fill values

0 (default) | scalar | 3-element vector

Output pixel fill values, specified as the comma-separated pair consisting of 'FillValues' and an array containing one or more fill values. When the corresponding inverse transformed location in the input image lies completely outside the input image boundaries, you use the fill values for output pixels. When you use a 2-D grayscale input image, you must set the `FillValues` to scalar. When you use a truecolor, `FillValues` can be a scalar or a 3-element vector of RGB values.

#### **'OutputView'** — Size of output image


'same' (default) | 'full' | 'valid'

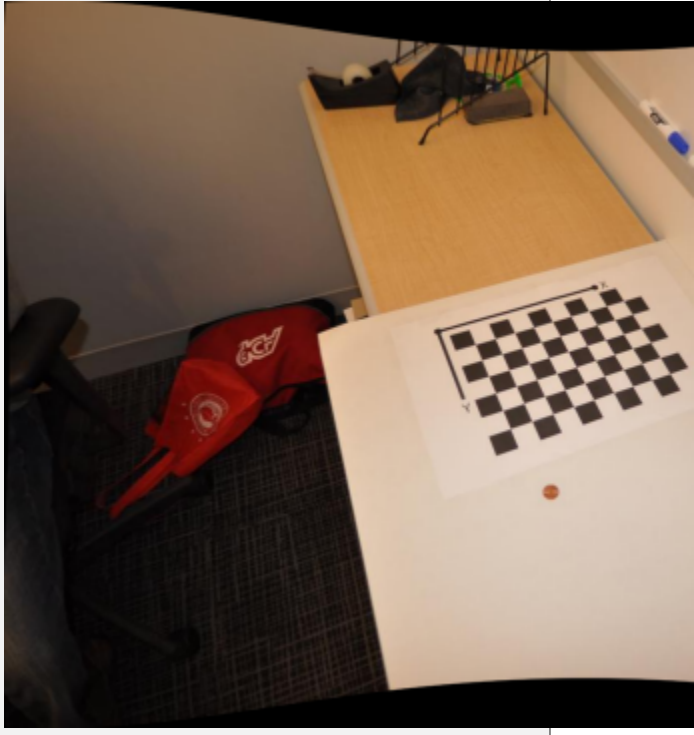
Size of output image, specified as the comma-separated pair consisting of 'OutputView' and the string 'same', 'full', or 'valid'. When you set the property to 'same', the function sets the output image to match the size of the input image. When you set the property to 'full', the output includes all pixels from the input image. When you set the property to 'valid', the function crops the output image to contain only valid pixels.


For the input image:





OutputView	Output Image
' same '	Match the size of the input image. 

OutputView	Output Image
'full'	<p data-bbox="792 302 1187 335">All pixels from the input image.</p>  A photograph of a room. In the foreground, a white surface (possibly a table or desk) has a chessboard on it. The chessboard is a standard 8x8 grid of black and white squares. To the left of the chessboard, there is a red bag or piece of clothing. In the background, there is a wooden desk with some items on it, and a black chair. The room has a grey carpet.

OutputView	Output Image
'valid'	<p data-bbox="793 303 1277 331">Only valid pixels from the input image.</p> 

## Output Arguments

### **J** — Undistorted image

*M*-by-*N*-by-3 truecolor image | *M*-by-*N* 2-D grayscale image

Undistorted image, returned in either *M*-by-*N*-by-3 truecolor or *M*-by-*N* 2-D grayscale.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

### **newOrigin** — Output image origin

2-element vector

Output image origin, returned as a 2-element  $[x,y]$  vector. The function sets the output origin location in terms of the input intrinsic coordinates. When you set `OutputView` to `'same'`, which means the output image is the same size as the input image, the function sets the `newOrigin` to  $[0,0]$ .

The `newOrigin` output represents the translation from the intrinsic coordinates of the output image `J` into the intrinsic coordinates of the input image `I`.

Let  $P_I$  represent a point in the intrinsic coordinates of input image `I`.

Let  $P_J$  represent the same point in the intrinsic coordinates of the output image `J`.

$$P_I = P_J + \text{newOrigin}$$

### See Also

`cameraParameters` | `stereoParameters` | `Camera Calibrator` |

`estimateCameraParameters` | `extrinsics` | `Stereo Camera Calibrator` |

`triangulate` | `undistortPoints`

**Introduced in R2014a**

# undistortPoints

Correct point coordinates for lens distortion

## Syntax

```
undistortedPoints = undistortPoints(points, cameraParams)  
[undistortedPoints, reprojectionErrors] = undistortPoints(points,  
cameraParams)
```

## Description

`undistortedPoints = undistortPoints(points, cameraParams)` returns point coordinates corrected for lens distortion. This function uses numeric nonlinear least-squares optimization.

`[undistortedPoints, reprojectionErrors] = undistortPoints(points, cameraParams)` additionally returns the errors used to evaluate the accuracy of undistorted points.

## Examples

### Undistort Checkerboard Points

**Create an `imageSet` object containing calibration images.**

```
images = imageSet(fullfile(toolboxdir('vision'), 'visiondata', 'calibration', 'fishEye'))  
imageFileNames = images.ImageLocation;
```

**Detect the calibration pattern.**

```
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);
```

**Generate the world coordinates related to the corners of the squares. Square size is in millimeters.**

```
squareSize = 29;
```

```
worldPoints = generateCheckerboardPoints(boardSize, squareSize);
```

#### **Calibrate the camera.**

```
params = estimateCameraParameters(imagePoints, worldPoints);
```

#### **Load an image and detect the checkerboard points.**

```
I = images.read(10);  
points = detectCheckerboardPoints(I);
```

#### **Undistort the points.**

```
undistortedPoints = undistortPoints(points, params);
```

#### **Undistort the image.**

```
[J, newOrigin] = undistortImage(I, params, 'OutputView', 'full');
```

#### **Translate the undistorted points.**

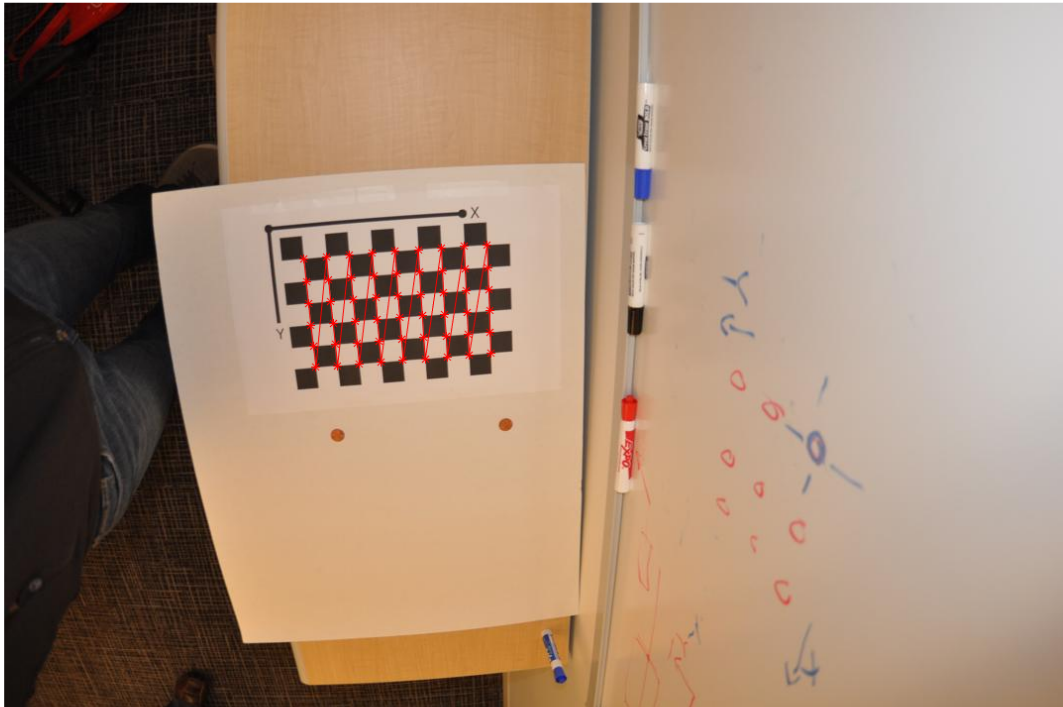
```
undistortedPoints = [undistortedPoints(:,1) - newOrigin(1), undistortedPoints(:,2) - newOrigin(2)];
```

#### **Display the results.**

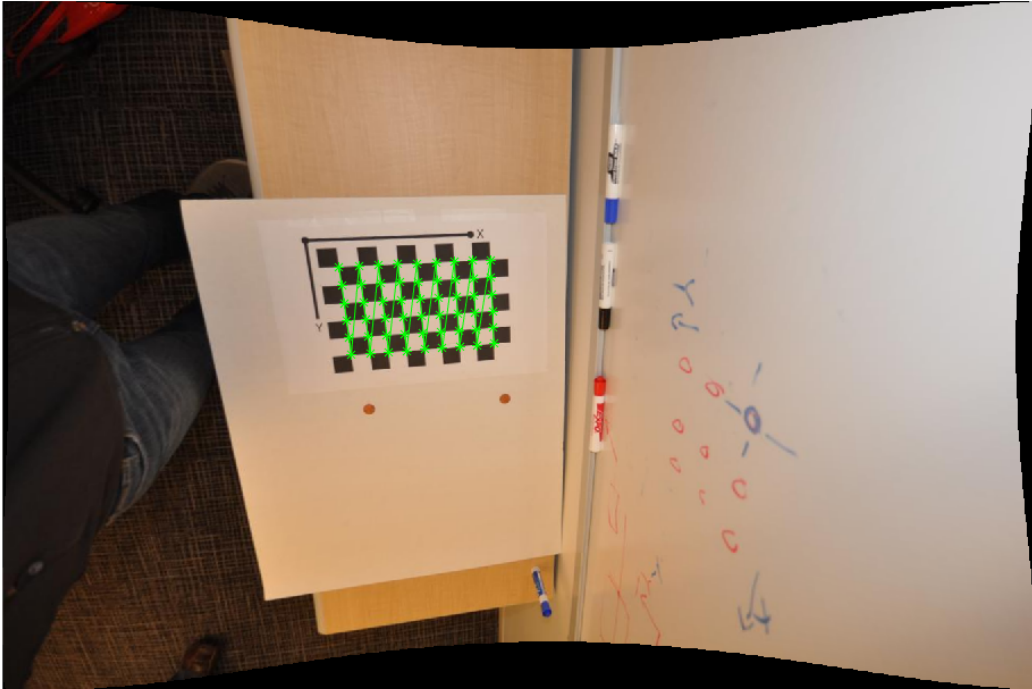
```
figure;  
imshow(I);  
hold on;  
plot(points(:, 1), points(:, 2), 'r*-');  
title('Detected Points');  
hold off;
```

```
figure;  
imshow(J);  
hold on;  
plot(undistortedPoints(:, 1), undistortedPoints(:, 2), 'g*-');  
title('Undistorted Points');  
hold off;
```

Detected Points



Undistorted Points



## Input Arguments

### **points** — Input points

cornerPoints object | SURFPoints object | MSERRegions object | BRISKPoints object |  $M$ -by-2 matrix

Input points, specified as either a BRISKPoints, SURFPoints, MSERRegions, or cornerPoints object, or an  $M$ -by-2 matrix of  $M$  number of [x y] coordinates.

### **cameraParams** — Object for storing camera parameters

cameraParameters object



Camera parameters, specified as a `cameraParameters` object. You can return this object using the `estimateCameraParameters` function. This object contains the intrinsic, extrinsic, and lens distortion parameters of a camera.

## Output Arguments

### **undistortedPoints** — Undistorted points

*M*-by-2 matrix

Undistorted points, returned as an *M*-by-2 matrix. The `undistortedPoints` output contains *M*  $[x,y]$  point coordinates corrected for lens distortion. When you input points as `double`, the function outputs `undistortedPoints` as `double`. Otherwise, it outputs `undistortedPoints` as `single`.

Data Types: `single` | `double`

### **reprojectionErrors** — Reprojection errors

*M*-by-1 vector

Reprojection errors, returned as an *M*-by-1 vector. You can use the errors to evaluate the accuracy of undistorted points. The function computes the errors by applying distortion to the undistorted points, and then taking the distances between the result and the corresponding input points. The `reprojectionErrors` output is in pixels.

## See Also

`cameraParameters` | `stereoParameters` | Camera Calibrator | `estimateCameraParameters` | `extrinsics` | Stereo Camera Calibrator | `triangulate`

**Introduced in R2014b**

## vision.getCoordinateSystem

Get coordinate system for Computer Vision System Toolbox.

### Syntax

```
vision.getCoordinateSystem(coordinateSystem)
```

### Description

`vision.getCoordinateSystem(coordinateSystem)` returns the coordinate system string `coordinateSystem` `RC` or `XY`.

When you set the coordinate system to 'RC', the function sets the Computer Vision System Toolbox to the zero-based, [r c] coordinate system prior to R2011b release.

When you set the coordinate system to 'XY', the function sets the Computer Vision System Toolbox to the one-based, [x y] coordinate system.

This function facilitates transition of Computer Vision System Toolbox MATLAB code prior to R2011b, to newer releases. Refer to the “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” R2011b release notes for further details.

---

**Note:** When you set the coordinate system to `RC`, only your current MATLAB session persists using the `RC` coordinate system. The coordinate system resets to `XY` when you restart MATLAB.

---

### Example

Get the Computer Vision System Toolbox coordinate system.

```
vision.getCoordinateSystem
```

**Introduced in R2011b**

# vision.setCoordinateSystem

Set coordinate system for Computer Vision System Toolbox.

## Syntax

```
vision.setCoordinateSystem(coordinateSystem)
```

## Description

`vision.setCoordinateSystem(coordinateSystem)` sets the coordinate system based on the string `coordinateSystem`.

When you set the coordinate system to 'RC', the function sets the Computer Vision System Toolbox to the zero-based, [r c] coordinate system prior to R2011b release.

When you set the coordinate system to 'XY', the function sets the Computer Vision System Toolbox to the one-based, [x y] coordinate system.

This function facilitates transition of Computer Vision System Toolbox MATLAB code prior to R2011b, to newer releases. Refer to the “Conventions Changed for Indexing, Spatial Coordinates, and Representation of Geometric Transforms” R2011b release notes for further details.

---

**Note:** When you set the coordinate system to RC, only your current MATLAB session persists using the RC coordinate system. The coordinate system resets to XY when you restart MATLAB.

---

## Example

Set the Computer Vision System Toolbox to the coordinate system prior to R2011b.

```
vision.setCoordinateSystem('RC');
```

Set the Computer Vision System Toolbox to the new coordinate system.

```
vision.setCoordinateSystem('XY');
```

**Introduced in R2011b**

## visionlib

Open top-level Computer Vision System Toolbox Simulink library

### Syntax

```
visionlib
```

### Description

`visionlib` opens the top-level Computer Vision System Toolbox block library model.


### Examples

View and gain access to the Computer Vision System Toolbox blocks:

```
visionlib
```

### Alternatives

To view and gain access to the Computer Vision System Toolbox blocks using the Simulink library browser:

- Type `simulink` at the MATLAB command line, and then expand the Computer Vision System Toolbox node in the library browser.
- Click the Simulink icon  from the MATLAB desktop or from a model.

**Introduced in R2011a**

## **visionSupportPackages**

Start installer to download, install, or uninstall Computer Vision System Toolbox data

### **Syntax**

```
visionSupportPackages
```

### **Description**

`visionSupportPackages` launches the Support Package Installer, which you can use to download, install, or uninstall support packages for Computer Vision System Toolbox.

#### **Computer Vision System Toolbox Support Packages**

“Install OCR Language Data Support Package”

“Install and Use Computer Vision System Toolbox OpenCV Interface”

### **Examples**

#### **Start Computer Vision System Toolbox installer**

```
visionSupportPackages
```

**Introduced in R2014b**

# ocvStructToKeyPoints

Convert MATLAB feature points struct to OpenCV `KeyPoint` vector

## C++ Syntax

```
#include "opencvmex.hpp"
void ocvStructToKeyPoints(const mxArray *
in,cv::vector<cv::KeyPoint> &keypoints);
```

## Arguments

in

Pointer to a MATLAB structure, `mxArray`, that represents a point feature. Format:

Field Name	Field Requirement	Field Data Type
Location	Required	Single
Scale	Required	Single
Metric	Required	Single
Orientation	Optional	Single
Octave	Optional	int32
Misc	Optional	int32

## Description

The `ocvStructToKeyPoints` function converts a point feature data structure from a MATLAB struct to an OpenCV's `KeyPoint` vector.

## See Also

`mxArray`, `ocvKeyPointsToStruct`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**



## ocvMxGpuArrayToGpuMat\_{DataType}

Create `cv::gpu::GpuMat` from `mxArray` containing GPU data

### C++ Syntax

```
#include "opencvgpumex.hpp"
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_double(const mxArray
* in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_single(const mxArray
* in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_uint8(const mxArray *
in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_uint16(const mxArray
* in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_uint32(const mxArray
* in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_int8(const mxArray *
in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_int16(const mxArray *
in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_int32(const mxArray *
in);
cv::Ptr<cv::gpu::GpuMat> ocvMxGpuArrayToGpuMat_bool(const mxArray *
in);
```

### Arguments

`in`

Pointer to a MATLAB struct, `mxArray`, containing GPU data. Supported data types:

<code>real_T (double)</code>	<code>real32_T (single)</code>	<code>uint8_T (uint8)</code>
<code>uint16_T (uint16)</code>	<code>uint32_T (uint32)</code>	<code>int8_T (int8)</code>
<code>int16_T (int16)</code>	<code>int32_T (int32)</code>	<code>boolean_T (bool)</code>

### Returns

OpenCV smart pointer (`cv::Ptr`) to a `cv::gpu::GpuMat` object.

### Description

The `ocvMxGpuArrayToGpuMat_{DataType}` function creates a `cv::gpu::GpuMat` object from an `mxArray` containing GPU data. This function requires the Parallel Computing Toolbox software.

### See Also

`mxArray`, `ocvMxGpuArrayFromGpuMat_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvMxGpuArrayFromGpuMat\_{DataType}

Create an mxArray from cv::gpu::GpuMat object

### C++ Syntax

```
#include "opencvgpumex.hpp"
mxArray * ocvMxGpuArrayFromGpuMat_double(const cv::gpu::GpuMat &
in);
mxArray * ocvMxGpuArrayFromGpuMat_single(const cv::gpu::GpuMat &
in);
mxArray * ocvMxGpuArrayFromGpuMat_uint8(const cv::gpu::GpuMat & in);
mxArray * ocvMxGpuArrayFromGpuMat_uint16(const cv::gpu::GpuMat &
in);
mxArray * ocvMxGpuArrayFromGpuMat_uint32(const cv::gpu::GpuMat &
in);
mxArray * ocvMxGpuArrayFromGpuMat_int8(const cv::gpu::GpuMat & in);
mxArray * ocvMxGpuArrayFromGpuMat_int16(const cv::gpu::GpuMat & in);
mxArray * ocvMxGpuArrayFromGpuMat_int32(const cv::gpu::GpuMat & in);
mxArray * ocvMxGpuArrayFromGpuMat_bool(const cv::gpu::GpuMat & in)
```

### Arguments

in

Reference to OpenCV cv::gpu::GpuMat object.

### Returns

Pointer to a MATLAB struct, mxArray, containing GPU data. Supported data types:

real_T (double)	real32_T (single)	uint8_T (uint8)
uint16_T (uint16)	uint32_T (uint32)	int8_T (int8)
int16_T (int16)	int32_T (int32)	boolean_T (bool)

### Description

The `ocvMxGpuArrayFromGpuMat` function creates an `mxArray` from a `cv::gpu::GpuMat` object. `GpuMat` supports 2-D arrays only. This function requires the Parallel Computing Toolbox software.

### See Also

`mxArray`, `ocvMxGpuArrayToGpuMat_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvMxArrayToSize

Convert 2-element mxArray to cv::Size.

## C++ Syntax

```
#include "opencvmex.hpp"
cv::Size ocvMxArrayToSize(const mxArray * in, bool rcInput = true);
```

## Arguments

**in**

Pointer to a MATLAB mxArray having 2 elements.

**rcInput**

Boolean flag that indicates if input mxArray is of the format  $[r\ c]$  or  $[x\ y]$ .

rcInput	in
true (default)	$[r\ c]$ ( <i>height, width</i> )
false	$[x\ y]$ ( <i>width, height</i> )

## Returns

OpenCV cv::Size

## Description

The ocvMxArrayToSize function converts a 2-element mxArray to cv::Size. Empty input ([ ]) returns cv::Size(0,0);

## See Also

mxArray, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvMxArrayToMat\_{DataType}

Convert column major mxArray to row major cv::Mat for generic matrix

### C++ Syntax

```
#include "opencvmex.hpp"
void ocvMxArrayToMat_double(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_single(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_uint8(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_uint16(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_uint32(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_int8(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_int16(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_int32(const mxArray *in, cv::Mat &out);
void ocvMxArrayToMat_bool(const mxArray *in, cv::Mat &out);
cv::Ptr<cv::Mat> ocvMxArrayToMat_double(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_single(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_uint8(const mxArray *in, const bool
copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_uint16(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_uint32(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_int8(const mxArray *in, const bool
copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_int16(const mxArray *in, const bool
copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_int32(const mxArray *in, const bool
copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToMat_uint8(const mxArray *in, const bool
copyData = true);
```

## Arguments

*in*

Pointer to a MATLAB struct, `mxArray`, having column major data. The data can be *n*-channel matrices. Supported data types:

<code>real_T</code> (double)	<code>uint8_T</code> (uint8)	<code>uint32_T</code> (uint32)	<code>int16_T</code> (int16)
<code>real32_T</code> (single)	<code>uint16_T</code> (uint16)	<code>int8_T</code> (int8)	<code>int32_T</code> (int32)

*copyData*

Boolean flag to copy data from `mxArray` to the `Mat` object.

- `true` (default) — The function transposes and interleaves (for RGB images) column major `mxArray` data into a row major `cv::Mat` object.
- `false` — No data copies from the `mxArray` to the `Mat` object. The function creates a new `Mat` wrapper and uses it to point to the `mxArray` data. Because OpenCV is row-based and MATLAB is column-based, the columns of the `mxArray` become the rows of the `Mat` object. If the image is 2-D, then `copyData` is `false`.

*out*

Reference to OpenCV `cv::Mat` with row major data.

## Returns

The functions that set `copyData` return an OpenCV smart pointer (`cv::Ptr`) to a `cv::Mat` object.

## Description

The `ocvMxArrayToMat_{DataType}` function applies to two C++ implementations. One set returns `void` and the other set returns an OpenCV smart pointer. The functions that return `void` reallocate memory for the `cv::Mat` if needed.

The `ocvMxArrayToMat_{DataType}` transposes and interleaves column major `mxArray` data into row major `cv::Mat`. This matrix conversion is a generic routine for any number of channels.



## See Also

`mxArray`, `ocvMxArrayToImage_{DataType}`, `ocvMxArrayFromMat_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvMxArrayToImage\_{DataType}

Convert column major mxArray to row major cv::Mat for image

### C++ Syntax

```
#include "opencvmex.hpp"
void ocvMxArrayToImage_double(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_single(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_uint8(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_uint16(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_uint32(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_int8(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_int16(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_int32(const mxArray *in, cv::Mat &out);
void ocvMxArrayToImage_bool(const mxArray *in, cv::Mat &out);
cv::Ptr<cv::Mat> ocvMxArrayToImage_double(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_single(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_uint8(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_uint16(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_uint32(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_int8(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_int16(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_int32(const mxArray *in, const
bool copyData = true);
cv::Ptr<cv::Mat> ocvMxArrayToImage_bool(const mxArray *in, const
bool copyData = true);
```

## Arguments

**in**

Pointer to a MATLAB struct, `mxArray`, having column major data that represents a 2-D or 3-D image. Supported data types:

<code>real_T</code> (double)	<code>uint8_T</code> (uint8)	<code>uint32_T</code> (uint32)	<code>int16_T</code> (int16)
<code>real32_T</code> (single)	<code>uint16_T</code> (uint16)	<code>int8_T</code> (int8)	<code>int32_T</code> (int32)

**copyData**

Boolean flag to copy data from `mxArray` to the `Mat` object.

- **true** (default) — The function transposes and interleaves (for RGB images) column major `mxArray` data into a row major `cv::Mat` object.
- **false** — No data copies from the `mxArray` to the `Mat` object. The function creates a new `Mat` wrapper and uses it to point to the `mxArray` data. Because OpenCV is row-based and MATLAB is column-based, the columns of the `mxArray` become the rows of the `Mat` object. If the image is 2-D, then `copyData` is **false**.

**out**

Reference to OpenCV `cv::Mat` with row major data.

## Returns

The functions that set `copyData` return an OpenCV smart pointer (`cv::Ptr`) to a `cv::Mat` object.

## Description

The `ocvMxArrayToImage_{DataType}` function applies to two C++ implementations. One set returns `void` and the other set returns an OpenCV smart pointer. The functions that return `void` reallocate memory for the `cv::Mat` if needed.

The `ocvMxArrayToImage_{DataType}` transposes and interleaves column major `mxArray` data into row major `cv::Mat`. The `ocvMxArrayToImage_{DataType}` function supports 2-D and 3-D images.

These functions are not a generic matrix conversion routine. For 3-D images, they take into account that the OpenCV format uses BGR ordering and manipulate the data to comply with that formatting.

### See Also

`mxArray`, `ocvMxArrayToMat_{DataType}`, `ocvMxArrayFromImage_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvMxArrayToCvRect

Convert a MATLAB struct representing a rectangle to an OpenCV `CvRect`

## C++ Syntax

```
#include "opencvmex.hpp"  
CvRect ocvMxArrayToCvRect(const mxArray *in);
```

## Arguments

`in`

Pointer to a MATLAB structure, `mxArray`, that represents a rectangle. The structure must have four scalar-valued fields: `x`, `y`, `width`, and `height`. The (`x`, `y`) fields represent the upper-left corner of the rectangle.

## Returns

OpenCV `CvRect`.

## Description

The `ocvMxArrayToCvRect` function converts a rectangle data structure from a MATLAB struct to an OpenCV `KeyPoint` vector.

## See Also

`mxArray`, `ocvCvRectToMxArray`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvMxArrayFromVector

Convert numeric vectorT to mxArray

### C++ Syntax

```
#include "opencvmex.hpp"
mxArray *ocvMxArrayFromVector(const std::vector<real_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<real32_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<uint8_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<uint16_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<uint32_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<int8_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<int16_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<int32_T> &v);
mxArray *ocvMxArrayFromVector(const std::vector<boolean_T> &v);
```

### Arguments

v

Reference to `vector<DataType>`. Supported data types:

<code>real_T</code>	<code>real32_T</code>	<code>uint8_T</code>
<code>uint16_T</code>	<code>uint32_T</code>	<code>int8_T</code>
<code>int16_T</code>	<code>int32_T</code>	<code>boolean_T</code>

### Returns

Pointer to a MATLAB struct mxArray.

### Description

The `ocvMxArrayFromVector` function converts numeric `std::vector<DataType>` to an mxArray.

## See Also

`mxArray`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvMxArrayFromPoints2f

Converts `vector<cv::Point2f>` to `mxArray`

## C++ Syntax

```
#include "opencvmex.hpp"
mxArray *ocvMxArrayFromPoints2f(const std::vector<cv::Point2f>
&points);
```

## Arguments

`points`

Reference to OpenCV `vector<cv::Point2f>`.

## Returns

Pointer to a MATLAB `mxArray`.

## Description

The `ocvMxArrayFromPoints2f` function converts `std::vector<cv::Point2f>` to an `mxArray`.

## See Also

“C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**



## ocvMxArrayFromMat\_{DataType}

Convert row major `cv::Mat` to column major `mxAarray` for generic matrix

### C++ Syntax

```
#include "opencvmex.hpp"
mxAarray *ocvMxArrayFromMat_double(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_single(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_uint8(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_uint16(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_uint32(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_int8(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_int16(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_int32(const cv::Mat &in);
mxAarray *ocvMxArrayFromMat_bool(const cv::Mat &in);
```

### Arguments

`in`

Reference to OpenCV `cv::Mat` with row major data.

### Returns

Pointer to a MATLAB struct, `mxAarray`, having column major data. Supported data types:

<code>real_T (double)</code>	<code>uint8_T (uint8)</code>	<code>uint32_T (uint32)</code>	<code>int16_T (int16)</code>
<code>real32_T (single)</code>	<code>uint16_T (uint16)</code>	<code>int8_T (int8)</code>	<code>int32_T (int32)</code>

### Description

The `ocvMxArrayFromMat_{DataType}` function creates an `mxAarray` from a `cv::Mat` object. The `mxAarray` contains column major data and `cv::Mat` contains row major data. This matrix conversion is a generic routine for any number of channels.

### See Also

`mxArray`, `ocvMxArrayToImage_{DataType}`, `ocvMxArrayFromImage_{DataType}`,  
`ocvMxArrayToMat_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”,  
“MEX File Creation API”

**Introduced in R2015a**

## ocvMxArrayFromImage\_{DataType}

Convert row major `cv::Mat` to column major `mxArray` for image

### C++ Syntax

```
#include "opencvmex.hpp"
mxArray *ocvMxArrayFromImage_double(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_single(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_uint8(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_uint16(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_uint32(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_int8(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_int16(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_int32(const cv::Mat &in);
mxArray *ocvMxArrayFromImage_bool(const cv::Mat &in);
```

### Arguments

`in`

Reference to OpenCV `cv::Mat` with row major data.

### Returns

Pointer to a MATLAB struct, `mxArray`, with column major data. Supported data types:

<code>real_T (double)</code>	<code>uint8_T (uint8)</code>	<code>uint32_T (uint32)</code>	<code>int16_T (int16)</code>
<code>real32_T (single)</code>	<code>uint16_T (uint16)</code>	<code>int8_T (int8)</code>	<code>int32_T (int32)</code>

### Description

The `ocvMxArrayFromImage_{DataType}` function creates an `mxArray` from a `cv::Mat` object. The `mxArray` contains column major data and the `cv::Mat` contains row major data.

This function is not a generic matrix conversion routine. For 3-D images, it takes into account that the OpenCV format uses BGR ordering and manipulates the data to comply with that formatting.

### See Also

`mxArray`, `ocvMxArrayToImage_{DataType}`, `ocvMxArrayFromMat_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvKeyPointsToStruct

Convert OpenCV `KeyPoint` vector to MATLAB struct

## C++ Syntax

```
#include "opencvmex.hpp"
mxAArray *ocvKeyPointsToStruct(cv::vector<cv::KeyPoint> &in);
```

## Arguments

in

Reference to an OpenCV's `KeyPoint` vector.

## Returns

Pointer to a MATLAB structure `mxAArray` that represents a point feature.

Format:

Field Name	Field Requirement	Field Data Type
Location	Required	Single
Scale	Required	Single
Metric	Required	Single
Orientation	Optional	Single
Octave	Optional	int32
Misc	Optional	int32

## Description

The `ocvKeyPointsToStruct` function converts a point feature data structure from an OpenCV `KeyPoint` vector to a MATLAB struct.

## See Also

`mxArray`, `ocvStructToKeyPoints`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvCvRectToMxArray

Convert OpenCV `CvRect` to a MATLAB struct

## C++ Syntax

```
#include "opencvmex.hpp"  
mxArray *ocvCvRectToMxArray(const CvRect *in);
```

## Arguments

`in`

Pointer to OpenCV `CvRect`.

## Returns

Pointer to a MATLAB structure, `mxArray`, that represents a rectangle. The structure must have four scalar-valued fields, `x`, `y`, `width`, and `height`. The (`x`, `y`) fields represent the upper-left corner of the rectangle.

## Description

The `ocvCvRectToMxArray` function converts a rectangle data structure from an OpenCV `KeyPoint` vector to a MATLAB struct.

## See Also

`mxArray`, `ocvMxArrayToCvRect`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvCvRectToBoundingBox\_{DataType}

Convert `vector<cv::Rect>` to *M*-by-4 `mxArray` of bounding boxes

### C++ Syntax

```
#include "opencvmex.hpp"
mxArray * ocvCvRectToBoundingBox_double(const std::vector<cv::Rect>
& rects);
mxArray * ocvCvRectToBoundingBox_single(const std::vector<cv::Rect>
& rects);
mxArray * ocvCvRectToBoundingBox_uint8(const std::vector<cv::Rect> &
rects);
mxArray * ocvCvRectToBoundingBox_uint16(const std::vector<cv::Rect>
& rects);
mxArray * ocvCvRectToBoundingBox_uint32(const std::vector<cv::Rect>
& rects);
mxArray * ocvCvRectToBoundingBox_int8(const std::vector<cv::Rect> &
rects);
mxArray * ocvCvRectToBoundingBox_int16(const std::vector<cv::Rect> &
rects);
mxArray * ocvCvRectToBoundingBox_int32(const std::vector<cv::Rect> &
rects);
```

### Arguments

`rects`

Reference to OpenCV `vector<cv::Rect>`.

### Returns

Pointer to a MATLAB `mxArray` having *M*-by-4 elements. Supported data types:

<code>real_T (double)</code>	<code>uint8_T (uint8)</code>	<code>uint32_T (uint32)</code>	<code>int16_T (int16)</code>
<code>real32_T (single)</code>	<code>uint16_T (uint16)</code>	<code>int8_T (int8)</code>	<code>int32_T (int32)</code>



## Description

The `ocvCvRectToBoundingBox_{DataType}` function converts `vector<cv::Rect>` to an  $M$ -by-4 `mxArray` of bounding boxes.

## See Also

`mxArray`, `ocvCvBox2DToMxArray`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

## ocvCvBox2DToMxArray

Convert OpenCV CvBox2D to a MATLAB struct

### C++ Syntax

```
#include "opencvmex.hpp"  
mxArray *ocvCvBox2DToMxArray(const CvBox2D *in);
```

### Arguments

`in`

Pointer to OpenCV CvBox2D.

### Returns

Pointer to a MATLAB structure, `mxArray`, that represents a rectangle. The structure must have five scalar-valued fields: `x_center`, `y_center`, `width`, `height`, and `angle`. The (`x_center`, `y_center`) fields represent the center of the rectangle.

### Description

The `ocvCvBox2DToMxArray` function converts a rectangle data structure from an OpenCV CvBox2D to a MATLAB struct.

### See Also

`mxArray`, `ocvCvRectToBoundingBox_{DataType}`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

# ocvCheckFeaturePointsStruct

Check that MATLAB struct represents feature points

## C++ Syntax

```
#include "opencvmex.hpp"
void ocvCheckFeaturePointsStruct(const mxArray *in);
```

## Arguments

`in`

Pointer to a MATLAB structure, `mxArray`, that represents point feature. Format:

Field Name	Field Requirement	Field Data Type
Location	Required	Single
Scale	Required	Single
Metric	Required	Single
Orientation	Optional	Single
Octave	Optional	int32
Misc	Optional	int32

## Description

The `ocvCheckFeaturePointsStruct` function performs the key point struct checker.

## See Also

`mxArray`, `ocvStructToKeyPoints`, `ocvKeyPointsToStruct`, “C/C++ Matrix Library API”, “MEX Library API”, “MEX File Creation API”

**Introduced in R2015a**

